TABLE OF CONTENTS

Lecture 1. Introduction
R.E. Nance. The time and state relationships in simulation modeling. Communications of the ACM 24(4), 1981. pp. 173-179..... 3

Lecture 2. System Theory
Paper: Russell L. Ackoff and Jamshid Gharajedaghi. On the Mismatch between Systems and their Models. 13

Lecture 3. System Specification
T.J. Schriber, D.T. Brunner, & J.S. Smith. Inside Discrete-Event Simulation Software: how it works and why it matters. In: Proceedings WSC 2015, L. Ylmaz at al. (Eds.). IEEE, 2015, pp 1-15..... 31

Lecture 4. DEVS
Chapter 1 from Zeigler, B.P., H. Praehofer and T.G. Kim (2000). Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, 2nd Ed.) Academic Press. 49
Yentl Van Tendeloo, Hans Vangheluwe. Introduction to Parallel DEVS Modelling and Simulation. SpringSim-Tutorial, 2017 April 23-26, Virginia Beach, VA, USA. SCS, 2017. 71

Lecture 5. DEVS Extensions
B.P. Zeigler. DEVS Today: Recent Advances in Discrete Event-Based Information Technology. Proceedings of the 11TH IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS'03). IEEE, 2003. 85
B.P. Zeigler. Embedding DEV&DESS in DEVS. DEVS Integrative M&S Symposium (DEVS 2006). Huntsville, Alabama, USA, April 2-6, 2006..... 99

Lecture 6. Object-oriented Simulation
P.H.M. Jacobs, N.A. Lang, A. Verbraeck. DSOL: A Distributed Java based discrete event simulation architecture. In: E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes (Eds.). Proceedings of the 34th Winter Simulation Conference. IEEE Computer Society Press. 2002. pp. 793-800..... 119
J.A. Joines and S.D. Roberts. Simulation in an Object-Oriented World. In: P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds. (Eds.). Proceedings of the 1999 Winter Simulation Conference. IEEE Computer Society Press. 2002. pp. 132-140..... 127

Lecture 7. Parallel and Distributed Simulation
Richard Fujimoto. 2015. Parallel and distributed simulation. In Proceedings of the 2015 Winter Simulation Conference (WSC '15). IEEE Press, Piscataway, NJ, USA, pp. 45-59. 139
Kalyan S. Perumalla. 2006. Parallel and distributed simulation: traditional techniques and recent advances. In Proceedings of the 38th conference on Winter simulation (WSC '06), L. Felipe Perrone, Barry G. Lawson, Jason Liu, and Frederick P. Wieland (Eds.). Winter Simulation Conference. 2006. pp. 84-95. 155

Lecture 8. Distributed Simulation using HLA
Christopher D. Carothers, Richard M. Fujimoto, Richard M. Weatherly, and Annette L. Wilson. 1997. Design and implementation of HLA time management in the RTI version F.0. In Proceedings of the 29th conference on Winter simulation (WSC '97), Sigrún Andradóttir, Kevin J. Healy, David H. Withers, and Barry L. Nelson (Eds.). IEEE Computer Society, Washington, DC, USA, 373-380. DOI=http://dx.doi.org/10.1145/268437.268511 169
Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. 1997. The Department of Defense High Level Architecture. In Proceedings of the 29th conference on Winter simulation (WSC '97), Sigrún Andradóttir, Kevin J. Healy, David H. Withers, and Barry L. Nelson (Eds.). IEEE Computer Society, Washington, DC, USA, 142-149. DOI=http://dx.doi.org/10.1145/268437.268465..... 177

Lecture 9. Interactive and Real-time Simulation

Young Kwan Cho, Xiaolin Hu and Bernard P. Zeigler. The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems. Simulation. Vol. 79, Issue 4, April 2003, pp. 197–210. DOI: 10-1177/0037549703038880 187

Peter H.M. Jacobs, Alexander Verbraeck, and William Rengelink. Emulation with DSOL. In: M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, (Eds.). Proceedings of the 2005 Winter Simulation Conference. IEEE, 2005. pp. 1453-1462. 201

Lecture 10. Simulation and Gaming

Alexander Verbraeck, Stijn-Pieter A. van Houten. From Simulation to Gaming: An Object-Oriented Supply Chain Training Library. In: M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, (Eds.). Proceedings of the 2005 Winter Simulation Conference. IEEE, 2005. pp. 2346-2354. 213

Rick van Krevelen, Martijn Warnier, Frances Brazier, Alexander Verbraeck and Thomas Corsi. Transparency, Consistency and Modularity of Strategic Reasoning: An Agent Architecture for Interactive Business Simulations. In: S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, (Eds.). Proceedings of the 2011 Winter Simulation Conference. IEEE, 2011, pp. 1617-1628. 223

Lecture 11. Multi-Paradigm Simulation

H. Vangheluwe, J. de de Lara, P.J. Mosterman. An introduction to multi-paradigm modelling and simulation. In: F. Barros and N. Giambiasi (eds.), Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems), Lisboa, Portugal, 2002. pp. 9-20. 237

H. Vangheluwe, H. and J. de de Lara. "Meta-models are models too", In: E. Yücesan, C.-H. Chen, J.L. Snowdon, and J.M. Charnes (eds). Proceedings of the 2002 Winter Simulation Conference. New York: ACM Press. 2002. pp. 597-605. 249

Lecture 12. Multi-Resolution Simulation

Y. Yilmaz, A. Lim, S. Bowen, & T. Ören. Requirements and Design Principles for Multisimulation With Multiresolution, Multistage Multimodels. In: S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, eds. Proceedings of the 2007 Winter Simulation Conference. IEEE, 2007. pp. 823-832 261

Mamadou D. Seck, H. Job Honig. Multi-perspective modelling of complex phenomena. Comput Math Organ Theory (2012) 18:128–144. DOI: 10.1007/s10588-012-9119-9 271

Lecture 13. Simulation Languages 1

Richard E. Nance. Simulation Programming Languages: An Abridged History. In: C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman (Eds.). Proceedings of the 1995 Winter Simulation Conference. IEEE, 1995. pp. 1307-1313 291

Ole-Johan Dahl and Kristen Nygaard. SIMULA - An ALGOL-based Simulation Language. Communications of the ACM, Vol 9, No 9, 1966. pp. 671-678 299

Lecture 14. Simulation Languages 2

T.W. Tewoldeberhan, A. Verbraeck and V. Hlupic. Implementing a discrete-event simulation software selection methodology for supporting decision making at Accenture. Journal of the Operational Research Society (2010) 61, 1446-1458. 309

Lecture 1. Introduction

Papers to study for the exam

- R.E. Nance. The time and state relationships in simulation modeling. Communications of the ACM 24(4), 1981. pp. 173-179.
  From: http://dl.acm.org/citation.cfm?id=358601

Simulation Modeling and Statistical Computing
N. Adam Guest Editor

The Time and State Relationships in Simulation Modeling

Richard E. Nance
Virginia Tech

Time and state descriptions form the core of a simulation model representation. The historical influence of initial application areas and the exigencies of language implementations have created a muddled view of the time and state relationships. As a consequence, users of simulation programming languages work in relative isolation; model development, simulation application, model portability, and the communication of results are inhibited and simulation practice fails to contribute to the recognition of an underlying foundation or integrating structure. A model representation structure has been forged from a small set of basic definitions which carefully distinguish the state and time relationships. This paper focuses on the coordination of the time and state concepts using "object" as the link. In addition to clarifying the relationships, the structure relates the concept of "state-sequenced simulation" to the variations in time flow mechanisms. In conclusion, some speculations are offered regarding alternative algorithms for time flow mechanisms.

Key Words and Phrases: model representation, definitions, event scheduling, activity scan, process interaction, simulation programming languages, model documentation, state-sequenced simulation, Conical Methodology, time flow mechanisms

CR Category: 8.1

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This work was done while the author was on study/research leave at the Combat Systems Department of the Naval Surface Weapons Center, Dahlgren, Virginia 22448.

Author's present address: Richard E. Nance, Department of Computer Science, Virginia Tech, Blacksburg, VA 24061.

© 1981 ACM 0001-0782/81/0400-0173 $00.75.

I. Introduction

The recognition of similar program structures and functions led to the development of several simulators between 1960–63, e.g., the General Simulation Program (GSP) [43], the Sequence Diagram Simulator [8], GPSS [13], SIMPAC [23], Control and Simulation Language (CSL) [26], SIMSCRIPT [27], GASP [18], and SIMULA [35]. The emergence of these general tools for discrete event modeling, while offering advantages over the problem-oriented (general purpose) languages (POL*s*), introduced different modeling perspectives as well as the influences of varying problem domains.

Lackner [24, p.3] describes a modeling perspective as a *Weltansicht* or world view that "... must be at least implicitly established to permit the construction of a simulation language." In a subsequent work [25], he identifies the world views that categorize the different simulation programming languages (SPL*s*). Later, Kiviat [19, 20] compares and contrasts the three world views:

(1) Event-oriented, which describes the model states in terms of the consequences of events, programmed to describe "... how system state changes take place" [19, p. 52].
(2) Activity-oriented, which describes the actions of objects comprising the model and the conditions for these actions to take place.
(3) Process-oriented, which considers "... a set of events that are associated with a system behavior description" so as to combine "... the run-time efficiency of event scheduling with the modeling efficiency of activity scanning" [20, p. 22].

(Lackner's original categorization differentiated between event-oriented and activity-oriented views, considering the process perspective as identical with the latter.)

One unfortunate legacy of the different perspectives has been a geographical communications gap. The activity-oriented view, stimulated by the utility of the "wheel chart" [44] (later called "activity cycle" and "entity cycle" diagrams), predominated in the U.K. The popularity of ALGOL contributed to the success of SIMULA among European simulation practitioners. Notwithstanding the success of GPSS with its easily learned process interaction view, the event-oriented view dominated the U.S. scene.

A more unfortunate legacy is the inability to generalize the simulation modeling task. Language implementations of world views have often obscured important distinctions and slight but significant differences in terminology. Kiviat [19, p. 11] notes that the long-term results are the "different formal mechanisms" and the "slightly different interpretation of the theory" afforded (or enforced) by each SPL.

Language differences or modeling differences are not a problem as such, but when an *interpretation* is perceived as a *theory*, confusion arises as to the primacy of concepts and the generality of application. The seemingly independent and concurrent development of several SPL*s* during 1960–63 and shortly thereafter magnified the

confusion caused by the "interpretation/theory inversion."

Is the simulation community really suffering from this language-induced inversion of theory and interpretation? Many would answer in the negative. However, simulation still carries the label of an expensive, uncertain problem-solving technique that represents the "court of last resort." This view persists despite the considerable advances in output analysis and improved capability of both hardware and software systems since 1970. Documentation of the shortcomings of computerized models, with simulation models prominent among those cited, is found in [6].

The concern with simulation extends beyond those who use the technique or those who underwrite the project costs. Others, admittedly a much smaller group, are concerned with the lack of recognition of a fundamental structure—a theory—after so many years of practice. The possibility is suggested that a causal relationship exists among these four factors:

(1) The absence of a theory, i.e., some fundamental definitional and integrating concepts spanning individual SPL domains;
(2) The theory/interpretation inversion in language-directed modeling;
(3) The multiplicity of languages, emanating from quite different application contexts;
(4) The high cost and questionable accuracy of simulation.

Efforts to reduce the deleterious effects of these factors include the application of computer-assisted modeling tools [22, 29, 38], research in model representation and documentation [32, 33], and the emergence of methodological approaches to simulation model development [14, 37, 34]. To date, Zeigler's work [46, 47] has progressed the farthest toward a descriptive theory of simulation modeling; however, other efforts such as using systems theory [16, 17] or the SIMSCRIPT-influenced entity/attribute/set approach [28] indicate a growing recognition of the need for an integrating general framework.

The objectives of this paper are:

(1) To illustrate the imprecision and subtle representational differences present in the approaches to discrete event simulation;
(2) To demonstrate the counterproductive influences of the imprecisions and differences;
(3) To present a set of definitions that clearly separate yet coordinate the time and state relationships;
(4) To consider the implications of this approach to simulation model development.

II. Time and State: The Model Coordinates

A. The Definitional Disorder

Early papers comparing SPL*s* utilized tabular presentations to assist the reader in understanding the variations in terminology relating to the same concept. These tables introduced the terminology used in particular SPL*s*, but did little to reveal semantic differences among languages. Fundamental differences in modeling views were hidden. However, the most serious misperceptions were fostered by the subtle distinctions in meaning applied to the same term in different SPL*s*.

A long list of these "pseudohomographs" serves no useful purpose. Instead, we focus on the term *event* and its relationship to the terms *activity* and *process* since these are principal terms in distinguishing among the three world views. At least four currently used SPL*s* are represented in the list below.

—(An event is) the instantaneous change in the value of one or more state variables [12, p. 3].

—This change of state is an event [4, p. 48].

—The events in a simulation are those particular times when something happens or should have happened [42, p. 83].

—An event occurs at any point in time beyond which the status of a system cannot be projected with certainty [41, p. 13].

—An event is an instant in time at which an activity starts or stops. The basic unit of action in a SIMSCRIPT II simulation is an activity [21, p. 282].

—Activity—a process that proceeds over a period of time.
—Event—a change in the state of the system that indicates the start or completion of an activity [5, p. 58–59].

—A process is an activity that proceeds over time. Its initiation, alteration, or conclusion is called an *event*. If, and only if, an event occurs, the state of the system changes [9, p. 76].

These examples describe an event as a point in time, a state change, or a state change at a point in time. An event is characterized as bounding a process or bounded by a process. An event is also described as initiating or terminating an activity, and the activity (initiation or termination) is characterized as causing an event. An event is defined in terms of a change in the system state in one case, and the change in state is predicated on the occurrence of an event in another definition. If the focus is shifted from event to process, for example, the situation can be made even more confusing (from [3, p. 91]):

—Descriptive units are the modules from which a description of the behavior of a system is constructed.
—*Activity.* The sequence of state changes prescribed by a descriptive unit.
—*Process.* A single instance or occurrence, during a simulation, of the activity prescribed in a descriptive unit.
—*Interaction Point.* A point within an activity at which control may be transferred to some other process.
—*Event.* The activity between two interaction points. An event always occurs instantaneously in simulated time.

The point is not that some definitions are "wrong" and others are "right," because each can be justified in its presentational context. The issue is: Can the differences among the definitions be reconciled within a clear, coherent understanding of *event* in the context of discrete event simulation? We think not—at least, not within the current understanding of the term.

B. Significance of the Disorder

The differences in terminology have contributed to the isolation of language users, albeit not as substantially as the geographical origins of the SPL or the availability of vendor-supplied language support. Installations acquired and supported only a single SPL. University courses emphasized a specific SPL available for student use, although possibly mentioning others (see [31]). Thus, few installations and fewer simulation analysts have used more than one SPL [2], and most simulations are written in a POL (also see [45]). This situation focused attention on the resolution of syntactic and semantic problems in a particular SPL, diverting interest from more fundamental issues. The persistence of the specific language focus is documented in the classical paper of Conway [7] as early as 1963.

Ironically, the diversion of attention from concepts and principles is now exacerbated by the spreading awareness of SPL developments. The subtle differences in terminology contribute more to this unfortunate circumstance than the glaring, more obvious examples. Students, novice users, and even experienced users assume a definitional uniformity that does not exist. Language extensions, especially when motivated by the desire to accommodate other world views, are achieved at the expense of altered definitions or "slight" redefinitions of the original terms. The consequence is a deviation from the original clarity of concept and cohesion in structure. Thus, the apparently beneficial spread of SPL awareness has proved detrimental in the sense that the efforts to increase language generality have contributed to the warping of concepts and the distortion of structures. This result contributes to increased impedance in the communication between model developer and model user, a significant factor in "cost overruns" and model disutility [6, p. 7–9].

The specificity of a SPL view imposes an insularity, beyond that characterized as user isolation above, by its mixing of syntactic, semantic, and modeling requirements during the model development and testing phases. The preoccupation with the *program representation* distracts from the more crucial aspects of the model itself. Zeigler [48] cites several reasons for the "model/program confusion" and emphasizes this misperception by citing a reference that defines a model as a program. Simulation analysts are given language training that contributes to the two problems cited above: Isolation and the relegation of concepts to a minor role. Insularity is promoted by the supposition that a model is a program. Furthermore, model portability is severely restricted since "documentation" is perceived to be "program documentation." Even if the documentation is adequate for the program representation, it is not likely to sufficiently meet the requirements for *model documentation* (see [33, p. 83–84]).

In summary, the definitional disorder discourages the realization of simulation foundations and inhibits the development and portability of models. It contributes to the difficulties in communication among model developers, model users, and potential model users (who are unable to assess utility or applicability). Therefore, the creation of order should concern simulation practitioners no less than those seeking a theory.

III. Coordinating the Time and State Relationships

In this section a set of definitions are offered that correspond closely to the terminology suggested by Kiviat [19, 20]. The differences arise primarily in the more detailed treatment of attributes and the clear distinction between state and time. Also essential to an understanding of the coordination of state and time is the structure imposed on the model during the development process. A brief explanation of the model development approach is provided.

A. Objects and Attributes

In typical fashion a simulation model is considered to be comprised of objects that are described in terms of their attributes (see [20]). In the definitions that follow, two conventions are utilized: (1) The use of indentation to reflect subordination among terms, and (2) The italicizing of a term that is subsequently defined.

—A model of a system is comprised of *objects* and the *relationships* among objects.
—An object is anything that can be characterized by one or more *attributes* to which *values* are assigned.
—Attributes are object descriptors of two types: *Indicative* and *relational.*

Indicative attributes describe an aspect of the object; i.e., provide some knowledge about the object.

Relational attributes relate the object to one or more other objects; i.e., describe the relationships among objects.

—Values assigned to attributes conform to an attribute typing much the same as in a POL; logical, numerical, and character string are examples of value types.

These definitions are a selected subset of those used in the Conical Methodology (see [34]), which requires a detailed classification of attributes.

B. Definition and Specification

The application of the Conical Methodology guides the model development progressively (but not necessarily sequentially) through two stages: Definition and specification. Definition proceeds through a top-down decomposition of the model into submodels, submodels into lower level submodels, and so forth until a base (atomic)

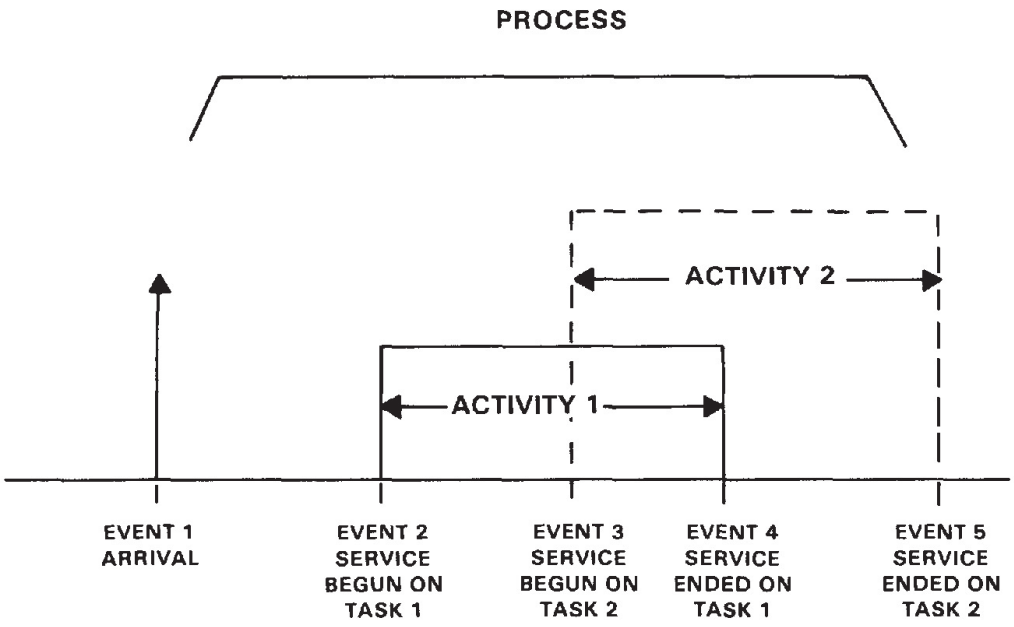


Fig. 1. Event, Process, and Activity. (From FISHG73, p. 24)

level is reached. Specification is accomplished by the description of the value assignment expressions for each attribute of an object and the synthesis of objects to form higher level objects (submodels). Note that, by the above definitions, each submodel is an object, and the model itself is an object.

Among other things, the top-down definition and bottom-up specification establish structural relationships of a hierarchical nature among objects. Thus, a scoping of objects and object attributes can be derived. Further, the relational attributes provide the means for describing other types of relationships among objects (termed "coordinate" to distinguish them from "hierarchical").

C. Time and State Definitions

Every simulation model must have an indexing attribute, that is, an attribute of the model object that enables state transitions. Time is the most common indexing attribute, and we employ the common term "system time" in the definitions below.[1]

—*An instant* is a value of system time at which the value of at least one attribute of an object can be assigned (altered).

—*An interval* is the duration between two successive instants.

—*A span* is the contiguous succession of one or more intervals.

—*The state of an object* is the enumeration of all attribute values of that object at a particular instant.

The above definitions establish three measures of time: A point (instant), a duration between two points (interval), and a contiguous succession of durations (span). Instants where no attribute value for an object changes have no meaning for that object (but must have meaning for some object). The hierarchical scoping imposes the recognition of an instant for all objects superior to the object experiencing a change in attribute value. Of course, every instant has meaning for one object—the model.

The time and state concepts are brought together in the definitions of the familiar terms below:

—*An activity* is the state of an object over an interval.

—*An event* is a change in object state, occurring at an instant, that initiates an activity precluded prior to that instant.

> An event is *determined* if the only condition on event occurrence can be expressed strictly as a function of system time. Otherwise, the event is *contingent*.

—*An object activity* is the state of an object between two events describing successive state changes for that object.

—*A process* is the succession of states of an object over a span (or the contiguous succession of one or more object activities).

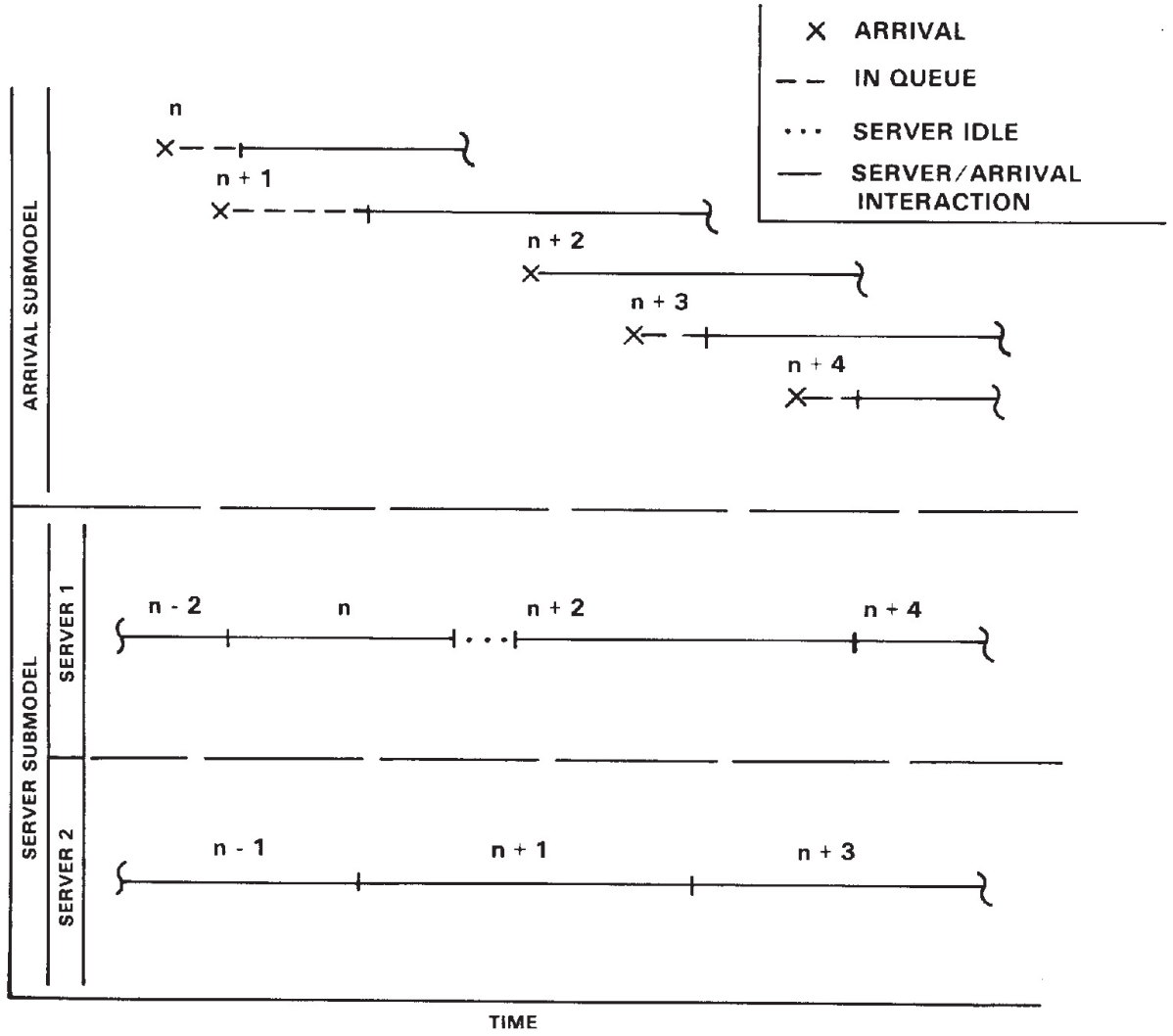
Keep in mind that an activity for a designated object is bounded by two successive events *for that object*. Intervening events, related to state changes for other objects, define intervals during which no state change occurs for the designated object. The object activity is terminated by the first instant such that an attribute value changes for the designated object.

IV. Implications and Speculations

The coordinated treatment of time and state, with clear distinctions between the two, has several implications for model development. The most important benefit is the removal of ambiguity surrounding the terms *event*, *activity*, and *process*. An equally important benefit which is currently being investigated but is not discussed here, is the potential for creating a primitive model description language that explicates the relations among the three world views of current SPLs (see [39]). Another impor-

[1] A model using space as an indexing attribute seems quite plausible.

Fig. 2. Two-Server Queueing Model Illustrating the Linking of the Activity and Process Concepts to Objects.



tant implication is the explanation of the relation of the concept of "state-sequenced simulation" to the existing time flow mechanisms (TFMs). Finally, the coordinated treatment of time and state permits some additional insights regarding the world views, the timing mechanisms, and the classification of SPLs.

A. Improved Precision in Definitions

A comparison by Fishman [10, p. 24] of the three world views indicates the current imprecision.

> . . . an event signifies a change in state of an entity. A process is a sequence of events ordered on time. An activity is a collection of operations that transform the state of an entity.

Figure 1, also taken from [10, p. 24], illustrates the three concepts.

The revised definitions of Sec. III extend these concepts by making the object relationship explicit. For example, the interval between the instants for Events 2 and 3 define an activity for the object of Task 1, but the Task 1 object activity is defined for the interval initiating at the instant for Event 2 and terminating at the instant for Event 4.

The definitions also remove the implied restriction of *process* to temporary entities (objects), which is revealed in the characterization ". . . the process interaction approach emphasizes the progress of an entity through a system from its arrival event to its departure event" [10, p. 25]. In accordance with the treatment of *process* in SIMULA, it is considered applicable for describing both permanent and temporary entities. The revised definition above is suitably general, only imposing the requirement that, just as an object activity is bound by events related to the designated object, a process is comprised of one or more object activities.

Figure 2 applies an extension of the Fishman diagram to a two-server queuing model to illustrate the above points. Activities and processes are defined in terms of events and objects (or object attributes, since objects are defined in terms of their attributes). The arrival submodel describes the "customer" objects labeled n, $n+1$, . . . from some arbitrary instant. The process for each "customer" can begin with a queueing activity followed by a service activity (e.g., customer n) or can consist only of a service activity (e.g., customer $n+2$).

During the time segment shown, the second server process includes no idle time; however, the server submodel process does contain an idle period. The hierarchical relation of the server submodel process to that for each server is quite apparent. With respect to the issue of simultaneity, the revised definitions specify that instants are unique; thus, intervals must take on positive values. The simultaneity complications treated in the interesting paper by Parnas [40] are avoided. All attribute value assignments at the same instant collectively define the event. Using the terminology of probability theory, an event can be simple—a single attribute value change, or compound—more than one. The ordering of value assignments comprising a compound event remains the responsibility of the modeler; however, continuing research effort is exploring techniques of model analysis to assist in the detection of potential ordering difficulties as well as model incompleteness and inconsistency.

B. State-Sequenced Simulation

In a 1975 presentation, Norton and Schaefer [36] described a state compression technique which would remove the requirement of strict time sequencing for some events. The result was to obviate the need for a time flow mechanism. This technique was referred to as "schedule independent simulation." At that time analysis of the requirements for schedule independence supported the conclusion that the technique had limited applicability. However, due in part to the obscurity surrounding the modeling concepts, the approach could not be related to the existing concepts of TFM*s* and the dynamic aspects of model representation.

A similar result is obtained with a technique described by Fishman [11, p. 126–129], based on a paper by Hordijk, Iglehart, and Shassberger [15]. Utilizing the property of exponential interevent times, the simulation execution can be reduced to the generation of states of a Markov chain, treating the time between transitions independently.

The schedule independent and Markov chain approaches are grouped under the term, "state-sequenced simulation." Figure 3 is intended as a conceptual aid to understanding the time and state relationships, in general, and in relating state-sequenced simulation to TFMs, in particular. Figure 3 is an expansion of an illustration of the continuum of TFM algorithms, presented nearly a decade ago [30], and re-examined recently by Atkins [1]. Time and state are depicted as orthogonal axes in Figure 3, and each axis is scaled conceptually as increasing, i.e., the time axis increases from very small to very large time increments. The state axis is scaled from very detailed changes (simple events) to highly aggregated changes (involved compound events). Thus, a TFM using fixed time increments lies along the time axis while a TFM might be postulated that strictly considers state sequences, with time treated implicitly, as lying along the state axis. The latter possibility is indicative of a state-sequenced approach. The relative position of a TFM on either axis is dependent on model attribute values. Employment of a state-sequenced approach is dependent on the property of time invariance: The division of the execution sequence into a repetitive subsequence of state changes that are not conditioned on time. A model with this property admits an execution that treats system time as a subordinate attribute.

The related Markov chain treatment described by Fishman is an example of the state-sequenced approach to the modeling situation where the state transitions remain probabilistic but the interevent times can be handled deterministically. This approach appears to be closely related to the use of a conditions list, which is described in the following paragraph.

Consider another implication of Figure 3: The possibility of forming a conditions list, analogous to the events list of an event scheduling SPL. An indexing variable would test for a listed condition (evaluate an expression of attribute values) to determine the state transition. This possibility, a more restrictive structure than used in activity scanning SPL*s* (see [47, p. 160–161]), represents a practical means for handling the typically complex problem of determining the time invariant subsequence of state changes. Is it possible that the conditions list could better match the inherent structure of some models than an events list?

V. Summary

In reference to the objectives stated in the Introduction, it has been demonstrated that the proposed definitions remove some difficulties caused by the confounding of state and time in the discrete event simulation literature and among SPL*s*. The separation of state and time

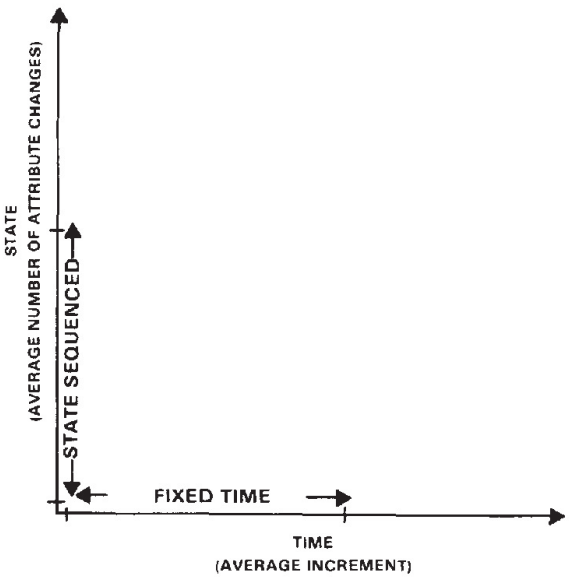


Fig. 3. Conceptual Illustration of the Conjunction of Space and Time in a TFM and the Contrast in the Fixed-Time and State-Sequenced Approaches

enables a more precise coordination of the concepts, helping to explain the relation of state-sequenced simulation to the explicit time treatments and stimulating some speculations on alternative TFM algorithms. This precision is a necessary precursor to the creation of an axiomatic approach to simulation model development.

Acknowledgments. The author acknowledges stimulating conversations with C. M. Overstreet, B. P. Zeigler, S. C. Mathewson, and J. G. Laski which contributed to the ideas developed in this paper.

Received 4/80; revised 10/80 accepted 12/80

References

1. Atkins, M. Stella A comparison of SIMULA and GPSS for simulating sparse traffic. *Simulation, 34,* 3 (March 1980), 93–100.
2. Beckwith, Richard E. Survey of simulation education: I. The perceived need. ORSA/TIMS Joint National Meeting, San Juan, Puerto Rico, October 16–18, 1974.
3. Blunden, G.P. and Krasnow, H.S. The process concept as a basis for simulation modeling. *Simulation, 9,* 2 (August 1967), 89–93.
4. Bobillier, P.A., Kahan, B.D. and Probst A.R., *Simulation with GPSS and GPSS V.* Prentice–Hall, Englewood Cliffs, N.J., 1976.
5. Colella, A.M., O'Sullivan, M.J., and Carlino, D.J. *Systems Simulation: Methods and Applications.* D.C. Heath, Boston, Mass., 1974.
6. Comptroller General of the U.S., Ways to improve management of Federally funded computerized models. LCD-75-111, General Services Administration, August 23, 1976.
7. Conway, Richard W. Some tactical problems in digital simulation. *Management Sci., 10,* 1 (October 1963), 47–61.
8. Dietmeyer, D. L., Gordon, G., Runyon, J.P., and Tague, B.A. An interpretive simulation program for estimating occupancy and delay in traffic-handling systems which are incompletely detailed. Proc. IEEE Pacific General Meeting, San Diego, CA, 1960.
9. Emshoff, James R. and Sisson, Roger L. *Design and Use of Computer Simulation Models.* MacMillan, New York, 1970.
10. Fishman, George S. *Concepts and Methods in Discrete Event Digital Simulation.* John Wiley, New York, 1973.
11. Fishman, George S. *Principles of Discrete Event Simulation.* John Wiley, New York, 1978.
12. Franta, William R. *The Process View of Simulation,* Elsevier–North Holland, New York, 1977.
13. Gordon, Geoffrey A general purpose systems simulator. (unpublished manual) IBM Corporation ASDD Commercial Department, October 25, 1960.
14. Holbaek-Hanssen, E., Handlykken, P., and Nygaard, K. System description and the DELTA language. Rept. No. 4 (Publication No. 523), Norwegian Computing Center, Oslo, Norway, 1977.
15. Hordijk, A., Iglehart, D.L., and Shassberger, R. Discrete time methods for simulating continuous time Markov chains. Rept. No. 35, Department of Operations Research, Stanford Univ, Stanford, CA, Jan. 1975.
16. Kindler, Evzen On the way to a mathematical theory of simulation. *Elektronische Informationsverarbeitung und Kybernetik, 12,* 1 (1976), 497–504.
17. Kindler, Evzen Dynamic systems and theory of simulation. *Kybernetika, 15,* 2 (1979), 77–87.
18. Kiviat, P.J. GASP—A general activity simulation program. Project No. 90.17-019(2), Applied Research Lab., United States Steel Corp., Monroeville, PA, July 1963.
19. Kiviat, Philip J. Digital computer simulation: Modeling concepts. RAND Memo. RM-5378-PR RAND Corp., Santa Monica, CA, Aug. 1967.
20. Kiviat, Philip J. Digital computer simulation: Computer programming languages. RAND Memo RM-5883-PR, RAND Corp., Santa Monica, CA, Jan. 1969.
21. Kiviat, P.J., Villanueva, R., and Markowitz, H. M. *SIMSCRIPT II Programming Language.* CACI Inc., Los Angeles, CA, 1973.
22. Kleine, Henry. Software design and documentation language. JPL Publication 77–24, Jet Propulsion Lab., Pasadena, CA, July 1, 1977.
23. Lackner, Michael R. SIMPAC: A research tool for simulation. SDC Document SP-220, System Development Corp., Santa Monica, CA, March 13, 1961.
24. Lackner, M.R. Toward a general simulation capability. Proc. of the SJCC, San Francisco, CA, May 1–3, 1962, 1–14.
25. Lackner, Michael R. Digital simulation and system theory. System Development Corp. SDC Document SP-1612, Santa Monica, CA, April 6, 1964.
26. Laski, John C. and Buxton, John N. C.S.L. Control and simulation language. Esso Petroleum Company, Ltd. and IBM United Kingdom, Ltd., London, England, 1962.
27. Markowitz, H.M., Hausner, B., and Karr, H.W. SIMSCRIPT: A simulation programming language. Rept. RM-3310-Pr; RAND Corp., Santa Monica, CA, 1962.
28. Markowitz, Harry M. SIMSCRIPT. Res. Rept. RC 6811 (29158), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, July 26, 1979.
29. Mathewson, S. C. Computer aided simulation modeling and experimentation. Proc. Eighth Australian Comptr Conf. Canberra, Australia, Aug. 28–Sept 1, 1978, 9–13.
30. Nance, Richard E. On time flow mechanisms for discrete systems simulations. *Management Sci. 18,* 1 (Sept. 1971), 59–73.
31. Nance, Richard E. A comparison of discrete event simulation courses based on a small sample survey. Tech. Rept. CS76010-E, Dept of Comptr Sci, Virginia Tech, Blacksburg, Virginia 24061, July 1976.
32. Nance, Richard E. The feasibility of and methodology for developing Federal documentation standards for simulation models. Final Rept. to the Natl. Bur. Stand., Dept of Comptr Sci., Virginia Tech, June 26, 1977.
33. Nance, Richard E. Model representation in discrete event simulation: Prospects for developing documentation standards. In *Current Issues in Computer Simulation,* N. Adam and A. Dogramaci, Eds., Academic Press, New York, 1979, 83–97.
34. Nance, Richard E. Model representation in discrete event simulation: The Conical Methodology. Tech. Rept. CS81003-R, Dept of Comptr Sci, Virginia Tech, Blacksburg, VA, March 15, 1981.
35. Nygaard, K. SIMULA, An extension of ALGOL to the description of discrete event networks. Proc. IFIPS Cong., North-Holland, New York, 1963, 520–522.
36. Norton, Suzanne and Schaefer, Brian M. A new methodology of discrete systems simulation: Schedule independent simulation. *Bull Operations Res. Soc. Am.,* Chicago, IL, April 30 and May 1–2, 1975, B-69.
37. Ören, Tuncer I. and Zeigler, Bernard P. Concepts for advanced simulation methodologies. *Simulation, 32,* 3 (March 1979), 69–82.
38. Ören, Tuncer I. Computer-aided modeling systems (CAMS). Plenary Address, Simulation '80 Symp., Interlaken, Switzerland, June 25–27, 1980.
39. Overstreet, C. Michael, Research progress report. Dept. Comptr Sci, Virginia Tech, (limited availability), 1978.
40. Parnas, David L. On simulating networks of parallel processes in which simultaneous events may occur. *Comm. ACM, 12,9* (Sept. 1969), 519–531.
41. Pritsker, A.B., and Young, R.E. *Simulation with GASP-PL/1.* John Wiley, New York 1975.
42. Reitman, Julian *Computer Simulation Applications.* Wiley-Interscience, New York, 1971.
43. Tocher, K.D. and Owen, D.G. The automatic programming of simulations. Proc. Second Int. Conf. on Operational Research, 1960, 50–68.
44. Tocher, K.D. Some techniques of modeling building proceedings. IBM Scientific Computing *Symposium Simulation Models and Planning. IBM Data Process Division, 1966 (Conference held in White Plains, New York, Dec. 7–9, 1964), 119–155.*
45. Tocher, K.D. Keynote address. Proc. Winter Simulation Conf., Volume II., San Diego, California, Dec. 1979, 640–654.
46. Zeigler, Bernard P. Towards a formal theory of modeling and simulation: Structure preserving morphisms. *J ACM, 19,* 4 (April 1972) 742–764.
47. Zeigler, Bernard P. *Theory of Modeling and Simulation.* John Wiley, New York, 1976.
48. Zeigler, Bernard P. Integrated model pluralism: An alternative to a universal modeling language. Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, 1979.

Lecture 2. System Theory

Papers to study for the exam

- Paper: Russell L. Ackoff and Jamshid Gharajedaghi. On the Mismatch between Systems and their Models. Retrieved from: http://www.acasa.upenn.edu/System_MismatchesA.pdf. This is a revised version of their original article: "Reflections on Systems and Their Models", by Russell Ackoff and Jamshid Gharajedaghi that was published in Systems Research, Vol. 13, No. 1, March 1966, pp. 13-23.

ON THE MISMATCH BETWEEN SYSTEMS AND THEIR MODELS

Russell L. Ackoff and Jamshid Gharajedaghi

Most contemporary social systems are failing. The educational system deteriorates as functional illiteracy increases. *Head Start* is said to be a failure. So-called healthcare institutions fail to provide needed care to more than 40 million people. In fact, there is considerable evidence that the alleged health care system itself produces and maintains illness and disability. The United States has a higher percentage of its population in prison than any other developed country, but nevertheless has the highest crime rate. Most of the corporations formed each year fail before the year is up. The number of corporate bankruptcies is on an increasing trend. The average life of an American corporation is now about fourteen and a half years. Half the corporations on the Fortune 500 list twenty-five years ago no longer exist. One could go on and on citing deficiencies in the management of our principal social systems.

The unsettling concern is that the same sets of social problems have been with us for most of the last fifty years. Little progress has been made in dealing with them despite the efforts of administrations with both left and right ideological orientations. As a result, many have come to believe that such problems as drug addiction, crime, poverty, inequitable health care, inadequate education and alienation from work and society, are an integral part of modern life and not much if anything can be done about them.

There is obviously no one reasons for all these system failures but, we will argue, there is one very large contributing factor to all of them: the way these systems are conceptualized, modeled. There is a very serious mismatch between most social systems and the models of them that are in use.

Barry M. Richmond, creator of the *Systems Dynamics* model *and I-think language* makes it clear that systems and the models of them in use are not the same. According to him "the way we think is outdated." He goes on to define thinking as:

> consisting of two activities: constructing mental models, and then simulating them in order to draw conclusions and make decisions. The mental model is a "selective abstraction" of reality that we create and carry around in our head. As big as some of our heads get, we still can't fit reality in there. Therefore all mental model are simplifications. They necessarily omit many aspects of the realities they represent.

To think about anything requires an image or a concept of it, a model. To think about something as complex as a social system we use models of similar, simpler, and/or more familiar systems. Unfortunately, as social systems become increasingly more complex, simpler mental models of them do not reflect their emerging properties. This unfortunate mismatch results in an inability to deal effectively with critical social problems.

Why does the mismatch of social systems and their models result in so many system failures? To answer this question we must (1) develop a clear understanding of what a system is, (2) identify the kinds of systems there are, (3) identify the kinds of mental models we often use to represent them, and (4) reveal the effects of mismatching a system and its representation?

THE NATURE OF A SYSTEM

We have dealt with the nature of systems extensively elsewhere. (See Ackoff, 1999, Chapter 1.) However, for our purpose here it is sufficient to recall the following:

1. *A system is a whole that is defined by its function(s) in one or more containing systems.*

2. *Every system contains at least two essential parts and these must satisfy three conditions.*

3a. *Every essential part of a system can affect its behavior or properties.*

3b. *The way an essential part affects the properties or behavior of the whole depends on the state or activity of at least one other part of the system.*

3c. *Groups of essential parts, subsystems, also can affect the behavior and properties of the whole system and none has an independent effect on it.*

Some important properties of a system derive from its definition.

When a system is taken apart it loses its essential properties.

No part of a system can carry out the function that defines the system.

When a system is taken apart its essential parts lose their ability to carry out the function they have in the whole.

Finally:

When the performance of any essential part of a system, taken separately, is improved, the performance of the whole may not be.

THE TYPES OF SYSTEM

There are many different ways of classifying systems. Different classifications have different uses. We have found the one we present here to be the most useful for understanding the poor performances of social systems. Since we believe *choice (*and purposeful behavior which derives from it) is at the heart of human and social development our classification depends on whether the essential parts of a system or the whole can display choice and, therefore, have purposes. This concern yields the four types of system shown in Table 1.

TABLE 1. TYPES OF SYSTEM

	Parts	**Whole**	**Example**
Deterministic	No choice	No choice	Clock
Ecological	Choice	No choice	Nature
Animate	No choice	Choice	Person
Social	Choice	Choice	Corporation

Significant continuities exist as we come down the Table and each category contains a variety of systems with important differences between them.[2] It is important to note that there is a hierarchical relationship among the four categories shown in Table 1: each type has or uses parts that are systems of the type above it in Table 1. Nevertheless, each type has a distinct emergent characteristic that distinguishes it from all other system types.

Deterministic Systems

All mechanisms are deterministic systems. Their behavior and the behavior of their parts are determined by their internal structure and their environments. A clock, for example, continues to run only if rewound or has its battery replaced, or is fed electricity, or is kept moving for much of the time. An automobile's behavior is determined by its users and the behavior of its essential parts. But the behavior of its essential parts is determined not by their choices, because they cannot make any, but on their physical state or that of their environment.

Although deterministic systems have no purposes of their own, they normally serve the purpose(s) of one or more complex systems; for example, their creators, controllers, or users. Their function is to provide such a service. Although the parts of deterministic systems do not have purposes of their own, they do have functions of their own: to serve the function of the system of which they are a part. Therefore, the behavior of all parts of a deterministic system is also determined.

The behavior and properties of deterministic systems are determined by their structure, causal laws, and—if they are open systems—by other systems in their environments. (The only completely closed system is the universe,) This is even true of computers; they are mechanisms although they appear to make choices, but it is not so. Their behavior is completely determined by their structure together with the information and program put into them by external sources. If we know these, in principle if not in practice, we can determine what a computer will do in any specified situation. The programmed instructions in a computer are its causal laws. These, together with its internal structure and externally provided inputs, completely determine its behavior.

Deterministic systems can be differentiated by the number of functions they have. For example, an ordinary clock has only one function, to tell time. However, some clocks also provide the date and the day and date, and even temperature, humidity and atmospheric pressure. The more functions a deterministic system performs, the more complex it is. The measure of complexity of a system is the number of variables and their interactions required to explain the behavior of the system.

Ecological Systems

Nature is the most familiar and inclusive ecological system. Some parts of ecological systems can display choice but the whole can not. As Daniel Quinn pointed out (1995), nature includes people, however unaware of this fact they may be. But the effects of their behavior are determined. We can affect our environments and often do, but the ways the environments react to our actions are determined. For example, the purposeful use of fluorocarbons as a propellant or DDT as a pesticide affect our environments, but their effects are determined, not a matter of choice.

Ecological systems are *living* systems in the sense contained in Zeleny's definition. They are self-organizing and self-maintaining. Life is currently defined in terms of *autopoiesis*:

> the maintenance of units and wholeness, while components themselves are being continuously or periodically disassembled and rebuilt, created and decimated, produced and consumed. (Zeleny, 1981, p. 5)

As we will see, it follows from this definition that animated, as well as, social systems are living systems.

Ecological systems serve the purposes of the organisms and social systems that are their parts; they provide inputs necessary for the survival of the biological and social systems (plants) that they contain. They also take the waste discharged by these systems.

Animate Systems

Not all biological systems are animate. Plants, for example, do not make choices but respond to external and internal conditions deterministically. Nevertheless they are also *organisms*. Animate organisms such as humans can make choices but their parts cannot. (Unless otherwise indicated, we will use the term 'organism' to refer only to animated systems.)

Plants *react* to external conditions in such a way as to make their survival possible. However, their reactions are determined, not matters of choice. Reactions are determined; responses involve choice.

Survival is obviously an important objective of animate systems. But their parts—for example, organs—have functions but no objectives of their own. Their functions, however, are necessary for the survival of the whole.

The organs of a human body like machines are mechanisms. The behavior of one's heart and lungs is determined in the same way that the behavior of a motor of an automobile is. Their function, however, is to make life possible, but they do not make choices.

Social Systems

All organizations are social systems. They are significantly different from organisms. They contain parts that are animate systems that display choice. (They may also use or have mechanical systems as parts; for example, production equipment in a manufacturing companies.) Organizations display choice. We commonly refer to choices made by corporations, government, and other types of social systems; for example, schools, hospitals, and government agencies. Note also that social systems are usually parts of larger social systems and that these usually contain other social systems. For example a corporation is part of an economic system that contains other corporations. A university is part of an educational system that contains many other schools. Some primitive societies that existed in complete isolation were not part of a larger social system, but became so when discovered.

TABLE 2. POSSIBLE SYSTEM-MODEL MATCHES

Types of Model	*Types of System*			
	Deterministic	**Ecological**	**Animate**	**Social**
Deterministic	DD	DE	DA	DS
Ecological	ED	EE	EA	ES
Animate	AD	AE	AA	AS
Social	SD	SE	SA	SS

Historically, animate systems have often been treated as if they were nothing but complicated deterministic systems. Mechanistic biology dominated biology for centuries. For example, the biochemist Roux is said to have taken the following position:

> According to Roux, biology admits of exact formulation because matter alone exists; there is no ground for a fundamental distinction between the living and the non-living. The animate, appearing as cells with nuclei, developed from the inanimate by the operation of mechanical laws, and is governed by them. (Flower in Clarke and Nahm, 1942, p. 72)

Reil, Lamarck, Rudolphi, Berzelius, Verworn and Loeb are among other eminent biologists who also had a mechanistic point of view of living systems.

Opposition to a deterministic conception of organisms surfaced in the mechanist-vitalist controversy that arose in the late nineteenth century. Mechanistic biologists were unable to adequately account for the nature of life. Today, when life tends to be defined in terms of self-organization and self-renewal, it is apparent that essential aspects of animate systems are not included in mechanistic models of them.

On the other hand, deterministic systems or entities, have seldom been conceptualized as organisms. An exception can be found among some primitive peoples whose beliefs are said to have been "animatistic," which is:

> the doctrine that a great part, if not the whole, of the inanimate kingdom, as well as all animate beings, are endowed with reason, intelligence, and volition, identical with that of man. *(Encyclopaedia Britannica*, Eleventh Edition, Vol. 2, p. 53)

We are not aware of effort to model organisms or mechanical systems as social systems. However, social systems have often been, and still are, modeled as organisms (Beer, 1972) and even deterministically (Forrester, 1961 and 1971). The sociologist Sorokin (1928) summarized the mechanistic interpretations of two prominent social physicists, Haret and Barcelo, as follows:

> In their works the translation of the mechanistic language of social science into that of mechanics goes on in the following way. The individual is transformed into a material point, and his social environment into a "field of forces,"…. As soon as this is done, there is no difficulty in applying the formulas of mechanics to social phenomena; all that is necessary to copy these formulas, inserting the word individual instead of material point, and the term social group instead of physical system or a field of forces. "An increase in the kinetic energy of an individual is equivalent to a decrease in his potential energy." "The total energy of an individual in his field of forces remains constant throughout all its modifications…and so on." (pp. 17-18)

In addition, Sorokin wrote that "H. C. Carey's principles of Social Science is one of the most conspicuous attempts in the second half of the nineteenth century at a physical interpretation of social phenomena." (p. 13) Carey applied such laws as those of gravitation to social phenomena. For example, if an individual is taken as a molecule and the social groups as a body, then the attraction between any two bodies is said to be in direct proportion to their masses (the number of individuals per unit volume) and inversely proportionate to the square of the distance between them. In addition, Carey took centralization and decentralization of populations to be the same as centripetal and centrifugal forces.

Herbert Spencer, the nineteenth century evolutionary philosopher, provides a striking example of biological modeling of social systems. His position was summarized by Hussong (1931) as follows:

> Spencer himself groups together under four heads those comparisons of life and society which result in showing three phenomena well known to characterize *life*, to be no less characteristic of anything to be called a *society*. They are: (1) growth; with which is associated (2) increasing differentiation of structure, and (3) increasing differentiation of function. (p. 23)

To clarify his first point Spencer argued as follows:

> In both biological and social organisms, growth is evidenced by the same phenomena. In both, there are increases in mass—in the biological individual, and expansion from germ to adult form; in the social, and expansion from small wandering bodies to great nations. In both, aggregates of different classes reach various sizes—among which biological organisms, the Protozoa rarely increase beyond a microscopic size; among social organisms the primitive Tasmanian seldom form large groups, while the empires of civilization include millions of people. In both, increases by simple multiplication of units is followed by union of groups and unions of groups of groups. In both, finally, a multiplication of individuals goes on within each group of units. (p. 23)

Stafford Beer (1972) introduced a celebrated model of what he called a "viable system," one that is based on a model of the human brain. He took the requirements for effective individual and organizational behavior to be the same. The difficulties in implementing his model derive from the purposeful nature of the parts of organizations. Organismic models do not take the purposes of the parts of an organism into account. However, such models may be usefully applied to social systems in those rare cases in which the purposes of the parts are very limited or are

apparently not relevant, for example, in organizations that are managed or ruled *autocratically*. The more autocratic an organization is, the more useful an organismic model of it can be.

A problem arises in applying an organismic model to social systems because of (1) the increasing education of the (human) members of a social system, (2) increasing technology that they must master to do the tasks assigned to them, and (3) the increasing variety of demands made on them. When those managed, governed or ruled know how to carry out their functions better than those who manage, govern or rule them, the less effective autocratic management or rule is. A *democratic* organization—one in which the members have considerable freedom and opportunity to make choices—can't be adequately modeled as organisms precisely because such modeling misses this most important characteristic of such a social system, the ability of its parts to make choices. These choices can (but may not) significantly affect the choices made by the whole. For example, plans and programs selected at the corporate level may fail to be implemented because of opposition to them by individual members of the corporation with responsibility for their implementation.

We recently asked a group of managers in an executive development program that had just completed working on a case, what would happen if they presented their solution to the relevant corporation's senior management. The class members said that these managers would probably finds a number of reasons for not accepting it, and if they accepted it, it would probably not be implemented as intended because of opposition to it by those who would be responsible for its implementation. We then pointed out that *the managers and the implementers were part of the problem, not external to it.* In the organismic model of the corporation which the class had unconsciously used, the purposes of those who had to approve of any proposed action, let alone those who had to carry it out, were not taken into account. Had the class used a social systemic model they would have treated acceptance and implementation of its solution to the problem as part of the problem, not as separate from it.

In the political arena finding what is normally thought of as a solution to a problem, and getting it accepted and implemented are usually treated separately rather than as necessary aspects of the problem. For example, many laws are simply not obeyed or enforced and therefore solve nothing. Laws that made homosexual behavior illegal are a case in point. President Clinton's proposed solution of the national health care problem was rejected by Congress whose approval was required before it could be enacted. Furthermore, many problem solutions that are implemented are sabotaged by those who implement them. This is the case when alleged solutions facilitate corruption.

MISMATCHING SYSTEMS AND MODELS

One can model a part or an aspect of a social system as a mechanism and by so doing may improve the performance of that part or aspect of the system, but this can reduce the performance of the whole. Optimization of the parts of a social system can affect performance of the whole For example, minimizing finished-goods inventory (hence the cost of carrying it) can reduce sales that, in turn, this can reduce corporate performance.

Recall that every essential part of a system can affect the performance of the whole but cannot do so independently of other essential parts. Therefore, in changing performance of any essential part, its effect on the system as a whole should be taken into account.

Deterministic Models Applied to Organizations

In the early stages of industrialization, machines replaced thousands of agricultural workers. This resulted in a very large number of unemployed unskilled workers that could have had a destabilizing effect on western societies. It was then that a new way of manufacturing "came to the rescue." Production processes were designed much as an assembly of parts each of which involves a very simple and repetitive task. Then unskilled workers could be assigned to these elementary tasks and be treated as replaceable machine parts. Subsequently, as Taylor (1911) showed, they could be studied as machines through time-and-motion studies and have their productivity increased. This mechanical model of production converted the army of former agricultural workers into industrial workers.

The impact of the use of a mechanistic model of organizations on productivity was so great that in one generation it provided an amount of goods and services that surpassed all previous expectations.

Enterprises created in the nineteenth century were conceptualized (as the universe was) as mechanisms created by their gods to do their work. Their owners had virtually unlimited control over their organizations; they were not yet subject to significant interventions by and constraints imposed by governments and unions.

The enterprise conceptualized as a mechanism had no purpose of its own, but it had a function that was to serve its owner's purposes. The principal purpose of its owner was to make a profit.

This concept of an enterprise could survive in the latter part of the nineteenth and early part of the twentieth century for the following reasons. First, the skills required of the work force were low and there was a plentiful supply of unskilled workers. To a large extent they were illiterate immigrants who had low aspirations for themselves but high aspirations for their children. Second, there was no form of social security, hence work was the only way to support oneself and ones dependents. Third, because there was a large pool of people looking for work, workers could easily be replaced. They were treated as if they were replaceable machine parts. The only alternatives available to the unemployed were to be supported by others or destitution. Therefore, most people were willing to work under any conditions, however difficult, and most of them had to do so.

Summarizing, deterministic models may work very well when applied to parts or aspects of a social system when these parts or aspects are considered in isolation. However, when the performance of each part or aspect of a system taken separately is improved, the performance of the whole may be reduced, and usually is.

The extent of the benefit or harm done by using a model that is not of the same type as the system modeled depends on the level of development of the system involved. For example, when it was initiated, Henry Ford's production system approximated a mechanical system for the previously noted reasons.
But, Henry Ford's phenomenal success in the creation of a mechanistic mass-production system marked the beginning of the end of production era. It contained the seeds of its demise. He failed to appreciate how his success had changed the game by transforming the problem from production to marketing. When he said, "they can have any color [of automobile] they want as long as it is black," this gave Alfred Sloan of General Motors the opportunity to gain domination of the market by focusing, not on production, but on how to sell. The marketing era emerged. It

gave rise to a new set of challenging questions, the most important of which were: (1) How respond to increasing demand for variety and diversity? and (2) How organize and manage the increases in size and complexity that resulted from increasing variety and diversity?

Organismic Models Applied to Social Systems

There were a number of developments between the two World Wars that made deterministic modeling of social systems less and less effective. (1) Technological developments required workers with increasingly greater skill. This in turn required more educated workers and subjecting them to more on-the-job training which became an investment. This made workers more difficult to replace. (2) Union and government interventions in management increased, particularly as related to the health and safety of the work force. (3) The threat of financial destitution associated with unemployment decreased as social security emerged. (4) Business and government organizations took growth to be necessary in order to respond effectively to demand for an increasing amount and diversity of products and services. The diversity of outputs and the new technologies required more growth than could *financed by internally generated profit*. Therefore, many businesses went public in order to raise the capital required to fuel continuous growth as well as technological improvements; management and ownership were separated. Finally, (4) as the size and complexity of organizations increased, decentralization of control became necessary, but this was incompatible with a mechanistic conception of organization. Deterministic systems require centralized control. No driver in his right mind would drive a car with front wheels that he/she could not control.

In an organization that requires invariant functioning of its parts (as deterministically conceptualized bureaucracies do), decentralization leads to disorganization, if not chaos. This is so because, as we have argued, improving the performance of an essential part of an organization taken separately, often reduces the effectiveness of other parts and the whole. This is why many organizations continually oscillate between centralization (to retain control of their parts) and decentralization (to cope with increasing size and complexity).

Because of these changes, enterprises went public; the company became a *corporation* (derived from 'corpus' meaning 'body') and the law came to treat it as an animate individual. The chief executive became the "head" of the firm. no longer its god. (Mechanisms do not have “heads.”) Managers became the clergy that facilitated communication between employees and the organization’s gods, its owners, the stockholders. Like the clergy, managers claimed to know the will of the corporate gods by revelation.

Because of these developments organismic models were increasingly used when dealing with enterprises in particular, and social systems in general. For example, at General Motors Sloan's concept of an organization was essentially that of a single minded animate system. This provided a relatively effective way of managing organizational growth and increasing the diversity of organizational outputs. In his (implicit) model, corporations, like human bodies, were divided into two distinct parts: (1) management, the *brain* (Beer, 1972), and (2) the operating unit, the *body*.

The operating unit, the body, was considered to have no choice, no consciousness. It was restricted to reacting deterministically to instructions coming from management, the brain, and/or events in its environment. Ideally, an operating unit would be a robot programmed to carry out without deviation pursuit of objectives by means both of which were specified by headquarters.

Military organizations, governmental bureaucracies, and autocratic corporations closely approximated the behavior of robots.

The health and safety of the organization and parts of its body, like that of an animate system, became the focus of government and union interventions. The work environment became their focus. The sweat shop and abusive labor practices were outlawed.

As business organizations came to be conceptualized as animate systems rather than mechanisms, production of profit was no longer taken to be its main function. As Peter Drucker pointed out, organizational profit came to be as oxygen is seen by a person, necessary for survival but not the reason for it. Survival became the principal objective of corporations and growth was considered to be essential for it. The opposite of growth, contraction, terminates with death.

Social-Systemic Models Applied to Social Systems

As a result of World War II a large portion of the workforce in western nations was drawn into the military. Replacements included Rosie the Riveter and Tillie the Toiler who were motivated more by patriotism than by the desire for additional income. In many cases, as dependents on those drafted into the military, the government provided them with enough income to survive. Such workers could not be treated as replaceable machine parts or organs of a body that only required consideration of the effects of their work on their health and safety. They had to be treated as human beings with purposes of their own. In addition, unions discovered that corporations working under cost-plus contracts or demands for increased production could easily be induced to make concessions to workers' interests. Work rules went through a profound transformation designed to maximize the number of employees required to do a job. Furthermore, because of the great increase in technological developments, the skills required of the workforce continued to increase dramatically. The more skilled the workers, the harder they were to replace—the more training and education had been invested in them. Technological developments required a significant investment in specialized training of the work force. Management had to obtain an adequate return on this investment. In order to obtain increased productivity from the work force all this generated a need to treat employees as human beings with purposes of their own.

The competence contained in the work force became the most important corporate resource. Exercise of this competence could not be ordered; it had to be induced, and this could only be done by taking the interests of those in the workforce into account.

Children of the post World War II workforce made up the permissive (Spock) generation whose members were not about to be treated as other than purposeful entities in their homes or at work. They expected their interests to be taken into account by their parents and employers and where they weren't, they were alienated from home and work. Their productivity decreased at work (*Work in America*, 1973). The quality-of-work-life movement was an effort to correct this. To summarize: organizations were forced to take the purposes of their employees into account.

In addition, protest groups formed outside organizations insisting that their interests be better served by the organization that affected them, for example, consumerists and environmentalists. The social responsibility of corporations and managerial ethics emerged as major concerns.

By the end of the 1960s it was apparent that the West was experiencing both an accelerating rate of change largely due to technological developments, and increasing complexity produced by growth and an explosion of interconnections resulting from continuously improving transportation and communications, for example, globalization. The socio-economic environment became turbulent, one in which predictability of the future diminished significantly and the only equilibrium that could be obtained was dynamic, like that or an airplane flying through a storm. These changes undermined whatever effectiveness had been obtained by applying organismic models to social systems. Centralized control and the treatment of subordinates as mindless parts became less and less effective.

Increasingly, employees could do their jobs better than their bosses, but only if they were given the freedom to do so. Therefore, the mechanistic and organismic concept of management as command and control, or even more lenient supervision, became less and less appropriate. The functions of management became (1) enabling and motivating subordinates to do as well as they knew how, (2) to develop them so they can do better in the future than the best they can do now, (3) to manage their interactions, not actions, and the interactions of the unit managed with other internal and external organizations. This can only be done with a social systemic model in mind.

Furthermore, in the second half of the twentieth century the unprecedented generation and distribution of both wealth and knowledge resulted in increasing choice and more interdependencies. This changed the nature of social settings and individual behavior. The greater the number of interactions and interdependencies, the more vulnerable social systems became to the actions of a few. The more knowledge available, the greater the value of communication and information. However, advances in information technology and communication did not yield the quality or quantity of control managers hoped for. It was assumed employees would behave like organs in a human body by reacting mechanistically to information provided by the brain, the head of the firm. Therefore, it appeared reasonable to conclude that the malfunctioning of organizations was due either to the lack of information or noise in the communication channels. Increasing effort was devoted to providing more information and better communication.

Unfortunately, this mode of thinking is ineffective in dealing with the complexities of increasing social interactions and interdependencies. It fails to recognize that members of an organization, unlike the parts of an organism, have a choice and do not react passively to the information they receive. Imagine a thermostat that developed a mind of its own. It would not react to information about the temperature in the room that it did not like. This would result in a chaotic air conditioning system. The effectiveness of a servo-mechanism is based on the fact that it does not have a choice and can only react in a predetermined way to the events in its environment. Our organs--heart, lungs, and so on--cannot decide not to work for us. Even when they are defective, we do not conclude that they "are out to get us."

Furthermore, increases in information eventually produces a condition Meier (1963) called "information overload." As the amount of information received increases beyond the amount its receivers can handle effectively, they use less and less of it. Not only do receivers become saturated with information—and therefore cannot receive any more—but they can and do become supersaturated and discard some of the (even useful) information they already have.
An organization with purposeful parts almost inevitably generates internal conflict. Wherever there is choice, conflict is likely. In conflict situations, organismic thinking is ineffective because it tries to resolve conflict by increasing the flow of information between the conflicting

parties. Unfortunately, when conflict is based on differing values or scarcity of resources, an increased flow of information, contrary to conventional wisdom, does not improve but aggravates the conflict. For example, the more information enemies at war have about each other, the more harm they can inflict on each other

The biological mode of organization can be successful, in the short term, in the particular context of paternalistic cultures; where loyalty, conformity, and commitment are considered to be the core virtues. These virtues are reinforced by the security of belonging to a group that in turn protects and provides for its members. For example, Japan, with a relatively strong paternalistic culture, closely approximates an organismic system. Therefore, it has been able to effectively grow economically on the strength of the biological model of organization. In a strong paternalistic culture, conflict can be resolved by the intervention of a strong father figure, whose are respected without hard feelings. To appreciate the power of this type of a leader, recall that such American corporate giants as Ford, DuPont, General Motors, and IBM owe much to their paternalistic founding fathers.

The nature of highly developed social systems is fundamentally different from that of a paternalistic culture. Members of societies that have matured past the secure and unifying umbrella of a paternalistic culture, insist on the right to make choices. But there is a price to be paid for this right; it can induce insecurity and conflict. Purposeful actors, individually or in-groups, who pursue incompatible ends and/or employ conflicting means, generate conflict. Consequently, because of its organismic orientation, corporate America is ill equipped to deal effectively with internal and external conflict. Furthermore, it finds it almost impossible to make the changes required to flourish in its rapidly changing and increasingly complex environment. A significant part of its energy is wasted on futile efforts to deal with such conflict. The frustration that results reinforces its inability to change. This in turn creates a feeling of impotence and hopelessness that immobilizes western governments, institutions, and organizations.

A conflict-free organization can be created by reducing choice, reducing members to robots. Fascist societies and autocratic organizations have attempted to approximate such a state. Such systems are dehumanizing and over time, result in reduced productivity of the workforce and reduced quality and variety of its outputs. This in turn produces a precipitous decline of an economy as is occurring in many western nations. On the other hand, relying exclusively—as organismic modelers of organizations do—on an increasing flow of information and compromise to reduce conflict does not produce encouraging results. Witness the situation in the UN, which has dramatically increased the flow of information between nations and compromise among them.

Therefore, the challenge before us is to create a type of organization that is capable of continuously *dis*solving conflict while *increasing* choice. This requires an organizational concept that is not compatible with either a deterministic or animate model of organization. It requires application of social-systemic models to social systems.

THE SOCIAL SYSTEMIC ORGANIZATION

In other places we (Gharajedaghi, 1985, 1986, and 1999; and Ackoff, 1981 and 1994) have proposed an organizational design that is based on a social-systemic model. It has the following features, none of which are compatible with other than a social systemic model.

1. It is a *democratic organization*, one in which every individual who is affected by what that organization does has a voice in deciding what it does, and in which anyone who has authority over others taken individually, is subject to their collective authority.

2. It has an *internal market economy,* one in which every part of the organization can purchase the goods and/or services it requires from any internal or external source it chooses, and can sell its output to any buyer it wants. Both these types of decision, buying and selling, are subject to overrides by higher authorities who must, nevertheless, compensate the organizational units affected for their loss of income or increased costs due to the higher-level intervention.

3. It has a *multidimensional organizational structure*, one in which units of three different types are located at each level of the organization: units defined by (a) their function (i.e., units whose output is primarily consumed internally), (b) their output (product or services primarily consumed externally), and (c) their users (markets, defined by type or location of customers). This type of organization eliminates the need for continual restructuring. Restructuring is replaced by reallocation of resources.

4. It uses Interactive *planning* which involves (a) formulation of the organization's mess (seeds of self destruction), (b) formulation of an idealized redesign of the organization, one that precludes such self-destruction, (c) determination of the closest approximation to that design that can be realized and (d) planning for it. Such planning then involves (i) selection of the means by which the approximation is to be pursued; (ii) provision of the resources required by the pursuit; (iii) specifying the implementation steps to be taken, when and by whom; and finally, (iv) design of monitoring and control of both the implementation and the effects of the plan.

5. It contains a *decision support system,* one that facilitates learning and adaptation by (a) recording the expectations associated with each decision of significance, (b) the assumptions and information on which they are based, and (c) the process by which the decision was reached, and by whom. It then monitors the implementation, assumptions, and effects of every decision, corrects them where assumptions turn out to be wrong or expectations are not met, and retains in an easily accessible memory what has been learned. Finally, it carries out continuous surveillance of the environment to detect changes that have occurred or are about to occur, changes that require adaptation by the organization.

These requirements are developed in detail in Ackoff (1999).

Any one or subset of these changes can significantly improve organizational performance. However, when all are made together, there is a powerful multiplicative effect, one that is much greater than the sum of its parts.

CONCLUSION

We have argued that it is useful to cast systems and their models into one of four types: deterministic, ecological, animate, and social systemic. The difference between them is a matter of "choice." Deterministic systems and their parts display no choice. The parts of ecological systems can display choice but not the whole. Animate systems can display choice but their parts can't. Social systemic systems display choice and their parts do as well. In addition, they are

usually parts of larger systems that also display choice and contain other systems that do so as well.

Our point has been that when models of one type are applied to systems of a different type, at least as much harm is done as good. The amount of harm (hence good) that is done depends on the level of maturity that social systems have reached. Society and the principal private and public organizations and institutions that it contains have reached a level of maturity that eliminates whatever effectiveness applying deterministic and animalistic models to social systems may once have had.

Finally, we showed five characteristics that we believe social systems designed as social systems should have in order to function as effectively as possible.

REFERENCES

Ackoff, Russell L., *Creating the Corporate Future*. New York, John Wiley & Sons, 1981.

_____, *The Democratic Corporation.* New York, Oxford University Press, 1994.

Beer, Stafford, *The Brain of the Firm*. London: Allen Lane, The Penguin Press, 1972.

Flower, E. F., "Two Applications of Logic to Biology," in *Philosophical Essays in Honor of Edgar Arthur Singer, Jr.*, ed. by F. P. Clarke and M. Nahm. Philadelphia: University of Pennsylvania Press, 1942.

Forrester, Jay W., *Industrial Dynamics*, Cambridge: Wright-Allen Press, 1961.

_____, *World Dynamics*, Cambridge: Wright-Allen Press, 1971.

Gharajedaghi, Jamshid, *Toward a Systems Theory of Organization*. Seaside, Calif.: Intersystems Publications, 1985

_____, *A Prologue to National Development Planning*. New York: Greenwood Press, 1986

_____, *Systems thinking, Managing Chaos and Complexity,* Boston, Butterworth-Heinemann, 1999

Hussong, A. M., *An Analysis of the Group Mind.* Philadelphia: University of Pennsylvania, a Doctoral Dissertation, 1931.

Meier, Richard L., "Communication Overload: Proposals from the Study of a University Library." *Administrative Science Quarterly*, 7 (1963), 521-544.

Quinn, Daniel, *Ishmael*, New York: Bantam/Turner, 1995.

Richman, Barry, *Introduction to Systems Thinking*, *Stella Software*, High Performance Systems: Hanover, NH, 2003.

Sorokin, P., *Contemporary Sociological Theories*, New York: Harper and Bros., 1928.

Taylor, Frederick, *The Principles of Scientific Management*, New York: Harper and Brothers, 1911

Work in America: Report of a Special Task Force to the Secretary of Health, Education, and Welfare. Cambridge, Mass: The MIT Press, 1973.

Zeleny, Milan (ed.), *Autopoiesis: A theory of Living Organization.* New York: North Holland, 1981.

NOTES

1. This article is a revision and extension of an article we published earlier: "Reflections on Systems and Their Models," *Systems Research*, Vol. 13, No. 1, March 1996, pp. 13-23.

2. The classification can be elaborated by distinguishing between choice of ends and choice of means, but doing so would take us beyond the scope of this paper.

3 The measure of complexity of a system is the number of variables and the number and the nature of their interactions that are required to explain the properties and behavior of that system.

Russell L. Ackoff is the Anheuser Busch Professor Emeritus of management science at the Wharton School of the University of Pennsylvania. RLAckoff@aol.com

Jamshid Gharajedaghi is the Managing Partner & CEO of INTERACT: The Center for Interactive System Design, Bryn Mawr, PA. Jghara@Earthlink.net

Lecture 3. System Specification

Papers to study for the exam

- T.J. Schriber, D.T. Brunner, & J.S. Smith. Inside Discrete-Event Simulation Software: how it works and why it matters. In: Proceedings WSC 2015, L. Ylmaz at al. (Eds.). IEEE, 2015, pp 1-15.
  On line available at http://www.informs-sim.org/wsc15papers/001.pdf

Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

INSIDE DISCRETE-EVENT SIMULATION SOFTWARE: HOW IT WORKS AND WHY IT MATTERS

Thomas J. Schriber

The University of Michigan
Stephen M. Ross School of Business
Ann Arbor, MI 48109-1234, USA

Daniel T. Brunner

Commonwealth Supply Chain Advisors
20 Park Plaza, Suite 400
Boston, MA 02116-4300 USA

Jeffrey S. Smith

Auburn University
Industrial & Systems Engineering
Auburn, AL 36849-5346, USA

ABSTRACT

This paper provides simulation practitioners and consumers with a grounding in how discrete-event simulation software works. Topics include discrete-event systems; entities, resources, control elements and operations; simulation runs; entity states; entity lists; and their management. The implementations of these generic ideas in AutoMod, SLX, ExtendSim, and Simio are described. The paper concludes with several examples of "why it matters" for modelers to know how their simulation software works, including discussion of AutoMod, SLX, ExtendSim, Simio, Arena, ProModel, and GPSS/H.

1 INTRODUCTION

A "black box" approach is often taken in teaching discrete-event simulation software. The external characteristics of the software are studied, but the foundation on which the software is based is touched on only briefly or totally ignored (for lack of time). Choices made in implementing the foundation and the impact of these choices on step-by-step model execution might not be studied at all. The modeler might then not be able to think things through when faced with developing good approaches for modeling complex situations, using interactive tools to come to an understanding of error conditions arising during model development, and using interactive tools to verify that complex system logic has been modeled correctly. The objective here, then, is to describe the logical underpinnings of discrete-event simulation and illustrate this material in terms of various implementations of discrete-event simulation software.

This paper is a revised version of a similar paper that initially appeared in the 1996 Winter Simulation Conference Proceedings (Schriber and Brunner 1996). The 1996 paper covered the entity-list management rules and "why it matters" for SIMAN (Arena), ProModel, and GPSS/H. A substantially expanded version of the 1996 paper containing figures, flow charts, and additional explanation can be found in Schriber and Brunner (1998). In Sections 2, 3 and 4 we comment on the nature of discrete-event simulation; basic simulation constructs such as entities, resources, control elements, and operations; and model execution. Section 5 discusses four specific implementations of entity management rules. Finally, Section 6 explores various aspects of "why it matters." Throughout this paper, terms we define are **boldfaced** on first use. Tool-specific terms are Capitalized or, where appropriate, are spelled out in ALL CAPS.

978-1-4673-9743-8/15/$31.00 ©2015 IEEE

2 DISCRETE EVENT SIMULATION IN THE TRANSACTION FLOW WORLD VIEW

The "transaction-flow world view" often provides the basis for discrete-event simulation. In this world view, a system is visualized as consisting of discrete units of traffic that move (*flow*) from point to point in the system while competing with each other for the use of scarce (capacity-constrained) resources. The units of traffic are sometimes called "transactions," giving rise to the phrase "transaction flow." Numerous systems fit the preceding description. Included are manufacturing, material handling, service, health care, communication, and information processing systems, as well as other general queuing systems.

Fundamentally, a discrete-event simulation is one in which the state of a model changes only at a discrete, but possibly random, set of simulated time points, called event times. Two or more traffic units often have to be manipulated at one and the same time point. Such "simultaneous" movement of traffic is achieved by manipulating units of traffic *serially* at that time point. This leads to logical complexities because it raises questions about the *specific real-time order* in which two or more units of traffic are to be processed at a given *simulated* time.

The challenges faced by a *modeler* escalate for the *designer* of a modeling language. The designer must take the logical requirements of discrete-event simulation into account in a generalized way. Choices and tradeoffs exist. As a result, although discrete-event simulation languages are similar in broad terms, they can differ in subtle but important particulars. We seek to expand on this notion and provide specific examples in this paper. Before we do this, however, the following sections provide additional general simulation concepts.

3 ENTITIES, RESOURCES, CONTROL ELEMENTS, AND OPERATIONS

The term **entity** is used here to designate a unit of traffic (a "transaction") within a model. Entities instigate and respond to **events**. An event is an instantaneous happening that changes the state of a model (or system). In a model of an order-filling system, for example, the arrival of an order (an event), might be simulated by bringing an entity into the model. There are two possible types of entities, here referred to as **external entities** and **internal entities**. External entities are those whose creation and movement is explicitly arranged for by the modeler (example: arrival of an order in at an order-processing point). In contrast, internal entities are created and manipulated implicitly by the simulation software. For example, internal entities might be used in some languages to trigger simulated machine *failures* and implement machine *schedules*, whereas external entities might be used to simulate the *use* of machines for processing parts.

The term **resource** designates a system element that provides service (such as a drill, an automated guided vehicle, a worker, or space in an input buffer). Entities typically use resources (for example, a work-in-process entity claims space in an input buffer, then captures an automated guided vehicle to move the entity to the input buffer). Resources are usually capacity-limited, so entities compete for their use and sometimes must wait to use them, experiencing delay as a result (queueing). The term **control element** designates a construct that supports other types of delay or logical alternatives based on a system's state. Control elements can take the form of switches, counters, user data values, and system data values built into the modeling tool. Complex conditions might be evaluated with arithmetic and/or Boolean expressions that examine the state of relevant control elements.

An **operation** is a step (or a series of steps) carried out by or on an entity while it moves through a system. The operations applicable to a ship at a harbor might include: arrive at the harbor; request a berth; capture a berth; request a tugboat; capture a tugboat; get pulled into the berth; free the tugboat; load cargo; request a tugboat; get pulled out of the berth; free the berth; get pulled into open water; free the tugboat; depart.

4 OVERVIEW OF MODEL EXECUTION

A simulation project entails running **experiments**. Experiments are differentiated by the use of alternatives in a model's logic and/or input data. An alternative part-sequencing rule might be tried, for example, in the model of a production system, and/or the quantity of various types of machines (resources) might be varied. Similarly, the number of loading and unloading berths in a harbor might be varied to assess the impact on system performance. Each experiment consists of one or more **replications** (trials). A replication is a simulation that uses the experiment's model logic and data but its own unique set of random numbers, and so produces unique statistical results that can be analyzed as part of a set of such replications (all of which are independent). A replication consists of initializing the model, running it until a run-ending condition is met, and reporting results. This "running it" phase is called a **run**.

During a run the simulation **clock** (an internally managed, stored data value) tracks the passage of *simulated time*. The clock advances (automatically) in discrete steps (typically of unequal size) during the run. After all possible actions have been taken at a given simulated time, the clock is advanced to the time of the next earliest event. Then the appropriate actions are carried out at this new simulated time, etc. In essence, the execution of a run therefore takes the form of a two-phase loop: "carry out all possible actions at the current simulated time," and then "advance the clock," with these two phases repeated again and again until a run-ending condition (usually a modeler-specified simulated time, number of processed entities, or some other condition) comes about. These two phases are respectively called the **entity movement phase** (EMP) and the **clock update phase** (CUP) here.

4.1 Entity States

As the simulation executes, entities migrate from state to state while moving through a model. The five states typically used are:

Active The state of the currently moving entity is the **active state**. By 'currently moving entity' we mean the entity that is currently executing the decision logic in the model. Only one entity moves at any instant of wall-clock time. This entity moves until it encounters a delay of one type or other. It then migrates to an alternative state.

Ready During an entity movement phase there may be more than one entity ready to move, and yet entities can only move (be in the active state) one-by-one. The **ready state** is the state of entities waiting to enter the active state during the current entity movement phase.

Time-delayed The time-delayed state is the state of entities waiting for a known future simulated time to be reached so that they can then (re)enter the ready state. A "part" entity is in a **time-delayed state**, for example, while waiting for the future simulated time at which an operation being performed on it will end.

Condition-delayed The condition-delayed state is the state of entities delayed until some specified condition comes about, e.g., a "part" entity might wait in the **condition-delayed state** until its turn comes to use a machine. Condition-delayed entities are transferred automatically from the condition-delayed state to the ready state when the specified conditions permit.

Dormant Sometimes it is desirable or necessary to put entities into a state from which no escape will be triggered automatically by changes in model conditions. We call this state the **dormant state**. Dormant-state entities rely on modeler-supplied logic to transfer them from the dormant state to the ready state. Job-ticket entities might be put into a dormant state, for example, until an operator entity decides which job-ticket to pull next, with consequent transfer of the job-ticket entity to the ready state. In this case, the specific conditions in effect when selecting the job ticket might not be known until the time comes for the operator entity to select a job ticket.

4.2 Entity Management Structures

In our generic model, we manage the entities by designating an Active Entity and a number of *entity lists*. The **active entity** (the **active-state entity**) moves nonstop until encountering an (attempted) step that causes it to migrate to another entity state (transfers it to another list) or removes it from the model. A ready-state entity then becomes the next active-state entity. Eventually there are no more ready-state entities at the current time. The EMP then ends and a CUP begins. The following lists are used to organize the entities.

Current Events List Entities in the ready state are kept in a single list we call the **current events list** (CEL). Entities migrate to the CEL from the future events list, delay lists, and user-managed lists (each described below). In addition, entities cloned from the active-state entity usually start their existence on the CEL. CEL entities are generally ranked in first-in, first-out (FIFO) order. Some software tools provide a built-in entity Priority attribute used to order Entities on the CEL by priority (with ties resolved using FIFO).

Future Events List Entities in the time-delayed state belong to a single list into which they are inserted at the beginning of their time-based delay. This list, called the **future events list** (FEL) here, is usually ranked by increasing entity move time. Move time is the simulated time at which an entity is scheduled to try to move again (the end of the treatment time for a patient in a clinic or the end of the processing time for a part in a manufacturing system, for example). At the time of entity insertion into the FEL, the entity's move time is calculated by adding the value of the simulation clock to the known (sampled) duration of the time-based delay. After an EMP is over, the CUP sets (advances) the clock's value to the move time of the FEL's highest ranked (smallest move time) entity. This entity is then transferred from the FEL to the CEL, migrating from the time-delayed state to the ready state and setting the stage for the next EMP to begin. The preceding statement assumes there are no other entities on the FEL whose move time matches the clock's updated value. In the case of move-time ties, some tools will transfer all the time-tied entities from the FEL to the CEL during a single CUP, whereas other tools take a "only one entity transfer per CUP" approach. Languages that provide internal entities usually use the FEL to support the timing requirements of these entities. (The FEL then includes internal and external entities).

Delay Lists Lists (there can be many) of entities in the condition-delayed state are called **delay lists**. These entities are waiting (e.g., waiting their turn to use a machine) until their delay is resolved so they can be transferred automatically into the ready state on the CEL. Delay lists, which are generally created automatically by the simulation software, are managed using one of two types of waiting. If a delay can be related easily to model events that might resolve the delay, then **related waiting** can be used to manage the delay list. For example, suppose a machine's status changes from busy to idle. In response, the software can automatically remove the next waiting entity from the appropriate delay list and put it in the ready state on the CEL. Related waiting is the prevalent approach used to manage conditional delays. If the delay condition is too complex to be related easily to events that might resolve it, **polled waiting** can be used. With polled waiting the software checks routinely to see if entities can be transferred from one or more delay lists to the ready state. Complex delay conditions for which polled waiting can be useful include Boolean combinations of state changes, e.g., a berth is empty and a tugboat is idle.

User-Managed Lists Lists (there can be many) of entities in the dormant state are called **user-managed lists**. The modeler must take steps to establish such lists and usually must provide the logic needed to transfer entities to and from the lists. Except for very simple one-line, one-server service points in a system, the underlying software has no way to know why entities have been

put into user-managed lists in the first place, and therefore has no plausible basis for automatically removing entities from such lists.

5 IMPLEMENTATION IN FOUR TOOLS

The tools chosen here for commentary on implementation particulars are AutoMod (Phillips 1997); SLX (Henriksen 2000; Schulze 2008); ExtendSim (Imagine That Incorporated 2015; Krahl 2012); and Simio (Kelton *et al.*, 2014). A previous version of this paper (Schriber and Brunner 1996) covered SIMAN (Arena) (Kelton *et al*., 2014), ProModel (ProModel Corporation 2015), and GPSS/H (Henriksen and Crain 2000) in similar detail. We think these tools are representative, but they clearly are not exhaustive (see Swain 2013 for a recent biennial survey of the many available tools).

5.1 AutoMod

The current events list is named the Current Event List (CEL) in AutoMod (see Table 1). Cloned Loads, Loads leaving the FEL due to a clock update, and Loads ordered off Order Lists are placed immediately on the CEL. The insertion rule is to rank first by priority (priority is a built-in attribute of every Load) and then FIFO within priority. When the CEL becomes empty, the Condition Delay List (see below) is checked, and Loads may be transferred from there to the CEL. This continues until the CEL is empty and no more Loads can be transferred, at which point the EMP is over and a CUP is initiated.

Table 1: AutoMod Terminology.

Generic Term	**AutoMod Equivalent**
External Entity	Load
Internal Entity	Logical Load
Resource	Resource; Queue; Block
Control Element	Counter; Process Traffic Limit
Operation	Action
Current Events List	Current Event List
Future Events List	Future Event List
Delay List	Delay List; Condition Delay List; Load Ready List
User-Managed List	Order List

The AutoMod Future Event List (FEL) is like future event lists in other tools. Loads arrive on the FEL in the time-delayed state by executing a WAIT FOR statement. AutoMod allows the specification of time units (day, hr, min, sec) in a WAIT FOR statement. The AutoMod CUP removes multiple Loads from the FEL if they are tied for the earliest move time, inserting them one by one into their appropriate place on the CEL. There are also internal entities in AutoMod, called Logical Loads, that do things such as wait on the FEL to trigger scheduled shift breaks.

Delay Lists (DL's) are lists of Loads waiting to claim capacity provided by any one of five finite-capacity elements (Resource, Queue, Block, Counter, or Process Traffic Limit; see Table 1). Each finite capacity element within the model has a DL associated with it. The waiting that results in these five cases is related waiting. Whenever capacity is freed, one Load from the head of the element's DL is tentatively placed on the CEL (but a placeholder is left on the DL). When that Load is encountered during the EMP, it tries to claim the requested capacity. If it fails (for example because it wants two units but only one is free), it is returned to the DL in its original place (this is default behavior that the modeler can override. See Section 6.2 for more general discussion). Immediately after this evaluation, if there is still unused capacity, the *next* Load (if any) on the relevant DL is placed on the CEL. Processing of the active Load

then continues, followed by evaluation of this next tentatively placed Load. And so on, tentatively, for each next Load (if any) during the EMP.

For conditional waiting apart from the five cases described above, AutoMod has a WAIT UNTIL statement that results in polled waiting. WAIT UNTIL conditions can be compounded using Boolean operators. If a Load executes a WAIT UNTIL and the condition is false, the Load is placed on a single global AutoMod list called the Condition Delay List (CDL). After the CEL has been emptied, but before the simulation clock is updated, *all* Loads on the CDL are moved to the CEL (actually, the CDL "becomes" the CEL) *if* there has been a state change for at least one element of the same general type (e.g. Queue) for which *any* Load on the CDL is waiting (this mechanism is primarily "polled," where the polling process is triggered by a change in state of at least one element of the same general type).

If the CEL is now non-empty, the EMP resumes. If the condition for which a CEL Load is waiting is not yet satisfied, AutoMod moves that Load from the CEL back to the CDL. The CDL in some cases may be emptied multiple times during one EMP until eventually the CEL has been emptied without having triggered a state change related to any Load on the CDL. A CUP then occurs. Because of the potential for repetitive list migration with WAIT UNTIL, AutoMod's vendor encourages the use of Order Lists or other explicit control mechanisms to manage complex waiting.

AutoMod implements the dormant state with Order Lists, which are user-managed lists of Loads. After a Load puts itself onto an Order List (by executing a WAIT TO BE ORDERED Action), it can only be removed by another Load (or another active model element such as a Vehicle) that executes an ORDER Action. An ORDER Action may specify a quantity of Loads, or a condition that must be satisfied for a given Load if that Load is to be ordered, or both. Loads successfully ordered are placed immediately on the CEL (one at a time according to how they were chosen from the Order List, and ranked on the CEL by priority, with priority ties resolved FIFO). Order Lists can achieve performance improvements over CDL waiting because Order Lists are never scanned except on explicit request. AutoMod Order Lists offer several interesting wrinkles, including: the ability for an ordering Load to place a back order if the ORDER quantity is not satisfied; the ability for a Load on an Order List to be ordered to *continue* to the next Action instead of to a Process (this feature is useful for control handshaking); and the ability to have a function called for each Load on the Order List (by using the ORDER…SATISFYING Action).

AutoMod has several material handling constructs that are integrated with Load movement. For vehicle systems there are three other types of lists (not included in Table 1). Loads on Load Ready Lists (LRL) (one list per vehicle system) are waiting to be picked up by a vehicle. Loads claimed (but not yet picked up) by a vehicle reside on the vehicle's Vehicle Claim List (VCL). Claimed loads that have been picked up reside on the vehicle's Vehicle Onboard List (VOL). The vehicle then becomes the active "load" and moves among AutoMod's lists (FEL, CEL, and possibly DL's) rather than the Loads themselves.

5.2 SLX

SLX is a hierarchical language in which the built-in primitives are at a lower level than most other simulation languages, facilitating user (or developer) definition of the behavior of many system elements. This design philosophy allows the SLX user (or developer) to create higher-level modeling tools whose constructs have precisely defined modifiable behavior. Equivalents for the generic terms for users of low-level SLX are given in Table 2. For example, SLX uses Control Variables to act as Control Elements. The "control" modifier can be attached to a Variable of any data type (integer, real, string, etc.). A Control Variable can be global, or it can be a local Variable declared in an Object's Class definition. (A Class-declared Variable is an attribute in other tools.)

SLX has two types of Objects: Active and Passive. The two are distinguished by the presence of actions – executable Statements – in an Active Object's Class definition (even without actions, Passive Objects are useful in their own right, functioning as user-defined complex data structures). Table 3 shows how higher-level SLX-based tools might exploit the definitional capabilities of SLX.

Table 2: SLX Terminology (Low-level).

Generic Term	**SLX Equivalent**
External Entity	Active Object and its Puck(s)
Internal Entity	none
Resource	Control Variable
Control Element	Control Variable
Operation	Statement
Current Events List	Current Events Chain
Future Events List	Future Events Chain
Delay List	Delay List
User-Managed List	Set (see below)

Table 3: Tools Based on SLX.

Generic Term	**SLX Equivalent**
Resource	Active or Passive Object
Control Element	Active or Passive Object
Operation	User-defined Statement
Delay List	User-defined (based on Set)
User-Managed List	User-defined (based on Set)

The current events list is named the Current Events Chain (CEC) in SLX. The members of the CEC are given the interesting name Puck. What is a Puck? SLX dissociates the concept of an Active Object (with its associated local data) from a Puck, which is the "moving entity" that executes the actions, carries its own entity scheduling data, and migrates from list to list. The effect of this dissociation is that a single Object can "own" more than one Puck. All Pucks owned by a single Object share the Object's local data (attributes). For example, one application of this "local parallelism" feature (as compared with the "global parallelism" offered by "clone" or "split" actions in other languages) is the use of a second Puck to simulate a balk time while the original Puck is waiting for some condition. (If the condition comes about before the balk time has elapsed, no balking occurs; otherwise, balking does occur.)

Activating a new Object creates one Puck and launches that Puck into action. In many cases no additional Pucks are ever created for that Object, and the combination of an Active Object and its Puck forms the equivalent of an entity (passive Objects have no actions and therefore own no Pucks). Newly activated Pucks, Pucks leaving the FEC due to a clock update, and reactivated Pucks (see below) are placed on the CEC, ranked FIFO by priority. The CEC is empty when an EMP ends.

The SLX Future Events Chain (FEC) is like future events lists in other tools. Pucks arrive on the FEC in the time-delayed state by executing an ADVANCE statement. The SLX CUP will remove multiple Pucks from the FEC if they are tied for the earliest move time, inserting them one by one into their appropriate place on the CEC. Because the SLX kernel functionality does not include downtimes or even repetitive Puck generation (scheduled arrivals), all activity on the SLX FEC unfolds as specified by the developer of the SLX model. More generally, if a user is using a model (or is using a model builder) that contains higher-level primitives defined by a developer, chances are that all kinds of things are going on behind the scenes, hidden from the higher-level user's view.

Delay Lists (DL's) in SLX are lists of Pucks waiting (via WAIT UNTIL) for state changes in any combination of Control Variables and the simulation clock value. A Puck waiting for a compound condi-

tion involving two or more Control Variables is listed on more than one DL. All higher-level constructs defined by developers can use this mechanism. Each Control Variable (which may be a local Variable, in which case there is one for each Object in the Class) has a separate DL associated with it.

A DL is ranked by order of insertion. All pucks on a DL are removed whenever the associated Control Variable changes value and are inserted one at a time into the CEC. Removed Pucks that are waiting for compound conditions are also tentatively removed from each of the other Delay Lists to which they belong. As these Pucks are encountered on the CEC during the EMP, those failing to pass their WAIT UNTIL are returned to the Delay List(s) for those Control Variables still contributing to the falseness of the condition. For conditions that include a clock reference, the Puck is inserted if necessary into the FEC, subject to early removal from the FEC if the condition becomes true due to other Control Variable changes. This low-level related waiting mechanism based on Control Variables is the default SLX approach to modeling all types of simple or compound condition-delayed states.

SLX handles the dormant state in a unique way. Instead of moving the Puck from the active state to a user-managed list and suspending it, all in the same operation, SLX breaks this operation into two pieces. First, the Puck usually joins a Set. But joining a Set does not automatically suspend the Puck. A Puck can belong to any number of Sets. Set membership merely provides any other Puck with access to the member Pucks. To go into the dormant state, a Puck executes a WAIT statement. It then is suspended indefinitely, outside of any particular list, until another Puck identifies the waiting Puck and executes a REACTIVATE statement for it. Often this other Puck is scanning a Set to find the Puck to REACTIVATE, but a Set is not exactly the same as a user-managed list in our terminology. A dormant-state Puck might be a member of no Sets (as long as a pointer to it has been stored somewhere) or of one or more Sets. An SLX developer can easily define a user-managed list construct, using Sets, WAIT, and REACTIVATE as building blocks, that mimics those of other languages or offers its own unique features.

5.3 ExtendSim

ExtendSim (originally named Extend) uses a message-based architecture for discrete-event simulation. Various types of messages are used to schedule events, propel Items (Entities) through a model, enforce the logic incorporated into a model, and force computation. The senders and receivers of messages are Blocks (Operations), including the Executive Block (master controller). In ExtendSim, it is *Block* execution that is scheduled (when a Block executes, for example, this can trigger the sending of messages back and forth among Blocks, with the effect of moving an Item along its Block-based path in a model). Table 4 summarizes ExtendSim equivalents for the terms introduced in the earlier generic discussion.

Table 4: ExtendSim Terminology.

Generic Term	**ExtendSim Equivalent**
External Entity	Item
Internal Entity	Blocks/Messages
Resource	Block; Resource; Item; Resource Pool; Advanced Resources
Control Element	Block
Operation	Block
Current Events List	Next Times Array and Current Events Array
Future Events List	Time Array; Linked-lists within Blocks
Delay List	List of Items Resident in a Pre-Programmed Block
User-Managed List	List of Items Resident in a User-Programmed Block; Database Tables; Dormant Items in a Queue Equation Block

A Block is the basic modeling construct in ExtendSim. Each Block has an icon, message-passing connectors, dialog capability, and behavior-defining code. Residence Blocks can hold Items while simulated time goes by, whereas Passing Blocks cannot (items go through Passing Blocks in zero simulated time). Models can be constructed by selecting pre-programmed Blocks from ExtendSim's Block libraries. The modeler can also modify the source code given for library Blocks (all Blocks in the base version of ExtendSim are open source). Finally, the modeler can create customized Blocks from scratch (user-programmed Blocks) using development tools that ExtendSim provides.

ExtendSim uses a Time Array to schedule future Block executions. For a given model, the Time Array contains one or more elements for each Block. A Time Array element records the future time for which execution of that Block has been scheduled, The potential for a Block to have more than one Time Array element is an enhancement in Version 7 of the language (Version 8 is the current version). This feature can be useful when a Block has multiple, dissimilar events, as for example in conveyor modeling.

Blocks not currently scheduled for future execution are temporarily "blacked out" by recording arbitrarily large time values for them in the Time Array. Residence Blocks that can hold multiple Items manage the corresponding event times internally, with only the earliest of the Block's event times kept in the Time Array. This is a two-stage event list, as Blocks can contain optimized linked-lists that serve as their own future event lists. Block execution can result in scheduling future Block executions. For example, if messages are passed that result in an Item entering a unit-capacity Residence Block designed to hold the Item until a sampled amount of simulated time has elapsed, then the Time Array entry for that Block will have its value set accordingly.

The number of Blocks in a given model is constant, which means the Time Array is of fixed and relatively small size. Because of its small size, the Time Array is searched to find the imminent event time; it is not kept in sorted order. This makes it straightforward for a Block to change its event time because no searching of the event list is required.

The Next Times Array is used to manage the execution of Blocks whose execution has been scheduled via the Time Array. The Next Times Array is populated just prior to a Block Execution Phase (ExtendSim's equivalent of an EMP) as follows. At each CUP, the Time Array is searched to find the earliest future time at which a Block execution has been scheduled. Identifiers for the corresponding Block (or Blocks, in case of time ties) is (or are) then put into the Next Times Array. The Block Execution Phase (BEP) then begins, with the Executive messaging the most highly qualified Block in the Next Times array to start its execution.

The Current Events Array is used to manage the *resumption* of execution of Blocks whose execution has been temporarily suspended during the course of a Block Execution Phase. For example, suppose a Block sends a message, and the receiving Block replies (returns control) immediately to the sending Block (even though the receiving Block still has to do additional processing at the simulated time in question). In this case, the receiving Block's identifier is added to the Current Events Array. When the sending Block is finished executing, the Executive sends a message to the most highly qualified Block in the Current Events Array to resume its execution. Eventually, the Current Events Array becomes empty. Then the Executive turns again to the Next Times Array, sending a message to the most highly qualified Block to start executing.

During a Block Execution Phase, Blocks can schedule themselves to be executed at the *current* simulated time (that is, during the ongoing BEP). The Current Events Array comes into play here, too, to manage the execution of Blocks in such cases. For example, if a capacity-constrained Block becomes non-full as a result of some other Block's execution, the non-full Block puts its identifier into the Current Events Array. The Executive will later (but at the same simulated time) send a message to the Block to start executing. The Block will then try to pull into itself Items (if any) that have been waiting to enter the Block (in ExtendSim, Items can be both pulled and pushed through a model). When the Current Events Array and the Next Times Array both become empty, this brings ExtendSim's Block Execution Phase to an end. Then the next CUP and BEP take place, repeating until a simulation-ending condition is satisfied.

Delay lists consist of Items delayed in Residence Blocks, waiting their turn to be pulled or pushed into their next Block(s). Message passing is used to accomplish the pulling and pushing when model conditions permit. ExtendSim provides related-waiting management of these Items based on user-specified first-in, first-out (FIFO), last-in, first out (LIFO), Priority, Attribute, Reneging, Matching, and Equation-based alternatives. Waiting for the resolution of compound conditions is normally achieved in ExtendSim by appropriately combining Blocks and exploiting ExtendSim's message-based architecture. We view this as a form of related waiting, because it is a change in an underlying value that triggers a re-evaluation of the condition that brought about the waiting in the first place.

Because of ExtendSim's messaging architecture, polled waiting is generally not necessary. A message is sent when a value changes and any conditions are evaluated at that moment. Waiting for a clock based event can be achieved by using a Block that schedules events, e.g., Shift; Lookup Table; Equation. These Blocks send a message at scheduled times. Polled waiting is available, however, with use of the Gate Block and selection of its "Check demand at each event" option.

The modeler can work with User-Programmed Blocks to create and manage lists of the modeler's own design. The code for custom Blocks can be written to achieve the modeler's objectives in this regard, just as the code for ExtendSim's pre-programmed Blocks has been written to specify the behavior of those Blocks. ExtendSim provides functions that can be used by Blocks to share lists (arrays) with other Blocks, further supporting customized list management in models.

5.4 Simio

Simio is an object-oriented language in which all model constructs are Simio Objects derived from the same base object. Most Simio users will primarily use objects from the Simio Standard Library for constructing models. Entities, Resources, Servers, Workstations, Sources, Sinks, Nodes, and Connectors are commonly used Objects from the Standard Library. Underneath, Simio is based on Processes that are comprised of individual Process Steps executed by Tokens. Standard Library objects' behaviors are implemented (by the object designer) using processes. While most models will be built using these high-level Objects, users (and developers) have complete access to Simio processes and can develop complete models as well as reusable Objects using Simio processes. In addition, users can augment the behavior of existing Objects using Add-on Processes and/or by Subclassing existing Objects. Simio models themselves are actually Objects that can be embedded in other models. The object structure of Simio facilitates the development of custom Object libraries that can simplify the modeling process in different application domains.

Equivalents for the generic terms for Simio users are given in Table 5. Note that any Simio object can act as a resource by setting the "Resource Object" property to True. The Resource object from the Standard Library is analogous to a resource in the context of this paper. The relationship between Simio entity objects and tokens is similar in concept to the relationship between the active object and pucks in SLX. In particular, as objects interact (e.g., an Entity object moving through a Server object to queue for and access a finite-capacity resource), one or more tokens associated with these objects are executing the processes (sequences of steps) that comprise the objects' behavior. The capability of having multiple tokens executing different processes on behalf of the same object provides modeling flexibility.

Object Properties and States serve as Simio control elements. Properties and States are both attributes of Objects. Properties are set at the start of a run and do not change during the run, whereas States can be set and changed at any time during a model run. Properties are typically used during model construction and experimentation and States are typically used during a model run to control flow and/or track and report statistics. Events are a bit more abstract in Simio than in some other packages. In Simio, events are characterized by an execution time (the simulation time when the event occurs), a reference to a procedure call (to be executed when the event occurs), and a reference to an Object (which provides additional data to the procedure, when necessary). An event's referenced Object may or may not be a user-visible Entity or associated Token. For events associated with Entity movement, the event Object refers to a To-

ken associated with the Entity and the event procedure executes a process step (called a "token arrived to a step" event). Most events that occur during a model run will be of this type. Examples of other, internal events include end-of-run events, States crossing a threshold events, Entity collision events, etc. Simio also supports user-defined events that can be fired, waited for, and used to trigger processes.

Table 5: Simio Terminology.

Generic Term	**Simio Equivalent**
External Entity	Entity/Token
Internal Entity	Token
Resource	Object (Resource, for example)
Control Element	States (Model states, Entity states, Resource states, for example)
Operation	Step
Current Events List	CEL – Current Event List
Future Events List	FEH – Future Event Heap
Delay List	Queue/Allocation Queue
User-Managed List	Storage Element

During model execution, once the CUP occurs and Simio enters the EMP, all events scheduled for the current simulation time are removed from the Future Event Heap (FEH) and are placed on the Current Event List (CEL). The heap data structure is used for future events because of its computational efficiency. Simio executes all of the events on the CEL (sequentially) before advancing the simulation time to the next event time. Events on the CEL are prioritized by event Type (normal, early, and late). Event procedures can create other events (future events as well as events scheduled for the current simulation time), update the system state, trigger the recording of statistics, and terminate the model run.

Allocation Queues are lists of entities waiting for resources. When a resource becomes available, Simio goes through a standard reallocation process whereby the "first" Entity in the Allocation Queue (determined by the queue ranking rule) will be placed at the front of the CEL and the Entity that released the resource will be processed through steps until it reaches a delay (either a time delay, a condition delay, or a user-specified storage). Under this process, the last releasing Entity is processed until it reaches a delay and the last Entity that was allocated resource capacity is the first to begin processing (at the same simulation time). Simio implements the dormant state using Storage or Station elements. A Storage element implements a *logical* queue where Objects can be "placed" by the user for later removal. Storage elements have no physical location and a given Object can simultaneously reside in multiple Storages. A Station element, on the other hand, defines a capacity-constrained physical location where Entity Objects can be stored for later removal and processing.

6 WHY IT MATTERS

In Sections 6.1-6.5 we describe situations that reveal some practical differences in implementation particulars among Arena (which uses the SIMAN language and processor), ProModel, GPSS/H, AutoMod, SLX, ExtendSim, and Simio. None of the alternative approaches mentioned is either intrinsically "right" or "wrong." The modeler simply must be aware of the alternative in effect in the simulation software being used and work with it to produce the desired outcome. Otherwise, it is possible to mis-model a situation and perhaps not become aware of it. In Section 6.6, we comment on how knowledge of software internals is needed to make effective use of model checkout tools. Finally, in Section 6.7, we point out that knowledge of internals aids in understanding performance monitoring.

6.1 Trying to Re-capture a Resource Immediately

Suppose a job in a flexible job shop releases a machine (for which other jobs are waiting), then, as what will be its next step, decides to re-capture that machine. Will the job re-capture the machine immediately, or will a waiting job (even whether less qualified or equally qualified) capture the machine instead?

Of interest here is the order of events following the releasing of a resource. There are at least three alternatives: (1) Coupled with the resource-release event is the *immediate* choosing of the next user of the resource, without the releasing entity having yet carried out its next step; (2) The choosing of the next resource user is *deferred* until the releasing entity has potentially become a contender; and (3) Without paying heed to other contenders, the releasing entity recaptures the idle resource immediately. Arena, ExtendSim, and Simio implement (1). ProModel implements (2). GPSS/H and AutoMod implement (3) by default. In SLX, using a Control Variable as the resource state, the result is also (3). (In some of the tools, modelers can implement higher-level resource constructs that behave according to the modeler's choice.)

6.2 The First in Line is Still Delayed

Suppose two condition-delayed entities are waiting in a delay list because no units of a particular resource are idle. Suppose the first entity needs *two* units of the resource, whereas the second entity only needs *one* unit. Now assume that one unit of the resource becomes idle. The needs of the first list entity cannot yet be satisfied, but the needs of the second entity can. What will happen?

There are at least three possible alternatives: (1) Neither entity claims the idle resource unit; (2) The first entity claims the one idle resource unit and waits for a second unit; and (3) The second entity claims the idle resource unit and migrates to the ready state. As in Section 6.1, each of these alternatives comes into play in the tools considered here. Arena (SEIZE) and ProModel (GET or USE) implement (1) and (2) respectively, by default. AutoMod (GET or USE), GPSS/H (ENTER or TEST) and SLX (WAIT UNTIL on a Control Variable) implement (3) by default. ExtendSim also implements (3) by default. But ExtendSim gives the modeler the choice of locally implementing (1) for resources specified by the modeler. The modeler does this by checking an "Only allocate resource pool to the highest ranked Item" option for each such resource. Simio allows the modeler to select option (1) or (2) by using the Number of Objects property and the "repeating resource seizes" option with the Seize step.

6.3 Yielding Control Temporarily

Suppose the active entity wants to give control to one or more Ready-State entities, but then needs to become the active entity again before the simulation clock has been advanced. This scenario might come into play, for example, if the active entity has opened a switch permitting a set of other entities to move past a point in the model, and then needs to re-close the switch after the forward movement of the other entities has been accomplished. As an example, consider a case where a group of identically flavored cartons of ice cream is to be transferred from an accumulation point to a conveyor leading to a one-flavor-per-box packing operation.

In Arena, the effect can be accomplished *approximately* with a DELAY that puts the active entity into a time-delayed state for an arbitrarily short *but non-zero* simulated time. In ProModel, "WAIT 0" can be used to put the active entity back on the FEC. The entity will be returned later (wall-clock time, but at the same simulated time) by the CUP to the active state. In GPSS/H, the active Xact (Transaction) can execute a YIELD Block to migrate immediately from the active state to the ready state (positioned last in its priority class) and force a restart of the CEC scan. Higher-ranked CEC Xacts are then given a chance to become active before the yielding Xact becomes active again at the same simulated time.

SLX (YIELD), AutoMod (Wait For 0), and Simio (Delay step with Math.Epsilon) all provide solutions in which the active Puck (SLX), Load (AutoMod), or Token (Simio) is shifted to the back of its pri-

ority class on the CEC, where it waits to become the active entity again before the clock has been advanced.

In ExtendSim, "yield and then eventually resume" is part of the architecture. A message is sent out through the appropriate Block connector when an Item moves into or out of a Block. This message propagates to other connected Blocks, perhaps changing system status or moving Items from one Block to another as a result. When the originating Block eventually receives the reply, it continues processing the original Item.

6.4 Conditions Involving the Clock

Every language provides a time-delay capability for FEL waiting. This works well when an entity needs to wait until a known clock value has been reached. But what if an entity needs to wait for a compound condition involving the clock, such as "wait until a specified input buffer is empty *or* it is exactly 5:00 PM?" A typical approach to this is to clone a dummy ("shadow") entity to do the time-based waiting. Management of such dummy entities can be cumbersome, particularly for very complex rules. ProModel does not use polled waiting, so a dummy entity would be the best approach available. (Otherwise, the condition would not be checked until the other component of the compound condition had a value change.) ExtendSim also does not use polled waiting, so a similar situation applies for ExtendSim, and any Block that can schedule events can be used. With Simio, entity objects can have multiple associated tokens, each waiting for a different component of the compound condition. This is similar in concept to using a dummy entity, but does not require an additional entity.

When a polled waiting mechanism is present, if a single entity tries to wait for a compound condition involving the clock, an interesting problem can arise. This is because the next polling time may not match the target clock time. Arena and AutoMod detect the truth of compound conditions via their end-of-EMP polling mechanisms. GPSS/H also detects the truth via its version of polled waiting (refusal-mode TEST). But in the absence of a something that waits on the FEL until exactly 5:00 PM (i.e., the approach recommended above for ProModel, ExtendSim, and Simio), all three of those tools are subject to the possibility that the first EMP that finds the condition true occurs when the clock has a value *greater* than 5:00 PM. This can be problematic if the exactness of 5:00 PM is important.

SLX recognizes the clock as a related wait-until target. A WAIT UNTIL using a future clock value in a way that contributes to the falseness of the condition will cause the Puck to be scheduled onto the FEL to force an EMP at the precise time referenced. This solves the greater-than-the-desired-time problem. Note that this Puck may also be waiting on one or more delay lists.

6.5 Mixed-Mode Waiting

Suppose many entities are waiting to capture a particular resource, while a user-created controller entity is waiting for the compound condition "the shift status is 'off-shift' *and* number waiting is less than six *and* resource is not currently in use" to take some action (such as shutting the resource down, in languages that allow user-defined entities to shut down resources; or displaying a status message). How can we guarantee that the controller entity will be able to "cut in front" of the waiting entities at the appropriate simulated time (before the idle resource is recaptured)?

One way to handle this would be through entity priorities in languages that offer this feature. However, as described below, that might not work even if the controller has relatively high priority. The key issue is the method used to implement the waiting. If it is "related" for the entities waiting to capture the resource and "polled" for the controller entity waiting for the compound condition (this is what we mean by the term "mixed-mode waiting"), things can get complicated. Every time the resource becomes free, a new entity will be selected from a delay list immediately in Arena and via the CEL in AutoMod, in both cases preceding the end-of-EMP checking for polled wait conditions (and thereby ignoring the entity pri-

ority of the controller). There are ways to work around this if desired, such as using a different type of operation to force a polled wait for entities wishing to use the resource.

In GPSS/H, using a high-priority controller Xact at a refusal-mode TEST Block, the controller waits at the front of the CEC. The Facility RELEASE triggers a scan restart and the controller does its job. In ProModel there is no polled waiting but there can be related waiting on compound conditions involving Variables. Variables would have to be defined and manipulated for each element of the Boolean condition and, to assure equal competition, the entities waiting to capture the resource might also have to use WAIT UNTIL instead of GET or USE. Another possibility with ProModel would be to have the entity that frees the resource do some state-checking right away (in effect becoming a surrogate for the controller). This is possible because of the deferred-selection method used by ProModel (see Section 6.2).

In the related waiting of SLX, a Puck awaiting a compound condition will be registered on the delay lists of those (and only those) Control Variables that are contributing to the falseness of the condition. The SLX architecture (in which only global or local Control Variables and the clock can be referenced in any sort of conditional wait at the lowest level) assures that there will already be Variables underlying the state changes being monitored. The modeler defines them as Control Variables.

As with ProModel and SLX, ExtendSim would use related waiting to detect and immediately respond to a change in the compound condition. Because the state change of the Resource is immediately broadcast as a message before the Resource is reallocated, this message can be used to control the sequence and logic of Item selection. A Resource is not allocated unless there is both downstream space for an Item and Resource availability, so blocking the output path from a Queue will prevent Resource allocation.

In Simio, Resource Objects can have their own logic implemented using Add-on Processes, so there is no need to use "controller entities" to implement the control mechanism. In this case, The "Off-Shift" and "Released" add-on processes can be used to check for the complex condition involving the shift status, queue size, and Resource status. Note that this is equivalent to related waiting in that the condition will only be checked when the resource goes off-shift and/or when a unit of capacity is released.

6.6 Interactive Model Verification

We now comment briefly on why a detailed understanding of "how simulation software works" supports interactive probing of simulation-model behavior. In general, simulation models can be run interactively or in batch mode. Interactive runs are of use when checking out (verifying) model logic during model building and when troubleshooting a model if execution errors occur. Batch mode is then used to make production runs.

Interactive runs put a magnifying glass on a simulation while it executes. The modeler can follow the active entity step by step and display the current and future events lists and the delay and user-managed lists as well as other aspects of the model. These activities yield valuable insights into model behavior for the modeler who knows the underlying concepts. Without such knowledge, the modeler might not take full advantage of the software's interactive tools or, worse yet, might not even use the tools.

6.7 Performance Issues

Simulation experiments can consume substantial amounts of computer time. Other things equal (including the model builder's skill), computer-time requirements depend on the design and implementation of the software used to build models. This dependency can be understood with knowledge of "how simulation software works." For example, consider user-managed lists vs. related waiting in models in which large numbers of entities contend for a resource. Performance is an important enough issue to motivate some simulation software (e.g., ExtendSim; SLX) to supply performance profilers which, for example, can produce histograms showing where CPU time is spent during model execution.

ACKNOWLEDGMENTS

Much of the information in this paper was provided by vendor personnel. The authors acknowledge the support of Deb Sadowski; Vivek Bapat; Charles Harrell (ProModel); Kenneth Farnsworth and Tyler Phillips (AutoMod); Robert C. Crain and James O. Henriksen (GPSS/H and SLX); David Krahl (ExtendSim); and David T. Sturrock and C. Dennis Pegden (both originally with Arena/SIMAN, and now with Simio).

REFERENCES

Henriksen, J. O. 2000. "SLX: The X is for Extensibility." *Proceedings of the 2000 Winter Simulation Conference*, edited by J. A. Joines, R. Barton, K. Kang, and P. Fishwick, 183-190, Piscataway, NJ: IEEE.

Henriksen, J. O., and R. C. Crain. 2000. "GPSS/H: A 23-year retrospective view." *Proceedings of the 2000 Winter Simulation Conference*, edited by J. A. Joines, R. Barton, K. Kang, and P. Fishwick, 177-182, Piscataway, NJ: IEEE.

Imagine That Incoporated. 2015. *ExtendSim 9.2 User Guide*. San Jose, California. Imagine That Inc.

Kelton, D., J. S. Smith and D. T. Sturrock. 2014. *Simio & Simulation: Modeling, Analysis, Applications*, 3rd Ed. Sewickley, PA: Simio, LLC.

Kelton, D., R. Sadowski, and N. Zupick. 2014. *Simulation with Arena*, 6th Ed. New York, NY: McGraw Hill.

Krahl, D. 2012. "ExtendSim: A History of Innovation." *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. Uhrmacher.

Phillips, T. 1997. "Know your AutoMod Current Events." In *AutoFlash*, 10(7). Bountiful, Utah: AutoSimulations, Inc.

ProModel Corporation. 2015. *ProModel Version 9.2 User Guide.* Orem, UT: ProModel Corporation.

Schriber, T. J., and D. T. Brunner. 1996. "Inside Simulation Software: How It Works and Why It Matters." *Proceedings of the 1996 Winter Simulation Conference*, edited by J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, 23-30. Piscataway, NJ: IEEE.

Schriber, T. J. and D. T. Brunner. 1998. "How Discrete-Event Simulation Software Works." Chapter 24 in *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*, ed. J. Banks. New York, NY: John Wiley & Sons.

Schulze, T. 2008. *Simulation Needs SLX*. Annandale, Virginia: Wolverine Software.

Swain, J. J. 2013. "Simulation Software Survey." *OR/MS Today* 40(5). (This is a biennial survey.)

AUTHOR BIOGRAPHIES

THOMAS J. SCHRIBER is a Professor of Technology and Operations in the Stephen M. Ross School of Business at the University of Michigan. He is a recipient of the INFORMS Simulation Society's Lifetime Professional Achievement Award, and is a 'Pioneer of Simulation' (http://d.lib.ncsu.edu/computer-simulation). Email: schriber@umich.edu.

DANIEL T. BRUNNER is a Senior Partner at Commonwealth Supply Chain Advisors. He holds a BSEE from Purdue University and an MBA from the University of Michigan. He has served as a WSC Business Chair (1992), General Chair (1996) and Transportation Applications Track Coordinator (1998). He can be reached at: dan.brunner@gmail.com.

JEFFREY S. SMITH is the Joe W. Forehand Professor of Industrial and Systems Engineering at Auburn University. He has served as the WSC Business Chair (2010) and General Chair (2004) and is currently on the WSC Board of Directors. He has a BIE from Auburn University and a MS and PhD (both in Industrial Engineering) from Penn State University. His email and web addresses are: jsmith@auburn.edu and http://jsmith.co.

Lecture 4. DEVS

Papers to study for the exam

- Chapter 1 from Zeigler, B.P., H. Praehofer and T.G. Kim (2000). Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, 2nd Ed.) Academic Press, NY.
  Retrieved from: http://acims.asu.edu/wp-content/uploads/2012/02/Introduction-to-Systems-Modeling-Concepts.pdf

- Yentl Van Tendeloo, Hans Vangheluwe. Introduction to Parallel DEVS Modelling and Simulation. SpringSim-Tutorial, 2017 April 23-26, Virginia Beach, VA, USA. SCS, 2017.
  Retrieved from: http://msdl.cs.mcgill.ca/people/yentl/papers/2017-DEVSTutorial.pdf

1 Introduction to Systems Modeling Concepts

(Chapter 1 from Zeigler, B. P., H. Praehofer and T. G. Kim (2000). Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, (2nd Ed.) Academic Press, NY.)

This chapter introduces some key concepts that underlie the framework and methodology for modeling and simulation (M&S) originally presented in "Theory of Modeling and Simulation" published in 1976 – referred to hereafter as TMS76 to distinguish it from the current revised edition TMS98. Perhaps the most basic concept is that of mathematical systems theory. First developed in the nineteen sixties, this theory provides a fundamental, rigorous mathematical formalism for representing dynamical systems. There are two main, and orthogonal, aspects to the theory:

- **levels of system specification** – these are the levels at which we can describe how systems behave and the mechanisms that make them work the way they do.
- **systems specification formalisms –** these are the types of modeling styles, such continuous or discrete, that modelers can use to build system models.

Although the theory is quite intuitive, it does present an abstract way of thinking about the world that you will probably find unfamiliar. So we introduce the concepts in a spiral development consisting of easy-to-grasp stages – with each spiral revolution returning to a more faithful version of the full story.

In this chapter we first introduce some basic systems concepts, then motivate the systems specification formalisms by describing their evolution over time. This also provides a way to point out the differences between TMS98 and its predecessor. Finally, we discuss the levels of system specification, illustrating them with familiar examples. In this ground stage of our spiral development, the presentation is informal and prepares the way for the framework for M&S that comes in the next chapter. Later, in the second part of the book, we return to a more rigorous development of the concepts to lay a sound basis for the developments to come in the third part.

1.1 Systems Specification Formalisms

System theory distinguishes between system *structure* (the inner constitution of a system) and *behavior* (its outer manifestation). Viewed as a black box (Figure 1) the external *behavior* of a system is the relationship it imposes between its input time histories and output time histories. The system's input/output behavior consists of the pairs of data records (input time segments paired with output time segments) gathered from a real system or model. The internal *structure* of a system includes its state and state transition mechanism (dictating how inputs transform current states into successor states) as well as the state-to-output mapping. Knowing the system structure allows us to deduce (analyze, simulate) its behavior. Usually, the other direction (inferring structure from behavior) is not univalent – indeed, discovering a valid representation of an observed behavior is one of the key concerns of the M&S enterprise.

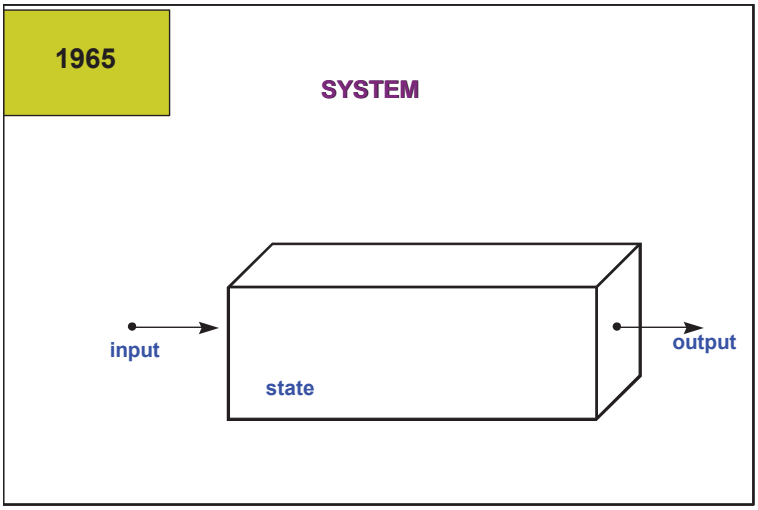


Figure 1 Basic System Concepts

An important structure concept is that of *decomposition* namely, how a system may be broken down into component systems (Figure 2). A second concept is that of *composition,* i.e., how component systems may be coupled together to form a larger system. Systems theory is *closed under composition* in that the structure and behavior of a composition of systems can be expressed in the original system theory terms. The ability to continue to compose larger and larger systems from previously constructed components leads *to hierarchical construction.* Closure under composition guarantees that such a composition results in a system, called its *resultant*, with well defined structure and behavior. *Modular* systems have recognized input and output ports through which all interaction with the environment occurs. They can be *coupled* together by coupling output ports to input ports and can have hierarchical structure in which component systems are coupled together to form larger ones.

The difference between a decomposed systems, as in Figure 2, and undecomposed systems, as in Figure 1, provides our first introduction to *levels* of systems specification. We'll say later that the former are at a higher level of specification than the latter since they provide more information about the structure of the system.

1.1.1 Relation To Object Orientation

Models developed in a system theory paradigm bear a resemblance to concepts of object-oriented programming. Both *objects* and system models share a concept of internal state. However, mathematical systems are formal structures that operate on a time base while programming objects typically do not have an associated temporal semantics. Objects in typical object oriented paradigms are not hierarchical or modular in the sense just described. The coupling concept in modular systems provides a level of delayed binding – a system model can place a value on one of its ports but the actual destination of this output is not determined until the model becomes a component in a larger system and a coupling scheme is specified. It can therefore: a) be developed and tested as a stand alone unit, b) be placed in a model repository and reactivated at will and c) reused in any applications context in which its behavior is appropriate and coupling to other components makes sense.

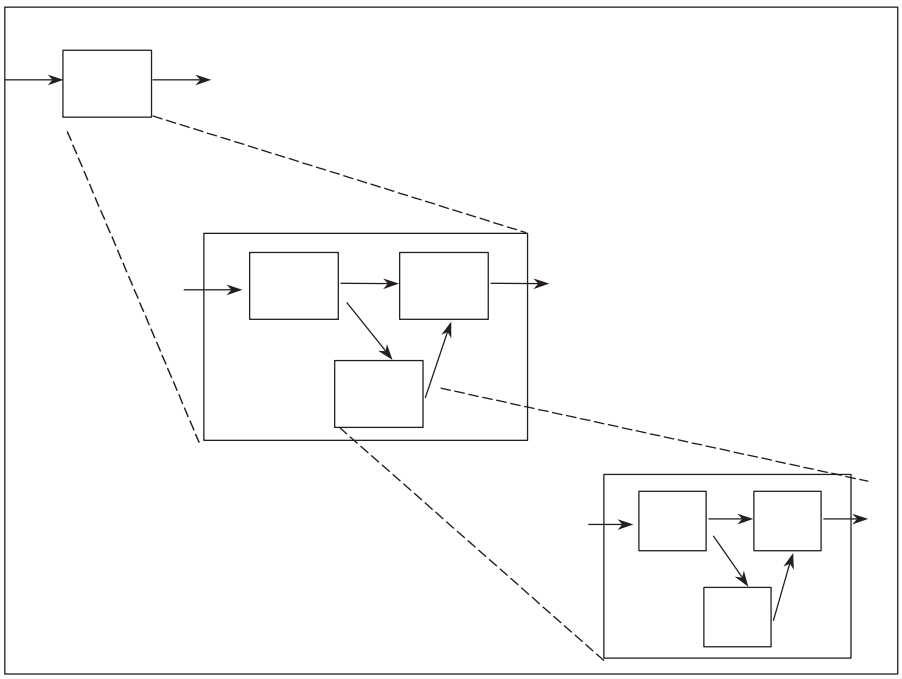


Figure 2 Hierarchical System Decomposition

While coupling establishes output-to-input pathways, the systems modeler is completely free to specify how data flows along such channels. Information flow is one of many interactions that may be represented. Other interactions include physical forces and fields, material flows, monetary flows, and social transactions. The systems concept is broad enough to include the representation of any of these and supports the development of M&S environments that can make including many within the same large-scale model.

Although systems models have formal temporal and coupling features not shared by conventional objects, object-orientation does provide a supporting computational mechanism for system modeling. Indeed, there have been many object-oriented implementations of hierarchical, modular modeling systems. These demonstrate that object-oriented paradigms, particularly for distributed computing, can serve as a strong foundation to implement the modular systems paradigm.

1.1.2 Evolution Of Systems Formalisms

As in many situations, portraying the evolution of an idea may help in the understanding of the complexities as they develop. 0 depicts the basic systems modeling formalisms as they were presented in the frist edition, TMS76. This edition was the first book to formulate approaches to modeling as system specification formalisms – shorthand means of delineating a particular system within a subclass of all systems. The traditional differential equation systems, having continuous states and continuous time, were formulated as the class of DESS (Differential Equation System Specifications). Also, systems that operated on a discrete time base such as automata were formulated as the class of DTSS (Discrete Time System Specifications). In each of these cases, mathematical representation had proceeded their computerized incarnations (it has been three hundred years since Newton-Leibnitz!).

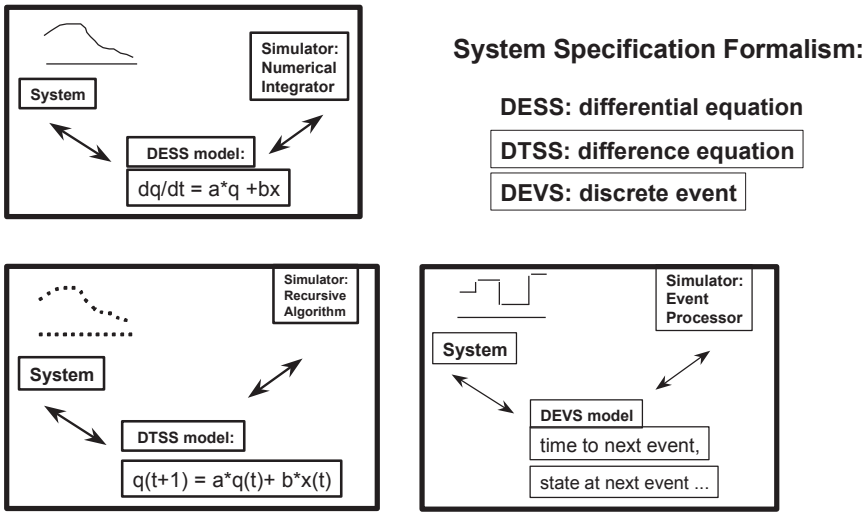


Figure 3 Basic Systems Specification Formalisms

However, the reverse was true for the third class, the Discrete Event System Specifications (DEVS). Discrete event models were largely prisoners of their simulation language implementations or algorithmic code expressions. Indeed, there was a prevalent belief that discrete event "world views" constituted new mutant forms of simulation, unrelated to the traditional mainstream paradigms. Fortunately, that situation has begun to change as the benefits of abstractions in control and design became clear. Witness the variety of discrete event dynamic system formalisms that have emerged [Ho 94]. While each one – examples are Petri Nets, Min-Max algebra, and GSMP (generalized semi-markov processes) – has its application area, none were developed deliberately as subclasses of the systems theory formalism. Thus to include such a formalism into an organized system-theory based framework requires "embedding" it into DEVS.

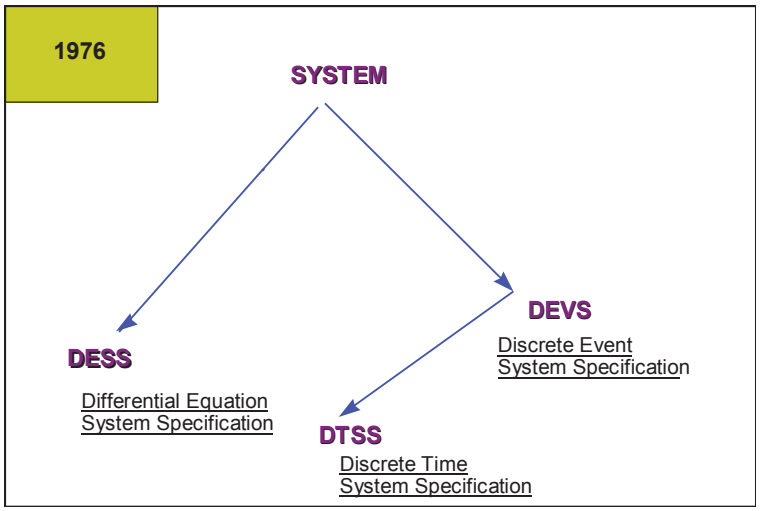


Figure 4 The Dynamics of Basic System Classes

"Embedding." What could such a concept mean? The arrows in Figure 4 indicate *subclass* relationships; for example, they suggest that DTSS is a "subclass of" DEVS. However, it is not literally true that any discrete time system is also discrete event system (their time bases are distinct, for example). So we need a concept of *simulation* that allows us to say when one system can do the essential work of another. One formalism can be *embedded* in another if any system in the first can be simulated by some system in the second. Actually, more than one such relationship, or *morphism*, may be useful, since, as already mentioned, there are various levels of structure and behavior at which equivalence of systems could be required. As a case in point, the TMS76 edition established that any DTSS could be simulated by a DEVS by constraining the time advance to be constant. However, this is not as useful as it could be until we can see how it applies to *decomposed* systems. Until that is true, we either must reconstitute a decomposed discrete time system to its resultant before representing it as a DEVS or we can represent each DTSS component as a DEVS but we can't network the DEVS together to simulate the resultant. The current edition, TMS98, establishes this stronger simulation relation (Chapter 15).

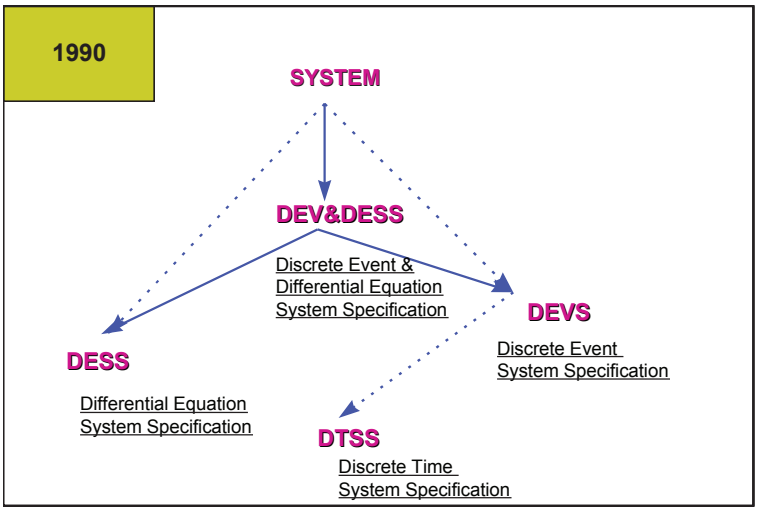


Figure 5 Introducing the DEV&DESS Formalism

1.1.3 Combining Continuous And Discrete Formalisms

Skipping many years of accumulating developments, the next major advance in systems formalisms was the combination of discrete event and differential equation formalisms into one, the DEV&DESS. As shown in Figure 5, this formalism subsumes both the DESS and the DEVS (hence also the DTSS) and thus supports the development of coupled systems whose components are expressed in any of the basic formalisms. Such *multiformalism* modeling capability is important since the world does not usually lend itself to using one form of abstraction at a time. For example, a chemical plant is usually modeled with differential equations while its control logic is best designed with discrete event formalisms. In 1990, one of the authors of this book showed that DEV&DESS was closed under coupling and in order to do so, had to deal with the pairs of input-output interfaces between the different types of systems. Closure under coupling also required that the DEV&DESS formalism provide a means to specify components with intermingled discrete and continuous expressions. Finally, simulator algorithms (so called *abstract simulator*) had to be provided to establish that the new formalism could be implemented in computational form (look ahead to Chapter 9 to see how this was all accomplished).

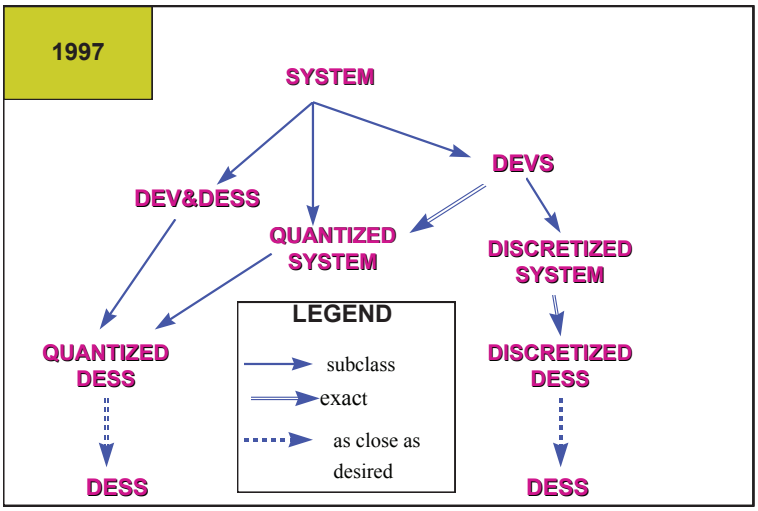


Figure 6 Introducing Quantized Systems

1.1.4 Quantized Systems

The current edition, TMS98, builds on the advances since 1976 especially in the directions pointed to by the introduction of DEV&DESS. Since parallel and distributed simulation is fast becoming the dominant form of model execution, and discrete event concepts best fit with this technology, our focus is on a concept called the *DEVS bus*. This concept, introduced in 1996 by another of the authors and his students, concerns the use of DEVS models, as a "wrappers" to enable a variety of models, to interoperate in a networked simulation. It is particularly germane to the High Level Architecture (HLA) defined by the United States Department of Defense. One way of looking at this idea is that we want to embed any formalism, including for example, the DEV&DESS, into DEVS. Another way is to introduce a new class of systems, called the Quantized System, as illustrated in Figure 6. In such systems, both the input and output are quantized. As an example, an analog-to-digital converter does such quantization by mapping a real number into a finite string of digits. In general, quantization forms equivalence classes of outputs that then become indistinguishable for downstream input receivers, requiring less data network bandwidth, but also possibly incurring error.

Quantization provides a process for representing and simulating continuous systems that is an alternative to the more conventional discretization of the time axis. While discretization leads to discrete time systems, quantization leads to discrete event systems. The beginnings of a theory of quantized systems are presented in Chapter 16 of this book. When we restrict quantization to differential equation systems, we can express the resulting class, Quantized DESS, within DEV&DESS and study its properties, especially from the point of view of the DEVS bus. We can then study the approximation capability and simulation efficiency of DEVS in distributed simulation in comparison with classical time stepped integration and simulation approaches. Particularly with respect to reduction of message passing and network bandwidth (a major concern in distributed simulation) promising results are being obtained.

1.1.5 Extensions of DEVS

Various extensions of DEVS have been developed as illustrated in Figure 7. They will be discussed in Chapter 10. These developments expand the classes of system models that can be represented in DEVS, and hence, integrated within both the DEVS bus and the parent systems theory formalism.

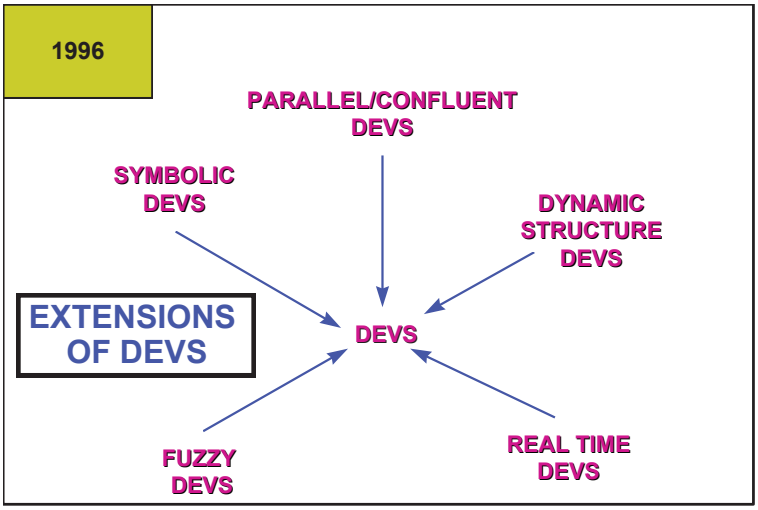


Figure 7 Extensions of the DEVS Formalism

These developments lend credence to the claim that DEVS is a promising computational basis for analysis and design of systems, particularly when simulation is the ultimate environment for development and testing (Figure 8). The claim rests on the *universality* of the DEVS representation, namely the ability of DEVS bus to support the basic system formalisms. This edition goes some distance toward substantiating the claim that DEVS is the unique form of representation that underlies any system with discrete event behavior. This *uniqueness* claim of DEVS, discussed in Chapter 15, offers the promise that the profusion of discrete event formalisms under development for control and management of systems can be embedded as subformalisms of DEVS in the DEVS bus and thus made accessible in an integrated distributed simulation environment.

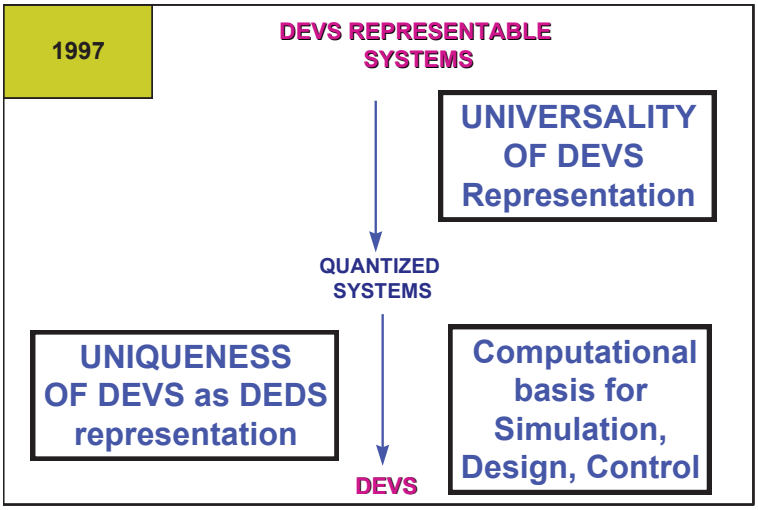


Figure 8 DEVS as a Computational Basis for Simulation, Design and Control

1.2 Levels of System Knowledge

As already mentioned, the systems specification hierarchy is the basis for a framework for M&S which sets forth the fundamental entities and relationships in the M&S enterprise. The hierarchy is first presented in an informal manner and later in Chapter 5 in its full mathematical rigor. Our presentation starts with a review of George Klir's [Klir 1985] systems framework.

Level	Name	What we know at this level
0	Source	what variables to measure and how to observe them
1	Data	data collected from a source system
2	Generative	means to generate data in a data system
3	Structure	components (at lower levels) coupled together to form a generative system

Table 1 Levels of System Knowledge.

Table 1 identifies four basic levels of knowledge about a system recognized by Klir. At each level we know some important things about a system that we didn't know at lower levels. At the lowest level, the *source level* identifies a portion of the real world that we wish to model and the means by which we are going to observe it. As the next level, the *data level* is a data base of measurements and observations made for the source system. When we get to Level 2, we have the ability to recreate this data using a more compact representation, such as a formula. Since typically, there are many formulas or other means to generate the same data, the *generative level*, or particular means or formula we have settled on, constitutes knowledge we didn't have at the data system level. When people talk about models in the context of simulation studies they are usually referring to the concepts identified at this level. That is, to them a model means a program to generate data. At the last level, the *structure level,* we have a very specific kind of generative system. In other words, we know how to generate the data observed at Level 1 in a more specific manner – in terms of component systems that are interconnected together

and whose interaction accounts for the observations made. When people talk about systems, they are often referring to this level of knowledge. They think of reality as being made up of interacting parts – so that the whole is the sum (or a sometimes claimed, more, or less, than the sum) of its parts. Although some people use the term 'subsystems' for these parts, we call them **component** systems (and reserve the term subsystem for another meaning).

As we have suggested, Klir's terms are by no means universally known, understood, or accepted in the M&S community. However, his framework is a useful starting point since it provides a unified perspective on what are usually considered to be distinct concepts. From this perspective, there are only three basic kinds of problems dealing with systems and they involve moving between the levels of system knowledge (Table 2). In *systems analysis*, we are trying to understand the behavior of an existing or hypothetical system based on its known structure. *Systems inference* is done when we don't know what this structure is – so we try to guess this structure from observations that we can make. Finally, in *systems design*, we are investigating the alternative structures for a completely new system or the redesign of an existing one.

Systems Problem	**Does source of the data exist? What are we trying to learn about it?**	**Which level transition is involved?**
systems analysis	The system being analyzed may exist or may be planned. In either case we are trying to understand its behavioral characteristics.	moving from higher to lower levels, e.g., using generative information to generate the data in a data system
systems inference	The system exists. We are trying to infer how it works from observations of its behavior.	moving from lower to higher levels, e.g., having data, finding a means to generate it
systems design	The system being designed does not yet exist in the form that is being contemplated. We are trying to come up with a good design for it.	moving from lower to higher levels, e.g. having a means to generate observed data, synthesizing it with components taken off the shelf.

Table 2 Fundamental Systems Problems.

The central idea is that when we move to a lower level, we don't generate any really new knowledge – we are only making explicit what is implicit in the description we already have. One could argue that making something explicit can lead to insight, or understanding, which is a form of new knowledge, but Klir is not considering this kind of subjective (or modeler dependent) knowledge. In the M&S context, one major form of *systems analysis* is computer simulation which generates data under the instructions provided by a model. While no new knowledge (in Klir's sense) is generated, interesting properties may come to light of which we were not aware before the analysis. On the other hand, *systems inference* and *systems design* are problems that involve **climbing up** the levels. In both cases we have a low level system description and wish to come up with an equivalent higher level one. For systems inference, the lower level system is typically at the data system level, being data that we have observed from some existing source system. We are trying to find a generative system, or even a structure system, that can recreate the observed data. In the M&S context, this is usually called *model construction*. In the case of systems design, the source system typically does not yet exist and our objective is to build one that has a desired functionality. By functionality we mean what we want the system to do; typically, we want to come up with a structure system, whose components are technological, i.e., can be obtained off-the-shelf,

or built from scratch from existing technologies. When these components are interconnected, as specified by a structure system's coupling relation, the result should be a real system that behaves as desired.

It is interesting to note that the process called *reverse engineering* has elements of both inference and design. To reverse engineer an existing system, such as was done in the case of the cloning of IBM compatible PCs, an extensive set of observations is first made. From these observations, the behavior of the system is inferred and an alternative structure to realize this behavior is designed – thus bypassing patent rights to the original system design!

1.3 Introduction to the Hierarchy of Systems Specifications

At about the same time (in the early 1970's) that Klir introduced his epistemological (knowledge) levels, TMS76 formulated a similar hierarchy that is more oriented toward the M&S context. This framework employs a general concept of *dynamical system* and identifies useful ways in which such a system can be specified. These ways of describing a system can be ordered in levels as in Table 3. Just as in Klir's framework, at each level more information is provided in the specification that cannot be derived from lower levels. As can be seen in Table 3, these levels roughly correspond to those of Klir's framework.

Level	**Specification Name**	**Corresponds to Klir's**	**What we know at this level**
0	Observation Frame	Source System	how to stimulate the system with inputs; what variables to measure and how to observe them over a time base;
1	I/O Behavior	Data System	time-indexed data collected from a source system; consists of input/output pairs
2	I/O Function		knowledge of initial state; given an initial state, every input stimulus produces a unique output.
3	State Transition	Generative System	how states are affected by inputs; given a state and an input what is the state after the input stimulus is over; what output event is generated by a state.
4	Coupled Component	Structure System	components and how they are coupled together. The components can be specified at lower levels or can even be structure systems themselves – leading to hierarchical structure.

Table 3 Relation between System Specification Hierarchy and Klir's levels.

The major difference between the two frameworks is that the System Specification Hierarchy recognize that simulation deals with **dynamics**, the way in which systems behave over time. Therefore, *time* is the base upon which all events are ordered. We also view systems as having *input* and *output* interfaces through which they can interact with other systems. As illustrated in Figure 9, systems receive stimuli ordered in time through their *input ports*, and respond on their *output ports*. The term "port" signifies a specific means of interacting with the system. Whether by stimulating it (input)or observing it (output). The time-indexed inputs to systems are *called input trajectories*; likewise, their time-indexed outputs are called *output trajectories*. Ports are the only channels through which one can interact with the system. This means that system are *modular*. While Klir's framework can include dynamics, input/output ports and modularity, it is not dedicated to these concepts. However, understanding these concepts is critical to effectively solving the problems that arise in M&S.

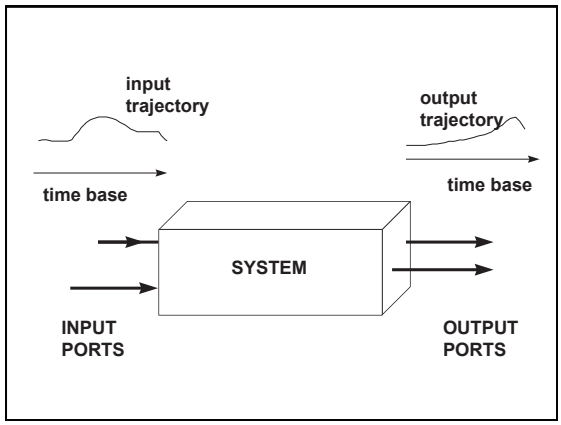


Figure 9 Input/Output System.

Before discussing each level of the specification hierarchy in some detail, let's observe that we could have the very same real world object specified simultaneously at each of the levels. Thus there should be a way to associate the next lower level specification with any given one. This association concept is illustrated in Figure 10. For example, if we have know the detailed structure at the Coupled Component level, then we ought to be able to construct the corresponding specification at the State Transition level. The hierarchy is set up to provide such an *association mapping* at each (other than the lowest) level. Indeed, this is the formal version of climbing down the levels just discussed. Since the association mapping is not necessarily one to one, many upper level specifications may map to the same lower level one. This is the underlying reason why climbing up the levels is much harder than climbing down the levels. Indeed, when we select one of the associated upper level specifications for a given lower level one, we are gaining knowledge we didn't have at the lower level.

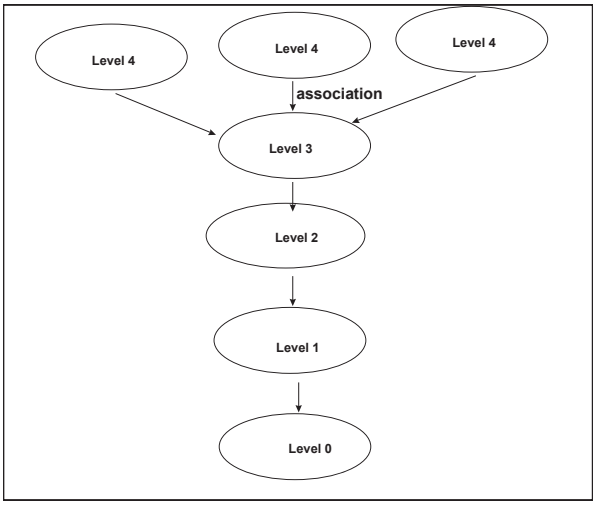


Figure 10 Association between levels of the System Specification Hierarchy

1.4 The Specification Levels Informally Presented

1.4.1 Observation Frame

The Observation Frame specifies how to stimulate the system with inputs; what variables to measure and how to observe them over a time base.

As an example, Figure 11 shows a forest subject to *lightning, rain* and *wind,* modeled as input ports and *smoke* produced from fire, represented as an output port. This is a level 0 or Observation Frame specification. Note the choice of variables we included as ports and their *orientation* (i.e., whether they are input or output ports). We could have chosen differently. For example, we could have included an output port to represent heat radiation. Moreover, rather than representing each variable by a single value, it can be distributed over space, i.e., represented by an array of values. Such choices depend on our modeling objectives and are specified through *experimental frames*, a concept which we discuss in Chapter 2.

Figure 12 shows some examples of input and output trajectories. The input trajectory on the *lightning* port shows a bolt occurring at some particular time t_0. Only one such bolt occurs in the time period shown. The *smoke* output trajectory, at the top, depicts a gradual build up of smoke starting at t_0 (so presumably, caused by a fire started by the lightning bolt). The possible values taken on by *smoke*, called its *range,* would result from some appropriate measurement scheme, e.g., measuring density of particulate material in grams/cubic meter. The pair of input, and associated output, trajectories is called a *input/output (or I/O) pair.* Figure 12 also displays a second I/O pair with the same input trajectory but different output trajectory. It represents the fact that there may be many responses to the same stimulus. In the second case, lightning did not cause a major fire, since the one that broke out quickly died. Such multiple output trajectories (for the same input trajectory) are characteristic of knowledge at Level 1. Knowing how to disambiguate these output trajectories is knowledge we will gain at the next level.

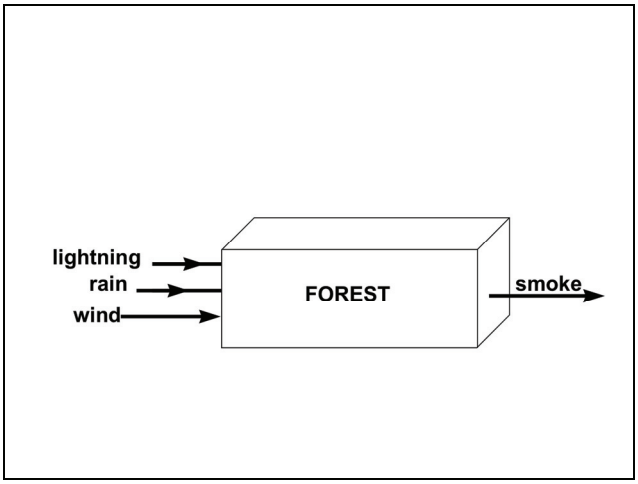


Figure 11 A forest specified as a system in the Observation Frame (Level 0)..

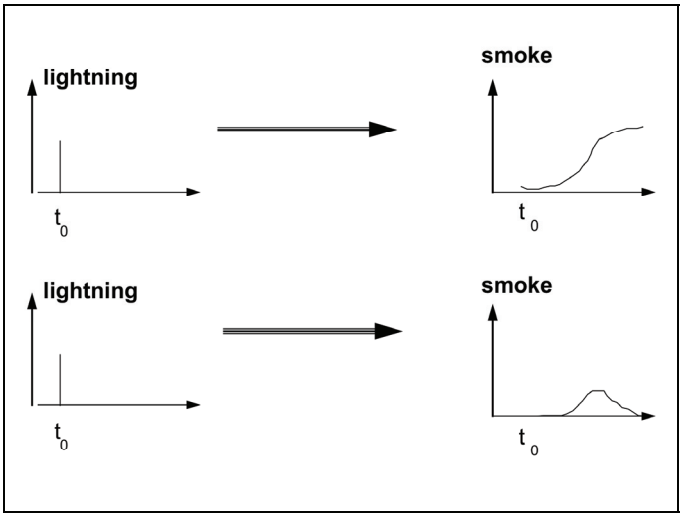


Figure 12 Some Input-Output Pairs for the Forest System Frame of Figure 11.

1.4.2 I/O Behavior and I/O Function

The collection of all I/O pairs gathered by observation is called the *I/O Behavior* of a system. Returning to Table 3, this represents a system specification at Level 1. Now suppose that we are able to uniquely predict the response of the smoke output to a lightning bolt. For example, suppose we know that if the vegetation is dry, then a major fire will ignite, but if the vegetation is moist then any fire will quickly die. Having such a factor represents knowledge at Level 2, that of the *I/O Function.* Here, in addition to lower level information, we add *initial states* to the specification – when the initial state is known, there is a functional relationship between input and output trajectories, i.e., the initial state determines the **unique** response to any input (Figure 13 a).

1.4.3 State Transition System Specification

At the next level (3) of system specification, we can specify not only initial state information but also how the *state changes* as the system responds to its input trajectory. Figure 13 b) and c) illustrate this important concept. Figure 13 b) presents the situation where the forest is in state (dry vegetation, unburned) when a lightning bolt occurs at time t_0. The state that the forest is in at time t_1 when a second bolt occurs is (dry vegetation, burnt) reflecting the fact that a fire has ignited. Since the forest is in a different state, the effect of this second bolt is different from the first. Indeed, since there is little left to burn, there is no effect of the second bolt.

In contrast, Figure 13 c) illustrates the situation where the forest is wet and unburned when the first bolt occurs. It does not cause a major fire, but it does dry out the vegetation so the resulting state is (dry, unburned). Now the second bolt produces a major fire, just as the first bolt did in Figure 13 b) – since both the state and subsequent input trajectory are the same, the response of the system is the same.

1.4.4 Coupled Component System Specification

At the highest level of system specification, we can describe more about the internals of the system. Until now, it was a black box, at first observable only through I/O ports. Subsequently, we were able to peer inside to the extent of observing its state. Now, at level 4, we can specify how the system is composed of interacting components. For example, Figure 14 illustrates how a forest system could be composed of interacting cells, each representing a spatial region, with adjacent cells interconnected. The cells are modeled at level 3, i.e., their state transition and output generation definitions are used to act upon inputs, and generate outputs, respectively to and from, other cells. The cells are coupled together using ports. The output ports of one cell are coupled to the input ports of its neighbors.

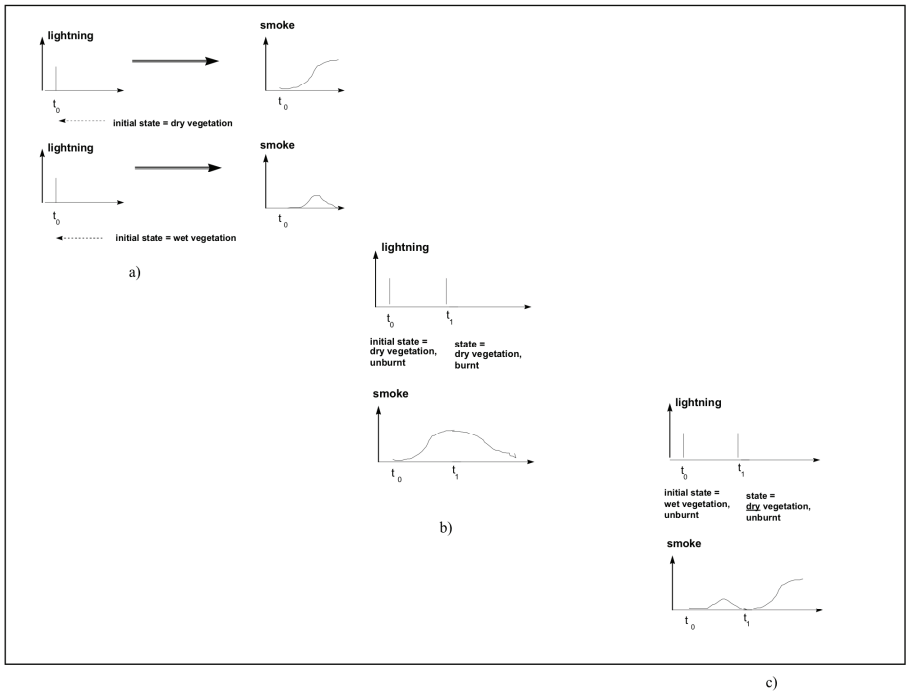


Figure 13 Initial State Concept: a specification at Level 3 (I/O Function) in which we have initial state knowledge about the forest.

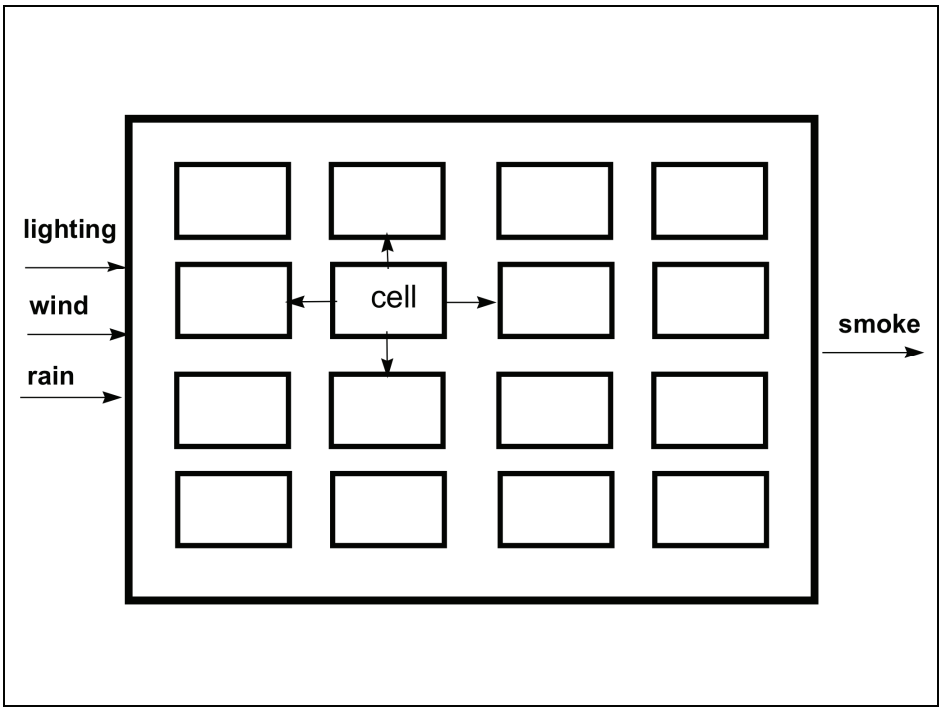


Figure 14 Component Structure System Specification for the Forrest System.

1.5 System Specification Morphisms: Basic Concepts

The system specification hierarchy provides a stratification for constructing models. But, while constructing models is the basic activity in M&S, much of the real work involves establishing relationships between system descriptions. The system specification hierarchy also provides an orderly way of presenting and working with such relationships. Figure 15 illustrates the idea that pairs of system can be related by morphism relations at each level of the hierarchy. A morphism is a relation that places elements of system descriptions into correspondence as outlined in Table 4.

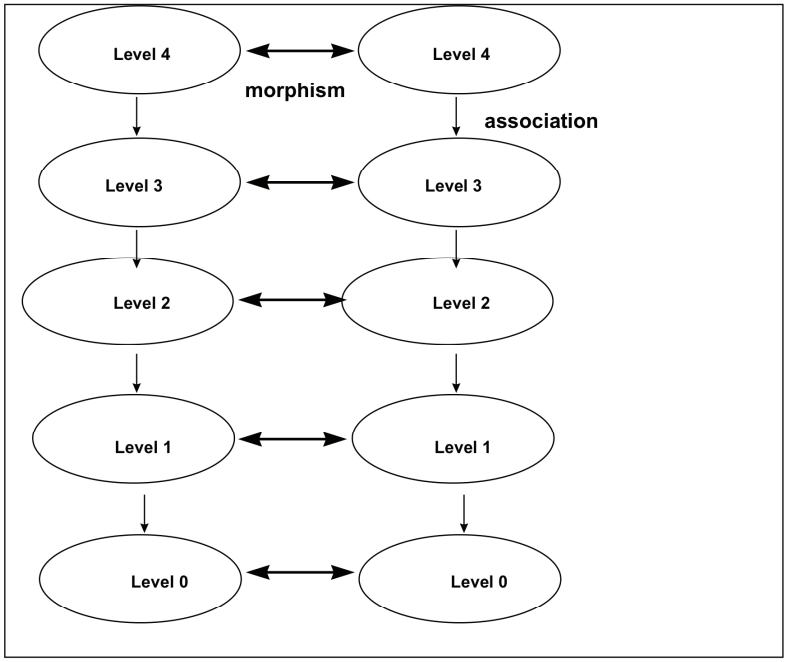


Figure 15 Morphism Concepts for System Specification Hierarchy

For example, at the lowest level, two Observation Frames are *morphic*, if we can place their defining elements – inputs, outputs, and time bases into correspondence. Such Frames are *isomorphic* if their inputs, outputs, and time bases respectively, are identical. In general, the concept of morphism tries to capture similarity between pairs of systems at the same level of specification. Such similarity concepts have to be consistent between levels. When we associate lower level specifications with their respective upper level ones, a morphism holding at the upper level must imply the existence of one at the lower level. The morphisms defined in Table 4 are set up to satisfy these constraints.

Level	Specification Name	Two Systems are Morphical at this level if:

Level	Specification Name	Two Systems are Morphical at this level if:
0	Observation Frame	their inputs, outputs and time bases can be put into correspondence
1	I/O Behavior	they are morphic at level 0 and the time-indexed input/output pairs constituting their I/O behaviors also match up in one-one fashion
2	I/O Function	they are morphic at level 0 and their initial states can be placed into correspondence so that the I/0 functions associated with corresponding states are the same
3	State Transition	the systems are homomorphic (explained below)
4	Coupled Component	components of the systems can be placed into correspondence so that corresponding components are morphic; in addition, the couplings among corresponding components are equal

Table 4 Morphism relations between systems in System Specification Hierarchy and Klir's levels.

The most important morphism, called **homomorphism**, resides at the State Transition level and is illustrated in Figure 16. Consider two systems specified at level 3, S and S', where S may be bigger than S' in the sense of having more states. Later, we'll see that S could represent a complex model and S' a simplification of it. Or S could represent a simulator and S' a model it is executing. When S' goes through a state sequence such as *a,b,c,d*, then S should go through a corresponding state sequence say *A,B,C,D*. Typically, a simulator has a lot of apparatus, represented in its states, necessary to accommodate the whole class of models rather than a single one. Thus we don't assume that states of S and S' are identical – only that there is a predefined correspondence between them illustrated by the shaded connecting lines in the figure. Now to establish that this correspondence is a homomorphism requires that whenever S' specifies a transition, such as from state *b* to state *c*, then S actually makes the transition involving corresponding states *B* and *C*. Typically, the simulator is designed to take a number of *microstate* transitions to make the *macrostate* transition from *B* to *C*. These are computation steps necessary to achieve the desired end result. It is not hard to see that if such a homomorphism holds for all states of S', then any state trajectory in the S' will be properly reproduced in S.

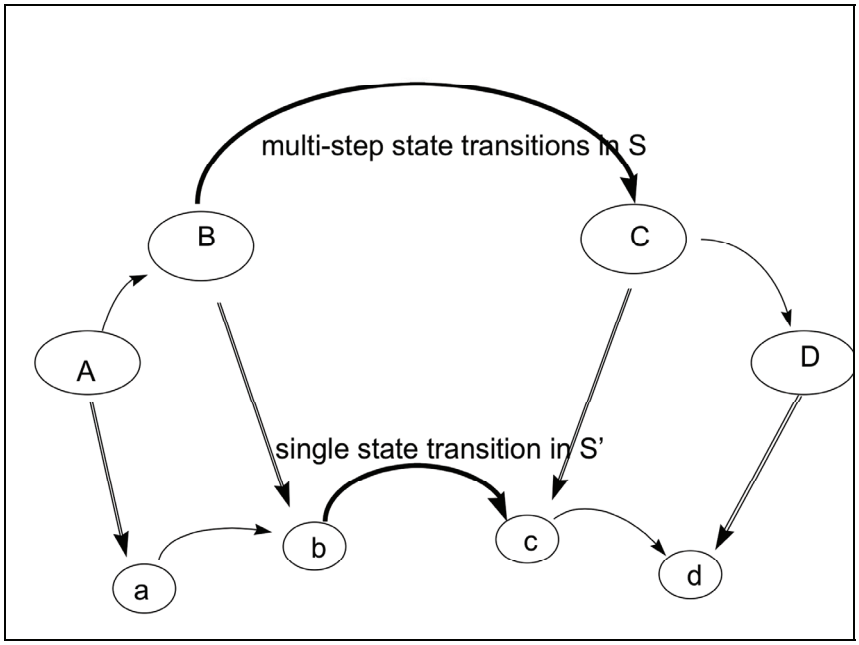


Figure 16 Homomorphism Concept. This figure illustrates the preservation of state transitions that a homomorphism requires.

Often, we require that the correspondence hold in a step-by-step fashion. In other words, that the transition from a to b is mirrored by a one-step transition from A to B. Also, as just indicated, we want the I/O Behavior's of homomorphic models specified at the I/O System level to be the same. Thus, as in Figure 17, we require that the outputs produced from corresponding states be the same. In this type of homomorphism, the values and timing of the transitions and outputs of the base model are preserved in the lumped model. Thus, in this case, the state and output trajectories of the two models, when started in corresponding states, are the same.

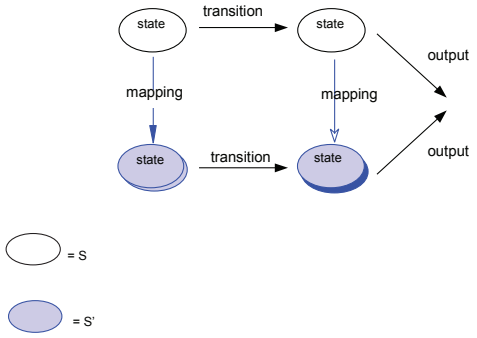


Figure 17 Homomorphism: a mapping preserving step-by-step state transition and output

1.6 Summary

We have outlined the basic concepts of systems theory: structure, behavior, levels of system specification and their associated morphisms. We have brought out the important distinctions that justify having different levels of specification. However, we have not considered all the possible distinctions and levels. For example, the important distinction between modular and non-modular systems has not been recognized with distinct levels. A more complete hierarchy will emerge as revisit the concepts introduced here in a more formal and rigorous manner in Chapter 5. We also have introduced the basic system specification formalisms and outlined the advances in the development of such formalisms that characterize this second edition, TMS98.

We now turn to a framework for modeling and simulation that identifies the key elements and their relationships. The systems specification hierarchy will provide the basis for presenting this framework. For example, we use specifications at different levels to characterize the different elements. The various system specification formalisms and their simulators provide the operational means to employ the framework in real world applications. We focus on real world application in the last part of the book.

1.7 Sources

The basic concepts of systems theory were developed by pioneers such as Arbib [Arbib 67], Zadeh [Zadeh 79] (later known more for his fuzzy sets theories), Klir [Klir 85], Mesarovic [Mesarovic 75] and Wymore. [Wymore 77]. Since the first edition of this book [Zeigler 76] there have been several trends – toward deepening the theory ([Mesarovic 89] [Wymore 93]), extending its range of applicability with computerized tools [Pichler 92] and going on to new more abstract formulations [Takahashi 95]. Also, somewhat independently, a new recognition of systems concepts within discrete event systems was fostered by Ho [Ho 94]. The DEV&DESS formalism was introduced by Praehofer in his doctoral dissertation [Praehofer 91]. The DEVS Bus originated in the research group of Tag Gon Kim [Yong 96]. Quantized system theory was first presented in [Zeigler 98]. A recent collection of systems concepts in computer science is given in [Allbrecht 98].

[Arbib 67] Arbib, M.A., *Theories of abstract automata*. 1967: Prentice-Hall.

[Allbrecht 98] *Systems: Theory and Practice,* Ed.: R. Allbrecht,. 1998, Springer: Vienna.

[Ho 94] Ho, Y.C. Discrete Event Dynamic Systems : Analyzing Complexity and Performance in the Modern World. 1994: IEEE Press.

[Klir 85] Klir, G.J., *Architecture of Systems Complexity*, Sauders, NewYork, 1985.

[Mesarovic 75] Mesarovic, M.D. and Y. Takahara, General Systems Theory: Mathematical Foundations. 1975, New York, NY: Academic Press.

[Mesarovic 89] Mesarovic, M.D. and Y. Takahara, Abstract Systems Theory. 1989, New York: Springer-Verlag.

[Padulo 72] Padulo, L., System Theory; *A Unified State-Space Approach to Continuous and Discrete Systems*. 1972: W B Saunders Co

[Pichler 92] Pichler, F. and H. Schwaertzel, eds. *CAST (Computer Aided System Theory) Methods in Modelling*. 1992, Springer-Verlag: Berlin.

[Praehofer 91] Praehofer, H., System Theoretic Foundations for Combined Discrete-Continuous System Simulation, . 1991, Johannes Kepler University of Linz: Linz, Austria.

[Takahashi 95] Takahashi, S. and Y. Takahara, Logical Approach to Systems Theory. 1995, London: Springer-Verlag.

[Wymore 77] Wymore, W., *Mathematical Theory of Systems Engineering : The Elements*. 1977: Krieger Publishing Company.

[Wymore 93] Wymore, A.W., Model-Based Systems Engineering : An Introduction to the Mathematical Theory of Discrete Systems and to the Tricotyledon Theory of System Design. 1993: CRC Press

[Yong 96] Yong, J. and T.G. Kim. A Heterogeneous Distributed Simulation Framework Based on DEVS Formalism. in AI, Simulation and Planning in High Autonomy Systems. 1996. San Diego: Web: www-ais.arizona.edu.

[Zadeh 79] Zadeh, L.A. and C.A. Desoer, *Linear System Theory*. 1979: Krieger Publishing Co.

[Zeigler 76] *Theory of Modeling and Simulation*. 1976, Wiley Interscience Co.

[Zeigler 98] Zeigler, B.P. and J.S. Lee. Theory of Quantized Systems: Formal Basis for DEVS/HLA Distributed Simulation Environment. in Enabling Technology for Simulation Science(II), SPIE AeoroSense 98. 1998. Orlando, FL.

Definitions, Acronyms, Abbreviations

DEDS – Discrete Event Dynamic Systems

DESS – Differential Equation System Specification

DEVS – Discrete Event System Specification

DTSS – Discrete Time System Specification

DEV&DESS – Discrete Event and Differential Equation System Specification

M&S – Modeling and Simulation

TMS76 – 1976 Edition of Theory of Modeling and Simulation

TMS98 – 1998 Edition of Theory of Modeling and Simulation

INTRODUCTION TO PARALLEL DEVS MODELLING AND SIMULATION

Yentl Van Tendeloo
University of Antwerp, Belgium
Yentl.VanTendeloo@uantwerpen.be

Hans Vangheluwe
University of Antwerp, Belgium
Flanders Make, Belgium
McGill University, Montréal, Canada
hv@cs.mcgill.ca

Abstract

DEVS is a popular formalism for modelling complex dynamic systems using a discrete-event abstraction. During a bounded time interval, only a finite number of relevant events can occur. The state variables are considered to change instantaneously, with the state being constant in between two state changes. Main advantages of DEVS are its rigorous formal definition, and its support for modularity: models can be nested. Thanks to its precise specification as well as its modularity, DEVS frequently serves as a simulation "assembly language" to which models in other formalisms are mapped. It is suited for both manual modelling, as well as for automated generation. This furthermore makes it possible to combine models in different formalisms together, by mapping both to DEVS.

This tutorial provides an introductory tutorial on the practical usage of the Parallel DEVS formalism. The formalism is explained in a bottom-up fashion, starting from simple autonomous Atomic (i.e., non-hierarchical) DEVS models, up to Coupled (i.e., hierarchical) DEVS models. Each aspect of the DEVS formalism is considered individually by means of a minimal running example: a simple traffic light. While the focus is on the practical use of DEVS modelling and simulation, the necessary theoretical foundations are interleaved, albeit at a high level of abstraction. More advanced topics, such as closure under coupling, are briefly mentioned.

Examples are presented using PythonPDEVS, though the foundations and techniques apply equally well to other DEVS simulation tools.

1 INTRODUCTION

This tutorial provides an introductory tutorial on the practical usage of the Parallel DEVS formalism (Chow and Zeigler 1994). Parallel DEVS is a discrete-event formalism with a long history, which started in 1976 with the introduction of Classic DEVS (Zeigler 1976). The content of this tutorial is primarily based on the influential book "Theory of Modelling and Simulation" (Zeigler 1976), which defined the original Classic DEVS formalism, with selected topics from "Multifacetted Modelling and Discrete Event Simulation" (Zeigler 1984) and the more recent edition of "Theory of Modelling and Simulation" (Zeigler, Praehofer, and Kim 2000). For the extension to Parallel DEVS, the founding papers of Parallel DEVS (Chow and Zeigler 1994, Chow, Zeigler, and Kim 1994) are integrated.

Contrary to other introductions to Parallel DEVS, our tutorial provides a bottom-up introduction that focusses on example-driven explanation of the various aspects of Parallel DEVS. And while we still briefly touch upon the theoretical foundations, these are interleaved with an actual application of the theory. This application is in the form of a traffic light: we start from a simple autonomous traffic light without output

SpringSim-Tutorial, 2017 April 23-26, Virginia Beach, VA, USA; *©2017 Society for Modeling & Simulation International (SCS)*

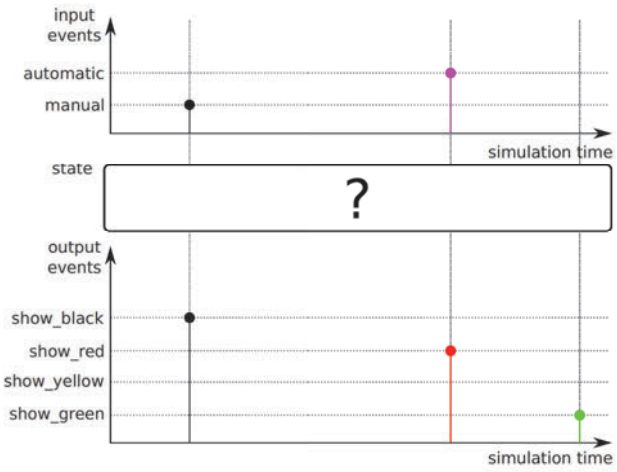


Figure 1: Discrete event abstraction.

and keep extending this to use all features of Parallel DEVS. Afterwards, hierarchy is introduced with the same example, and we glimpse at more advanced concepts.

As our tutorial is example-driven, we use PythonPDEVS (Van Tendeloo and Vangheluwe 2014, Van Tendeloo and Vangheluwe 2016b) as a simple Parallel DEVS simulator written in Python, a high-level, interpretted language. We use PythonPDEVS instead of another Parallel DEVS simulator, as Python is a simple language that is familiar to most people, and its semantics is, at least in the scope of this tutorial, intuitive. This makes PythonPDEVS models easy to understand, even without intimate knowledge of Python. Nonetheless, our tutorial focusses primarily on the theoretical aspects of Parallel DEVS, which is equally valid for other Parallel DEVS simulation tools. A comparison of several popular DEVS tools (Van Tendeloo and Vangheluwe 2016a) can be used to find the most appropriate tool. PythonPDEVS and all used examples can be found online at the PythonPDEVS project page[1], which also links to the documentation.

2 DISCRETE EVENT MODELLING

DEVS is a discrete-event formalism, so we start of with a brief description of the discrete-event abstraction. In a discrete-event abstraction, only a finite number of relevant events occur during a bounded time interval. This is in contrast to continuous models, which take *continuous signals* as input.

A discrete-event system is characterized by the output events it generates in response to a series of input events. There is no restriction on the internal state of the model, as the system is seen as a "black box".

An example of such a trace is shown in Figure 1, representing a traffic light system. As can be seen, there is a finite number of input and output events in any bounded time interval. It is not surprising that the input is a serie of events, as it is someone pressing a button to switch the mode of the traffic light. The output, however, also uses events, and thus only raise an event when the color of the traffic light changes. The receiver of the event therefore needs to remember the last event it received, in order to tell what light is currently active. As there is no restriction on the internal state, this part is hidden in the figure.

In the remainder of the tutorial, we go in-depth as to how we can define a discrete event system that generates this output trace for this specific input trace.

[1] http://msdl.cs.mcgill.ca/projects/PythonPDEVS/

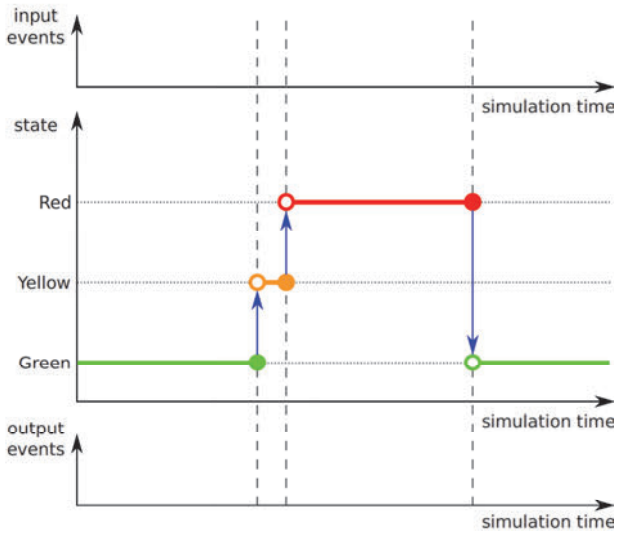


Figure 2: Trace of the autonomous traffic light.

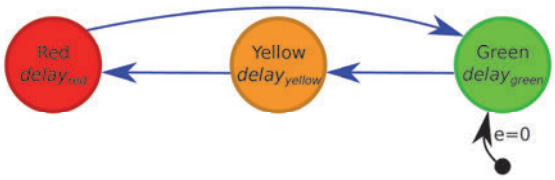


Figure 3: Model generating trace in Figure 2.

3 ATOMIC DEVS MODELS

We commence our explanation of DEVS with the atomic models. As their name suggests, atomic models are the indivisable building blocks of a model. It is these blocks that define behaviour. Nonetheless, they have no hierarchy whatsoever.

Throughout this section, we build up the complete formal specification of an atomic model, introducing new concepts when they become necessary. In each intermediate step, we show and explain the concepts we introduce, how they are present in the running example, and how this influences semantics.

3.1 Autonomous Model

The simplest form of a traffic light is an autonomous traffic light. We expect to see a trace similar to that of Figure 2. Visually, Figure 3 presents an intuitive representation of a model that could generate this kind of trace.

The atomic model can take external parameters, passed at initialization time. For example, it is possible for the atomic DEVS model to use parameters $delay_{green}$, $delay_{yellow}$, and $delay_{red}$ instead of defining constants. These parameters can then be used in the model as if they were constants.

Trying to formally describe Figure 3, we distinguish these elements:

1. **State Set** ($S : \times_{i=1}^{n} S_i$)
   The most obvious aspect of the traffic light is the state it is in, which is indicated by the three different colours it can have. These states are *sequential*: the traffic light is in exactly one of these states at all times.
2. **Time Advance** ($ta : S \to \mathbb{R}^{+}_{0,+\infty}$)
   For each of the states just defined, we notice the timeout in them: the time advance of the state. This duration can be any positive real number, including zero and infinity. Note that DEVS does not consider time bases: simulation time is just a real number and time units are left up to the user.
3. **Internal Transition** ($\delta_{int} : S \to S$)

$$M = \langle S, q_{init}, \delta_{int}, ta \rangle$$

$$S = \{\text{GREEN}, \text{YELLOW}, \text{RED}\}$$
$$q_{init} = (\text{GREEN}, 0.0)$$
$$\delta_{int} = \{\text{GREEN} \rightarrow \text{YELLOW},$$
$$\text{YELLOW} \rightarrow \text{RED},$$
$$\text{RED} \rightarrow \text{GREEN}\}$$
$$ta = \{\text{GREEN} \rightarrow delay_{green},$$
$$\text{YELLOW} \rightarrow delay_{yellow},$$
$$\text{RED} \rightarrow delay_{red}\}$$

Figure 4: Specification of an autonomous traffic light.

```
class TrafficLight(AtomicDEVS):
    def __init__(self, delay_green,
                 delay_yellow, delay_red):
        AtomicDEVS.__init__(self, "Light")
        self.state = "Green"
        self.elapsed = 0.0

        self.delay_green = delay_green
        self.delay_yellow = delay_yellow
        self.delay_red = delay_red

    def intTransition(self):
        return {"Green": "Yellow",
                "Yellow": "Red",
                "Red": "Green"}[self.state]

    def timeAdvance(self):
        return {"Green": self.delay_green,
                "Yellow": self.delay_yellow,
                "Red": self.delay_red}[self.state]
```

Figure 5: Implementation of an autonomous traffic light.

With the states and timeouts defined, the final part is the definition of which is the next state from a given state: the internal transition. For each state with a finite time advance, it returns the next state upon expiration of the time advance timeout.

4. **Initial Total State** ($q_{init} : Q = (s,e) | s \in S, 0 \leq e \leq ta(s)$)
   We also define the initial state of the system, even though it is not present in the original specification of the Parallel DEVS formalism. Nonetheless, it is a vital part of any model. But instead of being an "initial state (s_{init})", it is a total state. This means that we not only select the initial state of the system, but also how long we are already in this state: the *elapsed time*.

We describe the model in Figure 3 as a 4-tuple in Figure 4, and as PythonPDEVS code in Figure 5.

For this simple formalism, we define the semantics as in Algorithm 1. The model is initialized with the simulation time set to 0, the state set to the initial state (*e.g.*, GREEN), and the time of the last transition set to the initial elapsed time (*e.g.*, 0.0). Simulation updates the time with the returnvalue of the time advance function, and executes the internal transition function on the current state to get the new state.

Algorithm 1 DEVS simulation pseudo-code for autonomous models.

```
time ← 0
current_state ← initial_state
last_time ← -initial_elapsed
while not termination_condition() do
   time ← last_time + ta(current_state)
   current_state ← δ_int(current_state)
   last_time ← time
end while
```

3.2 Autonomous Model With Output

Recall that DEVS is a modular formalism, with only the atomic model having access to its internal state. This raises a problem for our traffic light: others have no way of knowing its current state (*i.e.*, its colour).

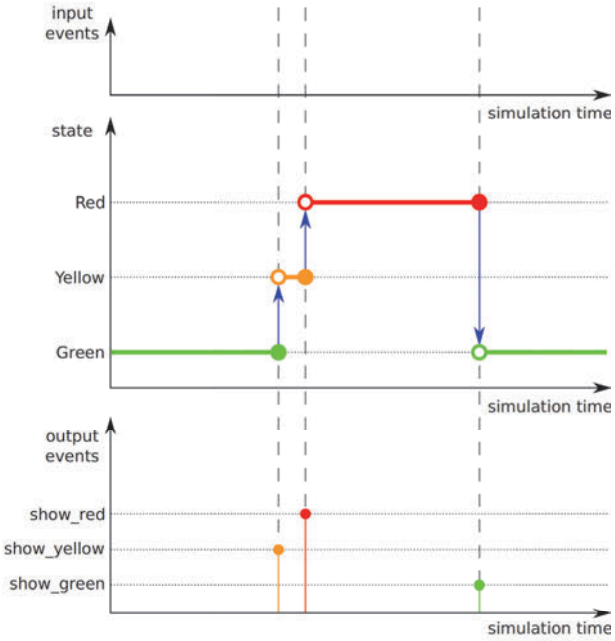


Figure 6: Trace of the autonomous traffic light with output.

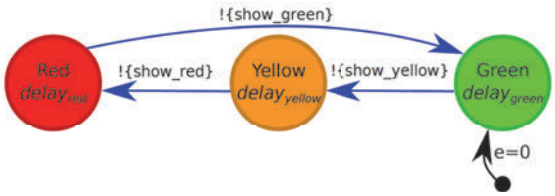


Figure 7: Model generating trace in Figure 6.

We therefore want the traffic light to output its colour, such that others can use it. Our desired trace is shown in Figure 6. We see that we now output events indicating the start of the specified period. DEVS is a discrete event formalism: the output is only a single event indicating the time and is not a continuous signal. The receiver of the event thus would have to store the event to know the current state of the traffic light at any given point in time. This model is shown in Figure 7, using the exclamation mark on a transition to indicate output generation. Our event is enclosed in curly braces, as we output a bag of events instead of a single event.

Analysing the updated model, we see that two more concepts are required to allow for output.

1. **Output Set** ($Y : \times_{i=1}^{l} Y_i$)
   Similarly to defining the set of allowable states, we should also define the set of allowable outputs. This set serves as an interface to other components, defining the events the model expects to receive.
2. **Output Function** ($\lambda : S \to Y^b$)
   With the set of allowable events defined, we still need a function to actually generate the events. As seen in the Figure of the model, the event is generated *before* the new state is reached. This means that instead of the new state, the output function still uses the old state (*i.e.*, the one that is being left). In the case of our traffic light, the output function needs to return the light to toggle, based on the *next* state instead of the current state. For example, if the output function receives the GREEN state as input, it needs to generate a *show_yellow* event, as it is entering the YELLOW state. We output a bag of events, meaning that, at the same time, multiple events can be raised simultaneously, potentially even the same event. This bag, however, can also be empty, indicating that the output function did not generate any output for this state.

We describe the model in Figure 7 as a 6-tuple in Figure 8, and as PythonPDEVS code in Figure 9.

$$M = \langle Y, S, q_{init}, \delta_{int}, \lambda, ta \rangle$$

$$\begin{aligned}
Y &= \{show_green, show_yellow, show_red\} \\
S &= \{\text{GREEN}, \text{YELLOW}, \text{RED}\} \\
q_{init} &= (\text{GREEN}, 0.0) \\
\delta_{int} &= \{\text{GREEN} \rightarrow \text{YELLOW}, \\
&\quad \text{YELLOW} \rightarrow \text{RED}, \\
&\quad \text{RED} \rightarrow \text{GREEN}\} \\
\lambda &= \{\text{GREEN} \rightarrow \{show_yellow\}, \\
&\quad \text{YELLOW} \rightarrow \{show_red\}, \\
&\quad \text{RED} \rightarrow \{show_green\}\} \\
ta &= \{\text{GREEN} \rightarrow delay_{green}, \\
&\quad \text{YELLOW} \rightarrow delay_{yellow}, \\
&\quad \text{RED} \rightarrow delay_{red}\}
\end{aligned}$$

Figure 8: Specification of an autonomous traffic light with output.

```
class TrafficLight(AtomicDEVS):
    def __init__(self, delay_green,
            delay_yellow, delay_red):
        AtomicDEVS.__init__(self, "Light")
        self.state = "Green"
        self.elapsed = 0.0

        self.delay_green = delay_green
        self.delay_yellow = delay_yellow
        self.delay_red = delay_red

        self.output = self.addOutPort("Output")

    def intTransition(self):
        return {"Green": "Yellow",
                "Yellow": "Red",
                "Red": "Green"}[self.state]

    def outputFnc(self):
        if self.state == "Green":
            return {self.output: ["show_yellow"]}
        elif self.state == "Yellow":
            return {self.output: ["show_red"]}
        elif self.state == "Red":
            return {self.output: ["show_green"]}
        return {}

    def timeAdvance(self):
        return {"Green": self.delay_green,
                "Yellow": self.delay_yellow,
                "Red": self.delay_red}[self.state]
```

Figure 9: Implementation of an autonomous traffic light with output.

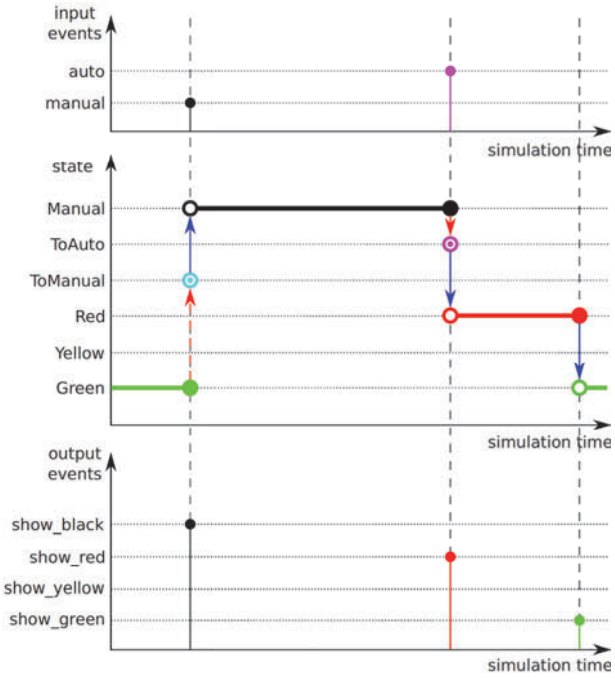


Figure 10: Trace of the autonomous traffic light.

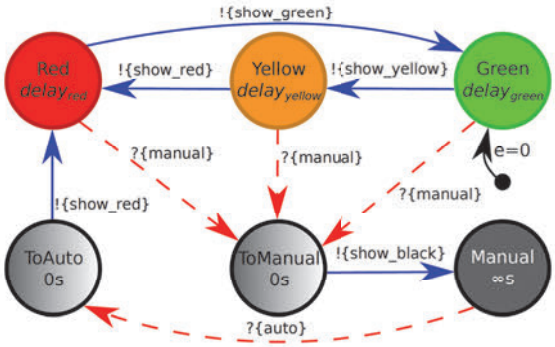


Figure 11: Model generating the trace in Figure 10.

The pseudo-code is slightly altered to include output generation, as shown in Algorithm 2. Recall that output is generated before the internal transition is executed, so the method invocation happens right before the transition.

Algorithm 2 DEVS simulation pseudo-code for autonomous models with output.

```
time ← 0
current_state ← initial_state
last_time ← -initial_elapsed
while not termination_condition() do
  time ← last_time + ta(current_state)
  output(λ(current_state))
  current_state ← δ_int(current_state)
  last_time ← time
end while
```

3.3 Interruptable Model

Our current traffic light specification is still completely autonomous. The police might want to temporarily shut down the traffic lights when they are manually directing traffic. To allow for this, our traffic light must process external incoming events: the event from the policeman to shutdown and to start up again. Figure 10 shows the trace we wish to obtain. A model generating this trace is shown in Figure 11, using a question mark to indicate event reception. Similar to event generation, events are received in the form of a bag of events. As it is a bag, the order of events is non-deterministic, multiple events can be received simultaneously, and events can arrive multiple times at the same time.

We once more require two additional elements in the DEVS specification.

1. **Input Set** ($X : \times_{i=1}^{m} X_i$)
   Similar to the output set, we need to define the events we expect to receive. This is again a definition of the interface, such that others know which events are understood by this model.
2. **External Transition** ($\delta_{ext} : Q \times X^b \to S$)
   Similar to the internal transition function, the external transition function is allowed to define the new state as well. First and foremost, the external transition function is still dependent on the current state, just like the internal transition function. The external transition function has access to two more values: the elapsed time and the bag of input events. The *elapsed time* indicates how long it has been for this atomic model since the last transition (either internal or external), similarly to how it was defined in q_{init}.

Note the special structure around the MANUAL state. The output function was defined as being called before an internal transition only. As such, external transitions do not raise events. Therefore, additional transient states TOMANUAL and TOAUTO need to be added, with a time advance of zero, on which internal transition the output events will be raised.

Finally, a corner case still needs to be covered: it is possible for the atomic model to receive an input event at exactly the same time as an internal transition was scheduled. The question is then which transition to take: the internal or external transition?

Parallel DEVS solves this by introducing the **confluent transition function** (δ_{conf}): when both the internal and external transition function are to be executed, the confluent transition function is executed instead. Often, the δ_{conf} is defined as first invoking the internal transition, followed by the external transition. Its definition is a mix of both: $\delta_{conf} : S \times X^b \to S$.

We describe this model as a 9-tuple in Figure 12, and as PythonPDEVS code in Figure 13.

Algorithm 3 presents the complete semantics of an atomic model in pseudo-code. Similar to before, we still have the same simulation loop, but now we can be interrupted externally. At each time step, we need to determine whether an external interrupt is scheduled before the internal interrupt. If that is not the case, we simply act as before, by executing the internal transition. If there is an external event that must go first, we execute the external transition.

4 COUPLED DEVS MODELS

While our traffic light example is able to receive and output events, there are no other atomic models to communicate with. To combine and couple atomic models together, we introduce coupled models. We do this in the context of our previous traffic light, which is connected to a policeman. The details of the traffic light are exactly like before. the details of the policeman are irrelevant here, with the exception of its interface: we know that it raises the *take_break* and *go_to_work* events.

4.1 Basic Coupling

Contrary to the atomic models, there is no behaviour whatsoever associated to a coupled model. Behaviour is the responsibility of atomic models, and structure that of coupled models. Therefore, to a coupled model, we only need connection concepts, such as submodels and connections.

We specify a simple coupled model as a 3-tuple $\langle D, \{M_i\}, \{I_i\} \rangle$.

$$M = \langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta \rangle.$$

$$\begin{aligned}
X &= \{auto, manual\} \\
Y &= \{show_green, show_yellow, \\
&\quad show_red, show_black\} \\
S &= \{\textsc{Green}, \textsc{Yellow}, \textsc{Red}, \\
&\quad \textsc{ToManual}, \textsc{ToAuto}, \textsc{Manual}\} \\
q_{init} &= (\textsc{Green}, 0.0) \\
\delta_{int} &= \{\textsc{Green} \to \textsc{Yellow}, \\
&\quad \textsc{Yellow} \to \textsc{Red}, \\
&\quad \textsc{Red} \to \textsc{Green}, \\
&\quad \textsc{ToManual} \to \textsc{Manual}, \\
&\quad \textsc{ToAuto} \to \textsc{Red}\} \\
\delta_{ext} &= \{((_,_), \{manual\}) \to \textsc{ToManual} \\
&\quad ((\textsc{Manual}, _), \{auto\}) \to \textsc{ToAuto}\} \\
\delta_{conf} &= \{(s,x) \to \delta_{ext}((\delta_{int}(s), 0), x)\} \\
\lambda &= \{\textsc{Green} \to \{show_yellow\}, \\
&\quad \textsc{Yellow} \to \{show_red\}, \\
&\quad \textsc{Red} \to \{show_green\}, \\
&\quad \textsc{ToManual} \to \{show_black\}, \\
&\quad \textsc{ToAuto} \to \{show_red\}\} \\
ta &= \{\textsc{Green} \to delay_{green}, \\
&\quad \textsc{Yellow} \to delay_{yellow}, \\
&\quad \textsc{Red} \to delay_{red}, \\
&\quad \textsc{Manual} \to +\infty, \\
&\quad \textsc{ToManual} \to 0, \\
&\quad \textsc{ToAuto} \to 0\}
\end{aligned}$$

Figure 12: Specification of an interruptable traffic light.

```
class TrafficLight(AtomicDEVS):
    def __init__(self, delay_green,
            delay_yellow, delay_red):
        AtomicDEVS.__init__(self, "Light")
        self.state = "Green"
        self.elapsed = 0.0

        self.delay_green = delay_green
        self.delay_yellow = delay_yellow
        self.delay_red = delay_red

        self.interrupt = self.addInPort("Interrupt")
        self.output = self.addOutPort("Output")

    def intTransition(self):
        return {"Green": "Yellow",
                "Yellow": "Red",
                "Red": "Green",
                "ToManual": "Manual",
                "ToAuto": "Red"}[self.state]

    def extTransition(self, event):
        if "manual" in event[self.interrupt]:
            return "ToManual"
        elif "auto" in event[self.interrupt]:
            if self.state == "Manual":
                return "ToAuto"
        return self.state

    def confTransition(self, event):
        self.state = self.intTransition(self)
        self.elapsed = 0.0
        return self.extTransition(self, event)

    def outputFnc(self):
        if self.state == "Green":
            return {self.output: ["show_yellow"]}
        elif self.state == "Yellow":
            return {self.output: ["show_red"]}
        elif self.state == "Red":
            return {self.output: ["show_green"]}
        elif self.state == "ToManual":
            return {self.output: ["show_black"]}
        elif self.state == "ToAuto":
            return {self.output: ["show_red"]}
        return {}

    def timeAdvance(self):
        return {"Green": self.delay_green,
                "Yellow": self.delay_yellow,
                "Red": self.delay_red,
                "Manual": INFINITY,
                "ToManual": 0,
                "ToAuto": 0}[self.state]
```

Figure 13: Implementation of an interruptable traffic light.

Algorithm 3 DEVS simulation pseudo-code for interruptable models.

```
time ← 0
current_state ← initial_state
last_time ← -initial_elapsed
while not termination_condition() do
  next_time ← last_time + ta(current_state)
  if time_next_event < next_time then
    elapsed ← time_next_event - last_time
    current_state ← δ_ext((current_state, elapsed), next_event)
    time ← time_next_event
  else if time_next_event > next_time then
    time ← next_time
    output(λ(current_state))
    current_state ← δ_int(current_state)
  else if time_next_event = next_time then
    time ← next_time
    output(λ(current_state))
    current_state ← δ_conf(current_state, next_event)
  end if
  last_time ← time
end while
```

1. **Model instances** (D)
   The set of model instances defines which models are included within this coupled model.
2. **Model specifications** ($\{M_i\}$)
   For each element defined in D, we include the 9-tuple specifying the atomic model. By definition, a submodel of the coupled DEVS model always needs to be an atomic model. Through closure under coupling, as mentioned later, we show that this can be extended to coupled models as well.

$$\{M_i\} = \{\langle X_i, Y_i, S_i, q_{init,i}, \delta_{int,i}, \delta_{ext,i}, \delta_{conf,i}, \lambda_i, ta_i\rangle | i \in D\}$$

3. **Model influencees** ($\{I_i\}$)
   The next step is the connection of submodels, which is defined through influencee sets. For each atomic model, its influencee set defines the models affected by its output. There are some constraints on couplings, to prevent invalid models:
   - A model should not influence itself. ($\forall i \in D : i \notin I_i$)
   - Only links within the coupled model are allowed. ($\forall i \in D : I_i \subseteq D$)

4.2 Input and Output

Our coupled model now couples two atomic models together. And while it is now possible for the policeman to pass the event to the traffic light, we again lost the ability to send out the state of the traffic light to an entity external to this coupled model. We augment the coupled model with input and output events, which serve as the interface to the coupled model. This adds the components X_{self} and Y_{self} to the tuple, respectively the set of input and output events, resulting in a 5-tuple $\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}\rangle$.

The constraints on the couplings are relaxed to accomodate for this change: a model can be influenced by the input of the coupled model, and likewise the models can also influence the output events of the coupled model. The previously defined constraints are relaxed to allow for $self$, the coupled model itself:

```
def z_function(evt_in):
    if evt_in == "take_break":
        return "auto"
    elif evt_in == "go_to_work":
        return "manual"

class TrafficLightSystem(CoupledDEVS):
    def __init__(self):
        CoupledDEVS.__init__(self, "TrafficLightSystem")
        self.light = self.addSubModel(TrafficLight(57, 3, 60))
        self.police = self.addSubModel(PoliceMan())

        self.connectPorts(self.police.output, self.light.interrupt, z_function)
```

Figure 14: Implementation of a Coupled DEVS model.

- A model should not influence itself. ($\forall i \in D \cup \{self\} : i \notin I_i$)
- Only links within the coupled model are allowed. ($\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}$)

4.3 Translation Functions

Recall that our traffic light listened to the *manual* and *auto* events. The policeman, however, outputs *go_to_work* and *take_break* events. This mismatch prevents them from being coupled directly, as $Y_{police} \neq X_{light}$.

To handle this problem, connections are augmented with a **translation function** ($Z_{i,j}$), which specifies a translation of output to input events.

$$\begin{array}{lll} Z_{self,j} & : X_{self} \to X_j & \forall j \in D \\ Z_{i,self} & : Y_i \to Y_{self} & \forall i \in D \\ Z_{i,j} & : Y_i \to X_j & \forall i,j \in D \end{array}$$

These translation functions are defined for each connection, including those between the coupled model's input and output events.

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\}$$

The translation function is implicitly assumed to be the identity function if it is not defined. In case an event needs to traverse multiple connections, all translation functions are chained in order of traversal.

With the addition of this final element, we define a coupled model as a 6-tuple.

$$\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle$$

4.4 PythonPDEVS

Implementing a Coupled DEVS model in PythonPDEVS is simple, as it merely instantiates the Atomic DEVS models and connects them. Of course, it is also possible for the submodels to be coupled DEVS models themselves. An example for the coupling of a policeman with a trafficlight is shown in Figure 14.

4.5 Closure Under Coupling

Similar to atomic models, we need to define the semantics of coupled models. But instead of explaining the semantics from scratch, by defining some pseudo-code, we map coupled models to equivalent atomic

models. Semantics of a coupled model is thus defined in terms of an atomic model. In addition, this flattening removes the constraint of coupled models that their submodels should be atomic models: if a coupled model is a submodel, it can be flattened to an atomic model, thus matching the definition.

In essence, for any coupled model specified as $CM =< X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$, we define an equivalent atomic model specified as $< X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$. A detailed discussion of this is omitted, as it is not essential to model using Parallel DEVS.

4.6 Abstract Simulator

Actually performing closure under coupling at runtime to provide semantics is an inefficient option. Therefore, an equivalent semantic mapping is defined in the form of operational semantics by the abstract simulator (Chow, Zeigler, and Kim 1994). A detailed discussion of the abstract simulator is omitted as well.

5 SUMMARY

In this tutorial, we have considered Atomic DEVS models and Coupled DEVS models through the use of a simple running example: a traffic light. Atomic DEVS models consider discrete event behaviour, and thus map each allowable input sequence of events to a sequence of output events. It was described by the 9-tuple $\langle X, Y, S, q_{init}, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta \rangle$. Coupled DEVS models consider structure and routing of the atomic DEVS models it contains. It was described by the 6-tuple $\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle$. For each aspect of the DEVS specification, we have given justification as to why it is necessary, provided the formal specification, and made the link to an implementation in PythonPDEVS.

ACKNOWLEDGMENTS

This work was partly funded by PhD fellowships from the Research Foundation - Flanders (FWO). This research was partially supported by Flanders Make vzw, the strategic research centre for the manufacturing industry.

REFERENCES

Chow, A. C. H., and B. P. Zeigler. 1994. "Parallel DEVS: a parallel, hierarchical, modular, modeling formalism". In *Proceedings of the 1994 Winter Simulation Multiconference*, pp. 716–722.

Chow, A. C. H., B. P. Zeigler, and D. H. Kim. 1994. "Abstract simulator for the parallel DEVS formalism". In *AI, Simulation, and Planning in High Autonomy Systems*, pp. 157–163.

Van Tendeloo, Y., and H. Vangheluwe. 2014. "The Modular Architecture of the Python(P)DEVS Simulation Kernel". In *Proceedings of the 2014 Spring Simulation Multiconference*, pp. 387–392.

Van Tendeloo, Y., and H. Vangheluwe. 2016a. "An Evaluation of DEVS Simulation Tools". *SIMULATION*.

Van Tendeloo, Y., and H. Vangheluwe. 2016b. "An Overview of PythonPDEVS". In *JDF 2016 – Les Journées DEVS Francophones – Théorie et Applications*, edited by C. W. RED, pp. 59–66.

Zeigler, B. P. 1976. *Theory of Modeling and Simulation*. 1st ed. Wiley Interscience.

Zeigler, B. P. 1984. *Multifacetted Modelling and Discrete Event Simulation*. 1st ed. Academic Press.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. 2nd ed. Academic Press.

Lecture 5. DEVS Extensions

Papers to study for the exam

- B.P. Zeigler. DEVS Today: Recent Advances in Discrete Event-Based Information Technology. Proceedings of the 11TH IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS'03). IEEE, 2003.
  Retrieved from: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1240652

- B.P. Zeigler. Embedding DEV&DESS in DEVS. DEVS Integrative M&S Symposium (DEVS 2006). Huntsville, Alabama, USA, April 2-6, 2006
  Retrieved from: http://acims.asu.edu/wp-content/uploads/2012/02/Embedding-DEVDESS-in-DEVS.pdf via official ACIMS web page at https://acims.asu.edu/journal-conference-and-workshop-articles/

DEVS Today: Recent Advances in Discrete Event-Based Information Technology

Bernard P. Zeigler
Arizona Center for Integrative Modeling & Simulation
University of Arizona
Tucson, AZ, 85721-0104, USA
http://www.acims.arizona.edu
zeigler@ece.arizona.edu

Abstract

We review the DEVS modeling and simulation framework Its fundamental concepts are discussed from the standpoint of discrete event information processing with an example drawn from recent experiments on infant cognition. We also cover the DEVS formalism's atomic and coupled models and its hierarchical, modular composition approach. Some industrial applications of the methodology are discussed in depth to highlight the formalism's utility in the development of commercial and defense information technologies

1. Introduction

Originally introduced as a formalism for discrete event modeling and simulation, the DEVS (Discrete Event System Specification) methodology has become an engine for advances within the wider area of information technology. A variety of systems theory and artificial-intelligence applications employing DEVS-based concepts were compiled by Sarjoughian and Cellier [1]. However, the wealth of research continues to expand and a second re-examination of recent advances seems in order. Particularly, this paper reviews advances from the perspective of success stories in real-world applications. It also lays the basis for claims that discrete event information processing is an important paradigm for both understanding human intelligence and developing new information technologies that can match or supercede it.

2. Primacy of discrete event perception

Our ability to process information using discrete events is a distinctly human trait and is evident from an early age – according to a very interesting new book by David and Ann Premack [2]. Discrete event processing is characterized by the ability to perceive the flow of sensory stimuli as discrete events, and to attend to both sequencing and timing of such events [6,7]. As illustrated in Figure 1, a spatial metaphor is used in which time is perceived as a continuous horizontal line flowing from left to right and events are stored as images located on the line. The *sequencing* of events is represented by locating them on the line, with earlier events to the left of recent events corresponding to how we read (e.g., in the order of occurrence of the events e1,e2,e3,e4, we have e1 as the earliest and e4 as the most recent).[1] Instead of storing absolute values of time, only interval durations are coded giving a sense of the elapsed time that processes consume. In other words, our remembered image of mowing the lawn might last a half-second while an image of a ten minute shower might last for a few milliseconds. Before discussing discrete event information processing further, we stop to review the basic DEVS formalism within a larger framework for modeling and simulation.

3. Framework for modeling and simulation

The Discrete Event System Specification (DEVS) formalism provides a means of specifying a mathematical object called a system [3,4,5]. Basically, a system has a time base, inputs, states, outputs, and functions for determining next states and outputs given current states and inputs. Discrete event systems represent certain constellations of such parameters just as continuous systems do. For example, the inputs in discrete event systems occur at arbitrarily spaced moments, while those in continuous systems are piecewise continuous functions of time. The insight provided by the DEVS formalism is in the simple way that it characterizes how discrete event simulation languages specify discrete event system parameters.

[1] Apparently, readers of Mandarin use a vertical line and use a top to bottom order of events [2].

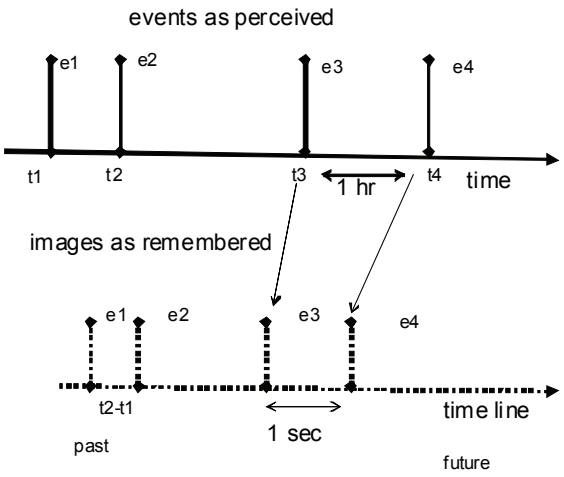


Figure 1 Representing events by their sequencing and inter-event durations.

Having this abstraction, it is possible to design new simulation environments with sound semantics that have a number of benefits to be described. Indeed, the DEVJAVA[2] environment [8] is an implementation of the DEVS formalism in Java which enables the modeler to specify models directly in its terms and derive the associated benefits.

3.1 Brief review of modeling and simulation concepts

Figure 2 depicts the conceptual framework underlying the DEVS formalism [3,4,5]. The modeling and simulation enterprise concerns three basic objects:

- the *real system,* in existence or proposed, which is regarded as fundamentally a source of data, or input/output pairs of time segments as described below.

- *model*, which (informally) is a set of instructions for generating data comparable to that observable in the real system[3]. The *structure* of the model is its set of instructions. The *behavior* of the model is the set of all possible input/output data that can be generated by faithfully executing the model instructions.

- *simulator*, which exercises the model's instructions to actually generate its behavior.

- *experimental frame*, is a specification of the conditions under which the system is observed or experimented with. An experimental frame is the operational formulation of the objectives that motivate an M&S project. A frame is realized as a system that interacts with the system of interest to obtain the data of interest under specified conditions.

.

The basic objects are related by two relations:

- *modeling relation* linking real system and model, defines how well the model represents the system or entity being modeled. In general terms a model can be considered valid if the data generated by the model agrees with the data produced by the real system in an experimental frame of interest.

- *simulation relation,* linking model and simulator, represents how faithfully the simulator is able to carry out the instructions of the model.

The basic items of data produced by a system or model are *time segment*s. These time segments are mappings from intervals defined over a specified time base to values in the ranges of one or more variables. The variables can either be observed or measured. Examples of concurrent data segments are in Figure 3.

The structure of a model may be expressed in a mathematical language called a *formalism*. The discrete event formalism focuses on the changes of variable values and generates time segments that are piecewise constant. Thus an event is a change in a variable value which occurs instantaneously. In essence the formalism defines how to generate new values for variables and the times the new values should take effect. An important aspect of the formalism is that the time intervals between event occurrences are variable

[2] DEVSJAVA is the intellectual property of the Regents of the State of Arizona. Since DEVS is a public formalism, many other implementations of the concepts exist, and anyone is free to develop their own based on literature in the public domain.

[3] See [3,5] for formal characterization of the objects and relations.

(in contrast to discrete time where the time step is generally a constant number).

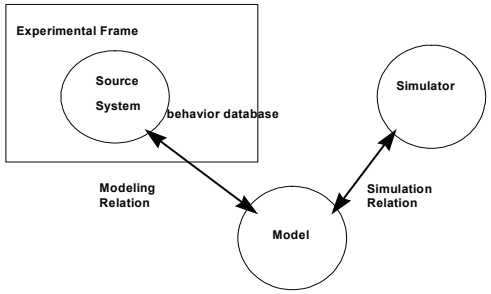


Figure 2 Basic Entities and Relations in Modeling and Simulation.

3.2 Basic models

In the DEVS formalism, one must specify 1) basic models from which larger ones are built, and 2) how these models are connected together in hierarchical fashion.

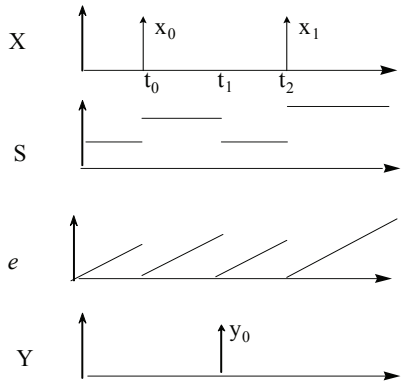


Figure 3 Four concurrent segments are shown for input X, state S, elapsed time e, and output Y. The X and Y segments are discrete event segments.

To specify modular discrete event models requires that we adopt a different view than that fostered by traditional simulation languages. As with modular specification in general, we must view a model as possessing input and output ports through which all interaction with the environment is mediated. In the discrete event case, events determine values appearing on such ports. More specifically, when external events, arising outside the model, are received on its input ports, the model description must determine how it responds to them. Also, internal events arising within the model, change its state, as well as manifesting themselves as events on the output ports to be transmitted to other model components.

A *basic model* contains the following information:

- the set of input ports through which external events are received,
- the set of output ports through which external events are sent,
- the set of state variables and parameters: two state variables are usually present, "phase" and "sigma" (in the absence of external events the system stays in the current "phase" for the time given by "sigma"),
- the time advance function which controls the timing of internal transitions – when the "sigma" state variable is present, this function just returns the value of "sigma",
- the internal transition function which specifies to which next state the system will transit after the time given by the time advance function has elapsed,
- the external transition function which specifies how the system changes state when an input is received – the effect is to place the system in a new "phase" and "sigma" thus scheduling it for a next internal transtion; the next state is computed on the basis of the present state, the input port and value of the external event, and the time that has elapsed in the current state,
- the confluent transition function which is applied when an input is received at the same time that an internal transition is to occur – the default definition simply applies the internal transition function before applying the external transition function to the resulting sate, and

- the output function which generates an external output just before an internal transition takes place.

The DEVS formalism uses set theory notation to concisely and rigorously state the above information. Once stated, the formalism can be used for both mathematical analysis as well as the basis for effective modeling and simulation methodology and technology development. Ashby's classic text, "Introduction to Cybernetics" provides a very readable introduction to the mathematical set theory and the systems concepts underlying the DEVS formalism. Fortunately, this historic ground-breaking work is available from http://pespmc1.vub.ac.be/books/IntroCyb.pdf. for download at no charge.

This is concisely as follows:

A *Discrete Event System Specification* (*DEVS*) is a structure

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$$

where

X is the set of input values

S is a set of states,

Y is the set of output values

δ_{int}: $S \rightarrow S$ is the *internal transition* function

δ_{ext}: $Q \times X^b \rightarrow S$

is the *external transition* function, where

$Q = \{(s,e) \mid s \in S, 0 \le e \le ta(s)\}$ is the *total state* set

e is the *time elapsed* since last transition

X^b denotes the collection of bags over X (sets in which some elements may occur more than once).

δ_{con}: $Q \times X^b \rightarrow S$

is the *confuentl transition* function,

λ: $S \rightarrow Y^b$ is the output function

ta: $S \rightarrow \mathbf{R}^+_{0,\infty}$ is the *time advance* function

The interpretation of these elements is illustrated in Figure 4. At any time the system is in some state, s. If no external event occurs the system will stay in state s for time $ta(s)$. Notice that $ta(s)$ could be a real number as one would expect. But it can also take on the values 0 and ∞. In the first case, the stay in state s is so short that no external events can intervene – we say that s is a *transitory* state. In the second case, the system will stay in s forever unless an external event interrupts its slumber. We say that s is *a passive* state in this case. When the resting time expires, i.e., when the elapsed time, $e = ta(s)$, the system outputs the value, $\lambda(s)$, and changes to state $\delta_{int}(s)$. Note output is only possible just before internal transitions.

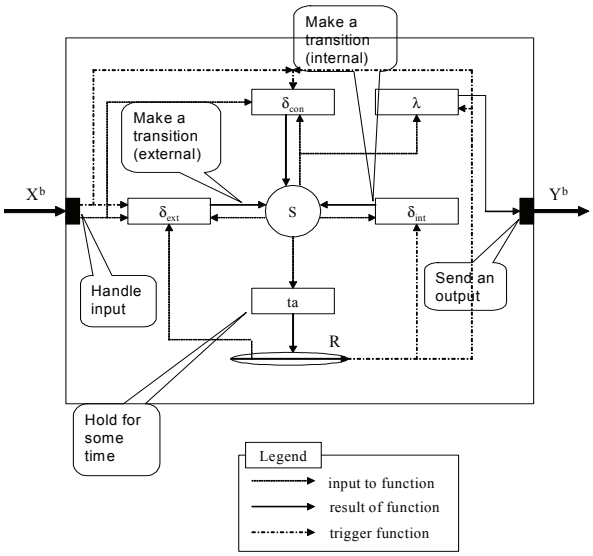


Figure 4 Portraying the operation of the DEVS formalism.

If an external event x in X^b occurs before this expiration time, i.e., when the system is in total state (s, e) with $e \le ta(s)$, the system changes to state $\delta_{ext}(s,e,x)$. Thus the internal transition function dictates the system's new state when no events have occurred since the last transition. While the external transition function dictates the system's new state when an external event occurs – this state is determined by the input, x, the current state, s, and how long the system has been in this state, e, when the external event occurred. In both cases, the system is then is some new state s' with some new resting time, $ta(s')$ and the same story continues.

Note that an external event x in X^b is a bag of elements of X. This means that one or more elements can appear on input ports at the same time. This capability is needed since DEVS allows many components to generate output and send these to input ports all at the same instant of time.

4. DEVS models of mental simulation

An experimental result that supports the formulation of discrete event perception portrayed in Figure 1 is described as follows [2, page 59]: "After watching an experimenter hide two cookies (one frosted, the other plain) and then reappear eating a cookie, children consistently go to the location of the cookie that is different from the one that the experimenter is eating….When however, the cookies are wrapped in layers of tissue paper before being hidden, children no longer head for the location containing the cookie different from that being eaten by the experimenter. In notable contrast, chimpanzees fail at this problem. They continue to head for the location holding the cookie different from the one that the experimenter is eating."

The events in the experiments are illustrated attached to the time lines of Figure 5. In the first experiment, Exp 1: both children and chimpanzees assume that the reappearing cookie is one of the ones hidden Therefore they head for the other one. In the second experiment, we assume that the experimenter reappears a short time (say, 1 second) after hiding the second cookie. The results indicated that children of age four are able to recall the time (duration) it takes to unwrap cookies. Comparing it to the time available to the experimenter to do so, they conclude that the reappearing cookie could not be one of the two that were hidden. Therefore, they are indifferent to which location has a cookie. Chimpanzees apparently don't pay attention to the wrapped cookies, or if they do, can't bring information on wrapping duration to bear. Interestingly, this is a case where having more reasoning capacity, makes the outcome more uncertain not less.

Figure 6 illustrates a discrete event simulation that justifies the children's reasoning. Children are able to infer that a hidden cookie would have to be unwrapped in order to be eaten. However, given the experimenter's early reappearance this would take too much time (duration) to be possible.

The logic of a DEVS model that can be employed in the simulation is illustrated in Figure 6. The "cookie unwrapped" state has a time advance of 8 (which means that it is passive, i.e., will not undergo an internal transition). When an external event "start wrapping" occurs, the model transitions to state "wrapping" (this is part of the external transition function). The time consumed in wrapping, "time to wrap," is given by the time advance function. Once this time has elapsed, the internal transition functions dictates moving to the passive state "cookie wrapped." Notice that an external input, "start eating cookie" can only cause a transition to "eating cookie" in the "cookie unwrapped" state. So a planning simulation is set up addressing the question: how can we get from the "cookie wrapped" state to the "cookie unwrapped" state where the eating can be started. In general, such a

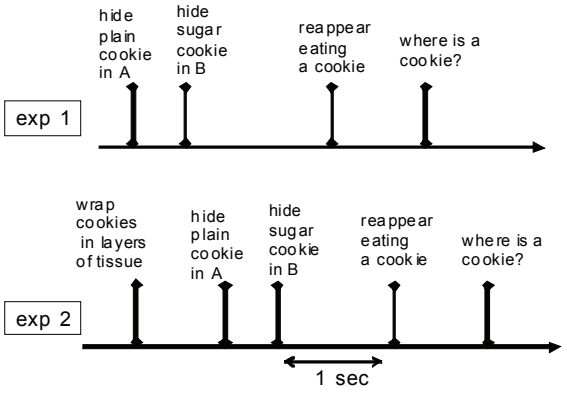


Figure 5. Time lines for experiments on discrete event perception

planning simulation will try to find a path in the state graph from the initial sate to the goal state (if there are more than one such path, an optimality criterion would be applied.) The path would reveal the sequence of external inputs and their timing to apply in order to bring the manipulated object from its current state to the goal state. In this case, the solution is apparent. As illustrated in red, a "start unwrapping" input will start the unwrapping process and one has to wait for the "time to unwrap" to elapse before the cookie will be in the "unwrapped" state ready for consumption.

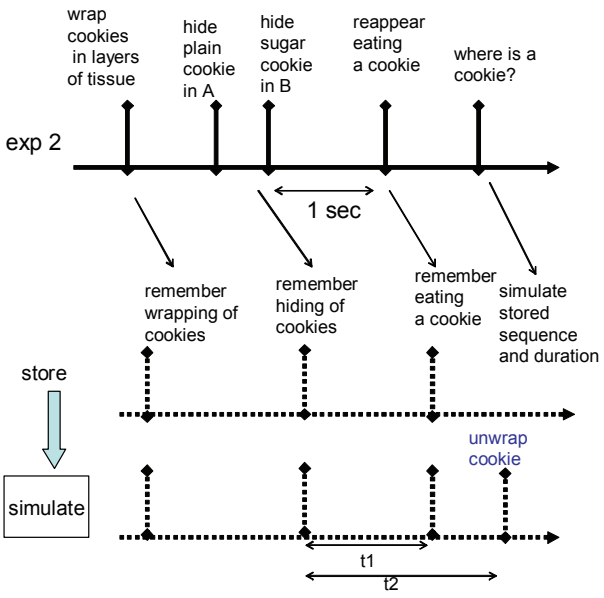


Figure 6. Illustrating how simulation is used to support temporal reasoning

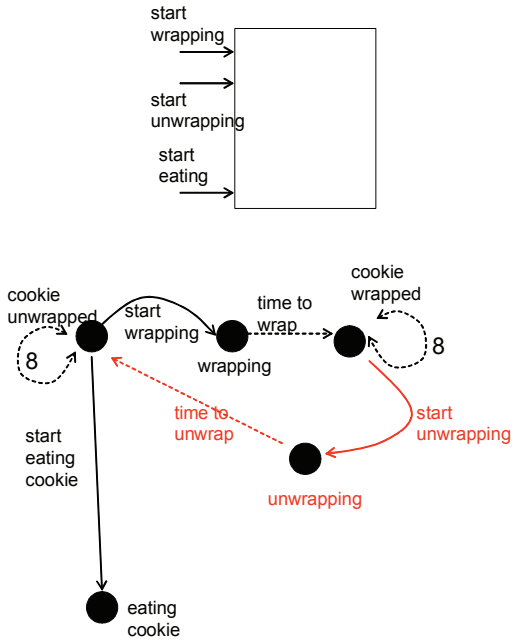


Figure 6. A DEVS model that illustrates how children reason about the wrapped cookie.

Humans and chimpanzees are able to store sequences of events and inter-event durations. Both primate species can compare stored event traces with a running event stream and with spatial analogs such as a sequence of toys laid out in some order. However, only humans can simulate stored event traces and reason with them. In this case, they can infer a necessary action, estimate its duration from past experience, compare event traces and infer their inconsistency based on incompatible durations (Figure 6). Humans, but not chimpanzees, can virtually reconstruct the past and can imagine the future. Therefore they can plan—which means they can try out alternative courses of action against a model of reality and choose what seems to be the best action to take.

5. DEVS perception-based control: an industrial success story

While our temporal information processing seems strongly founded on discrete-event principles, humans are still quite limited when it comes to the perception and control of technological phenomena that were not part of our evolutionary experience. So the question arises whether the DEVS formalism can help in developing intelligent systems that employ the discrete-event processing paradigm to control complex dynamical industrial processes.

The answer is provided in the affirmative, by Sachem, an extensive large-scale real-time system that uses DEVS as an overall framework for knowledge-based control of steel production [9]. Sachem was initially developed to monitor and to diagnose the blast furnaces of Usinor, a company in the Arcelor group, the world's largest producer of steel products. The original objective of Usinor was to save up to 1 euro per ton of pig iron [9]. Currently, Sachem saves approx. 1.6 euro per ton. and with six blast furnaces equipped with a Sachem system, Arcelor saves millions of euros annually. One the main hurdles overcome by such a system is the acquisition and the representation of the perception knowledge of the experts who monitor and diagnose the blast furnace. Such perception knowledge is sub-symbolic, meaning that experts are unable to use words to describe their own cognitive processes. Furthermore, such knowledge concerns temporal and spatial evolution of processes and humans have difficulty describing such dynamic processes with the needed fidelity. It is therefore very difficult to model

the reasoning of a control process expert.[4] Conventional artificial intelligence approaches are inadequate since they do not recognize the importance of temporal perception and discrete event processing in dealing with complex dynamical systems. The lack of mathematical models[5] of the blast furnace dynamic also inhibits subjecting it to conventional theories of process control.

The basic hypothesis underlying Sachem's design is that experts reason mainly directly on state transitions rather than on the state of the process. Because state transitions can be represented by discrete events [6,11], the knowledge to control the blast furnace can be represented with propositions about discrete events and temporal windows that constrain the time of the events

As illustrated in Figure 7, the basis of Sachem's DEVS-based perception is the recognition or successively more abstract classes of discrete events, called signal events, signal phenomena and process phenomena. The discrete event paradigm allows the design of this multi-layered self-similar discrete event abstraction process. At the lowest level, quantization [5] is employed to generate discrete event segments from continuous trajectories coming from thousands of spatially arrayed temperature, pressure and other sensors. Sachem continuously analyzes the flow of discrete events through the three levels of abstraction. The transition from one level of abstraction to the next is based on a similar event recognition process called signature detection. Since the same discrete event processing underlies each level of abstraction, a unified

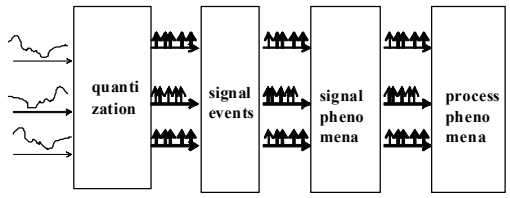


Figure 7. The continuous flow of Sachem's DEVS-based perception events

architecture can be developed which supplies essentially the same services for signature detection and reasoning at the different abstraction levels. The perception function of Sachem outputs its process phenomena events to a diagnosis function that analyzes the continuous flow of phenomena in order to recognize dangerous behaviors, to identify the hypothetical causes that explain the recognized behavior, and to issue warnings and advice to human operators on undesirable states that the process that could potentially reach in the future.

One of the advantages of the discrete event paradigm is *compactness*. For example, one year of blast furnace generates approximately 30,000 process phenomena. This results from the discrete event abstraction of a database containing the blast furnace raw data that is 50,000 this size. As a consequence, for the first time in the history of blast furnace study, it is possible to analyze the behavior of blast furnaces over years at such a high level of detail.

The success that Sachem has achieved is the result of a sustained effort with substantial investments in human and software-intensive system resources. A conceptual model of the knowledge specifies the entire system. This model contains 25,000 objects for 33 goals, 27 tasks, 75 inference structures, 3200 concepts and 2000 relations. The Sachem knowledge bases contain more than 1060 classes of objects, 1100 first order logic rules and 140 event report types. The total software volume represents approximately 400,000 lines of code. This represents 14 man-years of work for a team of 6 knowledge engineers and 12 experts during a period of 3 years.

Sachem developers are currently researching an approach to assist the expert in discovering signatures at the process phenomena abstraction level. Their approach is based on a Markov process representation of a discrete event flow. Integrated in a Java environment called the "ELP Lab", a set of tools has been developed to compute correlations between the beginnings of process phenomena. The ELP language is a high-level knowledge representation language of signatures [10]. The ELP signatures are operationalized using the DEVS formalism: the event reports are translated into DEVS models that a DEVS simulator uses in order to recognize the discrete events sequences that satisfy the signatures. The goal is to develop methods to infer an ELP signature from a set of significant sequences.

[4] Another important characteristic is the very long period (5-10 years) that is needed to learn to control the production of blast furnace

[5] One of the main characteristics of a blast furnace is the lack of a mathematical model of its internal dynamics due to the complexity of the physical and chemical interactions among gas, solids and liquids and the extreme conditions that prevail inside the blast furnace: very high temperatures, acidity, dust, high pressure, etc.

6. Coupled and Hierarchical DEVS models

Basic models may be coupled in the DEVS formalism to form a *coupled model*. A coupled model tells how to couple (connect) several component models together to form a new model (Figure 8). This latter model can itself be employed as a component in a larger coupled model, thus giving rise to hierarchical construction (Figure 9). A coupled model contains the following information:

- the set of components
- the set of input ports through which external events are received – ports are important for routing as specified by the coupling specification (below).
- the set of output ports through which external events are sent
- the coupling specification consisting of:

➢ the external input coupling which connects the input ports of the coupled to model to one or more of the input ports of the components – this directs inputs received by the coupled model to designated component models,

➢ the external output coupling which connects output ports of components to output ports of the coupled model – thus when an output is generated by a component it may be sent to a designated output port of the coupled model and thus be transmitted externally,

➢ the internal coupling which connects output ports of components to input ports of other components- when an input is generated by a component it may be sent to the input ports of designated components (in addition to being sent to an output port of the coupled model).

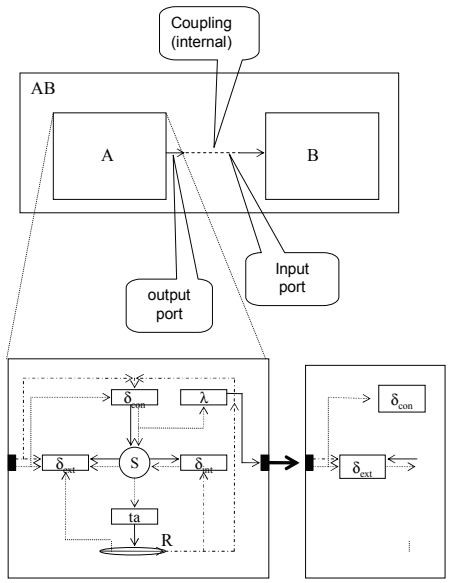


Figure 8. Coupled DEVS and the mechanism implementing the coupling specification

Figure 8 illustrates how internal coupling directs the flow of outputs to inputs in an illustrative coupled model, AB. When outputs are generated on an output port of a component, A, they are sent at the same time instant to the input port of component, B due to a coupling of the respective output and input ports defined from A to B.

6.1 Hierarchical Models

A coupled model can be expressed as an equivalent basic model in the DEVS formalism. Such a basic model can itself be employed in a larger coupled model. This shows that the formalism is closed under coupling as required for hierarchical model construction. Expressing a coupled model as an equivalent basic model captures the means by which the components interact to yield the overall behavior.

Closure under coupling and hierarchical construction form the basis of the DEVS composition framework. The framework deals with components which are modular, i.e., are self-contained and can stand alone or be incorporated, as components into a larger system. There are two types of components: atomic models and coupled models. Atomic models are expressed directly as basic models in the DEVS formalism. Coupled models are specified by providing the set of existing components and the internal and external coupling

specifications. We note that due to closure under coupling, coupled models have the same input and output port interfaces as atomic models and can be treated in the same manner as far as their external relations to other components. In particular, coupled models can become components in larger systems, just as atomic modules can, and this leads to hierarchical decomposition and construction.

Figure 9 a) then illustrates how by adding in a coupling specification to a set of models, we get a coupled model. By using this model as a component in a larger system with new components, and adding coupling information, we get a hierarchical coupled model. As illustrated in Figure 9 b), the models in a hierarchical coupled model are represented in a hierarchical tree, where the leaves represent *Atomic Models* and the interior nodes represented *Coupled Models*.

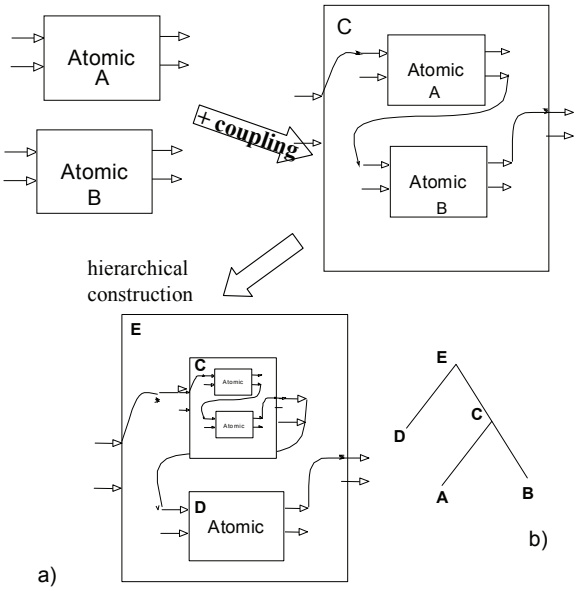


Figure 9. Coupled Modules formed via coupling and their use as components

7. DEVS-based system-of-systems simulation: a re-usability success story

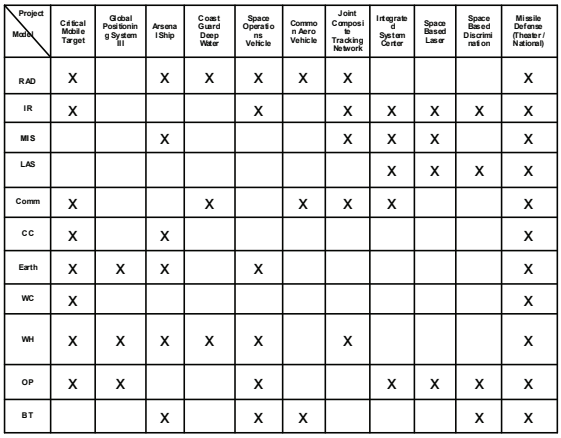
It is important to note that DEVS models, whether atomic or coupled can stand alone and go into a repository for reuse by the developers or others. In principle, the internals of any such model can be hidden (e.g., in relation to proprietary rights) – only the behavior as seen through the input/output ports needs to be communicated in a clear enough manner for others to use the component. This can foster a high degree of reuse and sharing among a growing DEVS community.

The Advanced Simulation Center (ASC), under Steve Hall, has a long history of M&S projects with significant legacy models and simulators[12,13,14]. It is oriented toward increasing emphasis on collaboration between different parts of Lockheed Martin and with other teammates, responding to the increasing interest in performing effectiveness analysis as well as performance analysis, and increasing focus on system-of-systems solutions. For nearly a decade, ASC has adopted a DEVS-based "managed model approach," which starts from an architectural framework for constructing simulation models that supports continued cumulative development and reuse The architectural framework for organizing models is supplied by the SES (System Entity Structure [4,5]), a generalization of the hierarchical tree structure in Figure 9b). The SES specifies a well-defined structure for model reuse, maintains kind-of, part-of, multiplicity relationships, and constraints on model compatibility. Models developed are highly parameterized and are network reusable – its system-of-systems simulations federate with other joint simulation systems using HLA (High Level Architecture)[14]. ASC has a collaborative development environment, strong support for third-party model 'wrapping', and employs object-oriented development of models in DEVS/C++, an environment co-developed by ASC and the University of Arizona. Under its managed modeling paradigm, ASC has made concerted efforts to develop its models with a view to their potential reuse as components for subsequent projects. The following table indicates the success that this approach has achieved for reusability. It shows the original project in which a model was developed and the new projects for which it was modified for re-use.

A more dramatic depiction of re-usability success is illustrated by the incidence matrix in Figure 10. The matrix shows that most models have been reused

Table 1: Listing DEVS models developed by ASC over an 8 year period, the original project of development and some of the subsequent projects in which the model was reused as a component.

Acronym	Model Component	Original Project Development	Subsequent Project Re-use and modification
RAD	Radar Model	Sea Shadow	Arsenal, JCTN,
IR	IR Sensor Model	SBL	Space Discrimination
MIS	Missile Model	Arsenal	Missile Defense
LAS	Laser Model	SBL	Missile Defense, ABL
Comm.	Communications Model	MUSIC	JCTN, CMT
CC	Command Control Model	Arsenal	All projects
EARTH	Earth & Terrain Model	IRAD	CMT
WC	Weather and Cloud Model	CMT	ABL
WH	Waypoint & Heading	IRAD	CMT
OP	Orbital Propagate	SBL	Space Discrimation, CMT, NMD, MSP
BT	Ballistic Trajectory Model	Arsenal	SBL, NMD



Project / Model	Critical Mobile Target	Global Positioning System III	Arsenal Ship	Coast Guard Deep Water	Space Operations Vehicle	Common Aero Vehicle	Joint Composite Tracking Network	Integrated System Center	Space Based Laser	Space Based Discrimination	Missile Defense (Theater / National)
RAD	X		X	X	X	X	X				X
IR	X				X		X	X	X	X	X
MIS			X				X	X	X		X
LAS								X	X	X	X
Comm	X			X		X	X	X			X
CC	X		X								X
Earth	X	X	X		X						X
WC	X										X
WH	X	X	X	X	X		X				X
OP	X	X			X			X	X	X	X
BT			X		X	X				X	X

Figure 10. An incidence matrix showing the reuse of models as components in projects – an X indicates the model (row) was reused in a project (column).

several times after their initial development and that the modifications required (as shown in Table 1) are relatively rare. This ability to re-use models as components reliably without much further modification has provided ASC with increased capability to rapidly respond to new projects and their requirements. In fact, ASC claims to be leaving other competitive system-of-systems environments that employ monolithic (non-hierarchical, modular) development further and further behind as time goes by.

7.1 How a DEVS simulator works

The DEVS formalism has an associated well-defined concept of simulation engine to execute models and generate their behavior. A coupled model in DEVS consists of component models and a coupling specification that tells how outputs of components are routed as inputs to other components. The simulator for a coupled model is illustrated in Figure 11. It consists of a coordinator that has access to the coupled model specification as well as simulators for each of the model components. The coordinator performs the time management and controls the message exchange among simulators in accordance with the coupled model specification. The simulators respond to commands and queries from the coordinator by referencing the specifications of their assigned models. The simulation protocol works for any model expressed in the DEVS formalism. It is an algorithm that has different realizations that allow models to be executed on a single host and on networked computers where the coordinator and component simulators are distributed among hosts. By replacing the simulators and their models at the leaves of Figure 11 by coordinators and their coupled models, we get a picture of how the DEVS protocol works for hierarchical coupled models.

Recently, ASC re-implemented the underlying simulation engine so as to achieve significantly reduced execution run times [13]. They could do this without disrupting the models due to the strict modular separation of model and simulator, afforded by the M&S framework within which DEVS resides.

We'll use the new ASC DEVS/C++ implementation to illustrate how an implementation of the DEVS simulation protocol works.

In DEVS/C++ Atomic models have the following methods to represent the DEVS formalism's functions:

- *delta-internal*: the internal transition function
- *delta-external*: the external transition function
- *delta-confluent*: the confluent transition function
- *output*: the output function
- *time advance:* the time advance function

Recall that the models in a hierarchical compostion are represented in a hierarchical tree (Figure 9). The role of the hierarchical tree and the couplings between pairs of models in the tree can be described as follows:

The models have *inports* through which input is provided. They also have *outports* that hold the aggregated output of the model, which will be drained when the output function of the model is exercised. A model is said to become *imminent* when a certain sleep has been completed: this results in the execution of the delta-internal function. The sleep is determined dynamically by the model through the time advance function. Coupled models contain other models (coupled or atomic) and this defines the natural parent-child relationship in the tree among the models. Atomic models, as their names imply, do not contain other models. An outport of a model can be connected to any subset of the inports of its siblings in the model tree, or to an outport of its parent coupled model. An inport of a model can also receive a connection from an inport of its parent.

The DEVS/C++ simulator maintains adjacency lists for ports and parent model pointers. In order to propagate messages, it precomputes the mail destinations for each port, dynamically and selectively updating these precomputed destination and origin sets when links, ports, or models are added to or deleted from the simulation. For example, when an outport is connected, it first calculates all the reachable inports on atomic models (the leafset) and on coupled models (the nonleafset). Likewise, when an inport is connected, similar reverse reachable sets are computed. The advantage of this obviously lies in the fact that these reachability calculations are done only when models (or ports or links) are added or deleted dynamically, and only in the part of the model tree that changed, thus adding to the efficiency of the implementation. In the following we distinguish between leafset elements and atomic models they correspond to. The leafset elements contain various pointers and slots such as the next event time and last event time that are not kept in the models themselves (for implementation independence).

A general-purpose templated priority queue for storing leafset elements by various keys, which could be a name, an next event time or other convenient index such as class name. An AVL tree structure is used to order leafset elements by their next event times: if two models have the same next event times (up to a time granule equivalence), tie breaking uses the pointer value. A reverse pointer in the element points back to its location in the priority queue makes searches efficient.

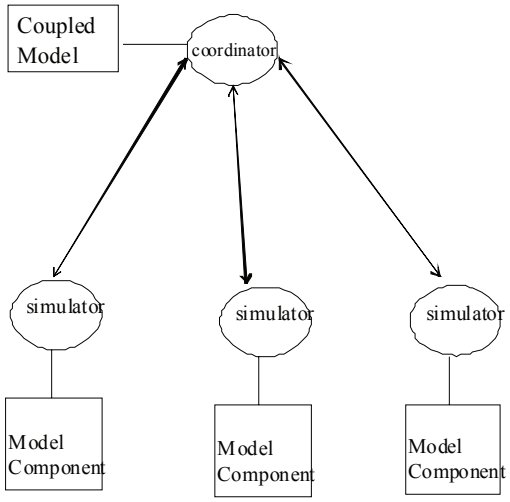


Figure 11. The DEVS Simulation Protocol

The simulation cycle is as follows:

1. Advance the clock to the small next event time in the priority queue .
2. Collect all the leafset elements that are first in the priority into a set I (representing the imminent models, i.e., those that have the smallest next event time, up to the time granule equivalence.)
3. Execute the output functions of the models pointed to by this set, and propagate the output messages directly to the inports in the leafset of the outports that produced the output. These messages are collected into the input message bags for the models in the receptive leafset.
4. Collect into a set M, all the leafset elements representing models that have non-empty bags waiting on their imports.
5. For all the elements in I intersect M (the confluent set), execute the associated model's delta-confluent function, feeding it the input bag and the elapsed time computed from the clock time and the last event time.
6. For the remaining imminents in I, execute the delta-internal function of the associated model.
7. For the remaining models in M, execute the associated model's delta-external function, feeding it the input bag and the elapsed time computed from the clock time and the last event time.

After steps 5), 6) and 7), reinsert the leafset element into the priority queue after updating its next event time obtained from the time advance function of its associated model.

Steps 1) through 7) are repeated until no models are imminent or a termination condition, such as exceeding a time or a number of iterations, is satisfied.

8. Separation of model, simulator and experimental frames: re-use in distributed simulation

We've already seen one advantage of modularity in the separation of model and simulator – the ability to change the underlying simulation engine without any change in the models themselves. There are a number of other advantages, one of which is the support of component re-usability in distributed simulation.

An experimental frame specification consists of 4 major subsections:

- *input stimuli*: specification of the class of admissible input time-dependent stimuli. This is the class of input time segments from which individual samples will be drawn and injected into the model or system under test for particular experiments.
- *control*: specification of the conditions under which an the model or system will be initialized, continued under examination, and terminated.
- *metrics*: specification of the data summarization functions and the measures to be employed to provide quantitative or qualitative measures of the input/output behavior of the model. Examples of such metrics are performance indices, goodness of fit criteria, and error accuracy bounds.
- *analysis*: specification of means by which the results of data collection in the frame will be analyzed to arrived at final conclusions. The data collected in a frame consists of pairs of input/output time segments.

When an experimental frame is realized as a system to interact with the model or system under test the 4 specifications become components of the driving system. For example, the class of input stimuli can be implemented by a generator of output time segments or alternatively, filtered from a larger set of stored or online segments. .

The HLA (High Level Architecture) [14] Federation Development and Execution Process (FEDEP) model proposes six steps in an "iterative waterfall software process" to support development of HLA-compliant simulations. The intent of HLA is to enable the construction of compositions, called federations, of existing simulations, called federates. Although, HLA and the FEDEP process provide a protocol and a methodology for such construction, the primary emphasis is on creating the right interfaces for sharing of data among federates rather than on assuring that the federates interact functionally, as models or as test environments, in the intended manner. A major problem, as suggested earlier, is that neither the HLA nor FEDEP can assure the encapsulated models, simulators, and experimentation constituents of federates match up properly for the intended application.

Figure 12 a) illustrates the inherent difficulties encountered when trying to develop testing infrastructures in the absence of separation of experimentation, model development and simulation execution concerns. Even assuming the most advanced middleware, incompatibilities at any of the three levels make it difficult for the monolithic boxes in which they reside to play meaningfully with each other. For example, the experimental frames might be incompatible because they were developed for different objectives and/or assuming different levels of resolution. Or the simulators might not be compatible because one uses event-based time advance and the other uses fixed time stepping.

In contrast, Figure 12 b) suggests an approach that allows more expeditious plug-and-play. Here, both models and frames are expressed as dynamic elements within a single formalism. The DEVS formalism supports composition of such elements into model-frame pairs and as we have seen, a standard simulation protocol supports efficient execution over existing middleware.

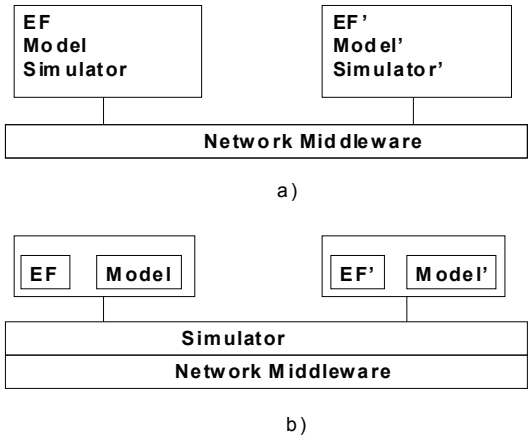


Figure 12. a) The difficulty encountered when trying to compose monolithic simulations. b) the distributed architecture for separating frame, model, and simulator.

In this case, the common use of the formalism in the simulator obviates simulation incompatibility as an issue. Further, by easily tearing the model-frame pairs apart, detecting incompatibilities at the frame and model levels can be reduced to smaller, more manageable consistency tests. For example, we can check frame consistency using the derivability relation [4]. A necessary condition for the models to form a useful composition is that their frames are derivable from (in the sense of [4]) the frame that characterizes the objectives we want to achieve with the proposed federation.

Separation of frames from models and simulators allows development of experimental frame *commodities* in both component and composite units to enable more automated and reusable test generators and the application of more powerful analytic, summarization, and visualization capabilities.

9. Endowing DEVS with more intelligence

Soft computing is a conglomeration of tools for intelligence stemming from approximate reasoning (fuzzy logic), learning (neural network, stochastic learning automaton) and optimization (genetic algorithms, genetic programming). DEVS provides a robust and generic environment for modeling and simulation applications employing single workstation, distributed, and real-time platforms. Fusing these two concepts together yields what is called IDEVS (Intelligent DEVS), an extension that is available within a virtual laboratory, called V-Lab [15]. The V-Lab environment builds on the three layers of Figure 12a). Its 3rd layer models are expressed in DEVSJAVA while an additional top layer supports distributed agent based simulation. Using the DEVS environment, V-Lab defines an appropriate structure in which to organize the elements of DEVS for a distributed agent based modeling and simulation. It separates the main components into different categories and defines the logical structure in which they communicate. It also provides the critical objects needed to control the flow of time and the flow of messages. Developed by the Autonomous Control Engineering (ACE) Center, University of New Mexico, under NASA sponsorship, V-Lab and IDEVS are targeted to development of small robotic rovers that can form cooperative teams in exploration of new terrains such as found on Mars.

V-Lab demonstrates the ability of the DEVS formalism to support the design of soft-computing intelligent behavior mechanisms that control continuous motion of vehicles over challenging terrain. V-Lab also demonstrates the benefits of working within the DEVS framework for hierarchical, modular composition and separation of model and simulation layers to provide reusable components for distributed agent simulation.

10. Moving ahead

DEVS is a framework rather than a particular technique, method or technology. As a consequence, real world applications of advanced DEVS-based information technology systems take considerable time and effort (measured in tens of person-years) to achieve. Indeed, as they develop, intelligent systems, human and artificial, must deal with a plethora of special circumstances and an exponentially growing number logic conditions. What we can hope to achieve with the DEVS formalism is a framework that scales up as such complexity increases. What are some of the ways in which further advances are needed to enable this scalability?

- Enhancing DEVS perception-based processing with the use of the soft-computing elements found within IDEVS as supported by VLab.

- Tapping the biological brain for more of its efficiency features – e.g., its ability to allocate resources (attention) to processing elements that

contribute to current goals or are currently active while not blocking out entirely less active alternatives.

- Incorporating the separation between model, experimental frame and simulators within a DEVS-based standard to enable sharing and re-use of components within broad communities of developers.

- Exploiting model continuity, the ability to transition core model specification through the different design stages of system development. DEVS model continuity allows distributed real-time systems to be designed, analyzed, and tested through DEVS-based modeling and simulation studies and then, by replacing simulators with real-time execution engine equivalents, migrated with minimal additional effort to be executed in the real-time distributed application.

There are some of the directions that are the subject of current DEVS research. Besides the collection of DEVS-related works described in [1], the reader may consult the special Issue of Transaction of Society for Modeling and Simulation International (SCS) on recent advances in DEVS Methodology. The proceedings of the recent Summer Computer Simulation Conference 2003 also contains a substantial collection of DEVS-related research.

11. Acknowledgements

This research has been supported by NSF Grant No. DMI-0122227, "Discrete Event System Specification (DEVS) as a Formal Modeling and Simulation Framework for Scaleable Enterprise Design."

12. References

1. Hessam S. Sarjoughian and Francois E. Cellier (Editors), *Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies*, Springer-Verlag, NY. 2001.

2. David Premack, Ann J. Premack, Original *Intelligence: The Architecture Of The Human Mind*, McGraw Hill, 2003.

3. Zeigler B.P. *Theory of modeling and simulation*. John Wiley Editor, New York, 1976.

4. Zeigler B.P.*Multifaceted Modeling and Discrete Event Simulation*. Academic Press, London UK, 1984.

5. Zeigler, B.P., T.G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*, 2nd Edition, Academic Press. New York, NY, 2000.

6. Zeigler, B.P.," Discrete Event Abstraction: An Emerging Paradigm For Modeling Complex Adaptive Systems", (Ed. L. Booker) *Advances in Adaptive Systems*, Sante Fe Institute/Oxford Press, Oxford (in press)

7. Zeigler, B.P., "The brain-machine disanalogy revisited", *BioSystems,* Vol. 64, pp. 127-140, 2002.

8. Zeigler, B.P. and H.S. Sarjoughian, *Introduction to DEVS Modeling and Simulation with JAVA* www.acims.arizona.edu 2001

9. Le Goc, M. and C. Frydman, "SACHEM, a Real Time Intelligent Diagnosis System based on the Discrete Event Paradigm", submitted to *Transaction of Society for Modeling and Simulation International* (SCS), 2003.

10. Frydman C., M. Le Goc, L. Torres and N. Giambiasi, "Knowledge-Based diagnosis in Sachem using DEVS models", Special Issue of Transaction of Society for Modeling and Simulation International (SCS) on Recent Advances in DEVS Methodology, Tag Gon Kim Ed.,Vol**. 1**8, N°3, 2001, pp147-158.

11. Giambiasi N., Escude B., and S. Ghosh, "Generalized Discrete Event Specifications: G-DEVS Coupled Models", Congress SCI2000, Orlando, USA, July 2000.

12. Steven B. Hall *Report to the National Research Council's Committee On Modeling And Simulation Enhancement For 21st Century Manufacturing And Acquisition*, November, 2000.

13. Steven B. Hall, Shankar M. Venkatesan, Donald B. Wood, "A Faster Implementation of DEVS in the Joint MEASURE Simulation Environment", in *Proc. of Summer Computer Simulation Conference*, Montreal, July 2003.

14. Zeigler, B.P, Steven B. Hall. and H.S. Sarjoughian "Exploiting HLA and DEVS to Promote Interoperability and Reuse in Lockheed's Corporate Environment", *Simulation*, Vol 73**, No.** 4, November 1999, pp. 288-295.

15. Osery, A. L, M. Jamshidi, et al, "V-Lab . – A Virtual Laboratory For Autonomous Agents", *IEEE Trans. SMC,* Vol 32, No. 8, pp791-803, Dec, 2003

Embedding DEV&DESS in DEVS

Bernard P. Zeigler
Arizona Center of Integrative Modeling and Simulation (ACIMS)
ECE Department, University of Arizona
Tucson, AZ 85721
zeigler @ece.arizona.edu

Abstract

DEV&DESS was defined to represent combined continuous and discrete model event simulation models and was shown to have the properties expected from a universal representation of such models in "Theory of Modeling and Simulation." Also in that theory, it was shown that DEVS can provide accurate simulations of DESS models. In this paper, we offer a proof that DEVS on its own is capable of accurately implementing the combination of DEVS and DESS formalisms as formulated in DEV&DESS, thus filling in the missing pieces to complete the picture. We elucidate the characteristic primitives needed to express hybrid or combined continuous/discrete event models and provide a rigorous proof of the ability to express these within a purely discrete event computational framework. We discuss the implications of these results for multi-formalism modeling and the implication of the results for a related, but alternative, formalization of hybrid systems called Heterogeneous Flow System Specification.

1 Introduction

Although DEVS (Discrete Event System Specification) has been employed in a number of studies in which both discrete and continuous elements appear [1,2,3, 4], it has not received a formal proof of its capability to express such combined or hybrid models as they are often called [5,6,7] . In this paper, we offer such a proof by supplying some of the missing pieces that complete the picture presented in Theory of Modeling and Simulation [8]. Praehofer [9] defined the combined formalism, DEV&DESS to represent combined continuous discrete models that are simulatable in simulation languages such as SLAM and ARENA [10]. As defined, DEV&DESS describes a subclass of dynamic systems that includes the subclasses specified by DEVS and DESS (Differential Equation System Specification), respectively. As such it provides a rigorous framework for considering properties of hybrid models but it does not provide a computational framework for simulating such models. Barros [11,12] has also provided a DEVS-based extension to include continuous system modeling with implementation on a SmallTalk platform [13], to which we shall return later. On the other hand, the DEVS formalism has been implemented in numerous platforms [14-19] thus affording an advantageous computational basis for simulation of combined models. Therefore, an official proof of the embedding of DEV&DESS into DEVS would provide a formal justification for the

use of the latter for combined or hybrid models. Furthermore, we show how multiformalism modeling [20] – the integrated use of many formalisms in the same framework – can be explicitly supported in such a framework.

The possibilities for expression of a formalism relative to DEVS are illustrated in Figure 1.

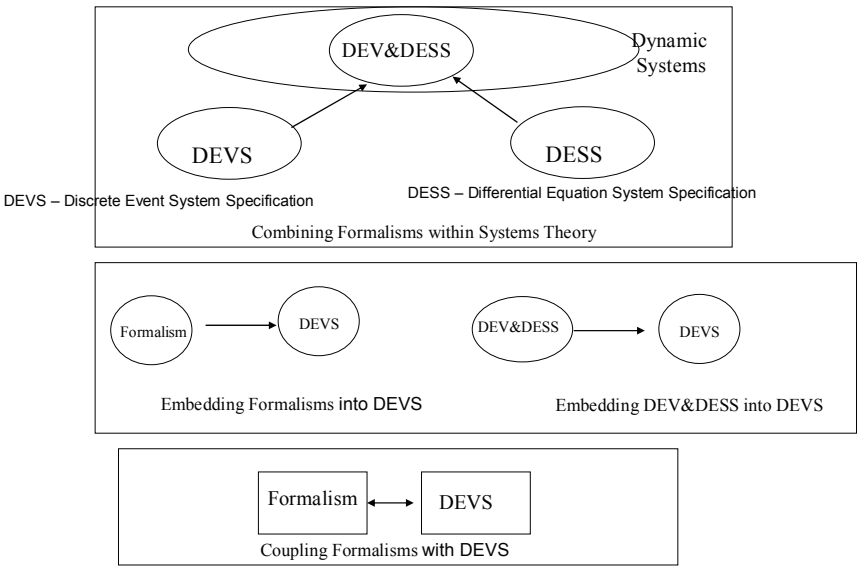


Figure 1. DEVS-based Multi-formalism Embedding and Compositional Approaches

Three approaches are depicted:

1. *Extending DEVS into a larger subclass of dynamic systems*: As mentioned earlier, the extension to DEV&DESS was performed by Praehofer [9] in his doctoral dissertation. He proved closure under coupling, thereby establishing that the structures and couplings included in the basic definitions are sufficiently expressive to represent arbitrary hierarchical compositions of discrete and continuous components. DEV&DESS is therefore a good candidate to consider as basis for hybrid and multi-formalism modeling theory investigations.

2. *Embedding a formalism into DEVS:* TMS2000 [8] also proved that DEVS is universal for discrete event systems, i.e., any system within the sub-class of dynamic systems having discrete event input/output behavior (when reduced to its canonical form) is isomorphic to a DEVS. This invites the possibility of exploiting the properties of special formalisms for discrete event systems by

expression them as subclasses of DEVS. Studies of this nature has been done for Petri Nets [21, 22], state charts [23] and timed automata [24].

3. *Interoperating a non-dynamic systems formalism with DEVS:* Sarjoughnian et al. [25] have shown the benefit of this approach, where linear and non-linear programs are coupled with discrete event simulation to optimize supply chains. The approach employs an interface specification to mediate between the two forms of knowledge (called a knowledge interchange broker) that provides a generic means of coupling a variety of simulation models with optimization strategy implementations.

Within this framework of discussion, we focus on embedding formalisms into DEVS. In particular, we show that the DEV&DESS formalism, the extension of DEVS with DESS, can be embedded within DEVS itself – within an accuracy specification that relates to the quantization-based representation of DESS within DEVS. As suggested above, the proof provides a justification for employing DEVS-based simulation environments as tools for hybrid and multi-formalism modeling – the latter taking into account the capabilities of DEVS to embed more restricted formalisms and to be interoperated with non-dynamic systems formalisms.

2 Characteristic Behaviors of the DEV&DESS Formalism

As background for the embedding of DEV&DESS into DEVS we first briefly review the DEV&DESS formalism presented in [8] and then go on to elucidate the characteristic behaviors need to achieve full DEV&DESS expressive capability.

2.1 Review of DEV&DESS

The DEV&DESS formalism [8] envisions a composition of discrete event and continuous parts as illustrated in Figure 2.

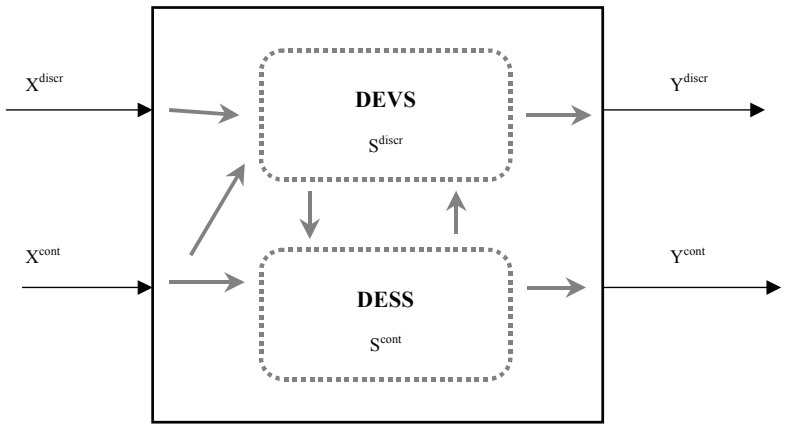


Figure 2. DEVS and DESS combined model

The DEV&DESS formalism prescribes the components and their interfaces. It is presented as follows:

$$\text{DEV\&DESS} = < X^{discr}, X^{cont}, Y^{discr}, Y^{cont}, S^{discr}, S^{cont}, \delta_{ext}, C_{\text{int}}, \delta_{\text{int}}, \lambda^{discr}, f, \lambda^{cont} >$$

X^{discr}, Y^{discr} are sets of discrete event inputs and outputs, resp.

X^{cont}, Y^{cont} are sets of continuous inputs and outputs, resp.

S^{discr}, S^{cont} are sets of discrete and continuous states, resp.

$\delta_{ext} : Q \times X^{discr} \times X^{cont} \to S$ is the external transition function

where $S = S^{discr} \times S^{cont}$ is the sequential state set, and

where $Q = \{(s,e) \mid s \in S, e \in \mathbb{R}_0^+\}$ is the total state set

$\delta_{\text{int}} : Q \times X^{cont} \to S$ is the discrete event internal transition function

$\lambda^{discr} : Q \times X^{cont} \to Y^{discr}$ is the discrete event output function

$\lambda^{cont} : Q \times X^{cont} \to Y^{cont}$ is the continuous output function

$f : S \times X^{cont} \to S^{cont}$ is the derivative function

$C : Q \times X^{cont} \to Bool$ is the event detection condition predicate

The semantics of the DEV&DESS formalism are given in terms of the subclass of dynamic systems that it defines [8]. A *state event* is considered to be the occurrence of a change in the value of the event condition predicate from false to true. The concept of state event derives from Pritsker's [26,27] original formulation of combined continuous and discrete simulation in which it was defined as a threshold crossing of the continuous state. In the DEV&DESS formulation, the concept of state event is generalized so that it can occur due to a change in any of its arguments, of which the continuous state is one component.

We note that as just formulated, the DEV&DESS formalism does not include a time advance function. Instead, triggering of the internal transition function is performed by the output of the event detection condition. We will show later that the time advance trigger can be captured within the formalism.

2.2 Capabilities Exhibited in the DEV&DESS Formalism

DEV&DESS defines certain behavior primitives that are manifest in the formalism that are not found in either of its components separately. As illustrated in Figure 3, these are: a) state event detection, b) derivative function change, c) integrator state reset, and d) event detector threshold change.

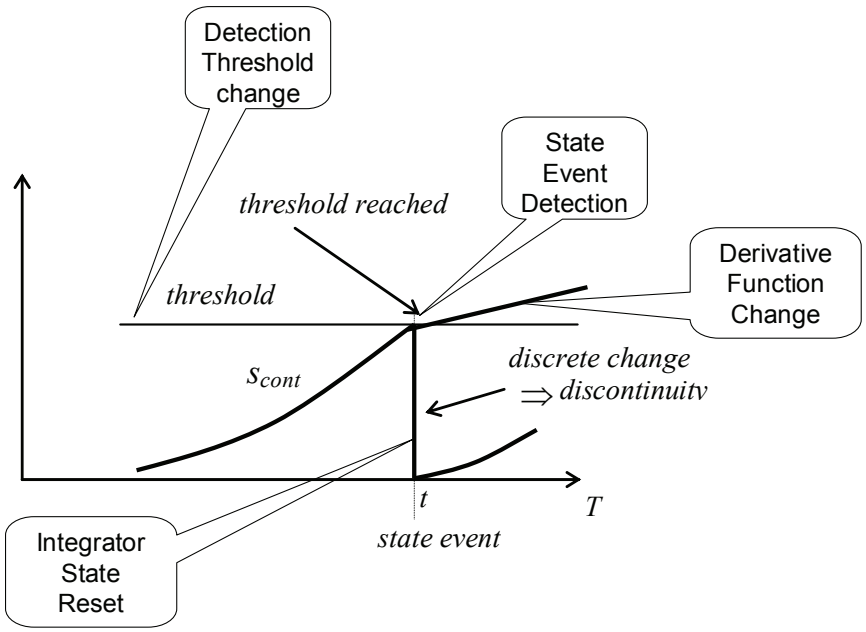


Figure 3.Behaviors in DEV&DESS

Let's see how the formalism makes use of these basic primitives to obtain behaviors characteristic of hybrid systems.

Behavior 1. *Discrete Events, both state and external, can control the derivative function, thus causing an instantaneous change in the derivative of the continuous state trajectory.*

By expressing the sequential state set as a cross product of the discrete and continuous state sets, the derivative function takes the form:

$$f : S^{discr} \times S^{cont} \times X^{cont} \rightarrow S^{cont}$$

Since the discrete state changes only through discrete events, we can consider f to be a family of functions indexed by S^{discr}, with the interpretation that the differential equation

$$\frac{ds^{cont}}{dt} = f_{s^{discr}}(s^{cont}, x^{cont})$$

holds for the interval during which s^{discr} is the state of the discrete event part as depicted in Figure 4.

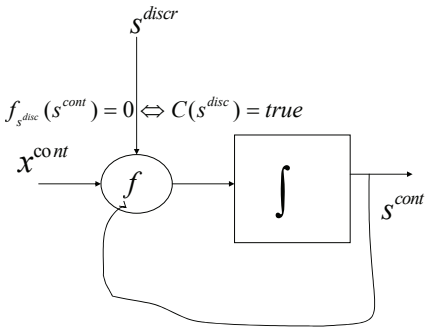


Figure 4. Illustration of discrete event control of derivative function

External events, through the invocation of the external transition function can change the discrete state and hence control the derivative function. Similarly, a state event, through the invocation of the internal transition function, can exhibit the same effect. An example given in TMS2000 concerns the filling of a container until full. This requires setting the derivative of the filling function to zero to shut off a flow, either externally, or when a level is exceeded (a state event).

Behavior 2. *Discrete Events, both state and external, can control the event detection condition, thus instantaneously changing the thresholds governing event detection.*

In the same manner as done for the derivative function, we can consider the event detection condition to be indexed by the discrete state set:

$$C_{s^{discr}} : S^{cont} \times X^{cont} \rightarrow Bool$$

where a particular predicate $C_{s^{discr}}$ holds during the interval in which the discrete state s^{discr} prevails (Figure 5).

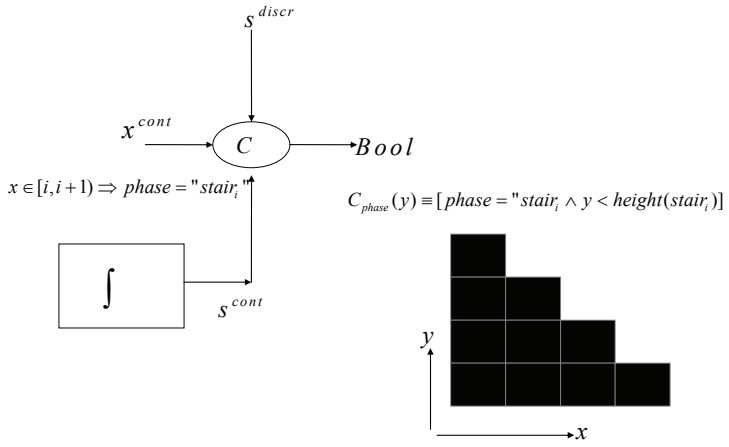


Figure 4. Illustration of discrete event control of event detection function

As an example, in a model first developed by Kofman [19], a ball bounces down a staircase. In our reformulation, the stair currently underneath the ball can be tracked by a phase variable in the discrete event part. The event detection condition controlled by this phase is made up of two parts corresponding to the vertical and horizontal coordinates of the ball. In the vertical component, the detection condition monitors the ball's vertical position for crossing the level of the current stair. In the horizontal component, the detection condition monitors the ball's horizontal position for the transition to the next lower stair. Thus the horizontal position of the ball determines the stair phase, e.g. $x \in [i, i+1) \Rightarrow phase = "stair_i"$. This in turn determines the height of the stair currently below the ball and thereby the level to be crossed in order to change the derivative function to represent the bouncing region, e.g. $C_{phase}(y) \equiv [phase = "stair_i \wedge y < height(stair_i)]$

Similarly, the continuous input or and/an external event can change the event detection function. Furthermore, in the latter case, the elapsed time can be taken into account. When an external event arrives after an elapsed time e since the last event, it can trigger

the external transition function which can change the discrete state based on the current state and the elapsed time e.

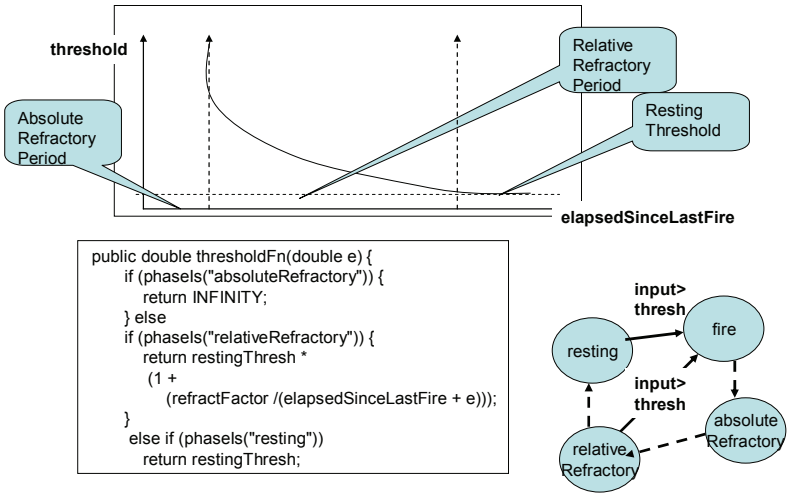


Figure 5. Firing threshold of a leaky integrator neuron

An example of dependence on external events is given by a model of the firing threshold of a leaky integrator neuron [28] as modeled in DEV&DESS. As depicted in Figure 5, the model has a discrete event phase variable such that the firing threshold depends both on the phase as well as the elapsed time since it was last fired. In the "resting" phase the threshold equals the resting threshold value. When the input exceeds the threshold, the phase changes to "absoluteRefractory" in which the neuron cannot fire (the threshold effectively takes on an infinite value). The passage from "absoluteRefractory" to "relativeRefractory" is an event that is triggered by the passage of a fixed refractory time. In the "relativeRefractory" phase, the threshold is given as a function of time since last fired, which itself is updated through the elapsed time, e.

Behavior 3. *Discrete Events, both state and external, can instantaneously change the continuous state (causing discontinuities from the point of view of differential equation systems)*

This is true because the continuous state is one of the components in the range of the external and internal transition functions. For example, we have

$$\delta_{ext} : Q \times X^{discr} \times X^{cont} \rightarrow S^{discr} \times S^{cont}$$

Since the occurrence of an external event triggers this function, the external event can directly change the continuous state, which is the composite state of the integrators underlying the differential equation of the continuous part.

The barrel filling model provides an example of a state event triggering an integrator reset. When the barrel becomes full (state event),e.g. $C(s^{cont}, s^{disc}) = [s^{cont} > fullLevel]$ the contents of the barrel (hence the integrator representing the barrel) is reset back to zero. An example of an external event triggering a reset is provided by a model of the synapse of a leaky integrator neuron [28]. Spike arrivals at the synapse (external events) cause an immediate jump in the soma integrator state (which then proceeds to decay to zero).

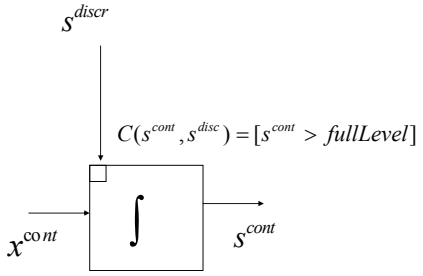


Figure 6. Illustration of discrete event control of integrator state

SIMULINK can reset integrator states during simulation but only to the state that was used in initializing the model – hence for a return to the initial state. This meets the requirements of the barrel filler example. However. it does work not for spikes arriving to a synapse where the reset is to a potentially different state on each occasion. In an inverted pendulum model [29], the angle of the pendulum must be set to 0 when it reaches 2π . This is not the same as the initial angle of the pendulum which is π).

2.2 Regaining the DEVS time advance function trigger of the internal transition function

In DEV&DESS as presented in [8], there is no explicit time advance and hence no triggering of the internal transition function due to time events. This is a simplification of the definition of DEV&DESS as first presented by Praehofer [9]. It is important to show that the simplified definition does not reduce the class of systems specified by the original definition. The time advance triggering functionality is regained by the approach illustrated in Figure 7. Incidentally, the approach also demonstrates the necessity of the three characteristic behaviors. We start by recognizing that time can be represented by the output of an integrator with constant unit input. To get the effect of a scheduled time

advance, we need a condition predicate that detects when the output crosses the given time advance threshold. The output triggers the internal transition function that resets the time integrator back to zero as well as sending the new time advance value to the condition predicate. Thus all three DEV&DESS characteristic behaviors are needed to capture the effect of the time advance trigger in DEVS.

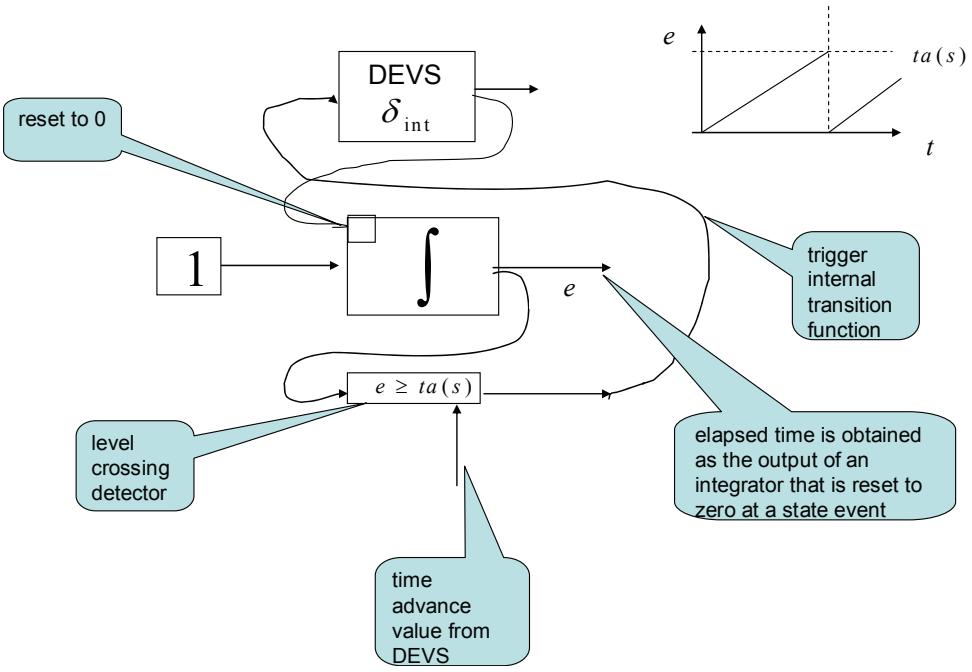


Figure 7. Expressing the time advance triggering of the internal transition function

3 DEVS implementation of DEV&DESS

3.1 Embedding DEV&DESS into DEVS

The DEVS implementation of DEV&DESS will be shown by the embedding of DEV&DESS into DEVS. This is done by showing how an arbitrary DEV&DESS model is realized by a DEVS model. The realization employs the same component-wise simulation concepts that were employed in [8] to provide the embedding of DESS into DEVS. The embedding in Figure 8 implements the individual DEVS and DESS parts of a DEV&DESS model as modular DEVS components coupled within the DEVS formalism. The DEVS part is implemented by a corresponding DEVS denoted by DEVS'. The DESS part is implemented using the quantized integrator approach [8,19], with accuracy determined by the quantum size. The instantaneous derivative functions in this representation are augmented with input ports that allow output events from the DEVS'

component to control the derivative functions. Likewise, the DEVS integrators have additional ports to allow output events from the DEVS' component to instantaneously change the integrator state. The event detection condition is implemented as a DEVS component with input ports receiving outputs of the integrators. Finally, the output ports of the event detector are coupled to the input ports of the DEVS' providing the event detection information to it. This realization which is, in part, an exercise in converting from non-modular to modular form, requires that zero time advances are employed to transmit state and control information that is immediately available in the non-modular form. In this manner, the three behaviors characteristic of DEV&DESS are implemented with DEVS itself.

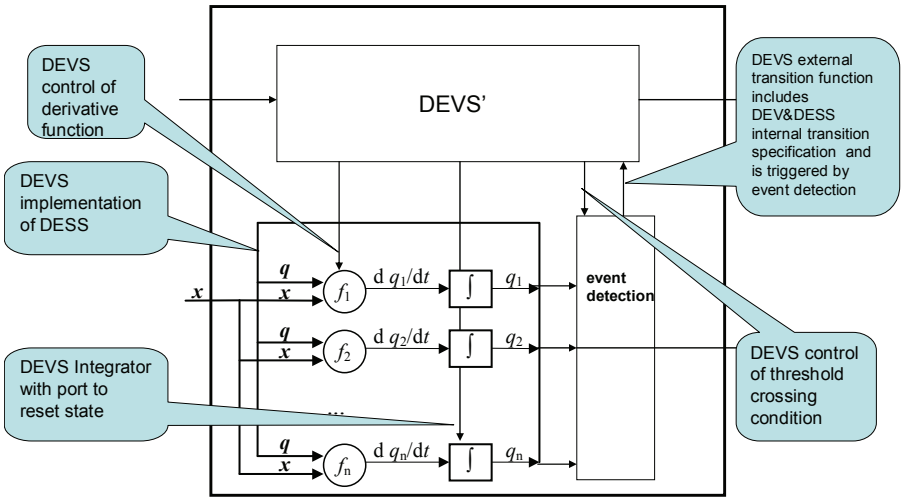


Figure 8. DEVS implementation of DEV&DESS

3.2 Well-defined DEV&DESS: Potential Zero-Time Loop in Integrator Resetting

We recall that the criterion for well-definition of any system specification requires that it specifies a unique system in the class of mathematical input/output systems [8]. The well-definition for DEV&DESS was formulated as a combination and extension of the separate well-definition criteria for DEVS and DESS separately. The DESS requirement is that solutions satisfying the underlying differential equation exist and define unique state trajectories and are guaranteed the Lipshitz condition as well as the absence of algebraic loops, (see page 164 of [8]). The essence of DEVS well-definition is the *legitimacy criterion* which requires that time always moves forward through the accumulation of time advances. An equivalent formulation was given in terms of events, namely, there are only a finite number of external and internal events in any finite time

interval of a simulation run. The extension of this criterion to DEV&DESS is called *state-event legitimacy* and requires that the number of state events be finite for every admissible input segment (page 213 of [8]). We recall that a loop of components all having zero time advances violates this legitimacy criterion. However, this is not the only source means by which illegitimacy arises. In general the convergence of infinite series of time advances must be avoided (see [8] page 142), akin to Zeno's paradox (see [3] for recent perspective.)

Let's now consider how this applies to the testing a DEV&DESS model for legitimacy by employing its embedding in DEVS as in Section 3.1. Theorem 5 of Chapter 15 in [8] is easily amended to state that that the absence of algebraic loops in DESS is equivalent to the absence of zero-time advance loops in its DEVS realization. Assuming Lipshitz conditions hold, this fact reduces the test of well-definition of the DESS part to that of legitimacy of its DEVS representation. However, there remains a source of state-illegitimacy that needs to be considered. It relates to the resetting integrator states by the discrete events.

Figure 10 a) extends Figure 6 by explicitly showing the feedback loop from an integrator through the condition function that decides when and how to reset the integrator. This is a zero-time loop if the integrator's output is instantaneously sent to the decision condition which instantaneously resets the integrator. The example in Figure 10 b) illustrates a feedback coupling between an integrator, *x* and a level crossing detector, *xUp.* The integrator broadcasts its state with zero time advance upon any resetting of its state. The level crossing detector has zero time advance to send the value to the integrator reset port.

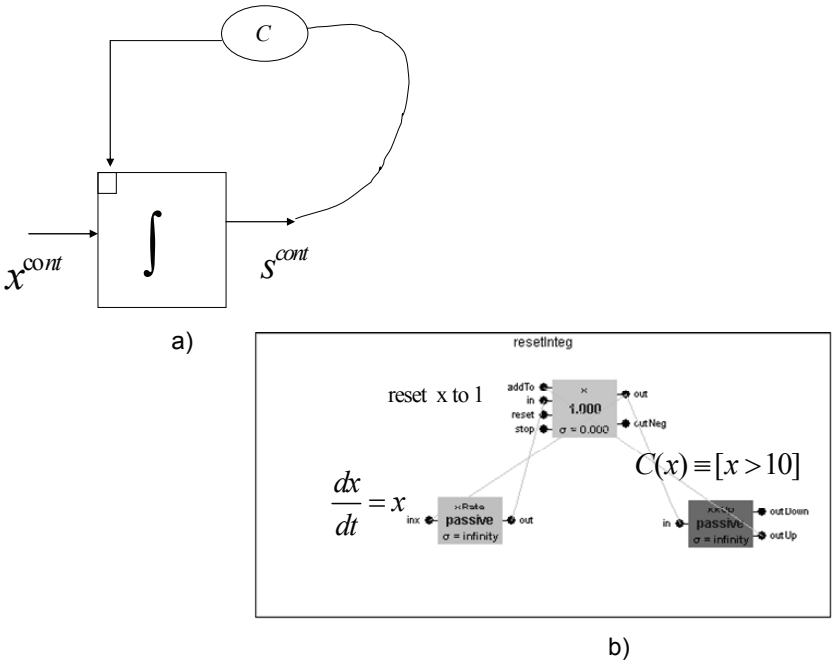


Figure 10. Potential Zero-time Loop in Integrator Reset

Under certain circumstances, this will form a loop of zero time advances. However, this cycle need not be manifest. For example, if the state to which the integrator is reset no longer satisfies the event detector's output criterion then the latter passivates, thus breaking the zero time advance loop. In Figure 10 b), when the level 10 is crossed from below, the integrator is reset back to 1, thus passivating the detector. Note that passivation (setting the time advance to infinity) is available only in a discrete event framework.[1]

We concluded that the well-definition of a DEV&DESS model can be tested in its DEVS embedding by testing for Lipshitz conditions on the DESS derivative functions and state-event legitimacy of the DEVS. A necessary condition for the latter is absence of zero time-advance loops in the DESS part as well as in the interaction between the DEVS and the integrator resets. Although it deals with a major source of illegitimacy, the condition is not sufficient since the latter may still arise through converge in the manner of Zeno's paradox as mentioned above.

4 The Need for Multiple Formalisms in Information Technology-based Systems

The need for multi-formalism modeling in developing modern information technology-based systems is increasingly recognized [2,20]. The ability of DEVS to support such modeling follows from the results just presented. Since DEVS implements DEV&DESS and leads to computationally effective and efficient environments, it offers the basis for combined discrete and continuous (hybrid) modeling. Moreover, as mentioned, since DEVS is universal for discrete event systems, special formalisms such as Petri Nets can be included when appropriate. Finally, DEVS-based simulation components can be interoperated with optimization and other non-dynamic software components. We now briefly discuss robot collaboration to show how this works to address new information-technology challenges.

[1] The same situation shows up in a continuous simulation language that allows integrator resetting as well. However, in that context, it may be treated as an algebraic loop that is detected by the syntactic analyzer. To resolve this situation, SIMULINK adds a state output port to the integrator which provides the last known state to the condition function. Syntactically, this breaks the algebraic loop but may not resolve the true problem. The formulation in Figure 10 within the realization of DEV&DESS in DEVS provides a useful context for examining the problem.

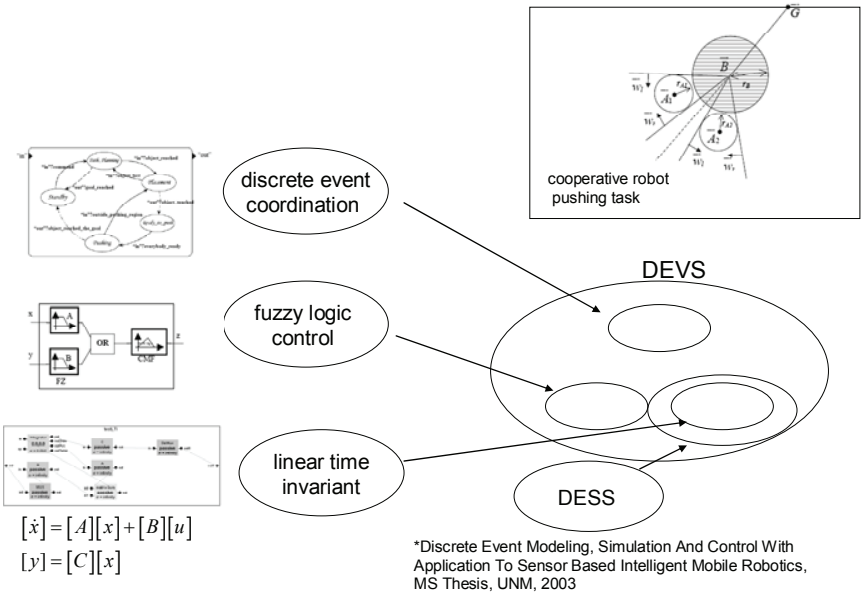


Figure 9.Mulltiformalism Modeling within DEVS

Figure 9 illustrates how several formalisms were employed in the development of cooperative robotics. To design robotic controls so that two or more robots can push an object to a given location, the following formalisms were implemented as packages in DEVSJAVA [4]:

- *Linear time invariant (LTI) systems*– implemented as coupled models that represent differential equations specified in the form:

  $$[\dot{x}] = [A][x] + [B][u]$$
  $$[y] = [C][x]$$

  where the lower case elements are real-valued vectors of inputs, states, and outputs, and the upper case elements are matrices that define the derivative and output functions. LTI is a subclass of DESS that forms the basis of classical control theory. Although not sufficient for robot dynamics, LTI simplifies specification, has a legacy of analytical support, and can be employed within sub-components of non-linear systems, for example, in representing robot and object motion with Newton's laws.

- *Fuzzy Logic Control (FLC) systems* – implemented as coupled models with components for fuzzification, fuzzy rule processing, and defuzzification. FLC is a sub-class of instantaneous functions that has been used in a variety of intelligent control applications [4,18]. FLC provides robust control logic for guiding robots toward goal locations, avoiding obstacles, and cooperatively pushing objects.

- Discrete Event Coordination (DEC) systems – directly implemented within DEVS to provide overall coordination and supervision of robot task executors. DEC provides control phases, analogous to the states of Statecharts, to direct and synchronize individual robot activities toward a global objective.

5. Conclusions: Comparison with Alternative Formalisms

The *Heterogeneous Flow System Specification* (HFSS) [11,12,13] formalism provides an alternative formulation of combined discrete and continuous systems. The formalism is an extension of DEVS that builds in an input sampling mechanism as well as variable structure capability. The approach allows multirate numerical methods for solving differential equations to be represented and supports event detection. Barros [11] claims that the HFSS formalism provides a unified framework for representing digital control, signal processing, numerical integration and hybrid systems. For example, it is possible to describe numerical integrators as part of the formalism and not as an external construct that needs to be incorporated into a pure modeling formalism. Barros [11] also claims that HFSS offers precise semantics to enable the interoperability of hybrid systems in a similar manner existing for discrete systems. However, there has been no discussion of the conditions under which models in the formalism are well-defined nor whether closure under coupling holds for the formalism. The discussion of these issues herein should carry over in large measure to HFSS. A precursor of HFSS, Continuous Flow Specified System (CFSS) was mapped into DEVS as a sub-formalism [30]. Essentially, the mapping shows how to simulate the sampling behavior of CFSS explicitly as a request and wait protocol in DEVS. Since HFSS is an extension of CFSS that preserves this sampling construct, the relationship of HFSS to DEVS an open question for further investigation. Other representations for combined systems within DEVS need attention as well [31,32].

This paper has explicitly pointed out the behavioral features in the DEV&DESS formalism that need to be supported in DEVS for the latter to be able to implement the former. In a broader perspective, the results elucidate the characteristic primitives needed for hybrid or combined continuous/discrete event models and provide a rigorous proof of the ability to express these within a purely discrete event computational framework. These results lay the foundation for a user-friendly modeling environment that puts the primitives directly at the user fingertips as templates for DEVS atomic and coupled models that can be easily instantiated and composed.

References

1. Enrique V. Kortright Modeling and Simulation, Article in Encyclopedia of Computer Science and Technology: Volume 40 -, Allen Kent, James G Williams Taylor and Francis, CRC Press. 1999, *pp* 177-190

2. R.W. Sierenberg, Combined discrete/continuous modeling, in: L. M. Tijskens (Editor), M. Hertog (Editor), B. Nicolai (Editor) Food Process Modelling, 2001

3. Le Goc, M. and C. Frydman, "SACHEM, a Real Time Intelligent Diagnosis System based on the Discrete Event Paradigm", *Transaction of Society for Modeling and Simulation International* (SCS), 2003.

4. Shahab, S., "Discrete Event Modeling, Simulation And Control With Application To Sensor Based Intelligent Mobile Robotics," MS Thesis, UNM, 2003

5. ALUR, R., GROSU, R. LEE, I., AND SOKOLSKY, O. 2001. Compositional refinement for hierarchical hybrid systems. In *Hybrid Systems: Computation and Control. Proceedings of the 4th International Conference (HSCC'01)*. Lecture Notes in Computer Science, vol. 2034. Springer-Verlag, New York, 33-48.

6. R. Alur and D.L. Dill, A theory of timed automata, Theoretical Computer Science, V.126, pp 183-235, 1994

7. ANTSAKLIS, P. J. 2000. A brief introduction to the theory and application of hybrid systems. *Proc. IEEE 88*, 7, 879–887.ACM Transactions on Modeling and Computer Simulation, Vol. 13, No. 3, July 2003.

8. Zeigler, B. P., T.G.Kim and H. Praehofer "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, second edition Academic Press, Boston", 2000. 510 pages

9. Praehofer, H., System Theoretic Formalisms for Combined Discrete-Continuous System Simulation. Int. J. Gen. Sys., 1991. 19(3): p. 219-240.

10. R.C. Huntsinger, Simulation Languages and Applications;, in: George A. Bekey, Boris Y. Kogan (Editors), Modeling and Simulation : Theory and Practice A Memorial Volume for Professor Walter J. Karplus, Kluwer, 2000.

11. F.J. Barros. "Towards a Theory of Continuous Flow Models," *International Journal of General Systems*, Vol. 31, No. 1, 29-39, 2002.

12. F.J. Barros. "Modeling and Simulation of Dynamic Structure Heterogeneous Flow Systems." *SIMULATION: Transactions of The Society for Modeling and Simulation International*, Vol. 78, No. 1, 18-27, 2002.

13. F.J. Barros, "Dynamic Structure Multi-Paradigm Modeling and Simulation," A*CM Transactions on Modeling and Computer Simulation*, Vol. 13, No. 3, pp. 259-275, 2003.

14. Q. Liu, G. Wainer "Simulating Market Dynamics with CD++".. In *Proceedings of the International Conference on Computational Science*. Lecture Notes in Computer Science. Atlanta, GA. 2005

15. Jong-keun Lee, Min-Woo Lee, Sung-Do Chi, “DEVS/HLA-Based Modeling and Simulation for Intelligent Transportation Systems”, SIMULATION, Vol. 79, No. 8, 423-439 (2003).

16. Bernard P.Zeigler, Doohwan Kim, Stephen J. Buckley, “Distributed supply chain simulation in a DEVS/CORBA execution environment”, December 1999 Proceedings of the 31st conference on Winter simulation: Simulation---a bridge to the future - Volume 2.

17. Bernard P.Zeigler, Yoonkeon Moon, Doohwan Kim, Jeong Geun Kim, “DEVS-C++: A High Performance Modeling and Simulation Environment”, January 1996, Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture.

18. Bernard P.Zeigler, Hessam S. Sarjoughian, Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models, http://www.acims.arizona.edu/SOFTWARE/devsjava_licensed/DevsJavaUserGuidev1.1.zip

19. Kofman, E. (2003). "Quantization-Based Simulation of Differential Algebraic Equation Systems". Simulation (Journal of The Society for Computer Simulation International). 79(7). pp 363-376.

20. Hans Vangelieuve and P. Mosterman, “Computer Automated Multi-Paradigm Modeling,” *TOMACS. 12*, 4, 1–7.

21. C. Jacques, G. Wainer Using the CD++ DEVS toolkit to develop Petri Nets. In Proceedings of the 2002 Summer Computer Simulation Conference. San Diego, CA. USA. 2002.

22. Carmen-Veronica Bobeanu, Eugene J. H. Kerckhoffs,Hendrik Van Landeghem,”Modeling of Discrete Event Systems: A Holistic and Incremental Approach Using Petri Nets,” ACM Transactions on Modeling and Computer Simulation, Vol. 14, No. 4, October 2004.

23. Spencer Borland and Hans Vangheluwe. Transforming Statecharts to DEVS. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference. Student Workshop*, pages S154 - S159. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada.

24. H. Dacharry, N. Giambiasi ``Formal Verification with Timed Automata and DEVS Models: a case study", *in: ASSE 2005 Simposio Argentino de Ingeniería de Software - 34 JAAIO Jornadas Argentinas de Informatica e Investigacion Operativa*, *pp. 251-265*, *Rosario, Argentine*, 29 août- 2 septembre 2005

25. H. Sarjoughian, “Hybrid Discrete Event Simulation With Model Predictive Control For Semiconductor Supply-chain Manufacturing,” *Proceedings of the 2005 Winter Simulation Conference*, Orlando, FL, 2005.

26.F.E. Cellier, 1979 *Combined Discrete/Continuous System Simulation by Use of Digital Computers*. Techniques and Tools. PhD Thesis, Swiss Federal Institute of Technology,.

27. Pritsker, A.A.B., The GASP IV Simulation Language, John Wiley & Sons, New York, N.Y., 1974

28. Rachel, Olivier, "Une approche evenementielle pour la modelisation et la simulation de reseaux de neurones impulsionnels," Doctoral Dissertation, Henri Pontcarre University, 2004.

29. K. J. Åström†and K. Furuta‡ Swinging Up A Pendulum By Energy Control, Paper presented at IFAC 13th World Congress, San Francisco, California, 1996

30. F.J. Barros, B.P. Zeigler, Model Interoperability in the Discrete Event Paradigm: Representation of Continuous Models, in: George A. Bekey, Boris Y. Kogan (Editors), Modeling and Simulation : Theory and Practice A Memorial Volume for Professor Walter J. Karplus, Kluwer, 2000.

31. M. D'Abreu, G. Wainer. M/CD++: modeling continuous systems using Modelica and CD++, MASCOTS 2005. Atlanta, GA. 2005

32. N. Giambiasi, G. Wainer ``Using G-DEVS and Cell-DEVS to model complex continuous systems", *in: Simulation: Transactions of the Society for Modeling and Simulation International*, *vol. 81*, *n° 2*, *pp. 137-151*, February 2005.

33. A. Cotaldo, E.L. Lee, X. Liu, E. Matsikouda and H. Zheng, "Discrete-Event Systems: Generalizing Metric Spaces and Fixed-Point Semantics", J. DEDS,

Lecture 6. Object-oriented Simulation

Papers to study for the exam

- P.H.M. Jacobs, N.A. Lang, A. Verbraeck. DSOL: A Distributed Java based discrete event simulation architecture. In: E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes (Eds.). Proceedings of the 34th Winter Simulation Conference. IEEE Computer Society Press. 2002. pp. 793-800. Retrieved from: http://www.informs-sim.org/wsc02papers/102.pdf

- J.A. Joines and S.D. Roberts. Simulation in an Object-Oriented World. In: P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds. (Eds.). Proceedings of the 1999 Winter Simulation Conference. IEEE Computer Society Press. 2002. pp. 132-140. Retrieved from: http://www.informs-sim.org/wsc99papers/017.PDF

Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

D-SOL; A DISTRIBUTED JAVA BASED DISCRETE EVENT SIMULATION ARCHITECTURE

Peter H.M. Jacobs
Niels A. Lang
Alexander Verbraeck

Delft University of Technology
Faculty of Technology, Policy and Management
Systems Engineering Group
Jaffalaan 5, 2628 BX, Delft, THE NETHERLANDS

ABSTRACT

Most discrete event simulation environments are based on a process-oriented, and therefore multi-threaded paradigm. This results in simulation environments that are very hard to distribute over more computers, and not easy to integrate with scattered external information sources. The architecture presented here is based on the event-based DES paradigm which is implemented by scheduled method invocation. Objects used in the simulation environment interact with remote, a-synchronous subscribed clients in order to produce representations of the simulated system. The environment, which is implemented in Java, consists of a simulation and representation library and is integrated with several statistical libraries.

1 INTRODUCTION

Discrete event simulation is used for dynamically modeling complex systems in order to analyze and understand these systems, and in order to (re)design and engineer these systems (Shannon 1975, Banks 1998). Statistical information is gathered and presented during and after the simulation run for providing insight into the most important performance indicators. In most cases, animation and visualization are added to provide insight and understanding into the system modeled during the run. Currently we see a lot of interest in object-oriented simulation environments, where real-world objects can be captured in the simulation environment as object representations. Many of the simulation applications in logistics, business process engineering, and information systems modeling deal with distributed systems with distributed information sources. Therefore, it seems logical to strive for a simulation environment that truly captures the distribution, and that does not reduce the effects of the distributed nature to an artificial time delay. Most current discrete event simulation packages are, however, not designed to cooperate in a distributed computing environment. There are several reasons why it would be interesting to have a simulation environment that has a truly distributed nature.

Simulation models of business processes may need to interact in complex ways with real-time information system components (Fishwick 1995). It might for example save a simulation model builder a lot of time to reuse an existing pricing component of a business system in a simulation component. However, in order to do so, the simulation and business system components should interact using a common communication protocol. Although such protocols (for inter-business component communication) have not yet matured, a simulation architecture can at least be expected to be extendible with such interaction mechanisms. This implies for example separating the core simulation service from the model components that use it. In this way the simulation will become a central service in a distributed, dynamic network where the exact capabilities and identity of the acting and reacting realtime components remains hidden for the simulation core. In the most extreme peer to peer case, objects will use the simulation service just as another way of scheduling their interactions.

Another issue is the lack of separation between visualization and simulation in current simulation environments. First, the place where the simulation can best be run (e.g. on a fast server) is not always the place where we want to view the simulation (e.g. in a Web browser at a client's location). Second, due to the global and complex nature of current business systems or logistical systems, the number of stakeholders involved in these systems is large. Moreover, these stakeholders do not reside in one location, but are spread all over the world. In order to provide them with a customized view on a simulation representing their business, there is a need to decouple visualization from the

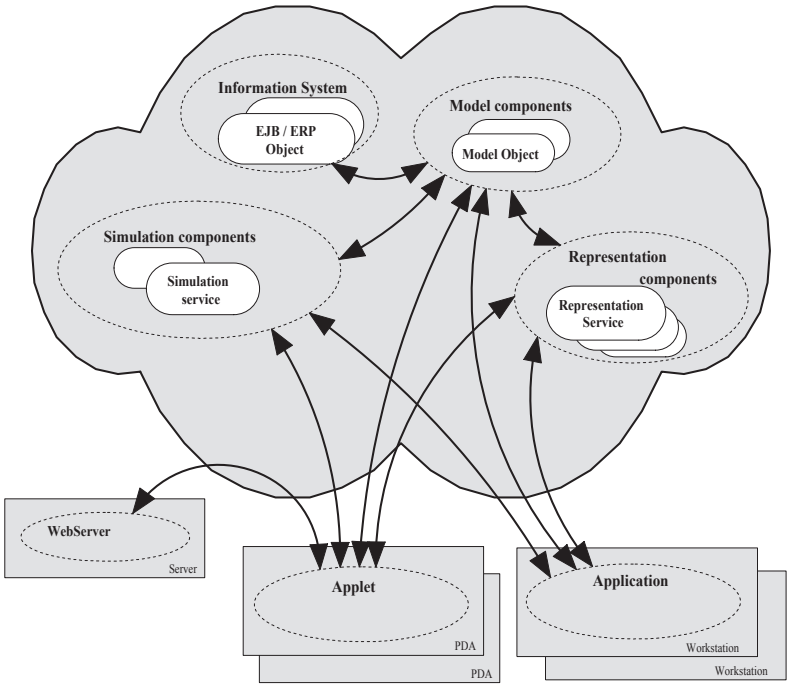


Figure 1: Architecture of Fully Distributed Simulation Environment

simulation and to deliver this visualization in an interactive way to distributed clients (e.g. a web browser or pda). When several stakeholders are in this way concurrently interacting with the simulation, distributed simulation can become distributed gaming.

A final argument for a distributed simulation architecture is rooted in the need to *specify* simulations of complex business systems. Gathering data from systems and databases, and carrying out measurements for the specification by hand is a costly procedure. However, most data needed for simulation studies is available in some automated form, such as in datawarehouses, in ERP databases, or in legacy systems. Since these systems probably contain the most consistent, accurate and actual information of the business system, it seems logical to use that information to specify a simulation. This implies the need to enable a simulation to interact with (distributed) information systems, perhaps even during the run of the simulation. In this way generators, statistical objects and even some of the transactional events could be automatically instantiated and initialized from underlying data sources.

This section has clearly stated the need for a distributed architecture on which discrete event simulations are built. As indicated, current implementations of discrete event simulation software are not sufficiently capable to support this distributed cooperation. The next section will present some background on existing object-oriented simulation packages. Section 3 will point out the approach that we have taken to develop a new distributed simulation and representation architecture called *D-SOL*, which stands for *Distributed Simulation Object Library*. In this section screenshots of a distributed supplychain example are included. The final section presents the conclusions and gives some pointers for further research.

2 EXISTING SIMULATION ENVIRONMENTS

This section gives an overview of current discrete event environments. Focusing on the distributed goal of simulation environments, three different levels of distribution can be distinguished in these environments. These are:

- Integrated simulation environments
- Semi-distributed simulation environments
- Fully distributed simulation environments.

Integrated simulation environments are not suited for a distributed deployment since visualization is strictly bound to the same runtime environment as the simulation. Further characteristics are single-user, single-computer, single-thread implementations of the environment. Most commercially available simulation packages fall in this category.

Semi-distributed simulation environments share a web-based characteristic. These packages can be divided in simulation models that are executed on the client (Page et al. 1997) and simulation models that are executed on the server and present their output to the client.

Examples of the first category of semi-distributed environments are most Java-based simulation environments like Silk, JavaSim, SimJava and DESMO-J (Garrido 2001). An example of the second category is the simulation package that can present its results during and after the run in html-format to distant viewers on the web.

Fully-distributed simulation environments consist of distributed simulation and visualization services. They provide integration with distributed information systems and endless variations in configuration. Figure 1 is an illustration of this architecture.

Since the Java programming language has strong support for distributed interconnectivity, it is reasonable to assume that current Java-based simulation languages characterized as semi-distributed can easily be used to deploy fully-distributed simulation models.

The next two sections will show though that current Java-based simulations are either based on a worldview that conflicts with the level of distribution that is sought in this research or share design patterns that focus on the easiness of model construction but conflict with a distributed deployment.

2.1 Current Java-based Simulation Languages Based on a Process Oriented Worldview

The main goal of environments like JavaSim, Silk, DESMO-J, etc. is to create (re)usable object-oriented simulation models and components while maintaining a natural process-oriented modeling capability. Since entities in the model are alternately suspended and resumed to coordinate time-ordered sequencing of entity movements in the model (Healy and Kilgore 1998), multithreaded execution is an essential aspect of the implementation for this process-oriented modeling capability. The implication of this multithreaded approach for the distributed goal are:

- Starvation occurs when one or more threads in a program are blocked from gaining access to a resource and thus cannot make progress. Deadlock is the ultimate form of starvation; it occurs when two or more threads are waiting on a condition that cannot be satisfied. Deadlock most often occurs when two (or more) threads are each waiting for each other to do something (Sun Microsystems, 1995). In a flexible network with thousands of entities it becomes nearly impossible to control the stability of a multithreaded simulation model.
- Inflexibility of processing intelligence. In distributed networks, remote method invocation in conjunction with dynamic class downloading has facilitated dynamic transfer of business and routing logic between the distributed nodes in a network (Sing 2000). The process-oriented modeling capability prevents this runtime learning mechanism because processing and routing intelligence is not stored in the stations or nodes but in the entities flowing through the model.
- Scalability limitations. Multithreading using operating system threads is strictly limited by the underlying operating system. Many of the current implementations that use operating system threads are therefore limited to around 2000 entities in the model and consume large portions of the underlying processing power.
- Since component libraries such as Enterprise Java Beans that are used in actual information systems are not built upon a multithreading model, the interchange of libraries between these realtime information systems and simulation models is limited.
- Since threads are fundamentally related to their underlying operating system, entities in the model cannot be streamed over networks.

2.2 Current Java-based Simulation Languages Based on an Event Based Worldview

Another worldview adopted by some discrete event languages is DES and its accompanying Future Event List on which object interaction is scheduled. Two of the core notions most Object-Oriented implementations of this worldview share are:

Objects involved in the simulation environment do not interact with each other by a direct method invocation, but schedule them by a constructing a simulation event. Such an event encompasses the execution time, a (remote) pointer to the object on which the method is intended to be invoked, the method itself and the arguments for the method to be invoked. This mechanism is referred to as scheduling method invocation.

Running a simulation model merely implies executing the time-sorted simulation events. The absolute simulation time therefore becomes the execution time of the last executed simulation event. Figure 2 illustrates this worldview by focusing on the way simulation events are stored (1) in the eventlist, ordered on time, retrieved (2) and finally executed (3) on the objects referred to in the simulation event. Examples of these simulation languages include Simkit (Buss 2001).

The advantages of a DES implementation are its single threaded approach, its increased flexibility (Buss 2001), and the lack of requirements on objects involved in the simulation.

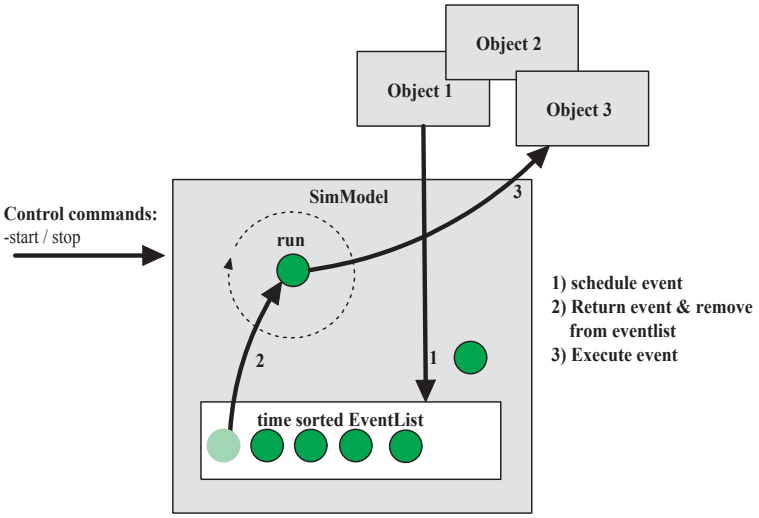


Figure 2: Event Based Simulation Core

3 ARCHITECTURE OF A FULLY DISTRIBUTED SIMULATION ENVIRONMENT

The architecture presented in this paper and illustrated in Figure 1 focuses on two remote network services. The first service is a DES-based remote simulation service which is described in subsection 3.1. The second service is a system representation service described in subsection 3.2.

Before focusing on the internal structures of these services, a notion must be made on the increased complexity of autonomous components in runtime. The basic question regarding this complexity is *"How are distributed clients, components, data sources, simulation services, etc. able to find each other in order to interact?"*.

The implementation presented in this paper made a first step in solving this issue by defining a asynchronous topic-subscription and lookup mechanism. The first step occurs when objects involved in the architecture present their characteristics and capabilities to lookup services.

The second step occurs when objects interested in others define their interest by creating a topic object and presenting this inquiry to a lookup service. Based on this topic object, lookup services are able to return all objects matching the characteristics described in the topic. More information on how objects initially find these lookup services can be found in (Sing 2000).

3.1 Discrete Event Simulation Implemented as a Remote Network Service

The simulation environment provided in this paper has fully adopted DES as its underlying worldview. Distribution formed once again the basis for not using current implementations but to design a new architecture.

As stated the disadvantage of DES-based simulation languages is the increased complexity in model construction. Most other implementations ease this complexity by one or more pragmatic design patterns. Most of these patterns conflict though with a truly distributed architecture.

Examples of these patterns are a static eventlist, relative time scheduling, non-serializable reflection classes, the lack of overloading primitive attributes, and the requirement to extend from an abstract simCoreObject which prevents interchange of simObjects and non-simulated components.

Concluding one can state that the simulation service is merely a remote service on which all objects are able to schedule their interaction. There are no requirements to objects either scheduling or being scheduled. Helper classes are available to facilitate overloaded methods with primitive and wrapped arguments.

3.2 System Representation Implemented as a Remote Network Service

The introductory notes on simulation use for complex systems engineering has already revealed minimal requirements for system representation. These can be summarized as follows:

The representation of a (modelled) system should be made available for *multiple stakeholders*, that *interact concurrently* with that system from a variety of possibly *remote* locations.

However, some additional observations specific for system representation can be made.

A first observation is that a stakeholder interacting with a system (model) is usually not interested in all the aspects of the system. This is especially true when large-scale,

complex systems are considered. This implies a need to 'browse' the system (Spence 2001).

Secondly, a user may interact with a model in several modes. We characterize the interaction as *passive*, if the user does not interact with the model during a run. In this mode the user is still able to dynamically define his area of interest (e.g. to 'browse' the system representation). However, these actions will not change the outcomes of the simulation run. The interaction is characterized as *active*, if the user is able to interact with the model during a run. The user's actions will then influence the actual outcomes of the simulation run. The distinction between passive and active model interaction has consequences for the relation between representation and simulation. With passive interaction, the representation data only needs to be generated once by a simulation run. After that, many users could concurrently iterate over the generated representation data. Given the composition of the user group and the ICT resources available, the representation data could be distributed among the users in several ways, given criteria like response time and effective and efficient resource usage. With active interaction, in which user interaction causes different simulation outputs, there is a tight link between the simulation execution and the representation presented to the user. This link becomes even more tight, when multiple users are interacting with the model concurrently. Ideally, the representation subsystem should be able to support both modes of interaction.

Thirdly, system representation as such does not *require* to be based on a simulation only. Instead, by separating the system representation from the simulation components, various combinations between real systems (e.g. information system) and simulated systems become possible. The representation presented to the user may be generated in each of them.

Finally, the remote aspect of system representation poses some challenges of its own. The devices and networks used by the various users may be as diverse as the users themselves. They may range from smart mobile phones up to powerful graphical workstations and use high speed LANs or low speed modems. As a consequence, a user might need to adapt the nature of the system representation to the actual communication and information processing capabilities available. Such adaptions are possible, since state change data can be represented in various ways. Information on the movement of a transport vehicle might for example be represented by a text message, but could also be represented in a 3D virtual world. Obviously, these representation modes pose different requirements on the communication and information processing resources available. It follows that the (modeled) system should somehow make these different kinds of representation available.

The system representation service has been developed with these notions in mind. A first proof-of-concept implementation has already been implemented. This implementation enabled various stakeholder to represent and simulate an actual, real-time supply chain. A screenshot is shown in Figure 4. In the remainder of the section, the representation service and the accompanying representation process will be described.

Figure 1 already introduced the relationship between model, simulation and representation components. Several of these components have a role in the system representation process:

- *Model components* represent the state of the system. They are sources of state change updates (events), which encompasses the basic data needed to update their system representation. The information system shown in Figure 1 is one potential source for state update events.
- *Representation components* enable clients to retrieve various *event handlers*, thus implementing the representation service. Event handlers can transform incoming state change updates into (visual) representations and may provide interaction with an underlying model component.
- *Clients* contain *representation contexts* used by event handlers to provide representations to the user. For visual representation such a context is a graphical region where the event handler can paint. The clients are able to find the other components using the lookup mechanism described previously.

The arrows in Figure 1 also show some interactions that take place between the various groups of components. This interactions are described in more detail in Figure 3.

The left side of Figure 3 illustrates the representation update process. Model components broadcast events to interested clients, that place received events in a local visualization event-list. Conceptually, the state update event-list does not need to reside client-side. Several clients could share one server-side event list. It may be noted that this event list contains state update events, in contrast with the simulation event list, that contains events triggering an action. Since they both contain time-ordered events they are similar components. The client-side dynamic representation process is then controlled by an EventController component, that iterates through the event-list usually in a time-ordered manner. Such a controller object may itself be controlled by some kind of timer, that updates the representation time of the system. By changing the properties of this timer, the user is able to control the speed and period of the system representation.

The process described thus far provides a mechanism to process a state-update event on the correct representation time. However, since these events basically only contain the state-change information itself, additional components are

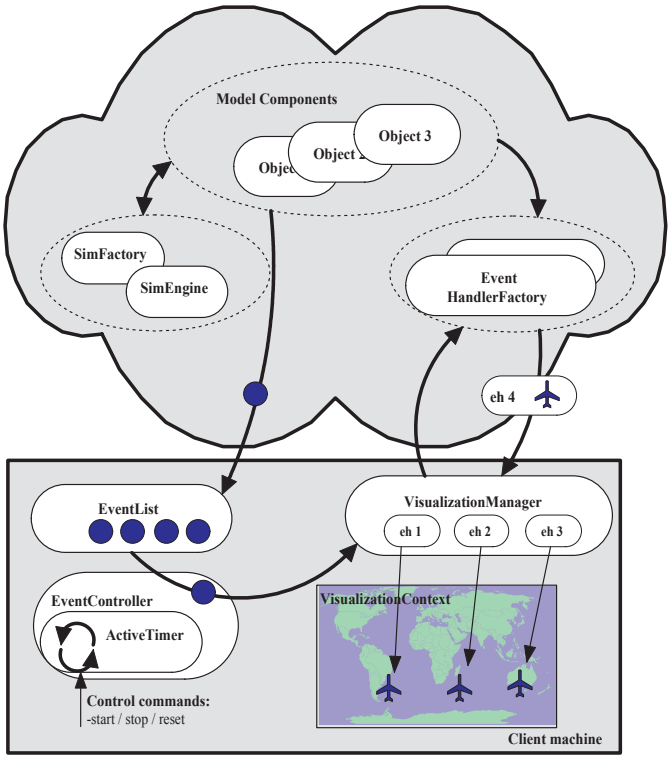


Figure 3: A-synchronous Distributed System Representation

needed to transform this information in an actual, dynamic representation. This is illustrated on the right side of Figure 3, by the interaction between the client-side visualization manager and the representation components in the network. Initially, the client has no event handlers available. When an event needs to be processed for which no event handler is yet available, the client tries to retrieve this event handler from the representation components, based on:

- The event's source model component
- Local representation contexts available
- Local information & communication processing resources available.

As a result, an event handler component is returned, that is able to process the incoming state-update event.

The event handler may consist of a controlling part and a representation part. The representation part shown in Figure 3 is a graphical representation of an airplane, that will be added to the client's visual representation context. The controlling part of the event handler can be an intelligent component: the movement of the airplane representations shown in Figure 3 could be extrapolated by the event handler until the time the next 'move' event arrives for that airplane. This mechanism reduces the amount of events that needs to be communicated to represent dynamic processes. This controlling part may also be enabled to interact with the original model component that created it. In this way active user-system interaction can be implemented.

4 OBTAINING THE SOFTWARE

The latest version of the software including its code, can be downloaded from the web at *http://www.simulation.tudelft.nl/research/d-sol/index.html*. License information, tutorials and examples can be downloaded and reactions are welcome.

5 DISCUSSION AND CONCLUSIONS

First of all this paper has resulted in a *"ready to use"* implementation of both a remote network simulation service as a remote network visualization service. The architecture presented here has shown the endless possibilities in both the distribution of simulation models as the integration and interaction of these models with realtime information systems.

Several topics remain though for further research and implementation. First of all the implementation of architecture presented in this paper has not yet completed a basic library of ready to use simulation components. Neither has the current implementation resulted in a graphical model environment. Both issues a very welcome for rapid model development and will become available soon.

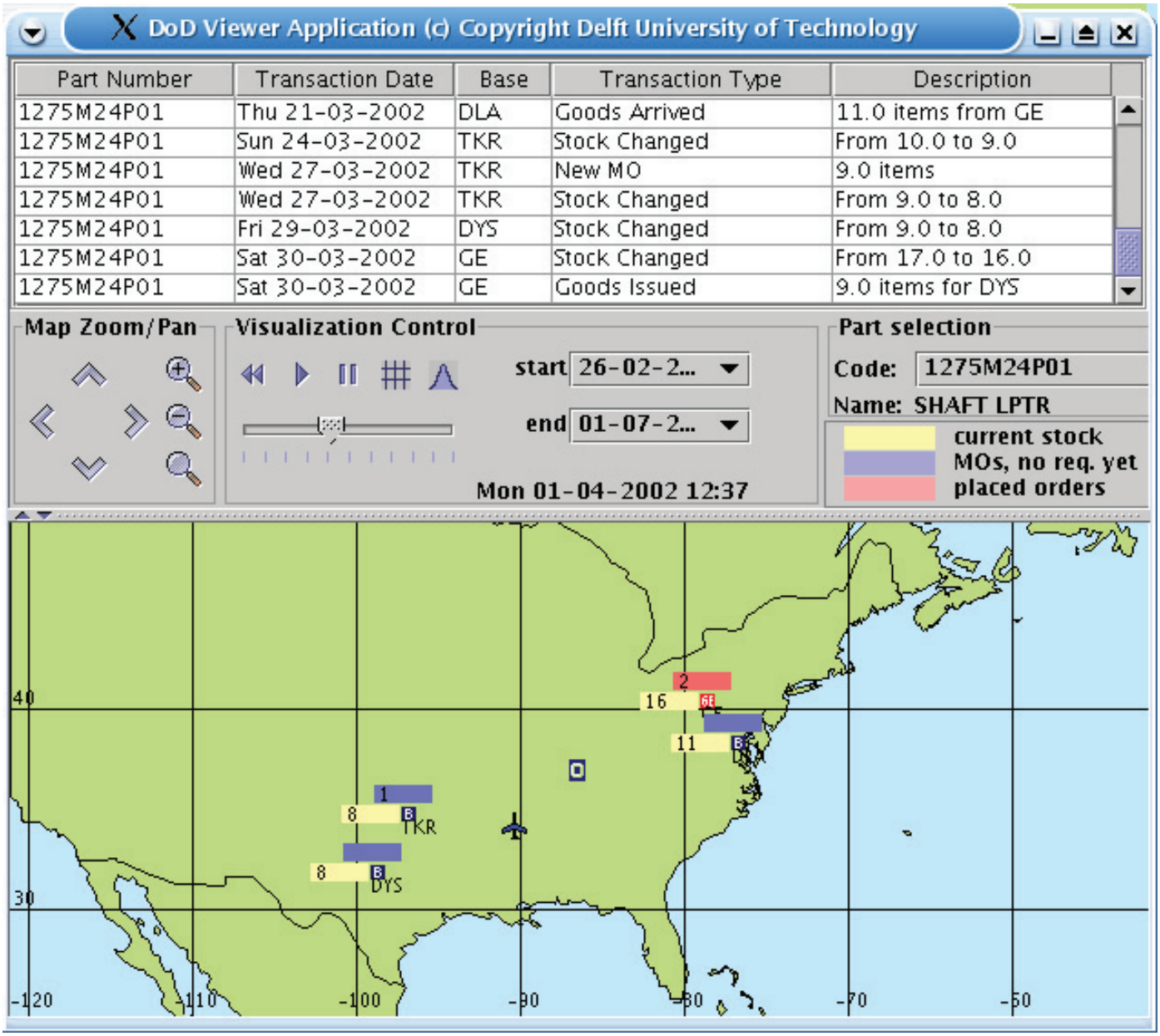


Figure 4: A Proof-of-concept Implementation for a Simple Supply Chain

As described the architecture has only partly resolved the *"searching and browsing"* issues by implementing a flexible topic-subscription model. These issues remain complex though. Bounded searching, federative grouping, flexible boundaries, and selective pattern recognition are still unreached.

A second research topic encompasses *time synchronization* between remote services. With respect to different simulation services, synchronization has not been touched by this paper.

Though only partly mentioned in this paper, synchronization also remains unsolved between visualization and simulation services. The lack of this synchronization has lead to visualization as *"replaying"* the simulation while a controlled synchronization might lead to distributed interaction or gaming.

Especially due to conflicts resulting from distributed concurrent model interaction, for now gaming remains beyond the scope of our horizon.

REFERENCES

Banks J., 1998 *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*.

Buss A.,2001 *Discrete Event Programming with Simkit*, published in Simulation News Europe.

Fishwick P.,1995 *Simulation Model Design and Execution: Building Digital Worlds*, Prentice Hall, New York.

Garrido J.M., 2001 *Object-Oriented Discrete-Event Simulation with Java*, Kluwer Academic/Plenum Publishers, New York.

Healy K.J. and Kilgore R.A.,1998 *Introduction to silk and java-based simulation*.

Page E.H., Moose, R.L, and Griffin, S.P.(1997) Web based simulation in SimJava using Remote Method Invocation, *Proceedings of the 1997 Winter Simulation Conference*, ed. A. Andrattotir, K.J. Healy, D.H. Withers, and B. L. Nelson. Institute of Electrical and Electronics Engineering, Piscataway, New Jersey.

Shannon, R.E., 1975, it Systems Simulation: The Art and Science, Prentice Hall, Englewood Cliffs, NJ.

Sing L., 2000 *Professional Jini*, Wrox Press Ltd, Birmingham UK.

Spence R., 2001 *Information Visualization*, Addison-Wesley, Essex UK.

Sun Microsystems, 1995 *Synchronizing Threads* <http://java.sun.com/docs/books/tutorial/essential/threads/multithreaded.html>.

AUTHOR BIOGRAPHIES

PETER H.M. JACOBS is a Ph.D. student at Delft University of Technology. His research focuses on coordination mechanisms of distributed loosely coupled networks. His e-mail address is <p.h.m.jacobs@tbm.tudelft.nl>.

NIELS A. LANG is a Ph.D. student at Delft University of Technology. His research focuses on using ICT's information and communication processing capabilities to support multi-actor decisionmaking processes on complex, dynamic systems. His e-mail address is <n.a.lang@tbm.tudelft.nl>.

ALEXANDER VERBRAECK is an associate professor in the Systems Engineering Group of the Faculty of Technology, Policy and Management of Delft University of Technology, and a part-time full professor in supply chain management at the R.H. Smith School of Business of the University of Maryland. He is a specialist in discrete event simulation for real-time control of complex transportation systems and for modeling business systems. His current research focus is on development of generic libraries of object oriented simulation building blocks in C++ and Java. Contact information: <a.verbraeck@tbm.tudelft.nl>.

Proceedings of the 1999 Winter Simulation Conference
P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds.

SIMULATION IN AN OBJECT-ORIENTED WORLD

Jeffrey A. Joines
Stephen D. Roberts

Department of Industrial Engineering
Campus Box 7906
North Carolina State University
Raleigh, NC 27695-7906, U.S.A.

ABSTRACT

An object-oriented simulation (OOS) consists of a set of objects that interact with each other over time. This paper provides a presentation of OOS design elements by contrasting OOS with its procedural counterparts. The elements of component technology is addressed along with the important issue of composition (components) versus inheritance that distinguishes object-based from object-oriented languages.

1 SIMULATION SOFTWARE CHALLENGE

There has been tremendous growth in the capability of computing hardware during the past three decades. An important question for simulation modelers is how can simulation can take full advantage of the computing power now available. Software engineering provides part of the answer. However, writing software "from scratch" is no longer advisable since software systems tend to be complex and several libraries exist for many of the common functions. Thus, simulation models need to include more than computational efficiency if they are to have wider utility and acceptance in a multi-media, virtual reality, and graphical interface software world.

Modeling and software have had a symbiotic relationship. The computer is more than a computational engine for simulation algorithms and should be regarded as a tool also for modeling. The technology of simulation is now a mature and developed methodology. Although there is plenty of room for additional research on fundamental areas (e.g., random number and variate generation, the next event simulation process, reliable and appropriate statistics), there are now widespread adoptions and use of computer simulation techniques.

However, the real limits on the future adoption of simulation may rest on our ability to represent complex systems and to do it easily, which can be construed as a matter of modeling style. The purpose of this paper is to describe ways to improve modeling style – through object-oriented simulation and to describe the fundamentals of object-oriented simulation. It is useful, however, to first consider the matter of programming style – modeling style usually follows programming style.

2 PROCEDURAL STYLE

As long as you can specify statement sequences, define variables, do branching, perform iteration, and have I/O, you can do everything a Turing machine can do which in turn means everything a computer can do. Thus the distinction in programming style is not what can and can't be done, but what can be done easily. However, what can be done easily may be a matter of judgement.

Early programs were long sequences of labeled statements (simulation instructions) where movement of program control used labels. Ironically, many fairly recent simulation languages maintain this same approach.

Owing to the repetition of some logical procedures, functions or subprograms were added to programming languages. To give these functions generality, argument lists were added that could change the computation within a function from call-to-call. Using functions to subdivide a programming problem gave rise to the notion of "functional decomposition," which remains today a popular approach to programming and simulation modeling. In fact, subnetworks are a recent form of functional decomposition.

GPSS, SLAM, and SIMAN were the simulation versions of the "library" approach to simulation, but the libraries were invoked much differently and invisibly to the user. Instead of writing general purpose programming code, users constructed text files containing a sequence of simulation "instructions." This approach provided a higher level of abstraction than programming in a low-level language, making it easier to model complex systems.

Simulation language instructions generally have a direct correspondence to a form of a flowchart (also called

network). While such input makes it easy to specify the simulation, it limits the direct impact the modeler can have on the execution of the simulation, since these simulation instructions did not constitute a programming language. Instead they are generic model templates for simulation.

Consequently, many models written in earlier versions of these languages were highly augmented by general programming code containing function calls to the simulation libraries, since function calls could not be directly invoked and users could not write functions with the native simulation instructions. In particular, Visual SLAM continues to promote extensive use of programming "inserts" with Visual Basic and C.

It is important to note that while simulation "languages" like GPSS, SLAM, and SIMAN were easy to use, although limited, there were more powerful simulation programming alternatives. For example SIM-SCRIPT and SIMULA were full programming languages with simulation functionality built into the language grammar and syntax. Using these languages, users "programmed" the simulation. SIMULA was not widely appreciated at the time as a simulation language but would, in fact, form the basis and motive for much of the modern object-oriented paradigm.

In all cases (except for SIMULA), the style was procedural based. A problem was decomposed into procedures and either represented by general components, like a queue, or represented in programming code with a data structures and code. Procedural programming represents today a fundamental style of programming usually learned in the first exposure to programming or modeling.

There are several fundamental problems with using the procedural style of modeling and simulation. Procedures do not correspond to real world components. Instead, they correspond to methods and algorithms. Therefore acts engaged by entities must be given a context for procedures to be easily specified. Many simulation contexts are based on networks of queues (often complicated queuing situations). The modeling approach is to let the queuing network create the procedural structure that is traversed by entities. When this structure is appropriate, as it often is in a manufacturing or communications application, the model is a convenient analog to the real system.

However modeling languages are limited when confronted with complicated circumstances, such as the need to code an algorithm that creates a schedule based on anticipated volume and current use of facilities. It is then that the need for general programming manifests itself. There is a fundamental difficulty in communication between the simulation code, provided by a simulation vendor, and user code from a general programming language. The only means of communication is generally through global data exchange or function calls. These mechanisms are vulnerable to inappropriate use and were dangerously visible to users.

Perhaps the greatest limitation of the procedural style is its lack of extensibility. From the earliest simulation languages until the early 1990s, the only way to adapt these simulations was through functional extension. In other words, you could add structural functionality to the simulation but not alter any of its basic processes, like giving properties to resources. For example, if you needed the simulation to include a bridge crane, you had to program it completely yourself or model it with the features available. One of the reasons for this lack of extensibility was that procedural changes were the only approach to model changes. Specifically, vendors had no way to partially hide implementation details and were either forced to give access to source code or restrict the access to the features. A module or file provided a form of encapsulation (which more recent simulation languages call templates or subnetworks), but these collections do not provide for autonomous objects.

3 OBJECT (COMPONENT) STYLE

It is very easy to describe existing simulation languages using object terminology. A simulation language provides a user with a set of pre-defined object classes (i.e., resources, activities, etc.) from which the simulation modeler can create needed objects or components. The modeler declares objects and specifies their behavior through the parameters available. The integration of all the objects into a single bundle provides the simulation model. Component is another word used for object.

Therefore, an object can be described by an entity that holds both the descriptive attributes of the object as well as defines its behavior. The class concept evolved out of the notion of encapsulation, an idea that originated in SIMULA. However SIMULA viewed objects as much more than encapsulation. Objects needed independence of action and a means to hide their implementation details, yet provide an interface for their use. Further, there needed to be way to construct objects and to communicate among them. C++ borrowed all these ideas from SIMULA (as did Smalltalk) and put them into the procedural programming language C. We use C++ to illustrate the object style.

3.1 An Example: Exponential Random Variable

Suppose you are modeling an exponential random variable in a simulation. The random variable may be described by a standard exponential statistical distribution, which has a set of parameters, a mean in this case. This mean would be considered an attribute of the exponential random variable component. It may be important to obtain observations from this random variable via sampling. One may want to obtain antithetic samples or to set the random seed. Sampling from the exponential random variable defines a particular behavior. Behaviors are more often called "methods."

3.2 Encapsulation

The entity "encapsulates" the properties of the object because all of its properties are set within the definition of the object. In our example, the exponential random variable's properties are contained within the definition of the random variable so that any need to understand or revise these properties are located in a single "place." Any users of this object need not be concerned with the internal "makeup" of the object.

In C++, the keyword "class" is used to begin the definition of an object followed by the name of the object class and then the properties of the entity are defined within enclosing {}. For example, the following object defines the `Exponential` class.

```
class Exponential{
…// Properties of the Exponential
};
```

Without encapsulation, properties could be spread all over, making changes to the object very difficult. Encapsulation also facilitates the ability to easily create multiple instances of the same object since each object contains all of its properties. For example, common random streams can easily be applied since each `Exponential` object contains its own individual random number stream.

3.2.1 Class Properties

The class definition specifies the object's properties, the attributes and methods. The attributes define all the singular properties of the object while the behaviors define how the object interacts within the environment with other objects. Attributes are considered the data members of an object. In the case of our `Exponential` random variable, its mean (given by the identifier `mu`) would be a real number attribute.

```
double    mu;
```

Other attributes would be similarly defined.

The methods of an object represent actions the object can perform or take. For example, if the exponential random variable needed to obtain a sample, the following member function can be used:

```
double   sample(){
   return –mu * log( 1.0 – randomNumber() );}
```

where the `randomNumber()` function yields a uniform random variable between 0 and 1. By representing methods with functions, the object can react to parameters passed in the function argument as well as change variable values within the function.

3.2.2 Classes and Instances

Notice, the word "class" not "object" is used in defining the object, which can be confusing, since it would seem that we are defining objects. Lets consider the more complete definition based on our prior discussion of encapsulation and properties (ignore the "public" for now), the `Exponential` class is defined as follows.

```
class Exponential{
   public:
     double mu;
     double sample(){ return –mu * log( 1.0 –
                        randomNumber() );}
};
```

Rather than defining an object directly, a class is defined where the class provides a "pattern" for creating objects and defines the "type." By defining a class (of objects), rather than a single object, the opportunity exists to use the class to create many objects (i.e., re-use existing code). Furthermore, as seen later, the class is a description of a pattern for constructing objects which can be easily extended. Now, objects can be created directly from this class once defined. These created objects are called "instances" of a class. For example, `serviceTime` is an instance of the `Exponential` class.

```
Exponential   interarrivalTime, serviceTime;
```

3.3 How Do Objects Communicate?

An OOS models the behavior of interacting objects over time. However before we can consider a simulation, we need to understand how objects interact or communicate with each other. The interaction among objects is performed by communication called "message passing." One object sends a message to another and the receiving object then responds. An object may simply publish a message that may be responded to by one of several objects. For example in a bank simulation, a customer arrives at a bank and may be served by any of several tellers. In a O-O context, the customer publishes their arrival and waits for service by a teller. There are several ways in transmitting messages in an object-oriented program and it depends on the programming language.

3.3.1 Directed Message

Perhaps the simplest form of message passing is a direction to the object's attributes or data members. For example, if the `interarrivalTime` object needed to have a mean of 5.5, then we could write:

```
interarrivalTime.mu = 5.5;
```

This message causes the object to receive the value and set its variable `mu`. It is a forced message because the object has no choice but to perform the action.

3.3.2 Data Methods or Functions

Rather than forcing a value upon an object, a value could be communicated to the object and then let it decide how to deal with the value. For example, if a new "member function" or data method to the Exponential class called `setMu()` was added as:

```
void setMu( double initMu ){ mu = initMu; }
```

Now the object is sent the `setMu` message with a message value of 5.5 which "communicates" our interest in changing the mean and `interarrivalTime` receives the message and changes its internal value of `mu`.

```
interarrivalTime.setMu(5.5);
```

Although this example really does the same thing as the direct reference, there are important distinctions. First, in our function call we simply "passed" the value of 5.5 to the object. Second, we didn't tell the object how to change the attribute `mu`. The object's function written by the designer of the `Exponential` class causes the mean parameter to change. The user of the function does not need to know how the function inside the class works. In fact, the class designer could change the internal name of `mu` to `expMean` within the class, and all exiting user code would remain the same. Further, the same message can be made to respond to several different message value types, a feature often referred to as "polymorphism."

3.3.3 Pointers

Another way to communicate is indirectly through pointers that are simply addresses of the location of an object. For example, a pointer to the `interarrivalTime` object can be created and the setMu message can be sent via the pointer.

```
Exponential * rnPtr = &interarrivalTime;
rnPtr->setMu(5.5);
```

Pointers have the advantage of not needing to know the particular object ahead of time, but only the address of the object. Thus, if we change the pointer to point to the `serviceTime` object, the format of the message remains the same. With a more complex message, use of pointers becomes very convenient.

3.4 How Are Objects Formed?

In our example, the exponential object has no ability to be created with different means. Instead, the object's mean was changed to a specific value. Although an object can be instantiated from a class without special instructions, often we want the creation to accomplish certain objectives. Likewise, we also might want to do something special when an object is destroyed.

3.4.1 Constructors and Destructors

Special member functions can be defined that act when an object is created and destroyed, which are called constructors and destructors, respectively. The constructor is recognized by having no return type and the same name as the class. For example, the following could be a constructor for the exponential object.

```
Exponential( double initMu ){ mu = initMu; }
```

This function accepts the invocation argument and sets the internal mean to it. An object whose initial mean is 4.3 can be specified upon creation as follows.

```
Exponential   serviceTime(4.3);
```

In C++, functions can be "overloaded" so that they differ only in their formal arguments (i.e., "polymorphism"). Therefore, a class can have multiple constructors. For example, if we wanted the exponential to accept an integer specification of its mean.

```
Exponential( int startMu ){ mu = startMu;}
```

Now, exponential objects with either a double or an int as arguments can be specified (actually C++ will make appropriate conversions among its built-in types but this example illustrates the way a user could provide conversions among user-defined classes). The following creates two objects using different argument types.

```
Exponential   arrival(9.3), inspect(6);
```

Users can also define a special member function called a destructor that acts when the object is destroyed. Only one destructor can be defined since it has no arguments. For example, a destructor for the exponential class has the following form.

```
~Exponential(){// print out how often used? }
```

3.4.2 Visibility of Properties

It should be clear that a user of a class does not really need to know the internal workings of the class. For example, they do not need to know what algorithm is used to obtain the sample (they may want to know for their own assurance). Furthermore, the designer of the class may not want the user of the class to know everything about the class. Thus, the class designer has the option of causing properties of the class to become invisible to users of the

class and to provide a public interface to those hidden properties. The two most frequently used labels are "public" and "private." Properties within a class that are public can be accessed directly by a user while those that are private are available only to the designer. For example, the variable containing the mean is made private within the class to prevent improper use (i.e., direct manipulation). Our class would then look like the following.

```
class Exponential{
 private:
  double mu;
 public:
  Exponential(double initialMu ){mu=initialMu;}
  double sample();
  void setMu( double changeMu ){mu = changeMu;}
  double getMu( ) { return mu; }
};
```

Now `mu` cannot be changed directly by a user. Thus the direct reference to `mu`, as done earlier, will fail. Communication to the exponential objects must be performed through member functions. The designer of the class can now protect the class data members from unwanted changes while the user of the class is unaffected.

3.5 How Are Objects Formed From Others?

One of the fundamental benefits of an O-O design is the ability to make other objects out of existing ones. We have already seen how to design a class of objects using the built-in types from C++. Suppose the following random number class has been defined which generates uniformly distributed numbers between 0 and 1.

```
class RandomNumber{
      long   seed;
   public
      RandomNumber( long seed = -1);
      void setSeed(long sd){seed=sd;}
      virtual double sample();
};
```

In this definition, the constructor argument can be specified or left blank to default to their initial values (i.e., -1 means use the next seed). The public member function `sample()` is used to obtain a sample and we will assume that the seed will be updated appropriately with each call. The "virtual" keyword will be discussed later.

There are two ways this random number generator could be used with our `Exponential` class. The first method is called **composition**, in which a random number object is included within the exponential class. The second method of using the random number generator is through **inheritance** which makes the exponential class a kind of random number. Inheritance is one of the major features that distinguish a "object-based" language from a true "object-oriented" one.

3.5.1 Composition

First, consider the case of composition where we simply compose the new class from the existing class:

```
class Exponential{
   private:
      double mu;
      RandomNumber rn;
   public:
      void setSeed(long sd){rn.setSeed(sd);}
… //Public Properties
};
```

Notice that the `Exponential` is defined simply to "have" a `RandomNumber`. In O-O parlance, the relationship between the `Exponential` and the `RandomNumber` rn is called a "has-a" relationship. The data member rn is used in the `sample()` function of the exponential. Notice, a `setSeed()` needs to be defined in order to access the one in the random variable.

3.5.2 Inheritance

The second kind of relationship among classes is called an "is-a" relationship and is based on inheritance or a parent-child relationship. In our example, the exponential can be considered a kind of random variable. It would be useful for the `Exponential` to be a child of `RandomNumber` and thus inherit all the random variable properties. Hence, what could be done to the random variable could also be done with the exponential. No additional `setSeed()` is required since the one in the random class can be used.

For example, sometimes a sample from an exponential is needed while other times a basic uniform generator is required. Suppose the following two objects and pointer are defined:

```
RandomNumber   uni;
Exponential   exp(5.5);
RandomNumber * pRN = &uni;
```

If at an activity in our simulation, a sample from a random variable is needed, the following message is sent to obtain an activity time.

```
pRN -> sample();
```

However, because Exponential is also a `RandomNumber`, the pointer `pRN` could be assigned to either an `Exponential` or a `RandomNumber` and the same message applies. In the **composition** example, two separate activities would be required (i.e., one which used an exponential and another one which used a uniform).

```
pRN = &exp;
```

In a true O-O language with inheritance, the message would be sent to the proper object and the sampling would be from the correct sampling function. In O-O terms, determining which variate to sample at run-time is called "run-time" binding and is performed by specifying the `sample()` to be "virtual" in the parent class.

To specify that `Exponential` inherits from `RandomNumber`, the header for the class definition would be modified as:

```
class Exponential: public RandomNumber{...
```

showing that `RandomNumber` is the parent and its visibility is "public".

Under inheritance, the child class inherits the public (and protected) properties of the parent. Now these properties are directly available to the child class and the class type resolves any conflicts. C++ also permits multiple inheritance, meaning a child can inherit from several parents.

4 OBJECT-ORIENTED VS. OBJECT-BASED

Because many simulation languages offer pre-specified functionality produced in another language, the user cannot access the internal function of the language. Instead, only the vendor can modify the internal functionality. Also, users have only limited opportunity to extend an existing language feature. Some simulation languages allow for certain programming-like expressions or statements, which are inherently limited. Most languages allow the insertion of procedural routines written in other general-purpose programming languages. None of this is fully satisfactory because, at best, any procedure written cannot use and change the behavior of a pre-existing object class. Also, any new object classes defined by a user in general programming language do not co-exist directly with vendor code.

4.1 Object-Based Extension

The object-based approach only allows extensibility in the form of composition (i.e., new objects can only be created out of existing objects). The simple `Event` object will demonstrate the limitations of extensibility only through composition. The Event object is used to move the simulation from one time to the next. Events are placed on the calendar and, when an event is removed from the calendar, the `processEvent()` function is called to handle the event. The following gives a portion of the `Event` class that can be used to process arrival of entities into the network and end of service events. Notice that depending on the type of event, the appropriate event handling function is called. This is an example use of composition.

```
class Event{
private:
    double eventTime, eventType;
    Source *source;
    Activity *activity; //… More properties
public:
    void processEvent(){
    select EventType{
    case ArrivalEvent:
        source->newArrival(Entity);  break;
    case EndofService:
        activity->endofService(Entity) break;}}
    //… Additional Properties
};
```

If the user wants to add additional events (e.g., a monitor event), it would require the designer of the `Event` class to add an appropriate data member, data methods, and then provide an additional case statement. Therefore, the designer has the impossible problem of anticipating every kind of event any user might need. Extensibility through composition only allows users to create new objects out of existing ones.

4.2 Object-Oriented Extension

An object-oriented simulation deals directly with the limitation of extensibility by permitting full data abstraction. Data abstraction means that new data types with their own behavior can be added arbitrarily to the programming language. When a new data type is added, it can assume just as important a role as any implicit data types and can extend existing types. For example, a new user-defined robot class can be added to a language that contains standard resources without compromising any aspect of the existing simulation language, and the robot may be used as a more complex resource. There are two basic mechanisms in C++ that allow OOS to provide for extensibility: **inheritance** and **genericity**.

4.2.1 Inheritance

Inheritance allows classes to exploit similarity through specialization of parent classes (i.e., child classes inherit the properties of the parent and extend them). All event types have an associated `eventTime` and `eventType` and the appropriate data methods to specify these properties. Therefore, specific event types would inherit these properties and provide additional ones (see Figure 1). Now, the class designer only has to provide the mechanism to extend the key classes.

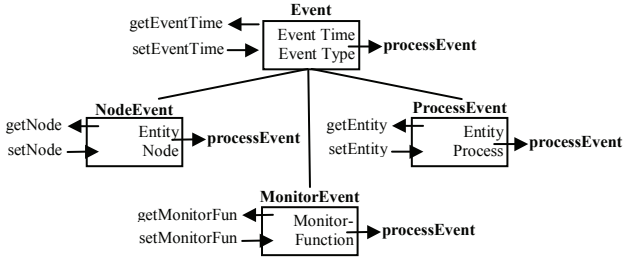


Figure 1: Inheritance Hierarchy

For example, `NodeEvent`, which provides events that occur at nodes (e.g., end of service at an activity), provides a pointer to the `Node` of interest and the `Entity` which caused the event. The `processEvent()` is declared virtual so that the appropriate `processEvent` is fired when the event is pulled off the calendar (i.e., run-time binding). The Event's `processEvent`() is a pure virtual function meaning any child classes must re-define it. The `NodeEvent`'s invokes the nodes `executeLeaving`() (another virtual function in the node hierarchy).

```
//Event's processEvent
void virtual processEvent() = 0

// ProcessEvent's processEvent
void virtual processEvent(){
  processPtr->executeProcess(entityPtr);}

//NodeEvent's processEvent
void virtual processEvent(){
      nodePtr->executeLeaving(entityPtr);}
//ExecuteLeaving -virtual function in Node
```

Now the designer does not have to anticipate every type of event. Users have the ability to define their own events provided they inherit from an existing event class and provide an appropriate `processEvent`() function. Given a pointer to an event, the simulation will invoke the appropriate event's `processEvent`() function at run time. Unlike Java, C++ provides for multiple inheritance that facilitates a very useful and powerful feature with some subtle idiosyncrasies. Multiple inheritance allows you to combine the collection of data and behavior of several classes. For example, when modeling a textile distribution network, there are nodes that are vendors, distribution centers (DCs), and stores. Vendors are suppliers that ship garments to consumers while stores are strict consumers that receive shipments. However, DCs are considered both suppliers and consumers (i.e., DCs can supply other DCs and stores while receive shipments from other suppliers (either DCs or vendors)). In a single inheritance hierarchy, the designer must repeat similar code for either the supplier or consumer behavior or force an unnatural inheritance hierarchy.

4.2.2 Parameterized Types

Even with inheritance, many O-O languages like Java and Smalltalk can still be limiting in terms of extensibility. Eiffel and C++ provides an additional method of extensibility called genericity or parameterized types (i.e. templates). Parameterized types are special forms of composition that exploit commonality of function. For example, most simulations would declare a source object that is used to place entities into the network. In an OOS environment, the user may want TVs or Orders to arrive rather than generic entities. The user can create several different source nodes by inheriting from the base Source class as seen in Figure 2. Each of the new classes defines a new type of object to be created (i.e., TV, Order) and the "virtual function" `executeLeaving`.

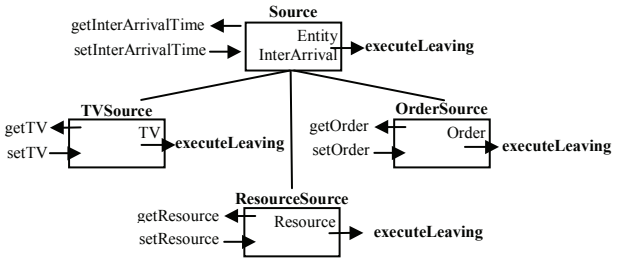


Figure 2: Inheritance Hierarchy versus Commonality

Notice, only the `Interarrival` object and methods are re-used in the child class. Each of the child classes must define its own `executeLeaving`() when the only difference is the type of object released into the network. When objects provide the same functionality, parameterized types are used (see Figure 3.). Now, the user specifies the type of entity to be released into the network and all remaining code is used. This ability is further demonstrated when a user wants to add statistics to the source node. The user only has to inherit from one class rather than create a `TVSourceStat`, `OrderSourceStat`, etc.

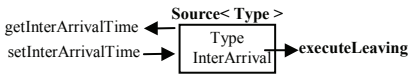


Figure 3: Parameterized Type

The following would declare two different source nodes.

```
Source<TV> tvSource(…);
Source<Order> orderSource(…);
```

5 CREATING A SPECIFIC OOS

A key to the creation of a fully integrated simulation package is the use of a *class inheritance hierarchy*. The formation of such a hierarchy is described in Joines and Roberts (1996). Object-based "frames" are used to collect classes into levels of abstraction. A *frame* is a set of classes that provide a level of abstraction in the simulation and modeling platform. A frame is a convenient means for describing various "levels" within the simulation class hierarchy and is a conceptual term.

While frames provide a convenient means to describe the levels of abstraction within the entire object-oriented simulation platform, another means of encapsulation is to place higher level complex interactions into "frameworks." For our purposes, *frameworks* are used to describe those collections of classes that provide a set of specific modeling facilities. The frameworks may consist of one or more class hierarchies. These collections make the use and reuse of simulation modeling features more intuitive and provide for greater extensibility. Special simulation languages and packages may be created from these object classes. For more information, see Joines and Roberts (1998b) in the creation of YANSL, which is just one *instance* of the kind of simulation capability that can be developed within an OOS environment.

5.1 Example Classes Specific to YANSL

Several classes are selected from the modeling frameworks (Joines and Roberts, 1998b) to create the YANSL modeling package. These classes are collected together to form a "simple" modeling/simulation language which can be extended to create more complicated features. The general simulation support classes, such as variate generation, statistics collection, and time management, are used indirectly throughout the modeling frameworks. The network concepts are somewhat enhanced, but are taken from the modeling framework. The node hierarchy for YANSL is shown in Figure 4.

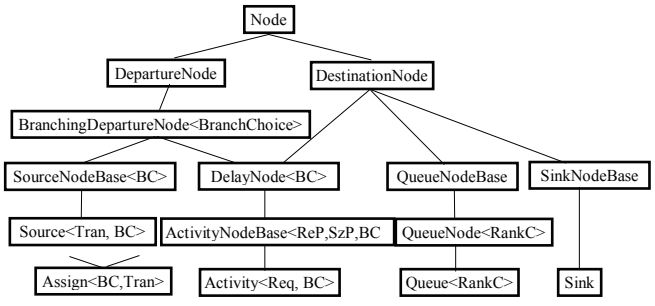


Figure 4: YANSL Node Hierarchy

The higher level nodes (`Assign`, `Activity`, `Queue`, `Source`, and `Sink`) are used directly by the YANSL modeler. Lower level nodes provide abstractions which are less specific, allowing specialization for other simulation constructs (e.g., the QueueNodeBase class excludes ranking and statistics). Sink and queue nodes can have transactions branched to them and are therefore destination nodes, while the source node is a departure node.

The delay and assign nodes are both departure and destination nodes, so they inherit from both the departure and destination node classes. Departure nodes may need a branching choice and called `BranchingDeparture Nodes`. An activity is a "kind of" delay but includes resource requirements. The properties of the YANSL nodes allow transactions to be created at source nodes, wait at queue nodes, receive attribute assignment at assign nodes, be delayed at activity nodes, and exit the network at sink nodes. Resources may service transactions at activity nodes. for YANSL, (see Figure 5) allows resources to be identified as individuals, as member of alternative groupings, or as members of teams.

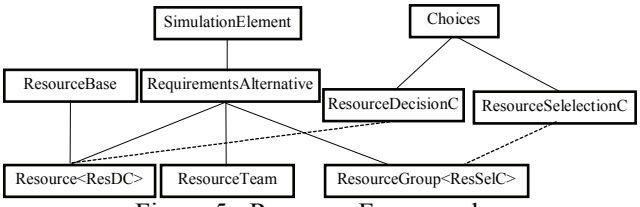


Figure 5: Resource Framework

The resource framework takes advantage of both inheritance and parameterized types. Activities actually request an `RequirementAlternative`. Because single resources, resource teams, and groups of resources inherit from `RequirementAlternative`, `Activity` nodes can except any type. If the three types are not suitable for your application, you can inherit from the three types (i.e., create a robot resource) or directly from `Requirement Alternative`. When there is a choice of resource service at an activity, then a resource selection method (i.e., a parameterized type) is used to select the individual resource among a group of resources `(ResourceGroup)`. The `Resource` object is parameterized with a resource decision object that allows the resource the ability to choose among several different queues to service (i.e., the resources are allowed to move through the network as well). The ability to request a resource service at run-time without specifying it explicitly is another example of polymorphism (e.g., the user my request either a single particular resource or a team of resources or select among a set of resources or teams). Owing to the extensibility through **inheritance** and **genericity**, the user has the ability to easily model complex resource decisions.

5.2 Modeling with YANSL

When modeling with YANSL, the modeler views the model as a network of elemental queuing processes (graphical symbols could be used). Building the simulation model requires the modeler to select from the pre-defined set of node types and integrate these into a network. Transactions flow through the network, can be assigned attributes, may require resources to serve them, and thus may queue to await resource availability. Unlike some network languages, resources in YANSL are active entities, like transactions, and may be used to model a wide variety of real-world items. The real power of a simple language like YANSL lies in the ability to extend and adapt the simulation language to directly model the complex problem rather than trying to adapt the problem to fit the modeling blocks given by standard simulation packages.

6 FINAL THOUGHTS

Modeling and simulation in an O-O language possesses many advantages. As shown, internal functionality of a language now becomes available to a user (at the discretion of the class designer). Such access means that existing behavior can be altered and new objects with new behavior introduced. The O-O approach provides a consistent means of handling these problems.

O-O systems view the world as a set of autonomous agents that interact or work together to solve some complex task. Each object is responsible for a specific task that helps one organize the complexity of complex systems which in turn simplifies the computer programming tasks. O-O designs yield smaller systems through the reuse of common mechanisms. They are more reliant to change and are better able to adapt over time. O-O designs greatly reduces the risk of building complex software systems because they are developed to evolve incrementally from smaller systems in which they have been tested for reliability and stability.

The O-O ideas have re-rooted in simulation, after being initiated by simulation through SIMULA. The Smalltalk environment is fully O-O and contains fully OOS. Obviously simulation languages based on C++, like C++/CSIM and C++SIM, possess all the object-oriented capability described in this paper. Simple++ and MODSIM III are further examples of object-oriented languages that employ most of these concepts within different simulation frameworks.

The queuing network based languages like Arena and AweSim have beginnings of object-based. Both languages provide a composition approach to creating network macros, through Arena templates and AweSim subnetworks. However neither are autonomous and independent objects in the sense described here and extensibility cannot be used to extend the active entities. Both have access to Visual Basic, which is itself only object-based. AweSim wraps its functionality in a few objects, whereas Arena contains a object model (not with Siman features) that is integrated with Visual Basic.

A simulation language called SLX from Wolverine Software provides a new object-based simulation product from the makers of GPSS/H. This language has all the object-based facilities but has none of the object-oriented facilities. It does contain an extended macro facility for adding statements and extended features for representing the simultaneous behavior of objects.

To take full advantage of object-oriented simulation requires more skill from the user. However, that same skill would be required of any powerful simulation modeling package but with greater limitations.

REFERENCES

Joines, J.A. and S. D. Roberts. 1996. Design of object-oriented simulations in C++. In *Proceedings of the 1996 Winter Simulation Conference*, ed., John Charnes, Douglas Morrice, Dan Brunner, and James Swain, 65-72. Institute of Electrical and Electronics Engineers, New Jersey.

Joines, J.A. and S. D. Roberts. 1997. An Introduction to Object-Oriented Simulation in C++. In *Proceedings of the 1997 Winter Simulation Conference*, ed., Sigrun Andradottir, Kevin J. Healy, David H. Withers, Barry L. Nelson, 78-89. Institute of Electrical and Electronics Engineers, New Jersey.

Fundamentals of Object-Oriented Simulation. In *Proceedings of the 1998 Winter Simulation Conference*, ed., Sigrun Andradottir, D.J. Medeiros, Edward F. Watson, John S. Carson, and Mani S. Manivannan, 141-150. Institute of Electrical and Electronics Engineers, New Jersey.

Joines, J.A. and S. D. Roberts. 1998b. Object-oriented simulations. In *Handbook of Simulation*, ed., Jerry Banks, 397-428. John Wiley & Sons, Inc. New York

AUTHOR BIOGRAPHIES

JEFFERY A. JOINES is a Research Associate in the Furniture Manufacturing and Management Center at NCSU. He received his B.S.I.E, B.S.E.E, M.S.I.E and Ph.D. I.E. from NCSU. He was the 1997 winner of the Pritsker IIE Doctoral Outstanding Dissertation Award. He is the WSC 2000 Proceedings Editor

STEPHEN D. ROBERTS is a Professor in the Department of Industrial Engineering at NCSU. He received his B.S.I.E., M.S.I.E., and Ph.D. from Purdue University. He was the recipient of the 1994 Distinguished Service Award. He has served as Proceedings Editor and Program Chair for WSC.

Lecture 7. Parallel and Distributed Simulation

Papers to study for the exam

- Richard Fujimoto. 2015. Parallel and distributed simulation. In Proceedings of the 2015 Winter Simulation Conference (WSC '15). IEEE Press, Piscataway, NJ, USA, pp. 45-59.
  Retrieved from: http://www.informs-sim.org/wsc15papers/004.pdf

- Kalyan S. Perumalla. 2006. Parallel and distributed simulation: traditional techniques and recent advances. In Proceedings of the 38th conference on Winter simulation (WSC '06), L. Felipe Perrone, Barry G. Lawson, Jason Liu, and Frederick P. Wieland (Eds.). Winter Simulation Conference. 2006. pp. 84-95.
  Retrieved from: http://www.informs-sim.org/wsc06papers/009.pdf

Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

PARALLEL AND DISTRIBUTED SIMULATION

Richard Fujimoto

School of Computational Science & Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA

ABSTRACT

Parallel and distributed simulation is a field concerned with the execution of a simulation program on computing platforms containing multiple processors. This article focuses on the concurrent execution of discrete event simulation programs. The field has evolved and grown from its origins in the 1970's and 1980's and remains an active field of research to this day. An overview of parallel and distributed research is presented ranging from seminal work in the field to address problems such as synchronization to recent work in executing large-scale simulations on supercomputing platforms. Directions for future research in the field are explored.

1 INTRODUCTION

The parallel and distributed simulation field emerged in the 1970's and 1980's from two distinct, overlapping research communities. On the one hand, the *parallel discrete event simulation* (PDES) community was concerned with accelerating the execution of discrete event simulations through the exploitation of high performance computing platforms. In approximately the same time frame the *distributed simulation* community formed growing out of research and development efforts in the defense community that focused on interconnecting separately developed simulations executing on computers interconnected via local and wide area networks. This research initially focused on simulations for training purposes, but quickly broadened to include areas such as analysis and test and evaluation of physical devices. While there are important differences between work in PDES and distributed simulation, there are also many common issues and problems. Here, we informally characterize *parallel and distributed simulation* as a field that encompasses issues arising from both of these communities stemming from the execution of individual simulation programs on platforms ranging from tightly coupled parallel computing platforms or loosely coupled machines connected via a wide area network.

2 PARALLEL DISCRETE EVENT SIMULATION

PDES is concerned with the technologies associated with distributing the execution of a single run of a discrete event simulation program across multiple processors in a high performance computing system. Such platforms include shared-memory multiprocessors and message-based cluster computers. The central goal of PDES is typically to accelerate the execution of the simulation.

A discrete event simulation captures the behavior of an actual or envisioned system over time. Unlike other simulations that operate in a time-stepped fashion and evolve the state of the system from one time step to the next, in a discrete event simulation changes in system state occur at distinct, typically irregular, points in simulation time. A sequential discrete event simulation program includes two fundamental concepts: state variables that capture the state of the system being modeled, and event computations or

978-1-4673-9743-8/15/$31.00 ©2015 IEEE

simply events that transition the state variables from one state to the next. In a discrete event simulation changes to simulation state occur *only* through event computations. Each event contains a timestamp that represents a point in simulation time at which the state transition occurs. For example, a simulation of an airport might include state variables representing the status of the runway, e.g., idle or busy, as well as other aspects such as the number of aircraft waiting to take off or land. Events might include aircraft arrivals or departures.

A parallel discrete event simulation program can be viewed as a collection of sequential discrete event simulations that interact by exchanging timestamped messages. Each message represents an event that is scheduled (sent) by one simulator to another. Each sequential simulator is referred to as a *logical process* or LP. For example, a simulation of the global air traffic system could be constructed by creating a sequential simulation of each airport and allowing each simulator to schedule events, i.e., send messages, to other simulators. Messages transmitted between airport simulators might represent the arrival of an aircraft flying from one airport to another.

Much of the work in PDES has focused on the synchronization issue. Briefly, the synchronization problem is concerned with managing the execution of a PDES program so that the results that are produced are exactly, or in some cases approximately, equal to a sequential execution of the same program where all events are processed one after the other in timestamp order. A large body of knowledge evolved around different approaches to addressing the synchronization algorithm, and associated algorithms. Other research focused on issues associated with parallel execution such as partitioning simulations, mapping LPs to the processors of a parallel computer, and redistributing computational workloads during program execution to improve efficiency. A number of studies focused on developing techniques to improve the performance of the parallel simulator, e.g., to increase the efficiency of the synchronization algorithm, often in the context of a particular application. Much of this literature is discussed in (Fujimoto 2000) and the references therein.

3 DISTRIBUTED SIMULATION

Distributed simulation is concerned with the execution of simulations over computing platforms that span a much broader geographic extent than parallel computers. In contrast to parallel simulations where the processors executing the simulation reside within a cabinet inside a machine room, a distributed simulation may execute on a set of machines interconnected through a local area network, globally distributed computers communicating via the Internet, or predictive simulations embedded within a physical environment such as a sensor network monitoring traffic in a city. A central object of utilizing distributed simulation systems has been to allow exploitation of geographically distributed resources such as equipment or people in the distributed simulation exercise, or executing simulations in close physical proximity to live data streams to predict future states of an operational systems.

A central issue in and often the principal motivation for utilizing distributed simulation concerns the desire to integrate several different simulators operating on different computing equipment into a single simulation environment. For example, much of the early work in distributed simulations was completed in the defense community for training – tank simulators, flight simulators, computer generated forces, and a variety of other models could be combined to create a distributed virtual environment into which personnel could be embedded to train for hypothetical scenarios and situations.

While synchronization, also known as time management in the distributed simulation literature, is also an issue in distributed simulation, it is not the only issue and typically is not the most important one. Much of the early research in distributed simulation focused on distributing information among the simulators participating in the distributed simulation in an efficient and timely manner. Much research focused on communication protocols such as multicast mechanisms or methods to reduce the amount of data that needed to be communicated through the use of techniques such as dead reckoning (Miller and Thorpe 1995) or data distribution management (Morse and Zyda 2001). Because a large emphasis in distributed simulation has been to achieve interoperability among separately developed simulators, a

substantial amount of effort has focused on developing standards to interconnect simulations. This has resulted in the Distributed Interactive Simulation (DIS) (IEEE Std 1278.1-1995 1995; IEEE Std 1278.2-1995 1995) and the High Level Architecture (HLA) (IEEE Std 1516-2010 2010; IEEE Std 1516.1-2010 2010; IEEE Std 1516.2-2010 2010) standards. Many of the technical issues addressed by the distributed simulation research community are discussed in (Fujimoto 2000; Tolk (ed.) 2012).

The remainder of this article is organized as follows. The next section focuses on conservative algorithms for addressing the synchronization problem. First and second generation algorithms are discussed. Optimistic algorithms are described next, including coverage of the so-called local and global control mechanisms. Finally, future directions for research in the field are discussed, highlighting six areas of research that require further investigation.

4 CONSERVATIVE SYNCHRONIZATION

As discussed earlier, the simulation is assumed to be composed of a set of logical processes (LPs) that communicate by exchanging timestamped event messages. In this section we use the terms events and messages synonymously. A synchronization algorithm is required to ensure that the parallel execution of the simulation produces exactly the same results as a sequential execution on a single processor. In some cases approximate results are acceptable, but the bulk of the research in synchronization algorithms has focused on producing exactly the same results. One can show that this can be achieved by ensuring that each LP processes events in timestamp order.

4.1 First Generation Algorithms

The parallel and distributed simulation field began in the late 1970's with seminal work by Chandy, Misra, and Bryant who defined the synchronization problem and a solution approach (Bryant 1977; Chandy and Misra 1978, 1979). Unlike a timestepped simulation execution where all processors can work on simulation computations in the current time step concurrently, in a PDES program there are typically very few events containing exactly the same time stamp. Thus, one must allow some LPs to advance ahead of others in simulation time in order to capture a sufficient amount of concurrent execution. Space does not permit a full exposition of the rich literature in synchronization algorithms. The following attempts to highlight some of the main technical points. A more lengthy description is presented in (Fujimoto 2000).

The Chandy/Misra/Bryant algorithm uses a mechanism that blocks the execution of an LP until it can guarantee that an event with a smaller timestamp will not later be received. This is accomplished by requiring the messages sent from one LP to another are sent in timestamp order, and the network guarantees ordered delivery, i.e., messages are received in the same order in which they were sent. This ensures events arriving on a single LP-to-LP link are received in timestamp order. Messages arriving on a single link may be stored in a first-in-first-out queue, which will hold messages in timestamp order (see Figure 1(a)). If each FIFO queue contains at least one message, the LP can simply pick the smallest timestamped message, remove it from its queue, and process it. If one or more FIFO queues are empty, the LP must block. Unfortunately, this may lead to deadlock situations where a cycle develops where one LP in the cycle is waiting for the next LP in the cycle (see Figure 1(b)). The CMB algorithm solves this problem by each LP sending *null* messages to neighboring LPs (LPs to which it may send event messages) to provide a guarantee indicating the smallest timestamp value of any message it will later send on that link. To make this guarantee each LP must declare a *lookahead* value. If an LP is currently at simulation time T, and its lookahead value is L, then any message later sent by the LP must have a timestamp of at least $T+L$. One can similarly associate lookahead values with links rather than LPs.

It can be shown that null messages avoid deadlock. However, this algorithm suffers from a problem known as *lookahead creep*. Consider the deadlock shown in Figure 1(b). Note that the smallest timestamped event in this snapshot of the system has a timestamp value of 7. Suppose each LP is at simulation time 5, and has a lookahead of 0.1. Each LP will send a null message to its neighbors

announcing its next message will have a timestamp of at least 5.1 (unless of course a message with a larger timestamp had already been sent), enabling each LP to advance to simulation time 5.1. Each LP will then send another null message with a timestamp of 5.2, enabling the LPs to advance to 5.2. This will repeat until each LP advances to simulation time 7, indicating the message with timestamp 7 can now be processed. Dozens of null messages will have to be sent and processed before the LP can determine that it is safe to process this event. This will clearly be very inefficient.

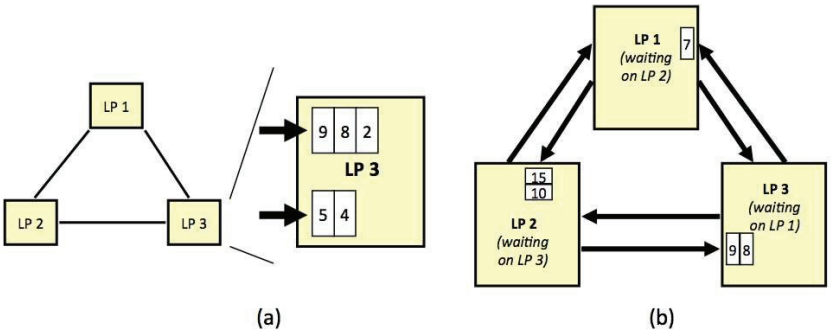


Figure 1: Conservative synchronization. (a) FIFO queues for CMB. (b) A deadlock situation.

4.2 Second Generation Algorithms

Second generation conservative algorithms addressed the lookahead creep problem. The solution to this problem lies in observing that the key piece of information that is required is the timestamp of the next unprocessed event in the system. If the LPs in the previous example knew this information they could immediately advance to simulation time 7 without sending rounds of null messages. Several synchronization algorithms were subsequently proposed that utilize this information, thereby avoiding lookahead creep (Chandy and Misra 1981; Chandy and Sherman 1989; Groselj and Tropper 1988; Lubachevsky 1989; Nicol et al. 1989).

Among these algorithms, we observe that algorithms such as the Bounded Lag (Lubachevsky 1989) and YAWNS (Nicol et al. 1989) algorithms differ significantly from CMB in that unlike CMB they are synchronous algorithms, meaning they utilize global synchronization points (barriers) as a fundamental element of the algorithm. These algorithms divide the computation into a sequence of cycles, or epochs. A global synchronization using a barrier primitive is used to separate the epochs. Each epoch involves (1) determining which events can be safely processed without risk of an LP later receiving a smaller timestamped event, (2) processing these safe events, possibly generating one or more new event messages, and (3) delivering these messages to their destination LPs. The computation repeatedly executes these epochs until the simulation has been completed.

All of the algorithms described above have the property that they strictly avoid allowing an LP to process events out of timestamp order. Algorithms that have this property are referred to as *conservative* synchronization algorithms. Among the many conservative synchronization algorithms that have been developed perhaps the most popular ones today are the original CMB algorithm and YAWNS, no doubt due to their simplicity.

All conservative algorithms rely on a sufficiently large lookahead value to achieve good performance. This is because if there is no lookahead constraint (or equivalently, the lookahead value is zero), the computation for the smallest timestamped event, say 7 in the above example, could in principle send a new event to every other LP in the system with a timestamp of 7. This implies the only concurrent execution that can occur arises from events with exactly the same timestamp. Further, the simulation program must be developed to have good lookahead, i.e., any new event must have a timestamp "reasonably" far into the simulated future. This may be difficult to accomplish for some applications, and can make the code difficult to understand and maintain.

The importance of lookahead is illustrated in Figure 2. This diagram shows a snapshot of a distributed simulation execution where each square box represents and event, plotted according to the timestamp of the event and the LP in which the event occurs. As can be seen, the smallest timestamped event in the system resides in *LP A* and has a timestamp of T_A. In the absence of any lookahead constraints it is possible that processing this event could cause a message with timestamp T_A to be sent to every other LP in the simulation. This implies none of the other events are safe to process because it is possible a smaller timestamped event will later arrive that should be processed first. This suggests that only those events with exactly the same timestamp can be processed concurrently. This severely limits the amount of concurrent processing that can be exploited because most discrete event simulations will have very few events containing exactly the same timestamp. On the other hand, if each LP has a lookahead of L, then the smallest new event that could be generated as a result of processing the event at time T_A must have a timestamp of T_A+L, implying the events shown in red are all safe to process. This example illustrates the importance of lookahead to achieve concurrent processing of events for any conservative synchronization algorithm.

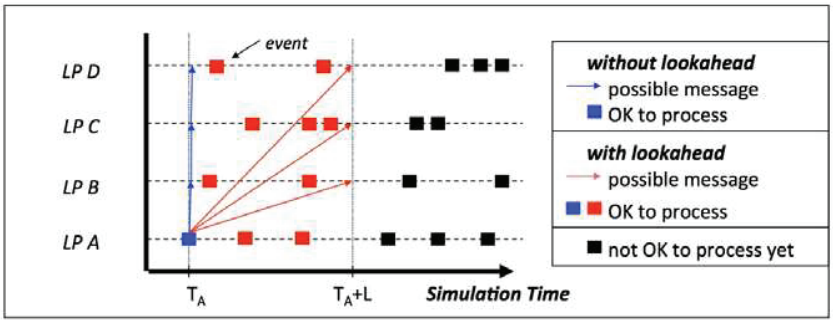


Figure 2: Lookahead. The events in red can be safely processed when exploiting a lookahead.

5 OPTIMISTIC SYNCHRONIZATION

In the early 1980's another approach to addressing the synchronization problem was developed by Jefferson and Sowizral. Their algorithm, called Time Warp, introduced a new approach to addressing the synchronization problem (Jefferson 1985). Unlike conservative approaches that avoid errors, Time Warp allows errors to occur, but the uses a rollback mechanism to recover. Time Warp spawned a number of new algorithms utilizing a rollback mechanism that are now collectively known as optimistic synchronization. The Time Warp algorithm consists of two parts, a local control mechanism and a global control mechanism. Each of these are described next.

5.1 The Local Control Mechanism

Time Warp, does not rely on rules to block the execution of LPs. Rather, LPs are allowed to process events, risking executing events out of timestamp order. When events are processed out of timestamp order, e.g., if an LP processes an event with timestamp 100 and then receives an event from another LP with timestamp 50, a rollback mechanism is invoked to undo the execution of those events processed out of order, i.e., events with timestamp greater than 50.

An event computation can perform two actions that must be undone. It may modify state variables, and it may send event messages to other LPs. Undoing modification of state variables can be accomplished by taking a snapshot of the state of each LP prior to processing each event. This can be done by simply making a copy of the state variables. This approach is called *copy state saving*. This may be expensive in both time to copy the variables, and memory used to hold the saved state. For some applications, state saving overheads may be prohibitive. An alternative approach is to use *incremental state saving* where code is inserted into the event routines to make a copy of a state variable before the

first time it is modified by an event. Unlike copy state saving, incremental state saving requires the address of the variable to stored as well as the copied data. Nevertheless, incremental state saving is much more efficient than copy state saving of an LP requires a large amount of memory to hold state variables, and relatively few variables are modified by each event. A third approach to addressing this problem is to use *reverse computation* where the inverse of the event computation is created and executed to undo an event computation when rollback occurs. For example, the inverse of a computation that increments a state variable is to decrement that variable. This approach has the advantage that it can reduce the amount of computation required in the forward execution of the program which often forms the critical path of the computation and often dictates the execution time, and it can significantly reduce the amount of memory required. However, for many computations the inverse computation cannot be generated. In this case, reverse execution relies on incremental state saving to enable rollback.

The second action that must be undone are message sends performed by the event computation. In contrast to undoing the modification of state variables that are confined to the LP being rolled back, a message sent to another LP may have been processed by that LP, and may have caused the generation of additional messages. These message may, in turn, have been processed resulting in still more messages, and so on. In fact, a single message send may have affected every other LP in the system. A mechanism is required to undo all of these computations! Perhaps the most clever aspect of Time Warp is a very simple mechanism called anti-messages was invented to undo a message send. An anti-message is an identical copy of the original (positive) message that was sent except it has a flag to marking it as an anti-message. To undo a message send, the LP need only send the anti-message to the same destination as the original message. When the anti-message is received by the destination LP, if the original positive message has not yet been processed, then the message/anti-message pair simply annihilate each other and storage for both is reclaimed. If the positive message has already been processed, the LP receiving the anti-message is rolled back to just before the positive message was processed, and then the message/anti-message pair can be annihilated. Rolling back an LP can result in the sending of additional anti-messages, which may in turn cause additional rollbacks. Recursively applying this procedure will erase all of the effects of the original message. The phenomena whereby anti-messages lead to rollbacks in other LPs that may result in the generation of additional anti-messages and still other rollbacks are known as cascaded rollbacks. Clearly cascaded rollbacks are undesirable because they result in the utilization of computation and communication resources without directly performing simulation event computations.

5.2 The Global Control Mechanism

To enable rollback one typically makes a copy of each LP's state variables so an old version of an LP's state can be later restored after a rollback. These check-pointing operations are called *state saves*. This introduces the problem that one must be able to later recover the memory utilized to hold the check-pointed states, as well as any event that has been processed in case it might have to be reprocessed later. A second problem is some computations cannot be rolled back, e.g., I/O operations performed by an LP. Both of these problems are addressed by computing a lower bound on the timestamp of any future rollback that might occur. This computed value is known as *global virtual time* (GVT). Memory used for saved state variables older than GVT (except one, in case a rollback to GVT occurs) can be reclaimed and used for other purposes, and irrevocable operations such as I/O that were performed at a simulation time less than GVT may be performed.

GVT may be defined as the smallest timestamp among those messages (events) and anti-messages that have not yet been processed. Because rollbacks are caused by receiving a message or anti-message in an LP's past, these messages clearly must be considered when determining a lower bound on the timestamp of any future rollback. Therefore, if one could obtain a global snapshot of the computation, GVT could be computed by simply finding the message or anti-message with the smallest timestamp. GVT computation is non-trivial because the GVT computation is typically performed asynchronously "in background" during the execution of the Time Warp simulation.

Several algorithms for computing GVT were developed. Samadi describes two key challenges that must be overcome in computing GVT (Samadi 1985). The first is the *transient message problem*. A transient message is defined as a message that has been sent but has not yet been received. Clearly such messages must be taken into account by the GVT computation. Samadi proposes using message acknowledgements to take these messages into account. The second is referred to as the *simultaneous reporting problem*. This problem arises because different LPs will report their local minimum at different points in wallclock time. Samadi addresses this problem by using a scheme to mark LPs and acknowledgement messages sent by marked LPs. Tha main drawback with Samadi's algorithm is the overhead associated with having the send acknowledgement messages.

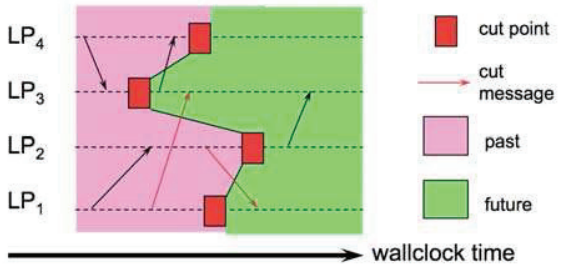


Figure 3: Snapshot of computation used in Mattern's algorithm for computing GVT.

Mattern proposes an elegant algorithm that avoids the need to send message acknowledgements (Mattern 1993). Mattern's algorithm is based developing a distributed snapshot of the execution of the program. Specifically, each LP places a cut point that divides the timeline for the LP's computation into a "past" and "future" part, as shown in Figure 3. Each LP places its cut point asynchronously of the others, so the cut points will in general occur at different points in wallclock time (not to be confused with simulation time). The set of cut points across all LPs forms a *cut* of the computation, and divides the entire distributed computation into a past and future part. A message that is sent in the past part of the computation, and received in the future part is referred to as a *cut message*. There are two cut messages shown in Figure 3. It can be shown that a valid GVT value may be computing by determining the minimum timestamp among (1) any unprocessed message or anti-message within a snapshot of the LP at its cut point and (2) any cut message. Like Samadi's algorithm, Mattern's algorithm must consider transient messages. Rather than using message acknowledgements, each LP keeps a count of the number of messages it has sent and the number of messages it has received. When the sum of these counters over all LPs yield the same value there are no transient messages. Mattern's algorithm utilizes these concepts to efficiently compute GVT. See (Fujimoto 2000) for a more detailed discussion of Mattern's algorithm as well as other algorithms for computing GVT.

5.3 Other Optimistic Algorithms

From a performance standpoint, two central problems faced by optimistic algorithms are an excessive amount of rollback and the time and memory required for state saving. A Time Warp system can suffer from an excessive amount of rolled back computation by, for example, some LPs may advance too far ahead of others leading to long rollbacks as well as excessive memory utilization. Many optimistic algorithms have been proposed to address this issue. Most attempt to limit the amount of optimistic execution, i.e., the amount of event computation that is performed that may be later rolled back. For example, an early technique involves using a sliding window of simulated time and not only allowing events whose timestamp fall beyond this window to be processed (Sokol and Stucky 1990). Another approach delays message sends until it is guaranteed that the send will not be later rolled back, i.e., until GVT advances to the simulation time at which the event was scheduled. This eliminates the need for anti-messages and avoids cascaded rollbacks, i.e., a rollback resulting in the generation of additional rollbacks

(Dickens and Reynolds 1990; Steinman 1992). A number of other techniques have been proposed to improve efficiency by controlling the amount of rollback, or improving the efficiency of the rollback mechanism (Chen and Szymanski 2002, 2003; Fujimoto 1989; Zhang and Tropper 2001). Early approaches to controlling the Time Warp execution used user-defined parameters that had to be tuned to optimize performance. Later work focused on adaptive approaches where the simulation executive automatically monitors the execution and adjusts control parameters to maximize performance. Examples of such adaptive control mechanisms are described in (Das and Fujimoto 1997; Ferscha 1995), among others.

The second problem, memory use in Time Warp, touches upon two main issues. The first concerns the amount of memory required for state saving. The second concerns the amount of time required to perform state saving itself. Increased memory usage can significantly impact performance because hardware caches may not operate very efficiently for programs consuming large amounts of memory. Several techniques have been developed to address this problem. State saving can be performed infrequently rather than before each event computation (Lin et al. 1993; Palaniswamy and Wilsey 1993). An alternative to making a copy of an LP's state variables, referred to as copy state saving, is to use incremental state saving where the original value of a variable is copied just before it is modified by an event computation (Ronngren et al. 1996; West and Panesar 1996). One can roll back computations to reclaim memory resources (Jefferson 1990; Lin and Preiss 1991). The memory used by some state vectors can be reclaimed even though their time stamp is larger than GVT (Preiss and Loucks 1995). Another approach attempts to do away with state saving altogether and instead use reverse execution to undo event computations (Carothers et al. 1999). A reverse execution compiler is used to automatically generate the code that computes the inverse of each event computation, and this inverse code is executed when for each event that must be rolled back. Reverse execution offers the advantage that the burden of executing code to enable rollback, i.e., state saving in conventional Time Warp implementations, is moved to rollback operations rather than the forward execution of events. This makes the overheads associated with state saving less likely to be on the critical path of the computation. In some case, the inverse operation for event computations cannot be created, so this technique typically relies on incremental state saving for such irreversible computations.

Synchronization is a well-studied area of research in the parallel and distributed simulation field. Neither optimistic nor conservative synchronization algorithms dominate the other with respect to performance; indeed, the optimal approach usually depends on the application. In general, if the application has good lookahead and programming the application to exploit this lookahead is not overly burdensome, conservative approaches are the method of choice. Indeed, much research has been devoted to improving the lookahead of simulation applications, e.g., see (Deelman et al. 2001). Otherwise, optimistic synchronization offers greater promise. Disadvantages of optimistic synchronization include the potentially large amount of memory that may be required, and the complexity of optimistic simulation executives. Techniques to reduce memory utilization may further aggravate the complexity issue.

6 DIRECTIONS FOR FUTURE RESEARCH

Below we briefly describe six areas for future research in parallel and distributed simulation.

PDES on Massively Parellal Supercomputers. Over the years numerous successes have been achieved demonstrating the capabilities of PDES technology to accelerate the execution of discrete event simulations. For example, (Fujimoto et al. 2003) examined packet-level simulation of computer communication networks on supercomputers. A metric was defined indicating the number of packet transmissions that could be simulated per second of wallclock time (PTS) by the parallel simulator. Experiments yielding performance as high as 106 million PTS were completed using a conservative synchronization algorithm executing on a supercomputer using 1,536 processors in 2003. By comparison, comparable simulators executing on a sequential machine yielded performance less than 100,000 PTS. In 2007 studies using a synthetic benchmark called PHOLD yielded performance exceeding 529 million

events per second (a packet transmission, discussed earlier, typically consists of two events) using an optimistic synchronization algorithm on a 16,384 processor machine (Perumalla 2007) and in 2009 performance of 12.26 billion events per second using 65,536 processors were achieved (D. W. Bauer Jr. et al. 2009), both using IBM Blue Gene machines. In 2013 Barnes et. al were able to achieve 504 billion events per second using almost 2 million cores of a Blue Gene/Q machine (Barnes et al. 2013). Studies of large-scale simulations of specific applications include epidemic spread (Bisset et al. 2009; Perumalla and Seal 2010) and electromagnetic signal propagation (D. W. Bauer Jr. et al. 2009). These studies yielded event rates *per core* of 138K (2003), 32K (2007), 187K (2009), and 256K (2013) events/second/core, representing only a factor of 2 improvement in single core performance over a span of 10 years. Performance improvements today are being driven almost entirely by increases in parallelism. Processor clock rates have seen only modest increases since 2005 due to physical constraints concerning heat dissipation, resulting in an explosion in the number of cores in supercomputer architectures since 2005. Throughout much of the 1990's and up until 2005 the most powerful supercomputers contained only thousands of cores. The most powerful machines today contain millions.

Exploiting GPU Platforms. A graphics processing unit (GPU) is a hardware accelerator that off-loads computational tasks from the central processing unit (CPU). A principal limitation of GPUs for PDES applications is the single-instruction-stream, multiple-data-stream (SIMD) execution paradigm. GPUs are designed to support data parallel applications where the same operation is performed across multiple data elements. Early work in PDES examined SIMD architectures for queueing network and Ising spin simulations (Lubachevsky 1989) and logic simulation of electronic circuits (Chung and Chung 1991). Work by (Kunz et al. 2012) using GPUs involves sorting events according to event type and clustering their execution to allow SIMD execution. Scheduling and load balancing are discussed in (Romdhanne et al. 2013). Use of GPUs for cellular automata simulations have been explored in application such as traffic simulation (Perumalla et al. 2009; Xu et al. 2014), systems biology (Falk et al. 2011), and Ising Spin (Hawick et al. 2011). In (Park and Fishwick 2011) a time-stepped-like mechanism to implement the discrete event simulation. Another approach utilizing a three step approach – generate events, sort them, process them and use of time parallel simulation techniques is described in (Li, Cai, et al. 2013). In (Liu et al. 2014) a method for hybrid simulation of telecommunication networks is described where a discrete event packet-level simulation is performed by the CPU while a continuous numerical fluid flow simulation of traffic is executed by the GPU.

Exploiting Cloud Computing Platforms. A significant impediment limiting widespread exploitation of PDES technology for large-scale simulation applications has been the need to have access to suitable high performance computing machines. The cloud's "pay-as-you-go" economic model eliminates the need to purchase, operate and maintain high performance computing equipment locally (Fujimoto et al. 2010). Further, by providing parallel and distributed simulation software as a service, cloud computing offers the ability to hide many of the complications of executing parallel and distributed simulation codes from the user, offering the potential to make exploitation of this technology less risky than is the case today. An important issue in cloud computing environments concerns resource sharing. Computing, communications, and I/O resources are shared among many users. Individual users are not guaranteed exclusive access to the processors assigned to that user's virtual cluster. This creates a significant amount of uncertainty and variability in the computation environment on which the parallel or distributed simulation code executes. This variability is cited as a likely cause of performance degradations of conservatively synchronized PDES codes executing on EC2 (Vanmechelen et al. 2012). This can lead to difficulties for parallel simulation applications, especially those that utilize optimistic synchronization techniques. Architectures designed to provide flexible resource sharing are described in (He et al. 2012; Li, Chai, et al. 2013; Liu et al. 2012).

Data Driven Distributed Simulation. Dynamic Data Driven Application Systems (DDDAS) (Darema 2004) are computational systems that dynamically incorporate data from a physical system into executing computations, thereby providing the ability of the application to dynamically steer the measurement

process. These approaches have been widely studied and applied to various science and engineering disciplines for a myriad of purposes. One typical application concerns system monitoring, such as examining the structural and material health (Cortial et al. 2007; Farhat et al. 2006), tracking wildfires (Brun et al. 2012; Douglas et al. 2006; Mandel et al. 2012; Mandel et al. 2005) and hurricanes (Allen 2007), or prediction and tracking regional scale weather phenomena (Plale et al. 2005). A second use concerns optimizing the operations of a physical system. For example, in an emergency situation alternate evacuation scenarios may be modeled and evaluated in order to minimize evacuation time (Chaturvedi et al. 2006). The evacuation plan may need to adapt as the evacuation evolves when unforeseen events arise (Chaturvedi et al. 2006). Additional examples include path planning for unmanned aerial vehicles (Kamrani and Ayani 2007; Madey et al. 2012), tuning parameters for computer networks (Ye et al. 2008), managing semiconductor manufacturing systems (Low et al. 2005), and managing surface transportation systems to mitigate congestion (Fujimoto et al. 2007; Suh et al. 2014). They have been proposed for decision support systems for manufacturing applications, as described in (Lendermann et al. 2005). An approach to distributed, online simulations of queueing networks is discussed in (Huang et al. 2010).

Power and Energy Consumption. Power consumption has become a major concern for many parallel and distributed computing applications. The need to reduce power consumption is clear in mobile and embedded computing where reduced power consumption can result in increase battery life or enable the use of smaller batteries thereby reducing the size and weight of devices. In high-end computing power consumption is a dominant cost associated with operating large data centers and supercomputers, and a substantial amount of effort has gone into developing techniques to reduce these costs. So far, work in power and energy aware computing has largely focused on low-level aspects of the computing system in terms of effectively utilizing specific hardware capabilities and the development of operating systems and compilers to reduce energy usage consistent with performance constraints. The design of a simulation, e.g., in terms of data structures and algorithms with respect to energy and power consumption has not been extensively studied. The relationships among model detail and fidelity and power consumption are not well understood. These tradeoffs must consider analysis of data produced by the simulation, which will likely be greater for more detailed model, in addition to the execution of the simulation model. Further, modeling results must be produced in a timely fashion when the model is used for managing operational systems, imposing real-time constraints on model execution time. The frequency and amount of data utilized by the simulation will have a large impact on power consumption and energy use. In addition to the power required to transmit the data, data analysis algorithms will also consume energy to process incoming data. Further, synchronization algorithms can significantly affect the amount of energy that is consumed.

Interoperable Models. In (Petty and Weisel; 2003) composability is defined as "the capability to select and assemble simulation components in various combinations into valid simulation systems to satisfy specific user requirements." Composability is not a binary, yes/no property, but rather is characterized by degrees. Three levels of integration are proposed in (Page et al. 2004), and refined in (Tolk 2012) to define seven levels that highlight several key issues to achieving composeability. A longstanding goal of the distributed simulation community has been to achieve "plug-and-play" interoperability where one is able to automatically compose separately developed simulations independent of the context into which they are embedded. This implies model composeability, and is far from the capabilities of current technologies. Multi-modeling or multi-paradigm modeling addresses the problem of constructing heterogeneous models of systems where the components of the system are modeled using different modeling approaches. Multi-modeling has been widely used in embedded system design and cyber-physical systems where models, computations, and networks are combined with physical devices. Computer Automated Multi-Paradigm Modeling (CAMPaM) strives to create domain-independent frameworks to support the development of heterogeneous models (Mosterman and Vangheluwe 2004). These systems must address issues concerned with integration of multiple modeling formalisms, integrations of models at different levels of abstraction, and metamodels. A metamodel

defines the modeling language. The same metamodeling language can describe multiple modeling languages, and thus can be used to specify systems that utilize multiple formalisms (Cetinkaya and Verbraeck 2011; Vangheluwe and Lara 2002). The metamodel provides a means of describing the overall system composed of the different constituent modeling formalisms supported by the modeling system. Compilers or translators provide a means to automatically transform a metamodel for a system into a simulation model utilizing multiple formalisms. Finally, the metmodeling language itself can be described by a meta-metamodel language. Ptolemy II and AToM3 are examples of multi-modeling frameworks (De Lara and Vangheluwe 2002; Ptolemaeus (ed.) 2014).

7 CONCLUSIONS

Parallel and distributed simulation continues to be an important field of research. A rich body of literature has developed addressing problems such as synchronization to ensure correct results are produced. The field continues to evolve, driven by requirements derived from application and changes in the underlying computing platforms. Future research will need to address requirements and constraints derived from applications as well as major changes in the underlying computing platform and software.

ACKNOWLEDGMENTS

Support for Fujimoto's research in parallel and distributed simulation has been provided by the National Science Foundation under Grant 1441208 and the Air Force Office of Scientific Research under Grant FA9550-13-1-0100.

REFERENCES

Allen, G. 2007. "Building a Dynamic Data Driven Application System for Hurricane Forecasting." In *Computational Science – ICCS 2007*, edited by Y. Shi, G. D. v. Albada, J. Dongarra, and P. M. A. Sloot, 1034–1041. Berlin: Springer Berlin Heidelberg.

Barnes, P. D., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. 2013. "Warp Speed: Executing Time Warp on 1,966,080 Cores." In *Principles of Advanced Discrete Simulation*, 327-336.

Bauer Jr., D. W. C. D. Carothers, and A. Holder. 2009. "Scalable Time Warp on Blue Gene Supercomputers." In *Principles of Advanced and Distributed Simulation*, 35-44.

Bisset, K. R., J. Chen, X. Feng, V. S. A. Kumar, and M. V. Marathe. 2009. "Epifast: A Fast Algorithm for Large Scale Realistic Epidemic Sim- Ulations on Distributed Memory Systems." In *International Conference of Supercomputing*, 430–439.

Brun, C., T. Artés, T. Margalef, and A. Cortés. 2012. "Coupling Wind Dynamics into a DDDAS Forest Fire Propagation Prediction System." In *Proceedings of the International Conference on Compuational Science*.

Bryant, R. E. 1977. "Simulation of Packet Communications Architecture Computer Systems." In *MIT-LCS-TR-188*.

Carothers, C. D., K. Perumalla, and R. M. Fujimoto. 1999. "Efficient Optimistic Parallel Simulation Using Reverse Computation." *ACM Transactions on Modeling and Computer Simulation* 9(3):224-253.

Cetinkaya, D., and A. Verbraeck. 2011. "Metamodeling and Model Transformations in Modeling and Simulation." In *Proceedings of the 2011 Winter Simulation Conference*, edited by S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, 3048-3058. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Chandy, K. M., and J. Misra. 1979. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs." *IEEE Transactions on Software Engineering* SE-5 (5):440-452.

Chandy, K. M., and J. Misra. 1981. "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations." *Communications of the ACM* 24 (4):198-205.

Chandy, K. M., and R. Sherman. 1989. "The Conditional Event Approach to Distributed Simulation." In *Proceedings of the SCS Multiconference on Distributed Simulation*, 93-99.

Chaturvedi, A., A. Mellema, S. Filatyev, and J. Gore. 2006. "DDDAS for Fire and Agent Evacuation Modeling of the Rhode Island Nightclub Fire." In *Computational Science – ICCS 2006*, edited by V. N. Alexandrov, G. D. v. Albada, P. M. A. Sloot, and J. Dongarra, 433–439. Berlin: Springer.

Chen, G., and B. K. Szymanski. 2002. "Lookback: A New Way of Exploiting Parallelism in Discrete Event Simulation." In *the 16th Workshop on Parallel and Distributed Simulation*, 153-162.

Chen, G., and B. K. Szymanski. 2003. "Four Types of Lookback." In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, 3-10.

Chung, M., and Y. Chung. 1991. "An Experimental Analysis of Simulation Clock Advancement in Parallel Logic Simulation on an Simd Machine." In *Advances in Parallel and Distributed Simulation*, 125-132. SCS Simulation Series.

Cortial, J., C. Farhat, L. J. Guibas, and M. Rajashekhar. 2007. "Compressed Sensing and Time-Parallel Reduced-Order Modeling for Structural Health Monitoring Using a Dddas." In *Computational Science – ICCS 2007*, edited by Y. Shi, G. D. v. Albada, J. Dongarra, and P. M. A. Sloot, 1171–1179. Berlin: Springer.

Darema, F. 2004. "Dynamic Data Driven Applications Systems: A New Paradigm for Application Simulations and Measurements." In *International Conference on Computational Science*.

Das, S. R., and R. M. Fujimoto. 1997. "Adaptive Memory Management and Optimism Control in Time Warp." *ACM Transactions on Modeling and Computer Simulation* 7 (2):239-271.

De Lara, J., and H. Vangheluwe. 2002. "Atom3: A Tool for Multi-Formalism and Meta-Modelling." In *Proceedings of the Fundamental Approaches to Software Engineering*, edited by R.-D. Kutsche, and H. Weber, 174–188. Springer.

Deelman, E., R. Bagrodia, R. Sakellariou, and V. Adve. 2001. "Improving Lookahead in Parallel Discrete Event Simulations of Large-Scale Applications Using Compiler Analysis." In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, 5-13.

Dickens, P. M., and J. Reynolds, P. F. 1990. "Srads with Local Rollback." In *Proceedings of the SCS Multiconference on Distributed Simulation*, 161-164.

Douglas, C. C., R. A. Lodder, J. D. Beezley, J. Mandel, R. E. Ewing, Y. Efendiev, G. Qin, M. Iskandarani, J. Coen, A. Vodacek, M. Kritz, and G. Haase. 2006. "DDDAS Approaches to Wildland Fire Modeling and Contaminant Tracking." In *Proceedings of the 2006 Winter Simulation Conference*, edited by L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, 2117-2124. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Falk, M., M. Ott, T. Ertl, M. Klann, and H. Koeppl. 2011. "Parallelized Agent-Based Simulation on Cpu and Graphics Hardware for Spatial and Stochastic Models in Biology." In *9th International Conference on Computational Methods in Systems Biology*, 73-82.

Farhat, C., J. G. Michopoulos, F. K. Chang, L. J. Guibas, and A. J. Lew. 2006. "Towards a Dynamic Data Driven System for Structural and Material Health Monitoring." In *Computational Science – ICCS 2006*, edited by V. N. Alexandrov, G. D. v. Albada, P. M. A. Sloot, and J. Dongarra, 456–464. Berlin: Springer Berlin Heidelberg.

Ferscha, A. 1995. "Probabilistic Adaptive Direct Optimism Control Iin Time Warp." In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, 120-129.

Fujimoto, R. M. 1989. "Time Warp on a Shared Memory Multiprocessor." *Transactions of the Society for Computer Simulation* 6 (3):211-239.

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems*: Wiley Interscience.

Fujimoto, R. M., M. Hunter, J. Sirichoke, M. Palekar, H.-K. Kim, and W. Suh. 2007. Ad Hoc Distributed Simulations. In *Principles of Advanced and Distributed Simulation*.

Fujimoto, R. M., A. W. Malik, and A. J. Park. 2010. "Parallel and Distributed Simulation in the Cloud." *SCS Modeling and Simulation Magazine, Society for Modeling and Simulation, Intl.* 1 (3).

Fujimoto, R. M., K. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley. 2003. Large-Scale Network Simulation: How Big? How Fast? In *Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*.

Groselj, B., and C. Tropper. 1988. "The Time of Next Event Algorithm." In *Proceedings of the Scs Multiconference on Distributed Simulation*, 25-29. Society for Computer Simulation.

Hawick, K. A., A. Leist, and D. P. Playne. 2011. "Regular Lattice and Small- World Spin Model Simulations Using Cuda and Gpus." *International Journal of Parallel Programming* 39 (2):183–201.

He, H., R. Li, X. Dong, Z. Zhang, and H. Han. 2012. "An Efficient and Secure Cloud-Based Distributed Simulation System." *Journal of Applied Mathematics & Information Sciences* 6 (3):729-736.

Huang, Y.-L., M. Hunter, C. Alexopoulos, and R. M. Fujimoto. 2010. "Ad Hoc Distributed Simulation of Queueing Networks." In *Principles of Advanced and Distributed Simulations*.

IEEE Std 1278.1-1995. 1995. *IEEE Standard for Distributed Interactive Simulation -- Application Protocols*. New York, NY: Institute of Electrical and Electronics Engineers, Inc.

IEEE Std 1278.2-1995. 1995. *IEEE Standard for Distributed Interactive Simulation -- Communication Services and Profiles*. New York, NY: Institute of Electrical and Electronics Engineers Inc.

IEEE Std 1516-2010. 2010. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules*. New York, NY: Institute of Electrical and Electronics Engineers, Inc.

IEEE Std 1516.1-2010. 2010. *IEEE Standard for Modeling and Simulation High Level Architecture (HLA) - Interface Specification*. New York, NY: Institute of Electrical and Electronics Engineers, Inc.

IEEE Std 1516.2-2010. 2010. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) -- Object Model Template (OMT) Specification*. New York, NY: Institute of Electrical and Electronics Engineers, Inc.

Jefferson, D. 1985. "Virtual Time." *ACM Transactions on Programming Languages and Systems* 7 (3):404-425.

Jefferson, D. R. 1990. "Virtual Time Ii: Storage Management in Distributed Simulation." In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, 75-89.

Kamrani, F., and R. Ayani. 2007. Using on-Line Simulation for Adaptive Path Planning of UAVs. *IEEE International Symposium on Distributed Simulation and Real-Time Applications*.

Kunz, G., D. Schemmel, J. Gross, and K. Wehrle. 2012. "Multi-Level Parallelism for Time and Cost Efficient Parallel Discrete Event Simulation on Gpus." In *Principles of Advanced and Distributed Simulation*, 23-32.

Lendermann, P., M. Y. H. Low, B. P. Gan, N. Julka, L.-P. Chan, L. H. Lee, S. J. E. Taylor, S. J. Turner, W. Cai, X. Wang, T. Hung, L. F. McGinnis, and S. Buckley. 2005. "An Integrated and Adaptive Decision-Support Framework for High-Tech Manufacturing and Service Networks." In *Proceedings of the 2005 Winter Simulation Conference*, edited by M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 2052-2062. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Li, B.-H., X. Chai, B. Hou, C. Yang, T. Li, T. Lin, Z. Zhang, Y. Zhang, W. Zhu, and Z. Zhao. 2013. "Research and Application on Cloud Simulation." In *Summer Computer Simulation Conference*, 157-170.

Li, X., W. Cai, and S. Turner. 2013. "Gpu Acceleratedthree-Stageexecution Model for Event-Parallel Simulation." In *Principles of Advanced Discrete Simulation*, 57-66.

Lin, Y.-B., and B. R. Preiss. 1991. "Optimal Memory Management for Time Warp Parallel Simulation." *ACM Transactions on Modeling and Computer Simulation* 1 (4).

Lin, Y.-B., B. R. Preiss, W. M. Loucks, and E. D. Lazowska. 1993. "Selecting the Checkpoint Interval in Time Warp Simulations." In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, 3-10.

Liu, J., Y. Liu, Z. Du, and T. Li. 2014. "GPU-Assisted Hybrid Network Traffic Model." In *Principles of Advanced Discrete Simulation*, 63-74.

Liu, X., X. Qiu, B. Chen, and K. Huang. 2012. "Cloud-Based Simulation: The State-of-the-Art Computer Simulation Paradigm." In *Principles of Advanced and Distributed Simulation*, 71-74.

Low, M. Y. H., K. W. Lye, P. Lendermann, S. J. Turner, R. T. W. Chim, and S. H. Leo. 2005. "An Agent-Based Approach for Managing Symbiotic Simulation of Semiconductor Assembly and Test Operation." In *Proceedings of the 14th International Joint Conference on Autonomous Agents and Multiagent Systems*, 85–92. New York: Association for Computing Machinery, Inc.

Lubachevsky, B. D. 1989. "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks." *Communications of the ACM* 32 (1):111-123.

Madey, G. R., M. B. Blake, C. Poellabauer, H. Lu, R. R. McCune, and Y. Wei. 2012. Applying Dddas Principles to Command, Control and Mission Planning for UAV Swarms. *Proceedings of the International Conference on Compuational Science*.

Mandel, J., J. D. Beezley, A. K. Kochanski, V. Y. Kondratenko, and M. Kim. 2012. Assimilation of Perimeter Data and Coupling with Fuel Moisture in a Wildland Fire – Atmosphere DDDAS. In *International Conference on Compuational Science*.

Mandel, J., L. S. Bennethum, M. Chen, J. L. Coen, C. C. Douglas, L. P. Franca, C. J. Johns, M. Kim, A. V. Knyazev, R. Kremens, V. Kulkarni, G. Qin, A. Vodacek, J. Wu, W. Zhao, and A. Zornes. 2005. "Towards a Dynamic Data Driven Application System for Wildfire Simulation." In *Computational Science – ICCS 2005*, edited by V. S. Sunderam, G. D. v. Albada, P. M. A. Sloot, and J. J. Dongarra, 632–639. Berlin: Springer Berlin Heidelberg.

Mattern, F. 1993. "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation." In *Journal of Parallel and Distributed Computing*.

Miller, D. C., and J. A. Thorpe. 1995. "Simnet: The Advent of Simulator Networking." *Proceedings of the IEEE* 83 (8):1114-1123.

Morse, K. L., and M. Zyda. 2001. "Multicast Grouping for Data Distribution Management." *SIMPRA - Journal of Simulation Practice and Theory* Fall (Elsevier).

Mosterman, P., and H. Vangheluwe. 2004. "Computer Automated Multi-Paradigm Modeling: An Introduction." *Simulation: Transactions of The Society for Modeling and Simulation International* 80 (9):433-450.

Nicol, D. M., C. Micheal, and P. Inouye. 1989. "Efficient Aggregaton of Multiple Lps in Distributed Memory Parallel Simulations." In *Proceedings of the 1989 Winter Simulation Conference*, edited by E. A. MacNair, K. J. Musselman, and P. Heidelberger, 680-685. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Page, E. H., R. Briggs, and J. A. Tufarolo. 2004. "Toward a Family of Maturity Models for the Simulation Interconnection Problem." In *Spring 2004 Simulation Interoperability Workshop*.

Palaniswamy, A. C., and P. A. Wilsey. 1993. "An Analytical Comparison of Periodic Checkpointing and Incremental State Saving." In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, 127-134.

Park, H., and P. Fishwick. 2011. "An Analysis of Queuing Network Simulation Using Gpu-Based Hardware Acceleration." *ACM Transactions on Modeling and Computer Simulation* 21 (3).

Perumalla, K. S. 2007. "Scaling Time Warp-Based Discrete Event Execution to 10**4 Processors on a Blue Gene Supercomputer." In *Proceedings of the ACM Computing Frontiers Conference*.

Perumalla, K. S., G. A. Brandon, B. Y. Srikanth, and K. S. Sudip. 2009. "Gpu-Based Real-Time Execution of Vehicular Mobility Models in Large-Scale Road Network Scenarios." In *Principles of Advanced and Distributed Simulation*, 95-103.

Perumalla, K. S., and S. K. Seal. 2010. "Reversible Parallel Discrete-Event Execution of Large-Scale Epidemic Outbreak Models." In *Principles of Advanced and Distributed Simulation*, 106-113.

Petty, M. D., and E. W. Weisel;. 2003. "A Composability Lexicon." In *IEEE Spring Simulation Interoperability Workshop*.

Plale, B., D. Gannon, and D. Reed. 2005. Towards Dynamically Adaptive Weather Analysis and Forecasting in Lead. *International Conference on Computational Science*.

Preiss, B. R., and W. M. Loucks. 1995. "Memory Management Techniques for Time Warp on a Distributed Memory Machine." In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, 30-39.

Ptolemaeus (ed.), C. 2014. *System Design, Modeling, and Simulation Using Ptolemy II*: Ptolemy.org

Romdhanne, B. B., M. S. M. Bouksiaa, N. Nikaein, and C. Bonnet. 2013. "Hybrid Scheduling for Event-Driven Simulation over Heterogeneous Computers." In *Principles of Advanced Discrete Simulation*, 47-56.

Ronngren, R., M. Liljenstam, J. Montagnat, and R. Ayani. 1996. "Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation." In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, 70-77.

Samadi, B. 1985. "Distributed Simulation, Algorithms and Performance Analysis." Computer Science Department, University of California, Los Angeles, Los Angeles, California.

Sokol, L. M., and B. K. Stucky. 1990. "MTW: Experimental Results for a Constrained Optimistic Scheduling Paradigm." In *Proceedings of the SCS Multiconference on Distributed Simulation*, 169-173.

Steinman, J. S. 1992. "Speedes: A Multiple-Synchronization Environment for Parallel Discrete Event Simulation." *International Journal on Computer Simulation*:251-286.

Suh, W., M. Hunter, and R. M. Fujimoto. 2014. "Ad Hoc Distributed Simulation for Transportation System Monitoring and near-Term Prediction." *Simulation Modeling Practice and Theory* 41:1-14.

Tolk (ed.), A. 2012. *Engineering Principles of Combat Modeling and Distributed Simulation*. Hoboken, New Jersey: John Wiley and Sons

Tolk, A. 2012. "Challenges of Distributed Simulation." In *Engineering Principles of Combat Modeling and Distributed Simulation*, edited by A. Tolk. John Wiley and Sons Inc.

Vangheluwe, H., and J. D. Lara. 2002. "Meta-Models Are Models Too." In *Proceedings of the 2002 Winter Simulation Conference*, edited by E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, 597 – 605. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Vanmechelen, K., S. De Munck, and J. Broeckhove. 2012. "Conservative Distributed Discrete Event Simulation on Amazon EC2." In *International Symposium on Cluster, Cloud, and Grid Computing*, 853-860.

West, D., and K. Panesar. 1996. "Automatic Incremental State Saving." In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, 78-85.

Xu, Y., G. Tan, X. Li, and X. Song. 2014. "Mesoscopic Traffic Simulation on Cpu/Gpu." In *Principles of Advanced Discrete Simulation*, 39-49.

Ye, T., H. Kaur, S. Kalyanaraman, and M. Yuksel. 2008. "Large-Scale Network Parameter Configuration Using an on-Line Simulation Framework." *IEEE/ACM Transactions on Networking* 16:777–790.

Zhang, J. L., and C. Tropper. 2001. "The Dependence List in Time Warp." In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, 35-45.

AUTHOR BIOGRAPHIES

RICHARD FUJIMOTO is Regents' Professor in the School of Computational Science and Engineering at the Georgia Institute of Technology. He received a Ph.D. in Computer Science & Electrical Engineering from the University of California-Berkeley in 1983. His email address is fujimoto@cc.gatech.edu.

Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

PARALLEL AND DISTRIBUTED SIMULATION: TRADITIONAL TECHNIQUES AND RECENT ADVANCES

Kalyan S. Perumalla

Oak Ridge National Laboratory
One Bethel Valley Road
Oak Ridge, Tennessee 37831-6085, U.S.A.

ABSTRACT

This tutorial on parallel and distributed simulation systems reviews some of the traditional synchronization techniques and presents some recent advances.

1 INTRODUCTION

Parallel and distributed simulation finds relevance in many applications, including civilian applications such as telecommunication networks, physical system simulations and distributed multi-player gaming, and non-civilian applications such as battlefield simulations and emergency event training exercises, to name only a few. It deals with ways of using multiple processors in a single simulation. Achieving correctness of parallel execution requires synchronization across processors. Parallel simulation, at its core, is concerned with accurately synchronizing simulations that run on multiple inter-connected processors. All processors together serve to collectively simulate an integrated set of application models. The simulation is partitioned spatially (or temporally), and the partitions are mapped to processors. Although multiple processors can be employed to execute multiple separate single-processor simulation runs in parallel, here we focus on parallel simulation methods in which all processors are together used to execute a single simulation run, be it an integrated set of simulators or a single monolithic parallel system.

To be meaningful, the results produced by a parallel simulation run must ideally match those that could be produced by an equivalent sequential simulation run. To achieve this match, parallel execution must be properly synchronized to preserve the right orderings and dependencies during computation of simulation state across processors. One of the challenges in this synchronization is in minimizing the runtime execution overheads (memory, computation and communication) incurred during parallel execution. It is thus important to keep the overhead within acceptable levels, in order for the parallel execution to deliver sufficient value above and beyond sequential simulation. A large amount of research has been focused on reducing this overhead by devising efficient methodologies, algorithms and implementations for synchronization in parallel and distributed simulation.

In this vein, parallel and distributed simulation techniques have been well studied in the past two to three decades. The literature is now sizable, and several comprehensive books and survey articles, such as (Banks, et al. 1996, Fujimoto 1990, Fujimoto 1993, 2000), already serve well to document traditional techniques in parallel and distributed simulation. Recent emergence of new application demands, techniques and hardware platforms are resulting in enhancements to traditional techniques and formulation of newer techniques. These have been appearing at various places in the literature and together represent newer advances in parallel and distributed simulation. Here we present a brief overview of traditional techniques followed by a presentation of some of the recent advances.

The structure of the paper is as follows. Section 2 presents quick overview of basic concepts and classical techniques. Section 3 serves as a practical case study that describes the parallel time synchronization functionality provided by the IEEE standard called the High Level Architecture. Section 4 documents some of the recent developments. Section 5 discusses the issues and challenges raised by the expanding set of hardware platform types on which parallel simulation is now being performed. Finally, a summary in Section 6 recapitulates the material.

2 CLASSICAL TECHNIQUES

Parallel and distributed simulation approaches can be broadly categorized into spatial parallel and time parallel schemes. In spatial parallel schemes, the application is partitioned along spatial dimensions underlying the application's models (e.g., three dimensional grid cells in physical system simulation, or computers and network routers in Internet simulations). Time parallel schemes partition the simulation along its time dimension (e.g., regular time intervals along the simulation time axis).

Spatial decomposition is by far the most commonly used parallel simulation scheme. In this scheme, applica-

1-4244-0501-7/06/$20.00 ©2006 IEEE

tion models are partitioned into logical processes (LPs). Each LP contains its own individual state variables, and interactions among LPs are only via exchange of time-stamped events. The simulation progresses via execution of these events in temporal order. The temporal ordering is either maintained at every instant during simulation, or is achieved in an asymptotic manner (i.e., the system guarantees eventual convergence to overall temporal order). Here we focus only on spatial parallel schemes. Further information on time parallel schemes can be found in (Fujimoto 2000) and other places.

2.1 Basic Concepts

Let us first examine some basic concepts that underlie any synchronized parallel/distributed simulation, followed by an overview of basic synchronization approaches.

2.1.1 Notions of Time

In simulations, there are generally three distinct notions of time. The first is the *physical time*, which is the time in the physical system that is being modeled (e.g., 10-11pm on January 1990). The second is the *simulation time*, which is a representation of the physical time for the purposes of simulation (e.g., number of seconds since 10pm of January 1990, represented in floating point values in the range [0..3600] corresponding to the simulated time period of the physical time). Finally, the *wallclock time* is the elapsed real time during execution of the simulation, as measured by a hardware clock (e.g., number of milliseconds of computer time during execution). For each, the notions of *time axis* and *time instant* can be defined – time axis is the totally ordered set of time instants along the corresponding timeline. In particular, for simulation time, the time axis is common across all processors, and a processor's simulation time is its current time instant along the time axis up to which the processor has advanced its simulation.

2.1.2 Execution Pacing

In general, there is a one-to-one mapping from physical time to simulation time. In contrast, there may or may not exist a specific relationship between simulation time and wallclock time. The mode of simulation execution determines this particular relationship. In an as-fast-as-possible execution, the simulation time is advanced as fast as computing speed can allow, unrelated to wallclock time. In real-time execution, on the other hand, advances in simulation time are performed in lockstep with wallclock time – one unit of simulation time is advanced exactly in one same unit of wallclock time. A variation of real-time execution is scaled real-time execution, in which simulation time period is some constant factor times an equivalent wallclock time period.

Synchronization algorithms are required to provide correct execution, avoiding undesirable effects such as deadlocks, live-locks and termination problems. What more, in analytic simulations, which are executed in an as-fast-as-possible fashion, an important system goal is to minimize the overheads of synchronization such that the simulation completes as faster than real-time as possible. This adds the need for delivering rapid simulation progress, on top of correctness of parallel operation.

2.1.3 Events and Event Orderings

An event is an indication of an update to simulation system state at a specific simulation time instant. Thus each event specifies a timestamp. When events are exchanged among processors, their delivery at the receiving processors needs to be carefully coordinated at runtime. In general, multiple different types of delivery ordering systems can be defined. Two commonly used orderings are (1) receive-order (2) timestamp-order. Other types (Lamport 1978), such as causal order (Lee, et al. 2001), could also be useful in certain cases, but they are not as commonly used.

In receive-ordered delivery (RO), events from other processors are delivered to the receiving processors as and when the events arrive at the receiving processor. In contrast, in timestamp-ordered delivery (TSO), events are guaranteed to be delivered in non-decreasing order of their timestamps. Typically, since RO delivers events right away, RO events incur lower delivery delay/latency from the moment they are sent by a processor to the moment the destination processor(s) receives them. TSO events on the other hand undergo runtime checks and buffering until their non-decreasing timestamp order can be ascertained and guaranteed, and hence TSO events incur relatively higher latency. However, a significant difference arises with respect to modeling accuracy afforded by RO and TSO. RO cannot always preserve "before and after" relationships, while TSO does guarantee preservation of such relationships. Similarly, with TSO, all processors see the exact same ordering of events, whereas with RO, identical ordering among events cannot be guaranteed across processors. Overall parallel execution can be made repeatable with TSO from one execution to the next, while RO cannot ensure such repeatability.

2.1.4 Timestamp-Ordered (TSO) Processing

The rationale behind timestamp-ordered processing is that it permits the models to be accurately simulated, such that events are processed in the same order as their corresponding actions in the physical system. To enable such processing order, a simple local rule to follow is that a processor whose simulation time is at T should not receive events with timestamps less than T. Hence, advances of a processor's current time have to be coordinated and controlled

carefully to prevent events appearing in processor's "past." This requirement gives rise to different synchronization approaches, and consequently, different algorithms.

2.2 Basic Synchronization Approaches

In parallel simulation, broadly four approaches are commonly used: conservative, optimistic, relaxed, and combined synchronization.

Conservative: This approach always ensures safe timestamp-ordered processing of simulation events within each LP (Chandy and Misra 1978, 1981). In other words, an LP does not execute an event until it can guarantee that no event with a smaller timestamp will later be received by that LP. However, runtime performance is critically dependent on a priori determination of an application property called *lookahead*, which is roughly dependent on the degree to which the computation can predict future interactions with other processes in the absence of global information. In a lookahead-based approach, events that are beyond the next lookahead window are blocked until the window advances later, sufficiently far to cover those events. Typically the lookahead property is very hard to extract in complex applications, as it tends to be implicitly defined in the source code interdependencies. The appeal of this approach however is that it is easier to implement than the optimistic approach (described next) if such a lookahead value can be specified by the application.

Optimistic: This approach avoids "blocked waiting" by optimistically processing the events beyond the lookahead window (Jefferson 1985). When some events are later detected to have been processed in incorrect order, the system invokes compensation code such as state restoration or reverse computation (described later). A key issue introduced by large-scale platforms is the increased delay of inter-processor communication. Optimistic synchronization offers the potential for greater resilience to delays in the sense that computations may progress despite the delay in generation/delivery of certain events. Since blocking is not used, the lookahead value is not as important, and could even be specified to be zero without greatly affecting the runtime performance. While this approach eliminates the problem of lookahead extraction, it has a different challenge – namely, support for compensating code. Traditional optimistic methods rely on state saving or other techniques to enable rolling back to a previous state in case an event arrives in the "past".

Relaxed synchronization: This approach relaxes the constraint that events be strictly processed in time stamp order (e.g., see (Fujimoto 1999, Rao, et al. 1998)). For example, it might be deemed acceptable to process two events out of order if their time stamps are "close enough." This approach offers the potential of providing a simplified approach to synchronization, but without the lookahead constraints that plague conservative execution. A key challenge with this approach is determining the extent that ordering constraints can be relaxed without compromising the validity of the simulation results. An additional challenge lies in ensuring that the execution of the simulation is repeatable. Repeatability means multiple executions of the same simulation with the same inputs are guaranteed to yield the same numeric results from one execution to the next. This property may not be preserved with relaxed execution because events within each process may be processed in a different order from one execution to the next unless special care is taken.

Combined synchronization: This approach combines elements of the previous three. For example, sometimes it might help to have some parts of the application execute optimistically ahead (e.g., parts for which lookahead is low or hard to extract), while other parts execute conservatively (e.g., parts for which lookahead is large, or for which compensation code is difficult to generate) (e.g., see (Jha and Bagrodia 1994, Perumalla 2005, Rajaei, et al. 1993)). In such cases, a combination of synchronization techniques can be appropriate. A practical application of such a combined synchronization approach is the High Level Architecture, described in greater detail in Section 3.

3 CASE STUDY: THE HIGH LEVEL ARCHITECTURE

The US Department of Defense (DoD) High Level Architecture (2000), heretofore referred to as the HLA, includes support for time-synchronized parallel/distributed simulations, built on fundamental concepts of parallel and distributed simulation (PADS). In the HLA, an integrated execution of simulations is called a federation. Individual simulators participating in a federation are called federates. Federates can be of different types: pure software simulators such as computer generated forces, human-in-the-loop simulators such as virtual simulators, or live components such as instrumented weapon systems. In this section, we will use the terms "federate" and "processor" interchangeably.

As mentioned earlier, a significant amount of literature exists in the PADS research community, which has explored issues in time synchronized simulations. The time synchronization module of the HLA, called the Time Management (TM) has, in large part, been built on insights from PADS research. Thus, the fundamental concepts in HLA TM are common with those in PADS.

3.1.1 Interoperability Challenge

The HLA's TM services address two important components: (1) overall event processing order by each federate (2) synchronized event delivery to each federate.

While enabling event processing order and synchronized event delivery, all in a single encompassing standard

framework, the HLA needs to accommodate a large variety of individual types of simulators. In general, there is a plethora of different simulator types – event-stepped vs. time-stepped, sequential vs. parallel, real-time vs. as-fast-as-possible, conservative vs. optimistic, etc. An HLA federation might include any combination of any of these simulator types. Moreover, the exact combination of the types is not always made known *a priori* to the HLA RTI, and hence the interface as well as the implementation must be sufficiently general to accommodate any/all of them. The HLA TM interface does an amazing job of accommodating any arbitrary combinations of, and any number of instances of, different types of simulators, all in one core, seamless interface.

3.1.2 Synchronization Services

The HLA Time Management module provides a clear interface that each federate must invoke in order to synchronize with other processors. The three most commonly used services are: Time Advance Request (TAR), Next Event Request (NER) and Flush Queue Request (FQR).

A federate undertaking fixed time increments in simulation time can use TAR(T) to unconditionally advance its simulation time to T. Events from other federates that arrive with timestamps less than T are all delivered to this federate before the runtime permits the federate to advance to time T. Time-stepped parallel simulations typically use this service to coordinate their time steps across processors.

A federate operating under a discrete event paradigm invokes the NER(T) primitive to conditionally advance its simulation time to T. If the runtime discovers that other events with timestamps less than T are generated by other processors, the earliest of those events are delivered to the federate, and time is advanced only to their timestamp. This service is most commonly used by conservative parallel discrete event simulators.

A federate equipped to execute its events in optimistic mode (i.e., ahead of receiving guarantees of correctness) can invoke FQR(T) to force the runtime to release any and all events that it currently has, irrespective of their timestamps. The runtime utilizes the supplied timestamp T to compute absolute global time advances.

3.1.3 Computing LBTS

A fundamental role of a TM implementation is in computing a quantity known as Lower Bound on incoming Time Stamps (LBTS). At each federate, the LBTS value specifies a guarantee on the least timestamp on any future incoming event. In other words, no event will ever arrive at that federate with a timestamp smaller than LBTS. Once this global value is known, it is rather straightforward to locally serve TM requests, such as TAR, NER and FQR.

In order to compute the LBTS value at each federate, a distributed algorithm is required that exchanges messages to coordinate the LBTS computation without deadlocks, live-locks or undue performance degradation. Several such algorithms have been proposed in PADS literature – see for example (Perumalla and Fujimoto 2001). A close cousin to the LBTS computation is Global Virtual Time (GVT) computation in optimistic simulation (Bellenot 1990). Another closely related work in general distributed processing is that of distributed "flush barrier" algorithms (Ahuja 1990). Analogous to these algorithms, several variants exist for LBTS computation.

One such algorithm is based on global asynchronous distributed reductions. In this algorithm, the minimum local (conditional) guarantee on timestamps of events that could be generated is taken at each federate, and a global reduction algorithm is used to find the minimum of all the local minima. This can be performed fairly quickly and scalably, in $\log(N_p)$ steps, where N_p is the number of federates, using a butterfly pattern of communication (Brooks 1986). Assuming there are no events in transit across federates, the minimum of the minima gives a tight lower bound on LBTS.

3.1.4 Transient Messages

What if there are some events that are in transit in the network while the global minimum of local minima is being computed? This is called the transient event problem, in which some events could become potentially unaccounted for if they are not considered into the global algorithm. There exist different schemes by which transient events can be accounted for, albeit at the cost of either additional messages being sent/received and/or additional time spent blocking while waiting for all transient events to reach their destinations. A popular one is called the Mattern's algorithm (Mattern 1993) in which distributed consistent cuts are used to mark and recognize events belonging to distributed different snapshots.

For conservative parallel simulations, it is clear that the larger the lookahead, the fewer the number of LBTS computations that need to be performed, because of increased concurrency enabled by the larger lookahead.

3.1.5 Serving Synchronization Requests

The RTI internally maintains a priority queue of TSO events, ordered by their timestamps. When a federate invokes TAR(T), the RTI first examines if LBTS is greater than T. If so, the request is trivially satisfied – the RTI delivers all events from its TSO queue whose timestamps are less than or equal to T, and then issues a TAG(T). If T is greater than LBTS, then the RTI initiates a new distributed LBTS computation (if one is not already in progress). The lesser of T and minimum timestamp in TSO queue is used

as this federate's contribution in the LBTS computation. The operation is similar for NER(T) invocations as well, except that the TAG time could be smaller than T if events with timestamps earlier than T are delivered.

3.2 Other HLA TM Services

In addition to supporting basic integration of conservative federates, the HLA TM services include primitives to integrate federates that use advanced simulation methods, such as event retractions and optimistic event execution.

3.2.1 Event Retractions

In simulations, models are sometimes written to un-schedule previously generated events. For example, although a move event is scheduled on an entity at T, it might have to be retracted later if the entity gets destroyed after the event is scheduled but prior to T. Such event retractions are called user-level retractions. Typically, user-level retractions are enabled as follows. When an event is scheduled, the system returns a handle to that event. Later, if and when that event needs to be retracted, a retract primitive is invoked to which the event handle is given. The system then un-schedules that event. The HLA RTI provides such a framework using event handles and retraction primitive. Interestingly, the same service is also used for "system-level" retraction in optimistic simulations, as described next.

3.2.2 Optimistic Event Execution

As mentioned previously, the HLA supports conservative federates as well as optimistic federates, as well as their arbitrary combinations. Optimistic federates differ from their conservative counterparts in that they do not discard events after processing them. Instead they keep the events around, and also maintain copies of simulation states before modifying them as part of event processing. Since optimistic federates do not rely on lookahead, they execute their events without blocking for safety. In particular, they use the FQR(T) service of the RTI to force the RTI to deliver events from its TSO queue even if LBTS has not progressed past T. The difference between FQR and NER is that FQR does not guarantee that it has delivered all events with timestamp less than T. Thus, the federate will have to rollback its computation if/when it later receives events whose timestamp is less than T. There are two main parts to such rollback: (1) undo local computation by restoring the state prior to erroneous event processing (2) undo all events erroneously sent to other federates. The first part is typically federate-specific, and hence the HLA does not provide a standard service for it. The second part is realized by using the event retraction service described previously. When an optimistic federate receives a retraction request, it performs an event annihilation procedure canceling the original event.

Note that the HLA RTI shields conservative federates from optimistic events by holding on to optimistic events in RTI TSO queues until such a time that LBTS sweeps past their timestamps. If the optimistically scheduled events happen to get retracted by their sending federates, those events will get annihilated within the RTI's TSO queues without ever getting delivered to the (conservative) destination federate.

4 RECENT ADVANCES

While parallel and distributed simulation as an area has been well studied, some advances are being made recently along multiple fronts. These advances are driven by new challenges raised by high-performance needs of applications such as large-scale telecommunication network simulation (e.g., Internet-scale TCP/IP simulations), hardware/VLSI system simulations. Some of these advances are examined next.

4.1 Mixed Synchronization

When large application scenarios are considered, it often is the case that the models are heterogeneous in nature, with varying amounts of computational loads across spatial dimensions and varying levels of modeling. For example, simulations of the Earth's magnetosphere inherently contain models of varying granularity and synchronization requirements (Karimabadi, et al. 2005).

In contrast to the heterogeneity needs, the traditional method of prevalence in building parallel simulation systems is to build a system specifically for one synchronization method (e.g., one conservative algorithm, or one optimistic variant). This tradition has two fallouts. First, additions to the underlying framework involve major overhauls. Secondly, modelers need to either determine and stick to one mechanism, or re-code their models to switch to a new mechanism.

A set of systems are being constructed to alleviate this problem when the application needs to accommodate multiple synchronization types and be resilient to future changes to parts of its models. An early system that provided this type of interface is the Maisie language (Bagrodia and Liao 1994), in which entities (equivalent to LPs) are capable of choosing and/or changing their own conservative/optimistic modes dynamically at runtime. Later, the HLA has been designed to accommodate multiple simulator types, including optimistic and conservative ones, although this has been leveraged mostly in the context of coarse-grained simulators, and is not readily amenable to high-performance execution of fine-grained simulations. More recently, the μsik system (Perumalla 2005) was developed that provides a fine-grained simulation en-

vironment for mixed mode synchronization across shared memory and distributed memory platforms.

4.2 Lookahead Extraction

In purely conservative parallel simulations, the single most important parameter is the lookahead value. A small lookahead can have a large detrimental effect on overall runtime performance. Thus lookahead extraction has been a subject of intense study in PADS literature.

Recently many important analytic applications have been developed using conservative synchronization, largely in the network simulation context. In these applications, the models are prohibitively complex to apply optimistic simulation techniques but are also very tightly-coupled, exhibiting very small lookahead between LPs. Wireless network simulation is one such application (e.g., predicting the performance of IEEE 802.11 Wi-Fi networks). Static lookahead in such applications is typically very small, given by the time for electromagnetic signal propagation (at close to speed of light) over short distances. When coupled with the need to use such simulations in real-time context (for network emulation), the gain from improving the lookahead even by a small amount can be quite significant.

It is precisely in this context that recent efforts have focused on extracting as much lookahead information as possible from the models. Moving from purely static estimates of lookahead, these new techniques examine the LP inter-dependency structures dynamically at simulation runtime, and manage to extract values of lookahead that are higher than those estimated statically. Two such efforts are presented in (Zhou, et al. 2004) and (Liu and Nicol 2002).

4.3 Critical Channel Traversal

Another method of improving performance of conservative simulations focuses on scheduling alternatives. In the most common event scheduling approach, events from the future event list are executed in an earliest-timestamp-first manner. While being very simple to implement, such a scheme is less than optimal when caching and other effects are considered. Newer scheduling algorithms such as Critical Channel Traversal are designed for exposing and taking advantage of inherent locality of event execution in conservative simulations. Benefits include better cache performance due to immediate reuse of event buffers across causal chain of events and improved spatial locality due to reflection of LP inter-dependence on event processing patterns. The gains from such sophisticated event scheduling approaches have been shown to be significant, with very high event rates achieved even on fine-grained applications such as ATM network simulations. LPs that are "closely related" with respect to event communication are grouped together as "tasks". Tasks are then used as the primary scheduling units, and LPs within a task form secondary scheduling units. When LPs within a task interact among themselves more heavily than with other LPs outside that task, locality is enhanced and event scheduling overheads are decreased. Other beneficial effects include automatic realization of dynamic load balancing on shared-memory multiprocessor platforms, because processors operate on a dynamic pool of active tasks rather than with a traditional static assignment of LPs to processors.

While CCT is useful for enhancing performance, it does require the application be amenable to grouping of LPs into tasks, and is specific to conservative (lookahead-based) parallel simulations.

4.4 Optimizations for Efficient Rollback

While lookahead extraction is the bane of conservative parallel simulations, rollback support mechanism is its counterpart in optimistic parallel simulations. Efficient rollback techniques thus play an important role in the making or breaking of optimistic schemes. For a long time, checkpointing-based approaches were the dominant form of enabling rolling back computations. However, these imposed undue amount of overheads, making optimistic simulation schemes unappealing to many applications. An alternative scheme called reverse computation has been recently developed to alleviate this problem, as described next.

4.4.1 State Saving for Rollback

One of the common approaches for rolling back incorrect computations is based on checkpointing, also called state-saving. Checkpointing methods make a copy of the state variables of LPs before the variables are overwritten by optimistic event computations. If and when those event computations have to be negated (due to arrival of a remote event with an earlier timestamp, or due to withdrawal/cancellation of an event by a remote processor), state variable values are fetched from the checkpoint log and overwritten, thereby restoring the LP state to their correct values corresponding to the fault point.

The most commonly used state-saving technique is copy state-saving (CSS), in which a copy of the entire state is made before an event is executed. In CSS, the state is saved every time an event is processed. A variation of copy state-saving is called periodic state saving (PSS). PSS is a generalized technique in which state is saved only periodically, say, every p^{th} event, instead of every event as is done with CSS. This implies that some events save state before processing, and others do not. The former set of events can be rolled back easily by restoring the state to the saved values. The latter set of events needs special treatment, since they do not have saved state. The state restoration for these events is achieved by starting with a past processed

event that does have saved state, and then re-executing the sequence of events from that past event to the event just before the rolled back event.

Incremental state-saving (ISS) is an alternative state-saving technique in which only the modified portions are saved just before modification. Modifications to state are logged as pairs of address-value pairs, and stored in a log array for each LP. Optimistic simulators can permit CSS, PSS and ISS to be used simultaneously together in the same application, although most applications tend to use them mutually exclusively, because either the LP states are uniformly too large (warranting ISS) or uniformly small (warranting CSS) or in between (warranting PSS). A detailed treatment of compiler-based techniques to optimize the appropriate state saving operations is given in (Gomes 1996).

As it turns out, state saving techniques imposes two types of penalties: (1) a memory copy overhead during forward computation (2) memory consumption beyond that of conservative execution. The degree of severity varies with the particular variant of state saving, but they carry across all variants. The first penalty reduces the efficiency of parallel execution (e.g., a four-processor optimistic simulation running at the same speed as that of a uniprocessor conservative simulation). The second type of penalty introduces not only undue memory overheads for fine-grained applications (fine grained events, by definition, execute rapidly, and therefore can produce a long state vector log equally rapidly), but also pollute the processor cache, thereby reducing forward computation speed.

4.4.2 Reverse Computation

Alternatives to checkpointing solutions evaded the community until recently a reverse computation (RC) approach was proposed. In this approach, values of state variables are not saved for rollback support. Instead, as and when rollback is initiated, a perfect inverse of event computation is invoked that serves to recover the modified state values to their original values. In other words, the overwritten state values are reconstructed by executing the forward code backwards.

The RC approach has been successfully applied in different application domains, such as telecommunication network simulations (Carothers, et al. 1999, Garrett, et al. 2003) and physical system simulations (Tang, et al. 2005). Among optimistic simulation systems that currently support RC are ROSS (Carothers, et al. 2002) and μsik (Perumalla 2005).

Note that PSS and RC can be seen as duals of each other. Figure 1 depicts the relation between PSS and RC using a generic snapshot of execution at which rollback is initiated. The rightmost vertical bar represents the latest optimistically executed event. The light vertical bar in the middle represents the point to which computation needs to be rolled back. The left-most dark vertical line indicates the most recent periodic checkpoint before the rollback point. If RC is used, reverse execution is invoked on all events from left to middle. If PSS is used, coast forwarding computation (event re-execution to regenerate state) is invoked from left to middle. Clearly, the efficiency of RC or PSS depends on expected distance of left or right points, respectively, from the rollback point.

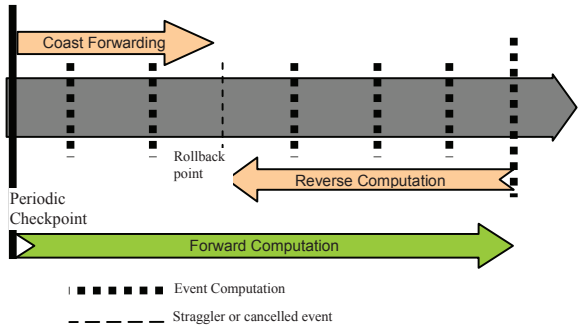


Figure 1: Comparison of Rollback Operation Using Reverse Computation vs. Periodic State Saving

However, a crucial difference between RC and PSS is that the state size is independent of the computing platform for RC, whereas it depends on sizes of data types in PSS. Also, PSS has the drawback that not all events from time points earlier than GVT/LBTS can be reclaimed for reuse (i.e., cannot be fossil-collected). Those events that fall between the periodic checkpoint time and GVT/LBTS need to be retained for coast-forwarding (Perumalla 1999).

4.5 Intra-Processor Lookahead

In large-scale application scenarios, each processor typically hosts many LPs. For example, in Internet simulations, each processor simulates thousands of network nodes. Traditional conservative parallel simulation maps network links crossing processor boundaries to in/out channels in null-message algorithms, and uses the minimum among all transmission delays as the lookahead values on all those out channels, irrespective of delays added by internal network nodes. However, time advances can be improved significantly if the additional topology information can be incorporated into the null message algorithm. The concept of "internal lookahead", illustrated in Figure 2, is defined as the minimum among all shortest paths between all pairs of in- and out-channels. This is easily computed as the sum of transmission delays along the shortest paths from all in channels to all out channels, which can add up to much more than lookahead along one transmission link. Adding this "internal lookahead" to null message timestamps significantly improves concurrency of the conservative simulation, delivering time advances in increments larger than one lookahead. The internal lookahead feature is supported in newer systems such as libSynk (Perumalla 2004).

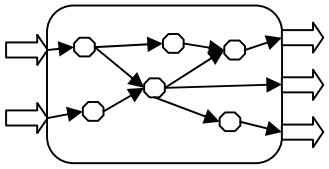


Figure 2: Illustration of Internal Lookahead

4.6 PADS Applied to *Sequential* Execution

Interestingly, even though parallel/distributed simulation techniques are developed specifically for execution on parallel computing platforms, they are in fact being discovered to be useful even in a sequential processing (uniprocessor) setting. When properly applied, the concepts developed for parallel execution can also be used to improve the speed of sequential simulation as an efficient alternative to traditional execution based on a central event list.

4.6.1.1 Conservative Execution on 1 Processor

The beneficial effects of parallel/distributed simulation techniques for sequential execution arise primarily due to (a) lower priority queue overheads (b) better caching effects (Curry, et al. 2005, Simmonds, et al. 2002). In a central event list implementation, events are organized into one priority queue (PQ), whereas in a parallel implementation, they are organized into multiple PQs. If N_e is the size of PQ encountered on average by each event, each event insertion/deletion incurs $O(\log(N_e))$ time overhead (assuming a general-purpose PQ implementation). On the other hand, if parallel simulation of the same application organizes the simulation as N_p logical processes, then, with algorithms such as critical channel traversal (Xiao, et al. 1999), each insertion/deletion of an event incurs priority queue sizes that are smaller than N_e/N_p. This makes the amortized event overhead less than $O(\log(N_e/N_p) + \log(N_p))$, where $\log(N_e/N_p)$ is the time to enqueue events into event priority queues and $\log(N_p)$ is the time taken to order the LPs by their earliest event timestamps. Moreover, reuse of memory buffers induces good caching effects which results in significant increase in performance. Such reuse of buffers for events processes immediately after one another naturally occurs for a chain of causally related events that fall within the same safe-time window.

4.6.1.2 Optimistic Execution on 1 Processor

Similar to application of conservative parallel simulation to sequential simulation, there are two different ways in which sequential simulation can benefit from optimistic parallel simulation methods: aggregate event processing and pipe-lined event processing.

In aggregate event processing, two or more events in the future event list are scheduled together (invoked with one "aggregate event handler"), with the expectation that no intermediate event computation arises between them to break their aggregate. In other words, the runtime behaves as though the events are scheduled as one event, rather than as multiple events. Rollback is initiated on the aggregate computation if violation of this assumption is detected at runtime (e.g., when a new event is scheduled by one of the events in the aggregate with timestamp intervening the aggregate's range of timestamps). The advantage of this approach is two-fold: event scheduling costs are reduced and compiler optimizations (such as common subexpression elimination) are realized on the aggregate event handler.

In pipe-lined event processing, newly generated events are executed at the tail of event handler of the generating event, bypassing the event queue, thereby reducing event queuing overheads and improving locality. Rollback is initiated on the tail-executed events if their time order is detected to be violated by later events.

4.6.1.3 Supercomputer-based Parallel Simulations

Another recent advance is in the realm of large-scale parallel computing platforms. Applications such as Internet simulations are serving to motivate the use of high-end computing for parallel discrete event simulation. With nascent interest in utilizing supercomputing facilities for such simulations, a few leading efforts have focused on porting, testing and improving traditional techniques on supercomputers. The state of the art in this area has now pushed the capabilities to one thousand processors and beyond.

Some of these include the demonstration of the PHOLD benchmark executed on 1,033 processors on the Pittsburgh Supercomputing Center using the DSIM Time Warp system (Chen and Szymanski 2005), and the RC-based PHOLD execution on 1,024 processors using the μsik system (Perumalla 2005) at the San Diego Supercomputing Center. The latter includes conservative, optimistic and mixed-mode simulations. Figure 3 shows processing time per event on PHOLD with increasing number of LPs and message population (MSG), with the largest recorded configuration containing 1 million LPs and 1 billion messages active at all times during the simulation. The amortized overhead of less than 20 microseconds per event brings it within the reach of even the most demanding fine grained simulation.

Conservative simulations have pushed the scalability limits to 512 processors on physical system simulations (Perumalla, et al. 2006) and to 1,536 processors (Fujimoto, et al. 2003) on network simulations. Additional refinements were shown using a variant of null message algorithm in (Park, et al. 2004).

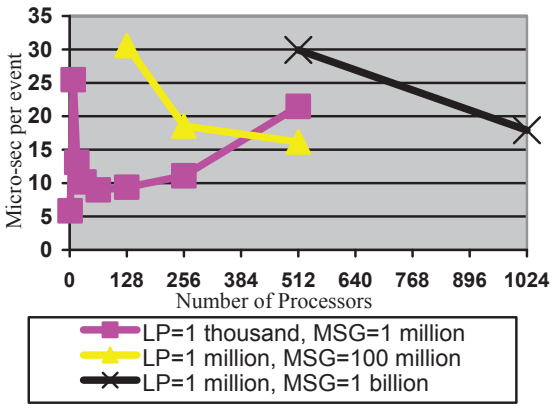


Figure 3: Runtime Performance Results from Optimistic Simulation of the PHOLD Benchmark on a Supercomputer

4.7 Approximate Time

Another exciting development in relatively recent past is the articulation of a new concept called Approximate Time. In this world model, the traditional notion of absoluteness of timestamps is relaxed into a range, rather than a point, on the simulation time axis. The range notion hinges on the realization that models typically have implicit fuzziness in the timestamps they generate (e.g., variation in the time taken for a battle tank to travel a given distance). This implicit uncertainty in timestamps can be exploited to relax the tight lookahead-based dependencies across processors, and consequently the parallel simulation speed can be improved. Systems that exploit temporal uncertainty are documented in (Fujimoto 1999, Loper 2002, Loper and Fujimoto 2004).

4.8 Fault Tolerance and Security

With the increase in the number of processors/federates participating in a large integrated simulation, the probability increases that one or more of the processors fail during simulation. Transparent techniques to sustain the progress of the simulation despite such crashes is a subject of recent interest. Schemes to incorporate such fault tolerance and/or continued optimistic execution are now being explored (Santoro and Quaglia 2006) (Chen, et al. 2006).

Also, secure exchange of information across processors without compromising confidentiality of information contained in models is an important facet of interoperable integrated simulations (e.g., preventing undesirable leaking of model information in joint exercises by different nations). This concern for security of information exchange during simulation raises new challenges, such as runtime overhead for encrypted communication, which are beginning to be addressed only recently.

5 EXPANDING SIMULATION PLATFORMS

Another exciting dimension of new advances in parallel simulation is in the use of novel computing platforms. With the advent of new computing hardware such as network co-processors and general-purpose graphical processing units, parallel simulation techniques are being revisited. Here we document some recent efforts in this light.

5.1 General-Purpose Graphical Processing Units

Traditionally, graphics cards for workstations and personal computers have been designed to handle intensive graphics operations to enable high speed rendering of complex objects and scenes. More recently, their programmability has reached a point to make them suitable for more general-purpose computation (Owens, et al. 2005, Pharr and Fernando 2005). As illustrated in Figure 4, certain applications have been shown to execute much faster on GPGPUs than on CPUs (Owens, et al. 2005). Texture data are input to a fragment processors (FPs), which "render" the results of their computation to target textures.

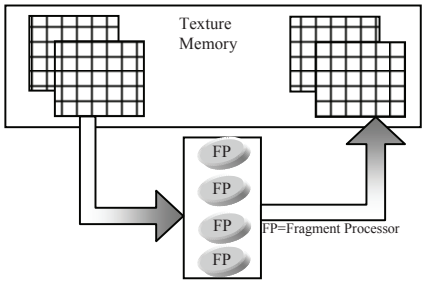


Figure 4: Simplified Schematic of GPGPU Operation

Since GPGPUs lack efficient random write access to memory (texture) locations, the traditional discrete event loop cannot be employed. Instead, algorithms such as the one shown in Figure 5 are more suitable. Computing the step $dt_{min}=min(dt_i)$, for example, is efficient on GPGPUs, taking only logarithmic time.

```
While not end of simulation
  /*Find next update times for all LPs*/
  For all (i): dt_i = compute_dt(i)
  /*Find minimum among all update times*/
  dt_min = min(dt_i) of all (i)
  /*Advance current simulation time*/
  t_now += dt_min
  /*Advance all LPs to current time*/
  For all (i): compute_state(i, dt_min)
```

Figure 5: A Variant of the Typical Sequential Discrete Event Algorithm Modified to Suit Execution on GPGPUs

While time-stepped approaches have been studied in simulations using GPGPUs, not enough work has focused

on discrete event simulation (DES) on GPGPU. Some recent work has shown the potential of GPGPUs for discrete event simulation (Perumalla 2006), showing four-fold speedup of GPGPU-based execution over that of regular processor.

5.2 Network Co-Processors and Hybrid Processors

The programmability of network co-processors is also being exploited to speed up certain operations. One of the uses is in off-loading state saving and other checkpointing functionality to programmable Myrinet network switches (Quaglia and Santoro 2003, Santoro and Fujimoto 2004). This helps free up the main processor to continue with main computation, minimizing rollback overhead. Another interesting use of co-processor is in fast computation of LBTS values in optimistic simulation (Rosu, et al. 1997), exploiting the low latency access to network operations by programs executing on the network card.

5.3 Other Architectures

Other architectures include Web-based execution, such as used in Extensible modeling and simulation framework (Moves-Institute 2006), and Internet/grid-based architectures, as used in the master-worker paradigm of (Park and Fujimoto 2006). The Cell processor from IBM is another new architecture that is poised to be suitable for high-performance parallel simulation, similar to the GPGPUs.

Tera- and Peta-Scale Computing Platforms such as the Blue Gene/L and Cray XT3 impose unprecedented scalability challenges for which it is unclear if traditional techniques will be adequate. For example, it is unclear if fully peer-to-peer architecture of traditional parallel simulation is appropriate in the presence of frequent node failures. Concerns such as fault tolerance and security are poised to become critical on such large-scale platforms. Similarly, recent multi-core processor architectures are yet to be fully explored. For example, key problems such as contention for I/O and communication among cores on a single system need to be resolved, possibly via staggered execution.

6 SUMMARY

This tutorial presented an overview of parallel and distributed simulation systems, their traditional synchronization approaches and a case study using the HLA standard interface and implementation. Recent advances, such as scalability to supercomputing platforms and novel rollback techniques have been presented. The interaction of parallel simulation with newly emerging hardware architectures is outlined.

The future outlook seems to warrant focus on needs from larger scale simulation scenarios (e.g., large-scale human behavior modeling & simulation, and large traffic simulations), to be achieved on high-end computing platforms. There is also interest on high-performance simulations on low-end platforms such as using the multi-core architectures, GPGPUs and other co-processor-based systems. However, practical challenges remain to be explored, including: (a) wide spectrum of network latencies (b) highly dynamic participation by processors (c) semantics and implementations of always-on presence for large simulations.

ACKNOWLEDGEMENTS

This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

REFERENCES

"IEEE Std. 1516. 2000. High Level Architecture," in *Institute of Electrical and Electronic Engineers*.

Ahuja, M. 1990. Flush Primitives for Asynchronous Distributed Systems. *Information Processing Letters*5-12..

Bagrodia, R. and W.-T. Liao. 1994. Maisie: A Language for the Design of Efficient Discrete-Event Simulations. *IEEE Transactions on Software Engineering*, vol. 20(4), pp. 225-238.

Banks, J., J. S. Carson II, and B. L. Nelson. 1996. *Discrete-Event System Simulation*. Upper Saddle River, N.J.: Prentice Hall.

Bellenot, S. 1990. Global Virtual Time Algorithms," in *Proceedings of the SCS Multiconference on Distributed Simulation*: Society for Computer Simulation, pp. 122-127.

Brooks, D. E. 1986. The Butterfly Barrier. *The International Journal of Parallel Programming*, vol. 14295-307.

Carothers, C., D. Bauer and S. Pearce. 2002. ROSS: A High-Performance, Low Memory, Modular Time Warp System. *Journal of Parallel and Distributed Computing*, vol. 62(11), pp. 1648-1669.

Carothers, C., K. S. Perumalla and R. M. Fujimoto. 1999. Efficient Optimistic Parallel Simulations using Reverse Computation. *ACM Transactions on Modeling and Computer Simulation*, vol. 9(3), pp. 224-253.

Chandy, K. M. and J. Misra. 1978. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, vol. SE-5(5), pp. 440-452.

Chandy, K. M. and J. Misra. 1981. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, vol. 24(4), pp. 198-205.

Chen, D. and B. K. Szymanski. 2005. DSIM: Scaling Time Warp to 1,033 Processors. In *Proceedings of the 2005 Winter Simulation Conference*. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Chen, D., S. J. Turner and W. Cai. 2006. A Framework for Robust HLA-based Distributed Simulations. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*.

Curry, R., C. Kiddle, R. Simmonds and B. Unger. 2005. Sequential Performance of Asynchronous Conservative PDES Algorithms. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*.

Fujimoto, R. M. 1990. Parallel Discrete Event Simulation. *Communications of the ACM*, vol. 33(10), pp. 30-53, 1990.

Fujimoto, R. M. 1993. Parallel Discrete Event Simulation: Will the Field Survive? *ORSA Journal on Computing*, vol. 5(3), pp. 213-230.

Fujimoto, R. M. 1999. Exploiting Temporal Uncertainty in Parallel and Distributed Simulations. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*.

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems*: Wiley Interscience.

Fujimoto, R. M., K. S. Perumalla, A. Park, H. Wu, M. Ammar, and G. F. Riley. 2003. Large-Scale Network Simulation - How Big? How Fast? In *Proceedings of the 11th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*.

Garrett, Y., D. C. Christopher and K. Shivkumar. 2003. Large-Scale TCP Models Using Optimistic Parallel Simulation. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation.*

Gomes, F. 1996. Compiler Techniques for State Saving in Parallel Discrete Event Simulation, thesis, University of Calgary, Canada.

Jefferson, D. 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems*, vol. 7(3), pp. 404-425.

Jha, V. and R. Bagrodia. 1994.A unified framework for conservative and optimistic distributed simulation. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*..

Karimabadi, H., J. Driscoll, Y. Omelchenko, K. S. Perumalla, R. M. Fujimoto, and N. Omidi. 2005. Parallel Discrete Event Simulation of Grid-based Models: Asynchronous Electromagnetic Hybrid Code. *International Conference on Applied Parallel Computing,* 2005.

Lamport, L. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21(7), pp. 558-565.

Lee, B.-S., W. Cai and J. Zhou. 2001. A Causality Based Time Management Mechanism for Federated Simulations. In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, pp. 83-90.

Liu, J. and D. Nicol. 2002. Lookahead Revisited in Wireless Network Simulations. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*.

Loper, M. 2002. Approximate Time and Temporal Uncertainty in Parallel and Distributed Simulation, thesis, Georgia Institute of Technology.

Loper, M. and R. Fujimoto. 2004. A Case Study in Exploiting Temporal Uncertainty in Parallel Simulations. *International Conference on Parallel Processing*, 2004.

Mattern, F. 1993. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing*, vol. 18(4), pp. 423-434.

Extensible Modeling and Simulation Framework (XMSF). 2006. Moves-Institute. Available online via <http://www.movesinstitute.org/xmsf/xmsf.html>.

Owens, J. D., D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. 2005. A Survey of General-Purpose Computation on Graphics Hardware. *Eurographics 2005*.

Park, A. and R. Fujimoto. 2006. Aurora: An Approach to High Throughput Parallel Simulation. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*.

Park, A., R. M. Fujimoto and K. S. Perumalla. 2004. Conservative Synchronization of Large-scale Network Simulations. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*.

Perumalla, K. S. 1999. Techniques for Efficient Parallel Simulation and their Application to Large-scale Telecommunication Network Models, thesis, Georgia Institute of Technology.

libSynk, Perumalla, K. S. 2004. Available online via <http://www.cc.gatech.edu/computing/pads/kalyan/libsynk.htm>.

Perumalla, K. S. 2005. µsik - A Micro-Kernel for Parallel/Distributed Simulation Systems. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*.

Perumalla, K. S. 2006. Discrete Event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs). In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*.

Perumalla, K. S. and R. M. Fujimoto. 2001. Virtual Time Synchronization over Unreliable Network Transport.

In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*.

Perumalla, K. S., R. M. Fujimoto and H. Karimabadi. 2006. Scalable Simulation of Electro-magnetic Hybrid Codes. In *Proceedings of the 6th International Conference on Computational Science*.

Pharr, M. and R. Fernando. 2005. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*: Addison Wesley Professional.

Quaglia, F. and A. Santoro. 2003. Non-blocking Checkpointing for Optimistic Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems*, vol. 14(6), pp. 593-610.

Rajaei, H., R. Ayani and L.-E. Thorelli. 1993. The Local Time Warp Approach to Parallel Simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*.

Rao, D. M., N. V. Thondugulam, R. Radhakrishnan and P. A. Wilsey. 1998. Unsynchronized Parallel Discrete Event Simulation. In *Proceedings of the 1998 Winter Simulation Conference*, pp. 1563-1570. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Rosu, M., K. Schwan and R. M. Fujimoto. 1997. Supporting Parallel Applications on Clusters of Workstations: The Intelligent Network Interface Approach. *IEEE Symposium on High Performance Distributed Computing*, 1997.

Santoro, A. and R. M. Fujimoto. 2004. Off-Loading Data Distribution Management to Network Processors in HLA-Based Distributed Simulations. *Distributed Simulations and Real-Time Applications*, 2004.

Santoro, A. and F. Quaglia. 2006. Transparent Optimistic Synchronization in HLA via Time Management Converter. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*.

Simmonds, R., C. Kiddle and B. Unger. 2002. Addressing Blocking and Scalability in Critical Channel Traversing. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*.

Tang, Y., K. S. Perumalla, R. M. Fujimoto, H. Karimabadi, J. Driscoll, and Y. Omelchenko. 2005. Optimistic Parallel Discrete Event Simulations of Physical Systems using Reverse Computation. In *Proceedings of the19th Workshop on Principles of Advanced and Distributed Simulation*.

Xiao, Z., B. Unger, R. Simmonds and J. Cleary. 1999. Scheduling Critical Channels in Conservative Parallel Discrete Event Simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*.

Zhou, J., Z. Ji, M. Takai and R. Bagrodia. 2004. MAYA: Integrating hybrid network modeling to the physical world *ACM Transactions on Modeling and Computer Simulation*, vol. 14(2), pp. 149-169.

AUTHOR BIOGRAPHIES

KALYAN S. PERUMALLA is a senior researcher in the Computational Sciences and Engineering Division at the Oak Ridge National Laboratory. He also holds an adjunct faculty appointment with the College of Computing, Georgia Tech. He received a Ph.D. in Computer Science from Georgia Tech (1999). Dr. Perumalla has over 10 years of research and development experience in the area of parallel and distributed simulation systems, including high-performance runtime infrastructures and large-scale simulation, and has published widely on these topics. He co-developed the Federated Simulations Development Kit (FDK), a widely-disseminated high-performance runtime infrastructure for HLA-like distributed simulator federations. He has also built several additional research prototype systems and tools (e.g., for distributed debugging, network modeling, interoperable simulations and parallel optimization), most of which are in use by researchers worldwide. He has served as co-PI on multiple federally-funded projects on scalable parallel/distributed discrete event simulation systems. His e-mail is <perumallaks@ornl.gov> and his Web address is <http://www.ornl.gov/~2ip>.

Lecture 8. Distributed Simulation using HLA

Papers to study for the exam

- Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. 1997. The Department of Defense High Level Architecture. In Proceedings of the 29th conference on Winter simulation (WSC '97), Sigrún Andradóttir, Kevin J. Healy, David H. Withers, and Barry L. Nelson (Eds.). IEEE Computer Society, Washington, DC, USA, 142-149.
  DOI=http://dx.doi.org/10.1145/268437.268465
  Retrieved from: http://www.informs-sim.org/wsc97papers/0142.PDF

- Christopher D. Carothers, Richard M. Fujimoto, Richard M. Weatherly, and Annette L. Wilson. 1997. Design and implementation of HLA time management in the RTI version F.0. In Proceedings of the 29th conference on Winter simulation (WSC '97), Sigrún Andradóttir, Kevin J. Healy, David H. Withers, and Barry L. Nelson (Eds.). IEEE Computer Society, Washington, DC, USA, 373-380. DOI=http://dx.doi.org/10.1145/268437.268511
  Retrieved from: http://www.informs-sim.org/wsc97papers/0373.PDF

DESIGN AND IMPLEMENTATION OF HLA TIME MANAGEMENT IN THE RTI VERSION F.0

Christopher D. Carothers
Richard M. Fujimoto

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280, U.S.A.

Richard M. Weatherly
Annette L. Wilson

The MITRE Corporation
7525 Colshire Drive
McLean, Virginia 22102-3481, U.S.A.

ABSTRACT

The DoD High Level architecture (HLA) has recently become the required method for the interconnection of all DoD computer simulations. The HLA addresses the rules by which simulations are designed to facilitate interoperability, the method by which information exchanged between simulations is described, and a standard set of software services provided by a common Runtime Infrastructure (RTI). The RTI is responsible for the coordination of collections of cooperating simulations. The familiarization version of the RTI, dubbed F.0, was developed at the MITRE Corporation. One of the core components of the RTI is Time Management and is the focus of this paper. In particular, we present the design and algorithms used to implement the HLA Time Management Services in F.0.

1 INTRODUCTION

The HLA consists of 3 parts: (1) HLA rules, (2) HLA Interface Specification (IFSpec), and (3) Object Model Template. The software that meets the interface requirements set forth by the IFSpec is called the Runtime Infrastructure (RTI). For more information on the HLA and its parts we refer the reader to the DMSO web-page at `http://www.dmso.mil`.

The familiarization version of the RTI, dubbed *F.0*, was developed at the MITRE Corporation. The purpose of this version is to (i) educate the user community in the correct usage of the RTI, (ii) serve as a platform for introducing new ideas and concepts that will improve quality and effectiveness of the RTI, and (iii) provide a capability until commercial RTIs are available.

Within the RTI, one of the core components concerns how time is managed and is the focus of this paper. Here, we describe the design decisions and algorithmic implementation of RTI F.0 that support the HLA Time Management Services. Section 2 sets the stage for this discussion by presenting the design requirements for Time Management and the key properties used to differentiate federate types. Based on these requirements, the RTI's time management services are described in Section 3 and their corresponding algorithmic implementation is described in Section 4. Section 5 concludes this study and presents directions for future work in Time Management.

2 REQUIREMENTS

To achieve the HLA's interoperability requirements, time management *must* appear transparent to all federates engaged in a given federation. Insupporting this transparency, the RTI must allow for: (i) federates that may or may not be constrained by a time-flow mechanism, (ii) constrained federates with different and dynamically changing time-flow mechanisms, e.g., event driven or time-stepped federates or those that switch between these two during execution, (iii) federates executing on parallel/distributed platforms using a conservative or optimistic synchronization protocol including those that dynamically change between them, (iv) federates with different event ordering requirements, e.g., receive or time stamp order, (v) federates using a mixture of event orderings and transportation services (e.g., reliable or best effort) and, (vi) dynamically changing federations where federates can join and resign from a federation at any time.

Motivated by the above requirements list, we divided the range of federate types into four major categories based on two key properties that are used in the *Aggregate Level Simulation Protocol (ALSP)* (Weatherly et. al. 1991): (i) *time constrained* and (ii) *time regulating* (see Figure 1). Federates that are *time constrained* , must receive events in time stamp order. Non-time constrained federates receives events in the order that they were received by the RTI. Federates that are *time regulating* , schedule events that must be received in time stamp order. *Non-time regulating* federates schedule events that will be delivered in the order received by the destination. Note that the order of event delivery need not be the order in which events are processed. Federates can reorder

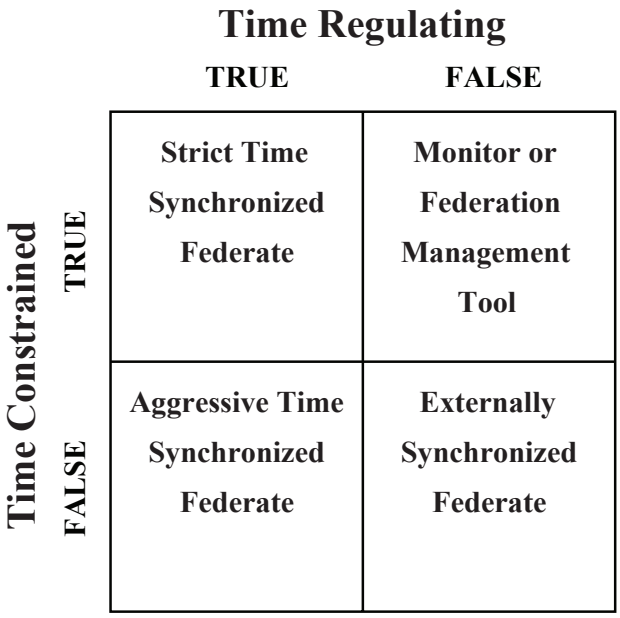


Figure 1: Simulator Types Based on the *Time Constrained* and *Time Regulating* Properties

events once delivered.

The first major federate category is *strict time synchronized*. To be in this category, a federate must be *time constrained* and *time regulating*. Example federates include simulations synchronized by conservative (e.g. ALSP) and/or optimistic (e.g. Time Warp (Jefferson 1985)) protocols. Optimistic federates are included here because despite allowing out-of-time stamp order execution, the order of committed events (i.e., events not rolled back) strictly adheres to the time stamp order processing rule.

The next major federate category is *aggressive time synchronized*. Here, federates are not *time constrained*, but still schedule time stamped events. A federate falling into this category would be a DIS federate (DIS Steering Committee, 1994) that would like to interoperate with other ALSP federates. In this example, the DIS federate is able to process events in "real-time" and not have its time advance constrained by the progress of other federates in the federation. The caveat is that the DIS federate could receive events in its past and process events out of order, which is allowed by the DIS protocol. Because, ALSP requires strict time stamp order processing of events, the DIS events sent to ALSP federates must be time stamped. By having a *time regulating*, DIS federate, events will be properly time stamped.

The third major federate category is *monitor*. This kind of federate is *time constrained*, but only schedules receive order events to be sent to other federates. The reasoning for the viewer category is that federations will need management tools that display time stamped information, but would schedule "out-of-time" receive order events, such as forcing a federate to resign from the federation or requesting certain status information, such as the length of the RTI's event queues.

The last major federate category is *externally synchronized*. These federates are synchronized by some mechanism outside of the RTI. Consequently, the are not constrained by the RTI's time management services and schedule only receive order events. Here, the RTI serves as a message passing library and time management services, while invoked, do not manage the federate's advancement of simulated time. This category of federates would be used in a DIS federation, such as Synthetic Theater of War (STOW) (Aronson 1996), which uses a real-time clock that is external to the RTI to control the advance of simulated time. Because this federation contains non-time constrained federates, there is no need to schedule time stamp events, thus all federates have their *time regulating* property turned off.

Additionally, the RTI must not only allow federates of different types to interoperate but also allow federates to change their type by toggling the *time constrained* and *time regulating* properties during federation execution.

3- TIME MANAGEMENT SERVICES

In this section, we review the HLA Time Management Service set forth in the IFSpec. These services encompass two aspects of federation execution: (i) *Transportation services* and (ii) *Time advance services*. Each of these is discussed in the sections below.

3.1- Transportation Services

The categories of transportation service are distinguished by (i) reliability of event delivery, and (ii) event ordering. With respect to reliability, version F.0 of the RTI offers two levels of service. **Reliable delivery** guarantees that any message sent using this service will arrive intact at the proper destination. This level of reliability comes at the cost of increased latency. Example federates using this service would be ALSP. **Best effort delivery** reduces latency, but does not guarantee delivery. Example federates using this service would be DIS.

Event ordering characteristics specify the order and time at which events may be delivered to federates and are central to the HLA time management services. In accordance with the 1.0 IFSpec, F.0 offers two ordering mechanisms. **Receive order** events are passed to the federate in the order that they were received. Logically, incoming events are placed at the end of the a first-in-first-out (FIFO) queue, and are passed to the the federate by removing them from

the front of this FIFO queue. **Time stamp order (TSO)** events utilizing this service will be delivered to *time constrained* federates in time stamp order. Further, the time advance services guarantee that no event is delivered to a federate "in its past", i.e., no TSO event is delivered that contains a time stamp less than the federate's current logical time. A conservative synchronization protocol is used to implement this service. TSO events sent to *non-time constrained* federates will be delivered as receive order events.

Earlier versions of the HLA Time Management Services called for the following additional ordering services: (i) **priority order**, (ii) **causal order** and (iii) **causal and totally order**. After several iterations through the IFSpec approval process, these services were excluded from the HLA Baseline Definition (IFSpec 1.0). Because the F.0 design team's charter was to build an RTI that strictly adheres to this baseline definition, these additional services were not to be included in F.0.

3.2 Time Advance Services

The time advance services serve several purposes. First, it provides a protocol for the federate and RTI to jointly control the advancement of logical time. The RTI can only advance the *time constrained* federate's logical time to T when it can guarantee that all TSO events with time stamp less than or equal to T have been delivered to the federate. At the same time, conservative federates must delay processing any local event until their logical time has advanced to the time of that event, while optimistic federates will aggressively process events and rollback when it receives a TSO event in its past and use T as an estimate of global virtual time (GVT) for fossil collection.

To insure the RTI properly delivers TSO events, a conservative synchronization protocol implements the TSO event delivery service and is used to advance logical time. The principal task of the protocol is to determine a value called *Lower Bound Time Stamp (LBTS)* for each federate, which is defined as a lower bound on the time stamp of future TSO events that it will receive from other federates.

Among *time constrained* federates, there are three subclasses of federates: (i) *conservative event-driven*, (ii) *conservative time-stepped*, and (iii) *optimistic*. For each of these federate subclasses, the following three time advance service have been devised.

Time Advance Request with parameter t requests an advance of the federate's logical time to t. This service is intended to be used by conservative time-stepped federates where t denotes the time of the next time step. Invocation of this service by a *time constrained* federate implies that the following events are eligible for delivery to the federate: (i) all receive order events, and (ii) all TSO events with the same time stamp that are less than or equal to t. When the RTI can guarantee that it has passed to the federate all such events, the RTI invokes the **Time Advance Grant** service, notifying the federate that its logical time has been advanced to t. Upon receiving the grant, the federate may proceed to the next time step.

Next Event Request with parameter t requests an advance to logical time t, or the time stamp of the next TSO event from the RTI, whichever is smaller. By invoking this request, the federate guarantees it will not produce any new TSO messages in the future with a time stamp less than t plus the federate's lookahead if that federate does not receive any additional TSO messages. This service is intended to be used by conservative event-driven federates where t denotes the time of the next local event within the federate that is to be processed. After invocation of this service, the RTI will deliver all receive order events and either (i) deliver the next TSO event (and all other time stamp order events with exactly the same time stamp) if that event's time stamp is less than or equal to t and advance logical time to the time of that TSO event, or (ii) not deliver any TSO events and advance logical time to t. In either case, the RTI calls the **Time Advance Grant** service, notifying the federate of the completion of this request and that logical time has been advanced.

Flush Queue Request with parameter t requests an advance to logical time t, or the time stamp of the next TSO event, or LBTS, whichever is smaller and deliver all receive order and TSO events currently residing within the RTI. This service is intended to be used by optimistic federates, where t denotes the time of the next local event within the federate that is to be processed. After invoking this service, the RTI will deliver all TSO events regardless of their time stamp, should any exist, then advance logical time to the minimum of t, LBTS, and the smallest time stamp of any delivered TSO event and then call the **Time Advance Grant** service, notifying the federate that logical time has been advanced.

To support optimistic federates, the RTI provides a **Retract** service that allows federates to "unschedule" previously sent events. When a federate schedules an event, the RTI provides the federate with a **Event Retraction Handle**. To retract an event, the federate invokes the **Retract** service with the appropriate **Event Retraction Handle** as the parameter. If the retracted event has already been passed to the federate, the retract service is forwarded to that federate.

For *non-time constrained* federates, the time advance services will deliver all receive order events (recall that for *non-time constrained* federates all events are treated as receive order) and immediately advance the federate's logical time to the requested time via

the **Time Advance Grant** service.

A lookahead value is specified for each federate. Lookahead defines the minimum distance into the future that a TSO message will be scheduled. A federate may change its lookahead during the federation execution, however, a decrease in the federate's lookahead by an amount k does not take effect until the federate has advanced k units of time.

An important assumption in using these time advance services is that federates must have non-zero lookahead. Because each service will not advance time until the RTI can unconditionally guarantee that *all* events less than or equal to some time t have been delivered, federates with a zero lookahead may cause the federation to deadlock (Fujimoto 1996). Changes to allow zero lookahead federates have been developed and should appear in RTI version 1.0.

4 IMPLEMENTATION

In designing the algorithms necessary to support all the requirements placed on the RTI by an unpredictable federate, every effort was made to avoid RTI modality. The RTI must be able to honor service calls in any order. This leads naturally to a finite state machine design philosophy for the time manager. The completion of every service call will leave the time manager in a position to accept any subsequent service call. The context of service calls, if any, is captured in the state variables of the time manager. As part of this design philosophy, the time management algorithms were constructed to ensure reliability and to facilitate implementation. Consequently, these algorithms avoid the use of potentially unreliable distributed programming constructs, such as global synchronizations.

While the IFSpec is quite detailed in the regard to the data flow into and out of each HLA Service, the design team made two assumptions to ease the development effort required to implement F.0.

(i) All federates are fully connected by point-to-point, reliable, FIFO communication links. The *Adaptive Communication Environment (ACE)* (Schmidt 1993) was chosen to support the communications layer of F.0 because it supported reliable communications between programs started by different users without having to resort to using "remote-shells", which has been shown to have security problems. Because ACE uses the TCP/IP protocol which is reliable, FIFO, point-to-point, it was decided all algorithms should take advantage of this communications property whenever possible.

(ii) All TSO messages are sent reliable and FIFO. The question of designing a synchronization algorithm that allows best-effort, TSO messages remains unresolved. Due to the F.0 tight delivery schedule, the design team did not have time to adequately investigate this issue. Accordingly, it was decided that all TSO messages must sent over a reliable communication link. The FIFO property assumption is added because the F.0's communications layer supported it, allowing for a straight-forward synchronization algorithm to be used. It should be noted that federates can send best-effort, TSO message, however, the synchronization algorithm currently used will not guarantee that messages will be delivered in TSO order.

In this section, we provide a detailed description of F.0's time management implementation. The first major design constraint concerned the issue of concurrency or "threadedness" within the RTI. This issue and its implications for F.0 are discussed. Next, we present an overview of F.0's object hierarchy and then devote the remainder of this section to the implementation of F.0's time management algorithms.

4.1 RTI Threading

The predecessors to the F.0 RTI are the 0.1 thru 0.33 versions. In this sequence of prototype RTIs, the core components of the RTI were divided into independent threads of computation taking the form of CORBA servers. This partitioning of the code was along the lines of the RTI services provided (i.e. the Time Manager thread serviced all time advance service requests). CORBA was used to implement prototype RTIs and CORBA IDL was used as the original application programmer interface language. The AMG selected CORBA IDL because of its operating system and programming language independence. The RTI implementation team choose CORBA for the support it provided in the rapid development of distributed systems. The CORBA-based RTI allowed experience to be gained with infrastructure software that was separate from the federate. This was the implementation approach already being taken by many large analytic systems such as JWARS (see `http://afmsrr.sc.ist.ucf.edu/resource/jwars.html`).

When the $0.X$ version series was delivered, it became increasingly clear that the performance demands of other systems in the HLA community would not be met, such as ModSAF (see `http://www.ait.nrl.navy.mil/modsaf`). The reason for this lies with the number of costly interprocess communications (IPC) hops taken in the RTI to deliver a single message to the federate.

To address the concerns of performance-demanding federates, the AMG decided that F.0 should be a singly-threaded library that links directly into the federate's program. By having a singly-threaded RTI, the possibility of race conditions and deadlock is reduced. Moreover, time consuming IPCs and context switches are eliminated. However, the consequence of this decision is that it complicates federate imple-

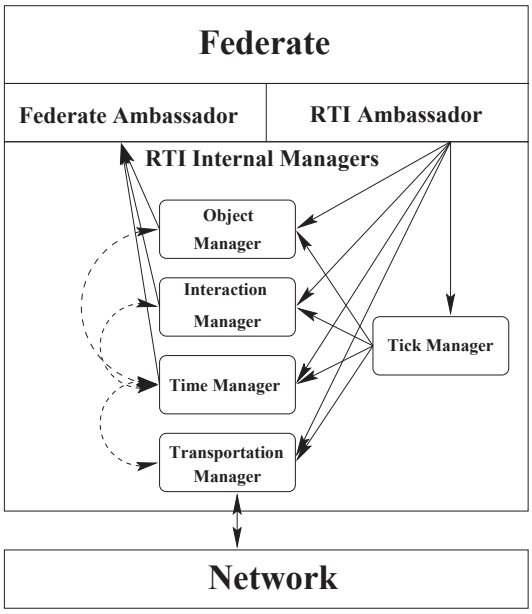


Figure 2: Version F.0's Object Hierarchy

mentations. In particular, the federate must explicitly yield control to the RTI so it can process service requests and deliver data to the federate. How this issue is addressed in F.0 is discussed in the next section below.

4.2 Object Hierarchy

The RTI, shown in Figure 2, is composed of three main parts: (i) *RTI Ambassador*, (ii) *Federate Ambassador*, and (iii) *Internal RTI Managers*. The solid lines represent service invocations either from the federate into the RTI or vice versa. The dashed represent intra-RTI service invocations between the different RTI Internal Managers. The first two parts of the RTI are used to pass information from the RTI to the federate and vice versa. The RTI Ambassador serves as an interface for marshalling service requests made by the federate to the appropriate RTI Internal Manager. The Federate Ambassador serves as an interface for marshalling service request responses, such as events or **Time Advance Grants**, from the RTI to the federate. Each federate must provide an implementation of the Federate Ambassador's services. The interface to both the RTI Ambassador and the Federate Ambassador are subject to HLA standards.

The Internal RTI Managers support five categories of run-time services. The *Object Manager* implements the object creation, object destruction, ownership, object publication and object subscription services. The *Interaction Manager* implements the services that create and destroy interactions as well as interaction publication and subscription services. The *Time Manager* implements the services for advancing logical time and will be discussed in detail. The *Transportation Manager* is used to send and receive data and supports the services provided by the other managers. The federation management services are implemented in the RTI Ambassador.

In keeping with the above design constraint, the RTI is designed to be linked in as part of the federate to form a single simulation with only one thread of control. To reduce the effort involved in integrating federates with a single threaded RTI, F.0 provides a **tick** service in the *RTI Ambassador* that gives the thread of control to the RTI and that ensures that all the necessary internal RTI functions and service requests are completed.

When a federate requests the **tick** service, that request is sent to the *Tick Manager*. The *Tick Manager* then invokes the **tick** service provided by each of the RTI Internal Managers to perform their necessary functions. For example, suppose a federate issues a **Time Advance Request** for some time, t. For the RTI to process this request, the **tick** service must be invoked. Upon doing so, the *Tick Manager* would invoke the *Transportation Manager's* **tick** service. Having the single thread of control, the *Transportation Manager* would deliver pending events from the network up to the *Time Manager*, where they are enqueued into the appropriate queue (either TSO or receive order). When the *Transportation Manager's* **tick** service completes, control is returned back to the *Tick Manager* who immediately gives control to the *Time Manager* by invoking its **tick** service. The *Time Manager*, seeing there is a pending **Time Advance Request**, examines the receive order and TSO queues and delivers the appropriate events to either the *Object Manager* or *Interaction Manager*, who then directly forwards the event, if necessary, to the *Federate Ambassador*. Note, the *Tick Manager* was bypassed in delivering events from the *Time Manager* to the *Federate Ambassador*. This was done to expedite the event delivery process. After the *Time Manager's* **tick** service completes, the *Tick Manager* gains control and similarly invokes the **tick** service on the *Object* and *Interaction Managers*. Once, the *Tick Manager* has "ticked" all the managers, control is returned to the federate. In practice, the federate will need to invoke the **tick** service more than once (i.e. "poll" the RTI) before the **Time Advance Grant** will be issued.

4.3 Time Management State Transitions

Internal to RTI Time Management there are six states used to determine what actions need to be completed and ensure that these actions occur in the correct order. These six states and their respective transitions are described below:

Initialized: Prior to joining a federation but after the RTI has been initialized, the *Time Manager* is placed in this state.

Joining: When the federate invokes the **Joining Federation** service, the *Time Manager* moves from the *Initialized* state to this state.

Idle: Once the federate joining process is completed, the *Time Manager* transitions to this state.

Time Pending: When the federate invokes the **Time Advance Request** service, the *Time Manager* moves from the *Idle* state to this state. Next, the *Time Manager's* **tick** service (called via the *Tick Manager*) will in turn invoke the **Do Time Pending** decision service (discussed below) based on being in this state and deliver the appropriate receive order and TSO events. Having completed the service request, the **Do Time Pending** service will issue the **Time Advance Grant** and transition back to the *Idle* state.

Event Pending: When the federate invokes the **Next Event Request** service, the *Time Manager* moves from the *Idle* state to this state. Next, the *Time Manager's* **tick** service will in turn invoke the **Do Event Pending** service (see below) based on being in this state and deliver the appropriate receive order and TSO events. Having completed the service request, the **Do Event Pending** service will issue the **Time Advance Grant** and transition back to the *Idle* state.

Flush Pending: When the federate invokes the **Flush Queue Request** service, the *Time Manager* moves from the *Idle* state to this state. Next, the *Time Manager's* **tick** service will in turn invoke the **Do Flush Queue Pending** decision service (see below) based on being in this state and deliver the appropriate receive order and TSO events. Having completed the service request, the **Do Flush Queue Pending** service will issue the **Time Advance Grant** and transition back to the *Idle* state.

4.4 TSO Event Queue Case Groupings

During a time advance request-grant cycle, the RTI must determine which TSO messages are eligible for delivery to the federate. In making this determination, the RTI must consider the relationship between (i) the time stamp of the event at the head of the TSO queue, (ii) the requested time to which the federate wishes to advance and (iii) LBTS. Shown in Figure 3 are the six relationship cases between these three factors, and how a decision is reached as to whether the RTI should (a) grant, (b) deliver a TSO event, or (c) do nothing. **Head** denotes the time stamp of the event at the head of the TSO queue. **LBTS** denotes the lower bound time stamp on any event that could be sent to this federate in the future. **t** denotes the time to which the federated has requested to advance.

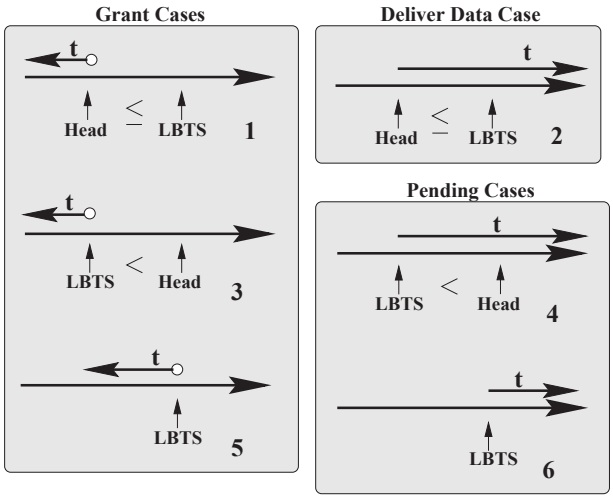


Figure 3: Time Stamp Order Event Queue Case Groupings

The white dot is used to denote a strictly less than relationship.

We argue that these six cases provide a complete coverage of all possible values for these three factors. To make this argument, consider the following case groups: (i) cases 1 and 2, (ii) cases 3, and 4, and (iii) cases 5, and 6. We observe that in each of these groups, the value of t, which denotes the request time, spans the entire time line. Thus, each of these groups considers all possible values of t. Next, we observe that groups (i) and (ii) together cover all possible relationships between the the head of the TSO queue, and LBTS. Last, we observe that group (iii) covers all the cases when the TSO queue is empty and no head exists. Consequently, these six cases provide complete coverage.

Now, the case grouping names shown in Figure 3 denote the action generally taking by the *Time Manager* when in that particular case. Consider the **Grant Cases**(cases 1, 3, and 5): we observe that the request time is always smaller than either the head of the TSO queue or LBTS. Because of this, we know it is safe for the *Time Manager* to issue a **Time Advance Grant** to the request time, t. Likewise, in the **Deliver Data Case** (case 2), since the head of the TSO queue is the smallest of the three factors, we can always deliver the event that is at the head of the TSO queue. Last, in the **Pending Cases** (cases 4 and 6), the RTI can neither grant nor deliver data since both the request time and head of the TSO queue are greater than or equal to LBTS. Note, there are some exceptions to the actions taken by the *Time Manager*, depending on the time advance service.

In the next section, we present the algorithm for servicing a **Time Advance Request**. Due to space limitations, we are unable to present the algorithms

```
Time Advance Request( FederationTime t)
    case state of
        IDLE
            if t < LT then
                throw FederationTimeAlreadyPassed()
            end if
            currentLookahead := max( specifiedLookahead,
                LT + currentLookahead − t)
            requestTime := t
            LT := t
            if timeRegulating then
                send newLT((LT + currentLookahead),
                    requestTime)
            end if
            state := TIME_PENDING
            more := TRUE
            fifoRemainingToDequeue := size of FIFO queue

        EVENT_PENDING
        TIME_PENDING
        FLUSH_PENDING
            error TimeAdvanceAlreadyInProgress
    end case
```

Figure 4: **Time Advance Request** Service

used to service **Next Event Request** and **Flush Queue Request**.

4.5 Time Advance Request

When the **Time Advance Request** service is invoked, shown in Figure 4, the *state* of the *Time Manager* is first checked. If the *state* is anything but *Idle* an exception is thrown indicating that a time advance service request is already in progress. Next, because the federate can shrink its lookahead during federation execution, the *Time Manager* must do it in a way that no causality errors result. This gradual shrinking of the lookahead (if necessary) is accomplished by the following expression: $currentLookahead = max(specifiedLookahead, LT + currentLookahead - t)$, where *currentLookahead* is the actual lookahead used by the *Time Manager*, *specifiedLookahead* is the smaller lookahead value that has been recently reset by the federate and t is the request time. This expression guarantees that the federate will advance k units of time before shrinking the lookahead by k units.

Then, the request time, t, is stored, and the federate's logical time, LT, is updated. This update of LT may seem inappropriate at this point given that other messages may exist in the TSO queue with a time stamp less than the request time. However, the semantics of this service as stated in the IFSpec stipulate that the federate will be granted to the request time. Moreover, the federate in making this request is stating that any future messages sent to other federates will have a time stamp greater than or equal to $t + currentLookahead$. Consequently, from the IFSpec's point-of-view this behavior is correct, however federates may have events delivered with a time stamp less than LT. Accordingly, we believe the AMG should consider amending the IFSpec to mitigate this ambiguity.

Next, a *newLT* message that contains $LT + currentLookahead$ and the request time is broadcast to all other federates along the reliable, FIFO, point-to-point communication links. This *newLT* message can be viewed as a null message from Chandy/Misra's null message algorithm (Chandy et. al. 1979). When another time constrained federate's *Time Manager* receives the *newLT* message, the logical time plus lookahead value is stored and used to compute LBTS. LBTS for a federate is defined as minimum of all other federates logical clock plus lookahead values and is computed every time a new "LT" value arrives. The *newLT* message's request time is stored separately and used to break situations where the next event in the federation is far into the future and the federation is creeping ahead by exchanging *newLT* messages because the lookahead is small. This creeping effect only occurs in the **Next Event Request** and **Flush Queue Request** services.

Having sent the *newLT* message, the *state* of the *Time Manager* is set to *Time Pending*. Now, to tell the *Tick Manager* that the *Time Manager* has more work to complete, *more* is set to TRUE. Last, the size of the receive order event queue is stored prior to returning control to the federate. The reason for storing the receive order event queue size will be discussed below.

When the federates yields control to the RTI via the **tick** service, the *Tick Manager* will invoke the *Time Manager's* **tick** service. In this service, the delivery of all receive order events occurs before any TSO events are considered for delivery. Delivering all the receive order events will take several invocations of this service since only one event is delivered per invocation. Also, as a means of flow-control, this service will only deliver the number of receive order events that is equal to the receive order queue size at the time the request was made. This was done to ensure that the amount of time spent processing this service is consistent between successive invocations and to prevent continuous FIFO message arrival from precluding TSO message delivery.

Next, after the receive order events have been delivered, TSO events may be delivered provided the federate is *time constrained*. Because the *state* of the *Time Manager* is *Time Pending*, the **Do Time Pending** decision service (see Figure 5) is invoked, which determines which, if any, TSO events are to be delivered to the federate and then the **Time Advance Grant** is issued. By comparing the request time, the time stamp of the head event of the TSO event queue and LBTS, the proper case grouping can be determined. If the case is *Deliver Data*, this service will deliver a single TSO event and set *more*

```
Do Time Pending()
    case findCases() of
        DELIVER_DATA:
            dequeue one TSO message
            more := TRUE

        GRANT:
            grant LT
            state := IDLE
            more := FALSE

        PENDING:
            fifoRemainingToDequeue := size of FIFO queue
            if fifoRemainingToDequeue = 0 then
                more := FALSE
            else
                more := TRUE
            end if
    end case
```

Figure 5: **Do Time Pending** Decision Service

equal to TRUE. If the case is *Grant*, then a **Time Advance Grant** is issued for the logical time of the federate, LT, the *state* is set to *Idle* and *more* is set to FALSE, indicating this service request is complete and the *Time Manager* has no pending work to perform. If the case is *Pending*, no actions can be taken and the number of receive order events is recalculated to be used in delivering of receive order events in future invocations of the *Time Manager's* **tick** service. If there are receive order events to deliver, *more* is set to TRUE. Otherwise it is set to FALSE.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we present the design and algorithms used to implement the HLA Time Management Services in version F.0 of the RTI. F.0 will be followed by RTI 1.0 in May 1997 and RTI 1.1 in the Fall of 1997. These future releases will address candidate functionality for inclusion in the HLA and services not provided in F.0. In future RTI versions, we hope to address the following such as, (i) zero lookahead federates, (ii) repeatability of results in the presence of zero lookahead federates, (iii) casually correct toggling of *time regulating* properties, and (iv) development of an efficient synchronization protocol that allows best-effort, TSO messages to be sent.

REFERENCES

Aronson, J. 1996. The STOW-97 system architecture and implementation design. *Technical Report CDRL:A005, STOW Web Page at* `http://www.stow.com`.

Chandy, K. M., and J. Misra 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering* SE-5(5):440-452.

DIS Steering Committee 1994. The DIS vision, a map to the future of distributed simulation. *Technical Report IST-SP-94-01, Institute for Simulation and Training.*

Fujimoto, R. 1996. HLA time management. *Technical Report, DMSO HLA Web Page at* `http://msis.dmso.mil/projects/hla/`.

Jefferson, D. R. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems* 7(3):404-425.

Weatherly, R. D. Sidel, and J. Weissman 1991. Aggregate level simulation protocol. In *Proceedings of the 1991 Summer Computer Simulation Conference*, 953-958.

Schmidt, D. 1993. The adaptive communication environment (ACE). *Technical Report, at* `http://www.cs.wustl.edu/~schmidt/ACE-papers.html`.

AUTHOR BIOGRAPHIES

CHRISTOPHER CAROTHERS is a Research Scientist and Ph.D. candidate in the College of Computing at the Georgia Institute of Technology. He was part of the F.0 development team as a summer intern at the MITRE corporation.

RICHARD FUJIMOTO is a professor in the College of Computing at the Georgia Institute of Technology. In addition to his many other research activities in the field of parallel simulation, he chairs the HLA Time Management Committee.

RICHARD WEATHERLY is a is Chief Engineer for The MITRE Corporation's Information Systems and Technology Division. He wrote the first version of the HLA Interface Specification and lead the RTI 0.X, F.0, and 1.0 software development teams.

ANNETTE WILSON is a Senior Systems Engineer at the MITRE Corporation. She served as the Technical Leader for RTI 0.X and F.0 Time Management and Data Distribution components.

THE DEPARTMENT OF DEFENSE HIGH LEVEL ARCHITECTURE

Judith S. Dahmann
Defense Modeling and Simulation Office
1901 North Beauregard Street
Alexandria, VA 22311, U.S.A.

Richard M. Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280, U.S.A.

Richard M. Weatherly
The MITRE Corporation
7525 Colshire Drive
McLean, VA 22102-3481, U.S.A.

ABSTRACT

The High Level Architecture (HLA) provides the specification of a common technical architecture for use across all classes of simulations in the US Department of Defense. It provides the structural basis for simulation interoperability. The baseline definition of the HLA includes (1) the HLA Rules, (2) the HLA Interface Specification, and (3) the HLA Object Model Template. This paper describes the motivations and processes used to develop the High Level Architecture and provides a technical description of key elements of the architecture and supporting software. Services defined in the interface specification for providing time management (TM) and data distribution management (DDM) for distributed simulations are described.

1 INTRODUCTION

The Defense Modeling and Simulation Office (DMSO), is addressing the continuing need for interoperability among new and existing simulations within the U.S. Department of Defense through its High Level Architecture (HLA) initiative. The HLA seeks to generalize and build upon the results from the Distributed Interactive Simulation (DIS) world and related efforts such as the Aggregate Level Simulation Protocol (ALSP)(Wilson and Weatherly 1994). A goal of the HLA activity was to develop and recommend an architecture to the Executive Council for Modeling and Simulation (EXCIMS) before the end of calendar year 1996. This was achieved, and the EXCIMS in turn, after appropriate review, recommended the architecture to the Under Secretary of Defense (Acquisition and Technology) for approval and standardization. The baseline HLA definition was approved as the standard technical architecture for all U.S. DoD simulations on 10 September 1996. This tutorial provides an overview of the HLA, updating and expanding upon (Dahmann 1997). Additional information concerning the HLA concept and the DMSO Master Plan is available at http://www.dmso.mil.

The High Level Architecture (HLA) provides a common architecture supporting reuse and interoperation of simulations across the U.S. Department of Defense. The HLA is based on the premise that no one simulation can satisfy all uses and users. An individual simulation or set of simulations developed for one purpose can be applied to another application under the HLA concept of the federation: a composable set of interacting simulations. The intent of the HLA is to provide a structure which will support reuse of capabilities available in different simulations, ultimately reducing the cost and time required to create a synthetic environment for a new purpose.

The HLA is intended to have wide applicability, across the full range of defense simulation applications including training, analysis, and engineering functions, at a variety of levels of resolution. These widely differing application areas include a variety of requirements which had to be considered in the development and evolution of the HLA.

The definition of architecture used in this effort -- "major functional elements, interfaces, and design rules, pertaining as feasible to all simulation applications, and providing a common framework within which specific system architectures can be defined" -- is one that is commonly accepted and is consistent with the IEEE definition of architecture for computer simulations. For the purpose of this effort, the emphasis is on the development of a high level architecture which pertains as widely as possible to all simulation areas and which will provide a framework for the development of specific system architectures. The HLA does not prescribe a specific implementation, nor does it mandate the use of any particular set of software or programming language. Over time, as technological advances become available, new and different implementations will be possible within the framework of the HLA.

2 THE HLA DEVELOPMENT PROCESS

The HLA was developed by the US Department of Defense (DoD) based on a process involving government, academia, and industry. In FY94, the Defense Advanced Research Projects Agency (DARPA) awarded three industrial contracts for the definition of a high level architecture. The results of these three contractor efforts were received in January 1995, and a core government and academic team combined the best of those inputs with additional insights from other DoD modeling and simulation (M&S) projects to arrive at the initial definition of the HLA. This definition was presented to the Defense Modeling and Simulation Office (DMSO)-sponsored Architecture Management Group (AMG) on 31 March 1995.

The AMG developed the architecture based on cooperative efforts to apply the HLA in a series of prototypes designed to ensure that it addressed the broad set of application requirements for DoD simulations. The result was the baseline HLA definition, completed in August 1996 and approved as the standard technical architecture for all US DoD simulations on 10 September 1996. The AMG continues to evolve the HLA as needed.

The AMG is made up of representatives of major U.S. DoD simulation programs. Programs were selected because they represent the principal simulation requirements, ranging from analysis to training to test and evaluation, from detailed engineering representation to campaign-level warfighting to man-in-the-loop interactive vehicle simulators. Members, along with their supporting government and industrial teams, participate in regular meetings every 6 to 8 weeks to address the key HLA issues, conduct cooperative prototyping efforts with other AMG members (prototype federations and experiments) and participate in technical working groups to evolve crosscutting technical aspects of the HLA.

It is also very important that the HLA be integrated into the broader technical community. Work is underway in a partnership with the Simulation Interoperability Workshop (SIW) and the Simulation Interoperability Standards Organization (SISO). The DIS Workshop (the precursor to the SISO and SIW) was represented on the AMG during the HLA baseline development period and has committed to establish industry standards to support the HLA. This effort was initiated at the 14th DIS Workshop in March 1996.

3 TECHNICAL ARCHITECTURE

3.1 Motivations

The HLA design is based on certain assumptions. First, the HLA is based on the premise that no one simulation can solve all the U.S. DoD functional needs for modeling and simulation. The needs of the users are too diverse. The technical complexity of needed implementations is beyond what has been shown to be possible today or is reasonably likely in the foreseeable future to be handled in a single simulation. Further, with changing user needs, it is just not possible to anticipate how simulations will be used in the future or in what combinations. Perhaps equally as important is the changing state of technology; the HLA will need to allow for leveraging new technical innovations.

Since it is not possible to anticipate all the uses of M&S in the U.S. DoD, it is important to think in terms of multiple simulations which can be reused in a variety of ways. This means that as simulations are developed, they must be constructed so that they can be easily brought together with other simulations, to support new and different applications. While the HLA attempts to impose as few constraints as possible on the design of individual simulations, given the drive for reuse, the HLA recognizes that simulations themselves will need to be constructed so that they can easily be extended to support new features needed by unanticipated applications.

These assumptions have affected the HLA design in several ways. Clearly, the architecture itself must have modular components with well-defined functionality and interfaces. Further, the HLA has separated the functionality needed for individual simulations from the infrastructure required for interoperability among simulations.

3.2 Functional Overview

Figure 1 shows how an HLA federation is partitioned into its major functional components. The first key component are the simulations themselves, or more generally, the federates. A federate can be a computer simulation, a manned simulator, a supporting utility (such as a viewer or data collector), or even an interface to a live player or instrumented range. All object representation is in the federates. The HLA imposes no constraints on what is represented in the federates or how it is represented, but it does require that all federates incorporate specified capabilities to allow the objects in the simulation to interact with objects in other

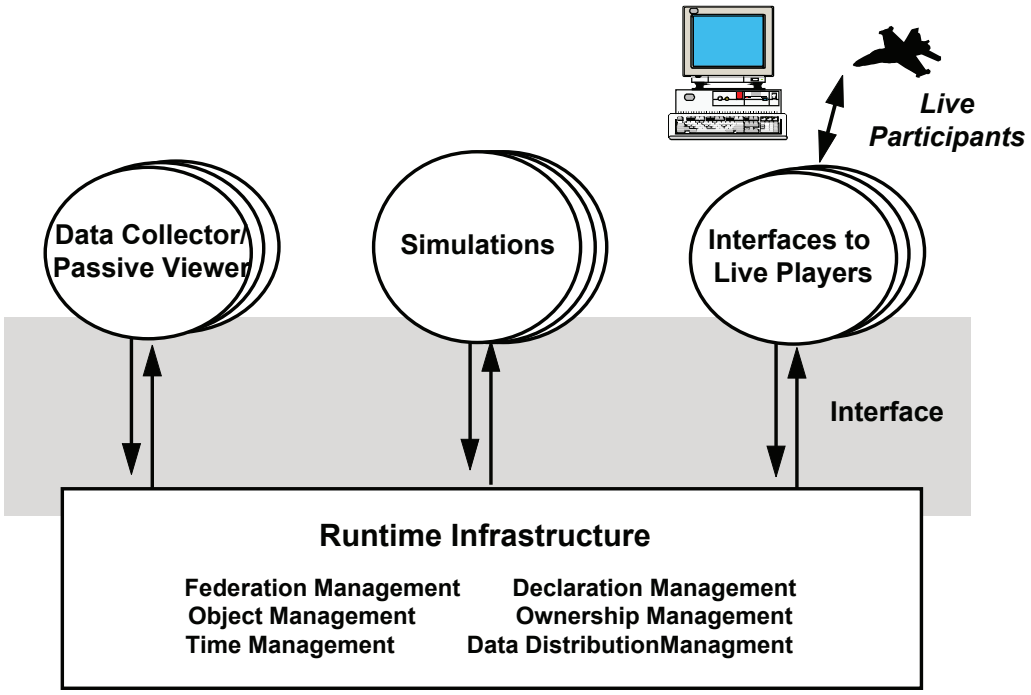


Figure 1: Functional View of an HLA Federation

simulations through the exchange of data supported by services in the Runtime Infrastructure (RTI).

The second functional component is the RTI. The RTI is, in effect, a distributed operating system for the federation. The RTI provides a set of services that support the simulations in carrying out these federate-to-federate interactions and federation management support functions. These services will be discussed later. All interactions among the federates flow through the RTI.

The third functional component is the runtime interface. The HLA runtime interface specification provides a standard way for federates to interact with the RTI, to invoke the RTI services to support runtime interactions among federates, and to respond to requests from the RTI. This interface is implementation independent and is independent of the specific object models and data exchange requirements of any federation.

Two other general capabilities of simulation systems are supported by the architecture. First, the HLA supports the passive collection of simulation data and monitoring of simulation activities. In the HLA, these tools act in the same way as simulations and interact with the RTI using the HLA interface.

Second, the HLA supports interfaces to live participants, such as instrumented platforms or live command and control (C2) systems. Live participants interact with the simulated world through something that acts like a simulation from the point of view of the HLA, that feeds a representation of the live world into the simulated world and that projects data from the simulated world back to the live system.

The HLA is formally defined by three components: the object model template(Defense Modeling and Simulation Office 1996), the interface specification(Defense Modeling and Simulation Office 1996), and the HLA rules(Defense Modeling and Simulation Office 1996).

3.3 HLA Object Models

HLA object models are descriptions of the essential sharable elements of the simulation or federation in 'object' terms. The HLA is directed towards interoperability; hence in the HLA, object models are intended to focus on description of the critical aspects of simulations and federations which are shared across a federation. The HLA puts no constraints on the content of the object models. The HLA does require that each federate and federation document its object model using a standard object model template. These templates are intended to be the means for open information sharing across the community to facilitate reuse of simulations. These completed templates will be openly available, and

tools are being developed to allow for automated search and reasoning about object model template data, to further facilitate cost-effective information exchange and reuse.

The HLA specifies two types of object models: the HLA Federation Object Model (FOM) and the HLA Simulation Object Model (SOM). The HLA FOM describes the set of objects, attributes and interactions which are shared across a federation. The HLA SOM describes the simulation (federate) in terms of the types of objects, attributes and interactions it can offer to future federations. The SOM is distinct from internal design information; rather it provides information on the capabilities of a simulation to exchange information as part of a federation. The SOM is essentially a contract by the simulation defining the types of information it can make available in future federations. The availability of the SOM facilitates the assessment of the appropriateness of the federate for participation in a federation.

While the HLA does not define the contents of a SOM or FOM, it does require that a common documentation approach be used. Both the HLA FOM and SOM are documented using a standard form called the HLA Object Model Template (OMT).

3.4 HLA Interface Specification

The HLA interface specification describes the runtime services provided to the federates by the RTI, and by the federates to the RTI. There are six classes of services. *Federation management* services offer basic functions required to create and operate a federation. *Declaration management* services provide a means for federates to declare what information the federate will provide and require during a federation execution. *Object management* services provide creation, deletion, identification and other services at the object level. State updates and interactions result in messages being transmitted to other federates that have indicated interest in receiving this information. *Ownership management* supports the dynamic transfer of ownership of object/attributes during an execution. This is necessary because at any instant, only one federate, termed the "owner," is allowed to update the value of a particular attribute. *Time management* services support synchronization of runtime simulation data exchange. Finally, *data distribution management* services support the efficient routing of data among federates during the course of a federation execution. The HLA interface specification defines the way these services are accessed, both functionally and in a programmer's interface. The time management and data distribution management services are described next in greater detail.

3.4.1 Time Management

Time management is concerned with the mechanisms for controlling the advancement of each federate in simulation time (referred to as the federation time axis in the HLA). In general, time advances must be coordinated with object management services so that information is delivered to federates in a timely and ordered fashion, e.g., to satisfy requirements for reproducing causal behavior in the system being modeled. Specifically, a federate may specify that the messages it sends should be delivered to federates in receive order or time stamp order (TSO). Advances in simulation time may or may not be paced to advances in wallclock time.

Federates will normally use one of the following approaches to time management:

- *paced, independent time advances.* Federate time advances are *paced* to occur in synchrony with wall clock time (or scaled wall clock time, derived as a linear function of wall clock time). The federate autonomously advances its own time without coordinating such advances with the RTI. "Human-in-the-loop" training federates and "hardware-in-the-loop" test and evaluation federates will typically utilize this approach.
- *paced, coordinated time advances.* Federate time advances are paced to occur in synchrony with (scaled) wall clock time, but time advances are coordinated with the RTI to ensure that before-and-after relationships in the physical system are properly reproduced. Analytic simulation models embedded in environments including humans and/or live elements or physical devices typically utilize this approach.
- *unpaced, coordinated time advances.* Federate time advances are not paced to occur in synchrony with wall clock time, and the federate coordinates time advances to ensure before-and-after relationships are properly modeled. Analytic simulation models intended to execute "as-fast-as-possible" typically utilize this approach.

It is anticipated that some federations may include combinations of federates using different approaches to time management (e.g., coordinated and independent time advance federates may be utilized in a single federation). The time management services are intended to support federations that include federates with different ordering and delivery requirements.

The time management services defined in the HLA are primarily concerned with coordinated time advance

federates. There are three key components of the time management services:

1. *Logical time:* Logical time is synonymous with federation time for coordinated time advance federates. Temporal relationships among events are specified by the logical time values (time stamps) assigned to the events. In general, at any instant during the execution, different coordinated time federates may be at different logical times.
2. *Mechanisms to advance logical time:* Services are provided for each federate to advance its own logical time. Federates must explicitly request logical time advances, and the advance does not occur until the RTI issues a grant. If advances in logical time must be paced to be in synchrony with wallclock time, such pacing must be done within the federate(s).
3. *Message ordering and synchronization:* The RTI will only grant an advance to logical time T when it can guarantee all time stamp ordered (TSO) messages with time stamp less than T (or in some cases less than or equal to T) have been delivered to the federate. This guarantee enables the federate to simulate the behavior of the entities it represents up to logical time T without concern for receiving new events with time stamp less than T.

Each federate using logical time must specify a non-negative *lookahead* value. If a federate's current logical time is T, all TSO messages generated by that federate must have a time stamp value of at least T plus the federate's lookahead. This constraint increases the amount of concurrent execution in the distributed simulation.

In order to determine which TSO messages can be delivered to a federate F without violating the guarantee that messages are delivered in time stamp order, the RTI must internally compute a lower bound on the time stamp (LBTS) of future messages that will be later received that are destined for F. Only TSO messages with time stamp less than or equal to F's LBTS value are eligible for delivery. At any instant, the RTI may hold one or more messages with time stamp less than LBTS in its internal queues. The minimum among the federate's LBTS value and messages stored in the RTI's queues gives a lower bound on the time stamp of any TSO messages that will be delivered to the federate in the future. LBTS is computed within the RTI using a conservative synchronization protocol, e.g., see (Fujimoto 1990).

The HLA provides several services to support the inclusion of optimistic federates, e.g., federates synchronized using the Time Warp mechanism (Jefferson 1985). The LBTS value computed by the RTI enables the federate to compute Global Virtual Time (GVT). A service for retracting previously scheduled events is provided, thereby enabling the implementation of anti-messages. A service is provided to allow optimistic federates to flush the time stamp ordered message queue, enabling the optimistic processing of events. An important property of the time management services is they ensure that optimistic events (events that may later be canceled) are not delivered to federates unless they explicitly request them via the flush queue primitive, thereby enabling conservative federates to operate in the same federation with optimistic simulations without being concerned with having to implement rollback and message cancellation mechanisms . At the same time, optimistic federates that provide this capability can freely exchange optimistic events.

A federate is said to be *time regulating* (or equivalently, time regulation is enabled) if it can send time stamp ordered (TSO) messages. Information such as the current logical time and lookahead of time regulating federates are used by the RTI in LBTS computations. TSO messages sent by a federate that is *not* time regulating are automatically converted to receive ordered by the RTI. Similarly, a federate is said to be *time constrained* (time constrained is enabled) if it can receive time stamp ordered (TSO) messages. TSO messages received by a federate that is not time constrained are automatically converted to receive ordered by the RTI.

3.4.2 Declaration Management and Data Distribution Management

Data management (DM) and data distribution management (DDM) in the HLA are used to specify which federates should receive messages for each state update and interaction (Morse 1996). DM services *Publish* and *Subscribe* allow a federate to update and receive updates to object attributes based solely on object class. The RTI uses information provided in publish/subscribe calls to set up filters that direct data among federates that need them. The object management's (OM) *Update Attribute Values* service call notifies the RTI that one or more attributes have been modified. For example, an object might subscribe to the attribute 'location' of all tanks in the battlefield. However, such filtering will only be appropriate for relatively small federations. DDM services provide more powerful data distribution services enabling value-based filtering(Defense Modeling and Simulation Office 1996). For example a tank might want to receive data from

other tanks only if they are in its visible range. Note that since the RTI does not know the meaning of object attributes it cannot provide range-based filtering in this example. Federates must agree on a filtering strategy to do this.

The fundamental concept used in the HLA to support value-based DDM is the *routing space.* A routing space is a normalized (coordinate values range from 0.0 to 1.0) multidimensional coordinate system in which federates indicate interest in receiving or providing updates via *subscription* and *update regions*. Subscription and update regions are rectangular (in N dimensions) and are specified by indicating *extents,* with one extent for each dimension. Each extent indicates the portion of that dimension covered by the region. For example, the extents [0.0,0.5][0.0,1.0] specify the left half of a two dimensional routing space. Federates express their interest in receiving updates from other federates through subscription regions defined over a routing space, and are called subscribers (to a specific attribute). On the other hand, federates that are providing updates define their update regions over the routing space, and are called publishers (of a specific attribute). The RTI detects when subscription and update regions associated with an attribute overlap, and provides data transfer from the publisher to all subscribers for all updates to that attribute.

Figure 2 shows a two-dimensional routing space with one update U1 and two subscription regions S2 and S3 that belong to federates F1, F2 and F3 respectively. Since U1 and S2 overlap, the RTI will transmit updates to attributes by F1 that are associated with this update region to subscriber F2, but not to F3. Regions can be changed dynamically by invoking the *Modify Region* service.

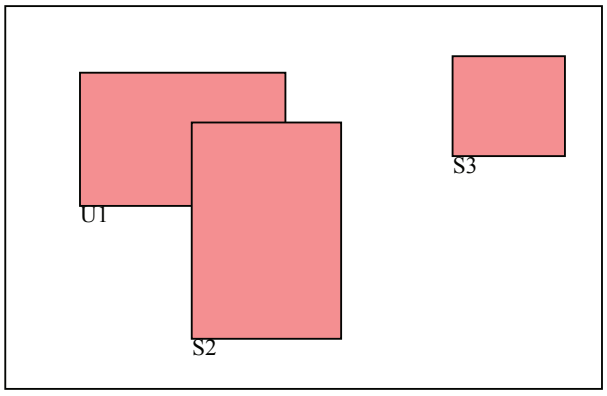


Figure 2: Update Region U1 and Subscription Regions S2 and S3 in a Two Dimensional Routing Space

3.5 HLA Rules

Finally, the HLA rules summarize the key principles behind the HLA. These are divided into two groups: federation and federate rules. Federations, or sets of interacting simulations or federates, are required to have a FOM in the OMT format. During runtime, all object representation takes place in the federates (not the RTI) with only one federate owning any given attribute of an instance of an object at any time. Information exchange among the federates takes place via the RTI using the HLA interface specification.

Additional rules apply to individual federates. Under the HLA, each federate must document their public information in their SOM using the OMT. Based on the information included in their SOM, federates must import and export information, transfer object attribute ownership, updates attributes and utilize the time management services of the RTI when managing local time.

4 HLA PROTOTYPING

The purpose of the HLA baseline development period was to take the initial HLA definition to develop the needed specifications and tools to actually use the HLA to support prototype applications. This prototyping was directed toward addressing a set of common issues(Defense Modeling and Simulation Office 1996) and to evolve the HLA to meet the broad set of needs. These issues include the technical viability of a single interface specification to support the wide application base of the HLA and the technical feasibility of building an RTI with the necessary range of tools needed by those applications, the impact of the HLA on simulation internal development, the utility of the object model concept and formats throughout the HLA life cycle, the ability to specify common testing methods, the ability to operate HLA federations in a secure mode and, finally, the applicability of the HLA to address the wide range of US DoD simulation applications.

There were four major prototype implementations of federations (“proto-federations”) using the HLA to support the HLA baseline development process. The simulations incorporated in these proto-federations were made available, along with their support technical development teams, by AMG members, as part of the HLA development process. Each proto-federation addressed a different corner of the simulation application domain, and as such, generally typifies one way simulations will be used in an HLA context.

Several overall points are warranted before the proto-federations themselves are described. The purpose of

these prototypes was to develop the HLA though hands-on experience, not to solve a specific real-world problem. In several cases the scenarios for the federations had to be stretched to support the mix of federates available from the AMG. Second, all four prototypes used the same architecture, the same interface specification, the same object model template and, in fact, the same RTI software implementation.

For HLA baseline development purposes it was decided to develop one prototype implementation of the RTI and use that implementation with all the proto-federations. This was seen as both the most cost-effective approach and a strategy that would allow for the examination of the desirability of having not only a common interface specification, but also one common implementation which could serve the entire community. The primary objective of the RTI prototyping was to provide the other proto-federations with the needed functionality for them to implement and use the HLA for their application needs. The RTI prototype also provided the mechanism to assess the functional and performance requirements for RTI service provision as input to future RTI development and transition. The RTI software is being re-engineered to incorporate the full set of services defined in the HLA Baseline Definition .

The first proto-federation was called the Platform Prototype federation. This was a federation of real-time platform level simulators and simulations which are current users of IEEE DIS 2.0. The key issues addressed by this proto-federation were the ability of the HLA to provide the capabilities now supported by DIS and to address the issues associated with adapting a current DIS-enabled system to work under HLA.

The second federation, called the Joint Training Prototype, was a federation of discrete event simulations. The key issues for this prototype were the ability to coordinate multiple discrete event simulations using the RTI time management services, the ability to hand-off ownership of attributes in a federation and options for providing dynamic environmental effects across a federation.

The third prototype federation was the Analysis Proto-federation. This proto-federation is comprised of closed form analytic applications; hence, runtime efficiency and replicability are important here, since these simulations will eventually require multiple executions for analysis purposes. The interest in this proto-federation was in the time management and data management services.

The Engineering Prototype federation examined use of the HLA to support simulations to support system acquisition using validated, engineering-level simulations. This proto-federation was particularly interested in object ownership management which affords the option of handing off computation of certain effects during select parts of a scenario to selected simulation facilities and in performance requirements to support hardware-in-the-loop applications.

These prototypes were implemented to apply the HLA to applications which typify its ultimate uses to support its definition. The results of the prototypes were fed into the specification and testing procedures through the working groups. The experiences of the individual federate developments support assessment of the effect of the HLA on simulations and offer an experience base for future users of the HLA.

Finally, the experience of developing these proto-federations provided the basis for the development of a general process view of the HLA. While HLA development focuses on the specification of the technical components of the architecture, it has been recognized that it is very important to understand the process which supports the use of the architecture for different applications. The baseline definition of the HLA was developed based on a set of proto-federations. Using the experience of these proto-federations as a driver, a general process for the development and execution of applications using the HLA is being formulated as a recommended practice to accompany the HLA definition(Defense Modeling and Simulation Office 1996). Understanding this process is viewed as a very important element in the development and application of the HLA.

5 HLA SUPPORT FOR SIMULATION REUSE AND INTEROPERABILITY

Simulation interoperability is defined as “the ability of a ... simulation to provide services to, and accept services from, other ... simulations, and to use the services so exchanged to enable them to operate effectively together.” (U.S. Department of Defense 1994) This definition of interoperability reflects the overall objective of the HLA that different simulations be able to effectively share data towards a common goal.

As the definition indicates, there are two elements in interoperation: effective data sharing and consistent data interpretation. The HLA requires that federates build the functionality required to interface with the RTI and exchange data with other federates via the HLA specified interfaces. This enables federates to participate in federations, to exchange data, and coordinate their operations with a federation. The HLA also requires all federates and federations to document characteristics of their object representations relevant to other potential users of the federate or federation. This documentation, in the form of completed object model templates,

facilitates the information exchange needed for a consistent interpretation of shared data.

Universal interoperability (the ability of any simulation to interoperate with any other simulation, regardless of original purpose or technical implementation) is not feasible with today's technology. Realistically, interoperability will be attainable in degrees, with the required level of interoperability determined by the needs of the federation user. Where interoperability deals with the logical exchange of information between distinct federates during runtime, reuse refers to the adaptation of components (e.g., ideas, whole simulations, lines of code) during the development of a new simulation. Reuse is assisted by having well-defined modular components (the federates) which share the common understanding of what is the meaning of the objects contained within them.

The HLA rules, interface specification, and object model template provide minimum essential tools for interoperability. They ensure that the basic capability for information exchange is in place, they establish the mechanisms for runtime data transfer, and they provide the means to identify appropriate simulations for different purposes. Beyond this, additional interoperability requires additional consistency in the internal representation of the simulations themselves. The extent to which this consistency is required for any particular application depends on the characteristics of that application. While the HLA in itself is insufficient to guarantee interoperability, it provides the technical framework for simulation and federation developers to achieve the degree of interoperability needed to achieve their objectives.

6 HLA TECHNICAL LIBRARY

DMSO has established an on-line HLA Technical Library, accessible through the DMSO Home Page on the World Wide Web (http://www.dmso.mil), which contains not only the HLA Baseline Definition documents, but a wealth of supporting documentation as well. This includes test procedures, design details, an HLA security architecture, examples of the use of the HLA, and an extensive archive of briefings and documents, including all of the AMG proceedings throughout the HLA prototyping phase and published HLA papers. The HLA Technical Library will continue to grow and evolve as the HLA itself is carried forward. It is intended to be an open source for use by implementors of the HLA.

REFERENCES

Dahmann, J. S. 1997. High level architecture for simulation. *Proceedings of the First International Workshop on Distributed Interactive Simulation and Real-Time Applications,* 9-14.

Defense Modeling and Simulation Office 1996. Data distribution and management design document, V0.2. Washington D.C. December (http://www.dmso.mil).

Defense Modeling and Simulation Office 1996. High level architecture baseline development plan, V1.7. Washington D.C. April 1 (http://www.dmso.mil).

Defense Modeling and Simulation Office 1996. High level architecture federation and execution process model, V1.0. Washington D.C. September 6 (http://www.dmso.mil).

Defense Modeling and Simulation Office 1996. High level architecture interface specification, V.1.0. Washington D.C. August 15 (http://www.dmso.mil).

Defense Modeling and Simulation Office 1996. High level architecture object model template. Washington D.C. August 15 (http://www.dmso.mil).

Defense Modeling and Simulation Office 1996. High level architecture rules, V1.0. Washington D.C. August 15 (http://www.dmso.mil).

Fujimoto, R. M. 1990. Parallel discrete event simulation. *Communications of the ACM* 33(10): 30-53.

Jefferson, D. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems* 7(3): 404-425.

Morse, K. 1996. Interest management in large scale distributed simulations, University of California, Irvine.

U.S. Department of Defense 1994. Department of Defense Directive on modeling and simulation management (5000.59). Washington D.C. January 4 .

Wilson, A. L., and R. M. Weatherly 1994. The Aggregate Level Simulation Protocol: An evolving system. *Proceedings of the 1994 Winter Simulation Conference*, 781-787.

AUTHOR BIOGRAPHIES

JUDITH S. DAHMANN is the Chief Scientist for the U.S. Defense Modeling and Simulation Office. She has lead the development of the High Level Architecture.

RICHARD M. FUJIMOTO is a professor with the College of Computing at the Georgia Institute of Technology. He served as the technical lead in defining the time management services of the HLA.

RICHARD M. WEATHERLY is the Chief Engineer for the MITRE Corporation's Information Systems and Technology Division. He wrote the first version of the HLA Interface Specification and lead the RTI 0.X, F.0, and 1.0 software development teams.

Lecture 9. Interactive and Real-time Simulation

Papers to study for the exam

- Young Kwan Cho, Xiaolin Hu and Bernard P. Zeigler. The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems. Simulation. Vol. 79, Issue 4, April 2003, pp. 197–210. DOI: 10-1177/0037549703038880

- Peter H.M. Jacobs, Alexander Verbraeck, and William Rengelink. Emulation with DSOL. In: M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, (Eds.). Proceedings of the 2005 Winter Simulation Conference. IEEE, 2005. pp. 1453-1462.
  Retrieved from: http://www.informs-sim.org/wsc05papers/180.pdf

The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems

Young Kwan Cho
Republic of Korea Air Force
Nonsan City
Choongnam, Korea 320-913

Xiaolin Hu
Bernard P. Zeigler
Arizona Center for Integrative Modeling and Simulation
Department of Electrical and Computer Engineering
University of Arizona
Tucson, AZ 85721
xiaolin@email.arizona.edu

The increasing complexity of large-scale distributed real-time systems demands powerful real-time object computing technologies. Furthermore, systematic design approaches are needed to support analysis, design, test, and implementation of these systems. The authors discuss RTDEVS/CORBA, an implementation of discrete event system specification (DEVS) modeling and simulation theory based on real-time CORBA communication middleware. With RTDEVS/CORBA, the software model of a complex distributed real-time system can be designed and then tested in a virtual testing environment and finally executed in a real distributed environment. This model continuity and simulation-based design approach effectively manages software complexity and consistency problems for complex systems, increases the flexibility for test configurations, and reduces the time and cost for testing. The layered architecture and different implementation issues of RTDEVS/CORBA are studied and discussed. An example application is then given to show how RTDEVS/CORBA supports a framework for model continuity in distributed real-time modeling and simulation.

Keywords: Distributed real-time systems, modeling and simulation, DEVS, CORBA, model continuity, simulation-based design, virtual testing environment

1. Introduction

Distributed real-time object computing technologies have been attracting lots of attention in the real-time computing area during recent years. Because of the effective object-oriented methodologies that enable engineers to reduce the development complexity and maintenance costs of large-scale software applications, object-oriented computing technology has been successfully applied to non-real-time software systems. However, real-time system engineering techniques have not fully adopted the concept of modular design and analysis, which are the main virtues of object-oriented design technologies. As a consequence, the demand for object-oriented analysis, design, and implementation of large-scale real-time applications has been growing.

This paper proposes that a discrete event system specification (DEVS)–based real-time modeling and simulation environment can provide capabilities required by real-time system engineering. DEVS is a sound formal modeling and simulation (M&S) framework based on generic dynamic systems concepts [1]. DEVS is a mathematical formalism with well-defined concepts of hierarchical and modular model construction, coupling of components, support for discrete event approximation of continuous systems, and an object-oriented substrate supporting repository reuse. DEVS is not, however, just a mathematical framework but also a practical M&S tool implemented in various object-oriented languages such as Scheme, C++, and Java. Recently, DEVS modeling and simulation environments also

SIMULATION, Vol. 79, Issue 4, April 2003 197–210
©2003 The Society for Modeling and Simulation International
DOI: 10-1177/0037549703038880

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

have been combined with middlewares such as High-Level Architecture (HLA) runtime infrastructure (RI) and Common Object Request Broker Architecture (CORBA) to support fast and easy construction of distributed models, as well as simulation of such models. These DEVS-based modeling and simulation environments have been shown to support an effective modeling and simulation methodology in various application areas, including design and implementation of real-time control systems [2].

Real-time systems design connotes an approach to software design in which *timeliness* (or *timing correctness*) is as important as the correctness of the outputs (or *logical correctness*) [3, 4]. Timeliness of response does not necessarily imply speed—although, this may be important—as much as predictability of response and reliable conformance to deadlines. For real-time systems, performance estimation to guarantee that the system under design meets performance requirements is crucial. Performance analysis often concerns schedulability, which involves checking the task schedule for feasibility or conformance with the required timing constraints. In distributed networked systems, quality of service (QoS) characteristics, such as the timely delivery of events between system components or priority-based bandwidth utilization, must necessarily be included in performance evaluation. To support the design and performance evaluation for a distributed real-time system, modeling and simulation technologies are developed.

Real-time considerations enter into the world of modeling and simulation in various ways. A real-time simulation is a real-time system in which some portion of the environment, or portions of the real-time system itself, is realized by simulation models [5]. When a simulation model interacts with a surrounding environment, such as software modules, hardware components, or human operators, the simulator must handle external events from its environment in a timely manner [6]. In more general terms, interfacing of abstract models with real-world processes requires that the (logical) time base of the simulation be synchronized as closely as possible to the clock time of the underlying computer system [2]. Work related to real-time simulation and control includes early research in DEVS-Scheme [2], the extension of the DEVS formalism to the DEVS real-time formalism [6], as well as its application to process control [7]. Current projects include the following: PORTS: A Parallel, Optimistic, Real-Time Simulation [5]; Operators Training Distributed Real-Time Simulation (OPERA) [8]; Ptolemy (Concurrent Discrete Event Simulation) [9, 10]; Time-Triggered, Message-Triggered Object (TMO)–Based Distributed Real-Time System Development Environment [11, 12]; and Cluster Simulation—Support for Distributed Development of Hard Real-Time Systems Using TDMA-Based Communication [13].

One concern of using modeling and simulation technology to support the system design of distributed real-time systems is how to maintain a model's continuity during the system's development process. This continuity means that the same model can be reused with minimal change during different design stages—from model design to performance evaluation and even to final execution. To support model continuity, one must develop techniques so that the modeling and simulation framework can treat the development of distributed real-time systems in a systematic way.

In this paper, we present how a distributed modeling and simulation framework, the real-time DEVS/CORBA (RTDEVS/CORBA), is developed to support the design of distributed real-time systems. To set the stage, we will review the concept of model continuity and show how RTDEVS/CORBA supports model continuity when traversing through different design stages. We then discuss different layers of the RTDEVS/CORBA modeling and simulation environment. With this background, we proceed to discuss in detail the implementation issues of RTDEVS/CORBA. Finally, an example is presented showing how a complex distributed real-time system can be developed using the framework of RTDEVS/CORBA and how model continuity is supported during the development process.

2. Support Model Continuity with RTDEVS/CORBA

Model continuity refers to the ability to use the same model of a system throughout its design phases. For the RTDEVS/CORBA environment, model continuity (or model transferability) means that models developed in any DEVS-based M&S environments, such as DEVS-Java, DEVS/HLA, or DEVS/CORBA, should be able to run in the RTDEVS/CORBA environment with minimal changes as long as they maintain the same interfaces [14]. With model continuity, distributed real-time systems can be designed, analyzed, and tested through DEVS-based modeling and simulation studies and then migrated with minimal additional effort to be executed in the distributed network. In other words, this framework provides generic supports to develop models of distributed real-time systems, evaluate their performance and timing behavior through DEVS-based modeling and simulation, and ease the transition from the simulation to actual executions.

Real-time systems deal with external stimuli from outside of the systems with time constraints. Therefore, real-time systems usually interact with environmental systems that could be hardware, software, human operators, and so forth that want to get timely responses from the real-time systems. When a real- time system is developed, the best way to evaluate and test the system under design might be for the system to interact directly with the existing environment. However, this is usually limited by cost and some other limitations such as safety and availability. Hence, it is good to provide a real-time modeling and simulation environment in which the environment model can be developed so that the system under design can be simulated

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

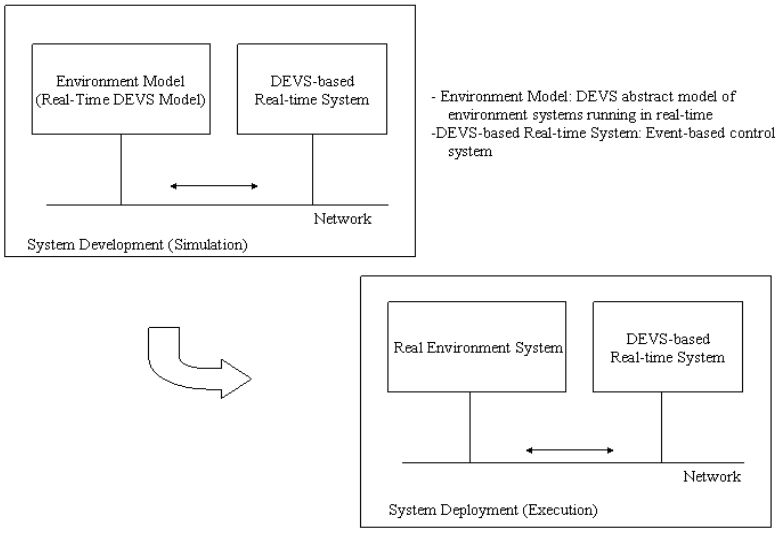


Figure 1. Real-time simulation and execution

along with the environment model. Once the system has been tested by the simulation, it should be easily deployed in the execution environment, in which the interfaces to the system are the same as those between the environment model and the system.

Figure 1 illustrates a conceptual overview of the relationship between real-time simulation and real-time execution in the RTDEVS/CORBA environment. High-level abstract models that provide logical and timing behaviors like the real environment systems can be developed and simulated in this environment so that real-time systems can be developed based on the interactions with the environment models. After finishing the development, the real-time systems could be easily migrated into the execution environment with minimal modifications.

Figure 2 shows in detail how this process works. In the modeling stage, the modeler defines DEVS models that capture the logic behavior and temporal behavior of the system under design. Then these models will be simulated in the DEVS-Java environment, which run a logic simulation and thus can check the models' logical behavior. After the models are checked by DEVS-Java, these same models can be distributed on different computers and simulated by DEVS/CORBA, which is a distributed logical simulation environment. With DEVS/CORBA, the model under design's logical behavior can be further checked in the distributed environment. Although DEVS/CORBA can check the models' logical correctness in the distributed environment, it cannot check the models' temporal behavior. So the next stage of the development is simulating the models in RTDEVS/CORBA, a real-time distributed simulation environment. Because RTDEVS/CORBA provides time-sensitive and QoS-supported distributed real-time simulations, the models' temporal behavior can be checked in this environment. Furthermore, after the models under design pass all these design and test stages, the same models can also be executed by the RTDEVS/CORBA real-time execution engine in a distributed environment.

As can be seen, during the whole process, the model's continuity is maintained, as the same model goes through different development stages—from modeling to testing and to the final execution. This model continuity is supported in the DEVS methodology by choosing different simulation and execution environments during different development stages while keeping the same model unchanged.

3. A Layered Modeling and Simulation Framework

To address design issues for real-time systems in the distributed computing environment, we adopted a layered architecture to support software design in which there is a separation of concerns underlying each layer. Figure 3 shows these layers of the RTDEVS/CORBA modeling and simulation environment.

The heterogeneous network, or the lowest layer in Figure 3, represents the actual physical hardware, computers, cables, routers, and so forth.

The second, or the real-time middleware layer, refers to newly emerging software that provides services to mediate the communication among nodes in the network with some degree of assured performance levels. In the RTDEVS/CORBA implementation, the real-time middleware is Adaptive Communication Environment/The ACE

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

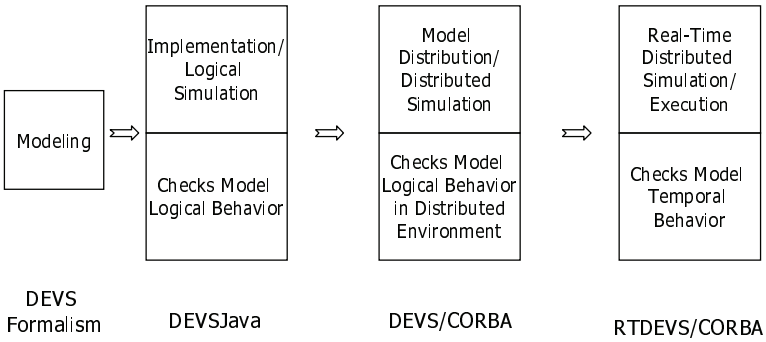


Figure 2. Model continuity during different development stages

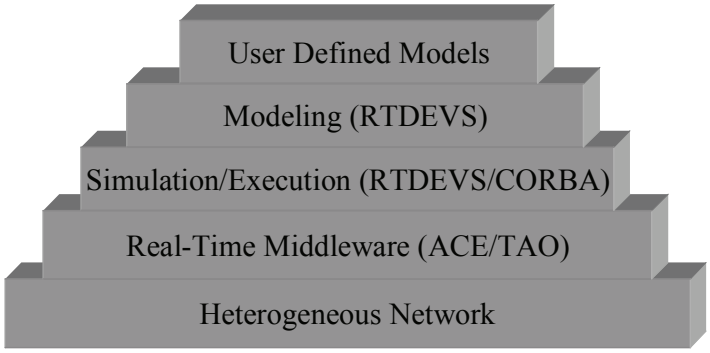


Figure 3. Layers of the real-time discrete event system specification (RTDEVS)/Common Object Request Broker Architecture (CORBA) environment

ORB (ACE/TAO), which is an extension of CORBA developed to demonstrate the feasibility of using CORBA for real-time applications [15]. Because conventional CORBA lacks real-time programming features and QoS support, it is not suited for high-performance, real-time applications. These shortcomings can be overcome by adopting the *ACE ORB* (TAO)—a real-time CORBA-compliant Object Request Broker (ORB) end system developed based on the ACE framework, which is a highly portable object-oriented middleware communication framework [16]. Since TAO is an ACE-based ORB, it runs on a wide range of operating system (OS) platforms and is compliant with most of the features and components in the CORBA 2.4 specification. In addition, TAO also provides real-time CORBA services such as the real-time event service and the real-time scheduling service, which are used to support the RT-DEVS/CORBA environment.

On top of the real-time middleware layer reside the DEVS-based modeling layer (RTDEVS) and the simulation/execution layer (RTDEVS/CORBA). As the RT-DEVS modeling layer provides a modeling environment in which real-time control models can be specified and implemented, the RTDEVS/CORBA layer provides the simulation/execution capability in which DEVS models can be simulated and executed in real time. Finally, based on specific applications, a user can develop his or her models and use the services of the underlying layers to simulate, test, and execute the models under design.

Note that without this separation of concerns, one would have much greater difficulty in trying to formulate and disentangle the control and network response issues just mentioned. Moreover, the layering makes it possible to reuse the software developed at the layers or, indeed, to purchase commercial off-the-shelf (COTS) software if that is available.

4. Implementation of Distributed RTDEVS/CORBA

Implementation of distributed RTDEVS/CORBA is based on the previous framework for the DEVS/CORBA distributed modeling and simulation environment [17]. DEVS/CORBA is a logical simulation environment,

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

whereas RTDEVS/CORBA is a real-time simulation environment. In the logical DEVS simulation environment, generation and delivery of events occur at the same logical time. In the distributed real-time simulation environment, however, this is not the case. There exist some time differences between event schedule, event generation, and event delivery in a real-time context. In a distributed simulation environment, time differences also exist between different distributed computers. Thus, the main concerns about the design and implementation of the real-time simulator in RTDEVS/CORBA are how to handle time in a distributed environment, how to synchronize simulation time with the physical time, and how to manage time differences in generating events at the scheduled time that is defined in the model and delivering them to the destination models in valid time boundaries, which requires QoS capabilities. In this section, these design and implementation issues are discussed.

4.1 Real-Time Simulator

In the distributed real-time DEVS simulation/execution context, running models in real time means that a model must be able to generate an "event" at the scheduled time and change its state based on the calculation caused by events. Furthermore, events generated by a model should be delivered to a remote model within some valid time boundaries specified as QoS by a modeler. The design and implementation of a real-time simulator focus on this fact.

The RTDEVS formalism defines two kinds of events—internal and external— which are handled by the internal transition function (δint) and the external transition function (δext), respectively. An internal event is an event generated by a model itself according to the time schedule specified in the model, whereas an external event is an event delivered from other models or from the external environment.

In Figure 4, the real-time simulator is illustrated as a model that shows the state transition flow of the simulator. The upper box in the figure represents the real-time simulator, and the other box is a DEVS model. The real-time simulator basically behaves almost the same as the previously implemented simulators, except that the time base is synchronized with physical time. When a simulation cycle begins, the simulator initializes and goes into the *nextTN* phase, in which the simulator calculates a time of next event (*tN*) based on the schedule specified in the DEVS model. Once the simulator gets the *tN*, it determines an amount of wait time by subtracting the current time from the *tN* and then goes into the *waitFor* phase, when the real-time simulator performs a *synchronization* step. In this phase, the simulator waits for either the *tN* (an internal event) or an external event. Either event triggers the real-time simulator to move out of the *waitFor* phase. Getting out of the *waitFor* phase is handled by two different simulator algorithms, as shown in Figure 5.

Figure 5a shows the internal transition-handling algorithm, whereas Figure 5b shows the external transition-handling algorithm. At the beginning of each algorithm, the simulator moves into the confluent phase in which the simulator is supposed to handle confluent problems caused by clock and network delay jitter. To compensate for jitter, the real-time simulator waits for a certain amount of time before it goes into the next step. This is explained in detail in the next section. After the confluent phase, the simulator goes through the same simulation steps such as *computeInputOutput*, *wrapDelFunc*, and *nextTN* as in the simulator of DEVS/CORBA.

The real-time simulator iterates one cycle of phases from *waitFor* to *nextTN*, while the simulator keeps interacting with the DEVS model whenever either an internal or external event arrives at the real-time simulator. In an ideal case, this iteration takes a zero time unit, as in the logical simulation. In the real world, however, each step of the iteration takes nonzero time units, which introduces the time discrepancy problem between the time of the event and the actual transition in the model. One aspect of this problem is delay accumulation. For example, suppose that a real-time simulator has reached time $t1$ and gets an internal event. So it needs to execute the internal transition-handling algorithm. Since executing this algorithm takes nonzero time $\varepsilon 1$, the actual internal transition of the model is carried out at time $t1 + \varepsilon 1$. Therefore, if the model calculates its new *tN* from this moment, then *tN* would be $t1 + \varepsilon 1 + ta$, which is not $t1 + ta$, the ideal time according to the original schedule. To make sure *tN* is scheduled at $t1 + ta$, the simulator should calculate the delay $\varepsilon 1$ and compensate this time of next event for the delay. Without delay compensation, accumulation of delay would continue to build up during the simulation cycle until the simulator misses the event schedule. A similar situation exists when an external event comes in and the external transition function needs to be called.

To compensate for the erroneous delays in the simulator, we record the time $t1$ when an internal or external event happens and set the last event time (tL) as $t1$. Then, we execute the event transition-handling algorithm and, when it is finished, schedule $tN = tL + ta$. This way, even the actual event transition happens at time $t1 + \varepsilon 1$, and tN is set to $t1 + ta$ instead of $t1 + \varepsilon 1 + ta$. This is because tN is calculated from tL, which is independent of $\varepsilon 1$, the time it takes to process the event transition-handling algorithm.

4.2 Time Synchronization and the Confluent Problem

4.2.1 Time Synchronization

In logical time DEVS simulation, the coordinator maintains the logical DEVS time and strictly controls the entire simulation cycle. All the participating simulators are closely synchronized with the logical DEVS time under the control of the coordinator. The same is true for any

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

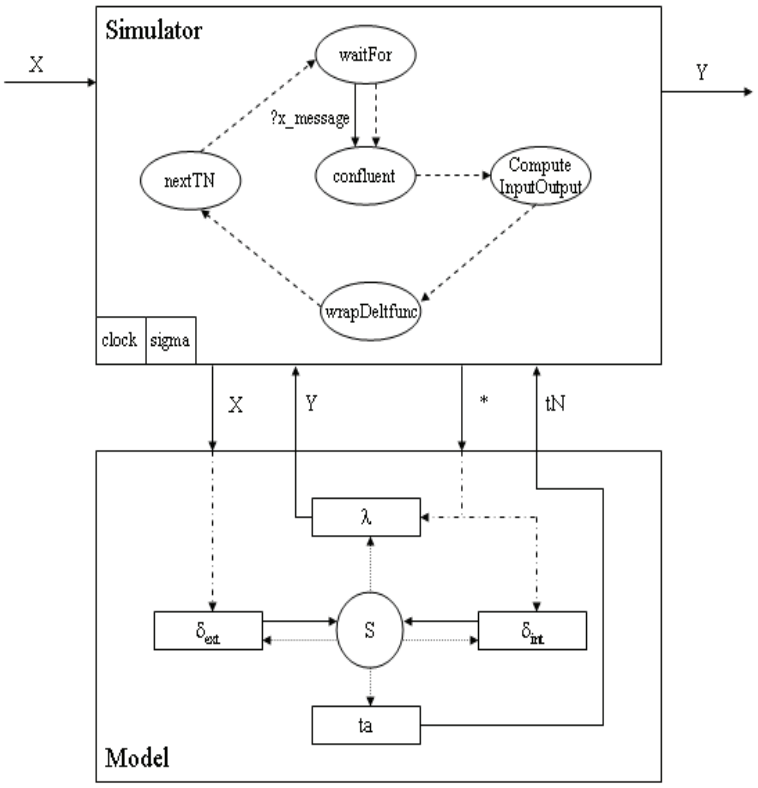


Figure 4. Real-time simulator and its model

```
while(true)
        wait for σ;
        t := checkCurrentTime();
        checkConfluent();
        y := λ(s);
        s' := δint(s);
        tL := t;
        tN := tL + ta(s');
end
```

(a)

```
when receive an external event
        t := checkCurrentTime();
        checkConfluent();
        e := t – tL;
        s' := δext(s,e,x);
        tL := t;
        tN := tL + ta(s');
end
```

(b)

Figure 5. Simulator algorithm

conservative parallel DEVS implementation in the distributed computing environment (i.e., DEVS/HLA and DEVS/CORBA), which performs logical simulation. Although this central control approach manages the time by using one central coordinator, it brings problems such as response latency and central bottleneck. To avoid this disadvantage, in the RTDEVS/CORBA implementation, each computer has its own simulator/executor that simulates/executes the model of that computer autonomously. This autonomous timer approach supports fast response and scalability, but it brings the time synchronization problem in which time consistency has to be maintained among different simulators.

For distributed real-time simulation or execution, synchronization usually includes two aspects: one is how to keep clocks synchronized with each other, which is referred to as *internal synchronization*; another is how to keep them synchronized with the world standard time, which is referred to as *external synchronization*. External synchronization provides distributed systems with the use of a synchronized time base on a global scale, and it enables the systems synchronized with its surrounding environment.

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

Two ways for time synchronization have been studied by Cho [14]. The first one is "soft" synchronization, in which simulators or executors use TAO's time service for time synchronization. The other one is "hard" synchronization, in which a utility such as "AboutTime" is used to reset the hardware clock of the computer, synchronizing with world standard time. With this approach, simulators or executors can get synchronized time reference directly from the local machine's clock through system calls. RTDEVS/CORBA uses the second approach for time synchronization.

4.2.2 Time Confluence

In real-time simulation or execution, due to the nondeterministic delay or delay jitter, it is hard to get event messages as scheduled. According to the International Telecommunications Union (ITU-T) definition, *jitter* is defined as those phase variations with respect to a perfect reference that happen in a clock or data signal as a result of noise, patterns, or other causes. Jitter refers to delay variations in the network context. It describes the variations in the latency of a message transmission. In data networks, too much packet jitter causes a voice to sound garbled. Network components usually compensate for jitter with buffers. Jitter buffers store incoming packets and send them in a more constant stream. In this case, the size of jitter buffers affects the performance of the network. However, there is no optimal size of a jitter buffer because the buffer size will vary from network to network.

A similar concept is used in RTDEVS/CORBA to control time jitter in real-time simulation or execution. Sources of jitter in the RTDEVS/CORBA environment vary. One of the main sources of jitter is the instability in the computer clock. Other sources could be general-purpose OS, network configuration, and other software configurations that are used in the environment. Any system involved in the environment could be the source of jitter. The goal here is not to identify or control the major sources of jitter but to identify problems that jitter could introduce and provide the simulators or executors with a capability to handle jitter in a proper way.

Jitter hinders the real-time simulators or executors from getting messages as scheduled. This situation can be explained by the *GPT* model, which is shown in Figure 6a. This coupled *GPT* model has three atomic models: a *generator* model, a *processor* model, and a *transducer* model. The *generator* produces jobs every 1 second and then sends them to the *processor* and the *transducer*. The *processor* accepts incoming jobs when it is passive and then processes the job and sends the finished job to the *transducer*. The processing time for every job is 1 second, too. The *transducer* collects unprocessed and processed jobs and calculates the performance statistics of the *processor*.

In this example, the *generator* generates a job every 1 second, which is the same as the processing time of the *processor*. This means, in the ideal case, that the *processor* will get an internal event and external event at the same time. However, job generation in the *generator* does not occur at the exact moment that is scheduled in practice. Furthermore, it takes nonzero time to deliver the job to the *processor*. Therefore, as shown in Figure 6b, an external event can be delivered right before or after the internal transition has been executed, which forces the *processor* model to make unnecessary transitions. In the *transducer*, same thing happens. Two external events scheduled to arrive at the same time could be delivered within a small interval.

To reduce this kind of undesirable behavior, a confluent time window is introduced to act like the jitter buffer discussed earlier. Either an external or an internal event can trigger the simulator to perform a confluent checking routine. At the beginning of this confluent checking routine, the simulator or executor sets the confluent time window and waits for any successive events during this period. Figure 6c shows the confluent time windows set forth by an internal or external event in the *processor* and the *transducer*. At the end of the confluent checking routine, the real-time simulator or executor executes the confluent function, which specifies the state transition order between internal and external events that occur together.

The size of the confluent time window is provided by the modeler, and this size determines the time granularity of real-time simulation or execution. It is desirable for the modeler to provide an optimal size of the confluent window based on the delay characteristics of the network environment. If the size of the confluent window is too small compared to the jitter, it will not handle all the confluent problems. On the other hand, if the size is too large, this could result in overlapping with next scheduled event.

4.3 Delivering Messages in Real Time

Delivering messages in real time is crucial for the RTDEVS/CORBA framework. This feature can be enabled by using the real-time event service in TAO. The TAO's real-time event service provides end-to-end QoS guarantees between objects communicating over the network. To get the desired QoS, the real-time requirements for each operation must be provided on the supplier side. A real-time scheduler propagates this information to consumer *RT_Infos* based on the dependency graph. The scheduler then uses the propagated information to order dispatches within a set of operations whose dependencies have been met.

We mapped DEVS ports to suppliers and consumers in the real-time event service, as shown in Figure 7. All the DEVS models that participate in a simulation session are coupled by DEVS ports so that messages generated by a source model can be routed correctly to the destination model based on the *RT_Info* specified. DEVS ports consist of two port classes: input ports and output ports. Output ports are mapped into suppliers, and input ports are mapped into consumers of the event service architecture. *pushInPort* class and *pushOutPort* class were created to incorporate these mapping relations in the framework. These two

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

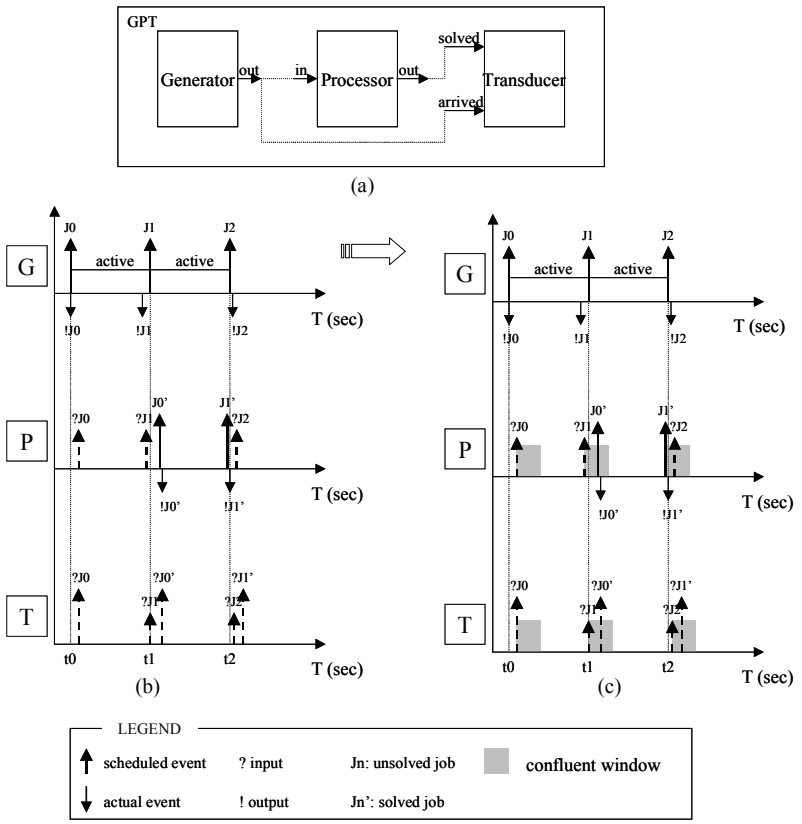


Figure 6. Confluent problem in real-time simulation or execution

classes maintain the port name, source identification, and *RT_Info* associated with an event to be handled as their member variables. When the coordinator downloads the coupling information at the initialization phase, the coordinator retrieves coupling information and the *RT_Info* table from the top coupled model and downloads it to the simulators. The *RT_Info* table contains real-time parameters associated with each event, and the events are distinguished by the name that is created by combining the model name and port name. The modeler must identify QoS information for each event before the assignment and provide the required QoS information through this *RT_Info* table. The coordinator parses the *RT_Info* table and then creates an event list based on the QoS information specified in the table before downloading the coupling information to the simulators. Upon receipt of the coupling information, each simulator creates *pushOutPort* and *pushInPort* accordingly and makes a connection to the event service, which makes a virtual event channel between the pair of *pushOutPort* and *pushInPort*. This virtual event channel is identified by the source identification assigned to the supplier connected to the event channel.

After downloading the coupling information, the coordinator requests the scheduler to compute priorities based on the given *RT_Info* and assigns dispatching priorities to each event. In this framework, each *pushOutPort* and *pushInPort* pair handles only one event, so that the priority assigned to the event can be inherited to the virtual event channel created between the *pushOutPort* and the *pushInPort*.

For DEVS messages to be delivered through the real-time event channel, the DEVS message must be converted into TAO's event format. The event consists of two fields: the event header and the payload. The event header contains routing information, and the data go into the payload. The data are serialized into a byte stream before they go into the payload.

4.4 Getting Data from External Environment Systems

For the real-time simulation and execution, there need to be means to get external input data from the sensors and to send out data to actuators. It is desirable for the

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

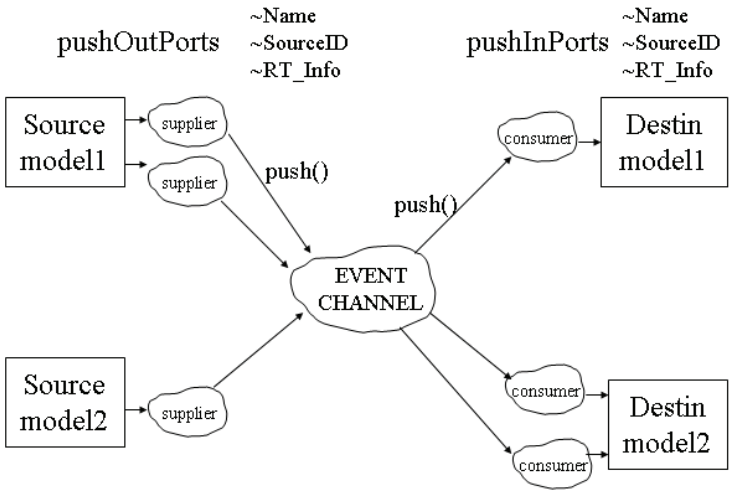


Figure 7. Mapping suppliers/consumers to discrete event system specification (DEVS) ports

real-time model to handle all the external events in the same external transition function in the same manner, regardless of their type. In RTDEVS/CORBA, a modeler can specify an external interface type using the *setExternalInterface()* method. When the simulator initializes the model, the simulator checks the specified external interface types and configures the interfaces accordingly. For example, as depicted in Figure 8, if the model expects external data through socket connections from the sensors, the modeler can specify a socket interface in the model, and the simulator sets up the connection. The real-time simulator creates proxies: sensor proxy and actuator proxy. These proxies are waiting for any data stream, and once data arrive, then they call the simulator's external input-handling method immediately. This method converts socket stream data into a DEVS message and puts it into the model as an external event message so that the model handles the external sensor data as a DEVS message. Of course, a model still can be coupled with other models while receiving data from the external environment.

User-friendly external interfaces are enabled by employing a specialized coupling concept, namely, reserved internal coupling (RIC). The RIC couples the ports attached to the model with the simulator. The real-time simulator has reserved internal ports such as *extInputFromSocket* and *extInputFromKeyboard* as input ports and *extOutputToSocket* and *extOutputToScreen* as output ports. The interface between activities and the model is defined in the same manner. In this case, the simulator starts an activity thread instead of proxies, and these are connected through other reserved internal couplings such as *resultFromActivity*, *cancelActivity*, and so forth.

4.5 Implementation of Activity

An activity is defined to incorporate real computations within simulation models in the real-time DEVS formalism. Such an activity is implemented as an activity thread in RTDEVS/CORBA so that a modeler can assign any computation tasks to the thread.

In the DEVS formalism, a model schedules the next event time using the *hold_in* method, which has the format of *hold_in(phase, sigma)*, where the *phase* is the control state in which the model must be kept for the amount of time specified by *sigma*. The *hold_in* method is also used in the RTDEVS/CORBA environment to schedule the next event time. In this case, however, *sigma* is mapped into the real-time clock (i.e., the value of 5 for *sigma* means 5 seconds in the physical real-time clock). To assign an activity to a certain state for a certain time period, the modeler can use the extended version of the *hold_in* method, which has the format of *hold_in(phase, sigma, activity)*. The activity is a real computation that must be performed to get some results. For example, this could be a DEVS simulation or a piece of computing software that takes some time to get the result. Through this version of the *hold_in* method, the modeler can assign a deadline to be associated with the activity. In this case, the modeler must make sure that the computation can be performed within the specified time window before he or she assigns the activity to the model.

The two versions of the *hold_in* methods also help to maintain the model continuity of the system under design. This is because when a real-time model is developed in the logical simulation environment such as DEVS-Java, the model can use the old version of the *hold_in* method. When the logical behavior of the model is verified and

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

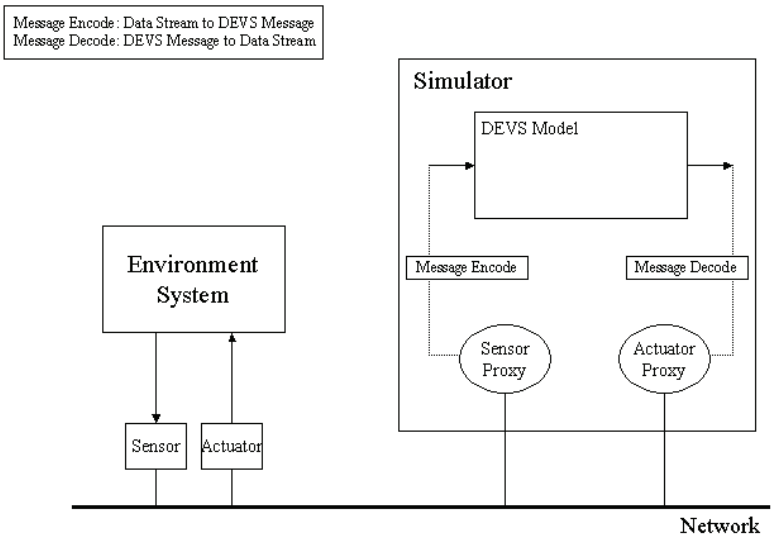


Figure 8. Design of an external interface for the real-time simulator

the model moves to the execution stage, the model can be migrated into the RTDEVS/CORBA environment with the extended *hold_in* method with activity.

5. An Example of Distributed Temperature Control with Hierarchical Scheduling

In this example, we show the ability of the distributed RT-DEVS/CORBA environment to support development of a complex distributed real-time system using real-time simulation. We show how model continuity may be maintained from the very beginning of the development stage to the execution of the model. The example is a distributed intelligent temperature control applied to a building using a hierarchical scheduling scheme. The building could be a hotel, an office structure, a hospital, or any other buildings that have many floors with multiple rooms on each floor. The requirement calls for each room in this building to be controlled individually, as specified by a manager at all times. The overall model architecture is depicted in Figure 9. The model consists of three components: the control system, the monitoring unit, and the building unit. The control system is the real-time system that needs to be developed, and the building unit is the environmental system model that would provide a virtual environment for the real-time system to be developed. Therefore, the building unit must provide outputs through the specified interfaces with highly accurate timing. The system also has a monitoring unit to provide the real-time monitoring capability.

The control system is further divided into the master scheduler and several control units, each of which consists of a master controller and subcontrollers. A master controller is assigned to each floor and controls the subcontrollers, each of which is assigned to a room. The master scheduler is the main interface unit through which the manager can specify the master control schedule, and the master scheduler maintains and distributes the master control schedule to each master controller accordingly. The master controller delivers the control schedule to each subcontroller and monitors the behavior of the subcontrollers. If an error should occur in a subcontroller, then the subcontroller would send out an error message to the master controller, and the master controller would process an error routine for the subcontroller. The building unit consists of multiple floors, each of which is again composed of many rooms.

To construct the model, we assumed that the building had the same floor plan on each floor. Since each floor of the building has the same structure, we only constructed a model for one floor with 10 rooms, as shown in Figure 10. One control unit is assigned to one floor unit model while each subcontroller is interacting with its counterpart, which is a room unit model. The real-time system, which would be developed and migrated from the simulation environment to the execution environment, is the control unit. As discussed before, the control unit model is composed of a master controller and subcontrollers, and the master controller is supposed to get updated control schedules from the master scheduler.

The master scheduler is to provide interfaces for a manager to input, as well as modify and retrieve the schedule. It is also required to store the schedule in an organized

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

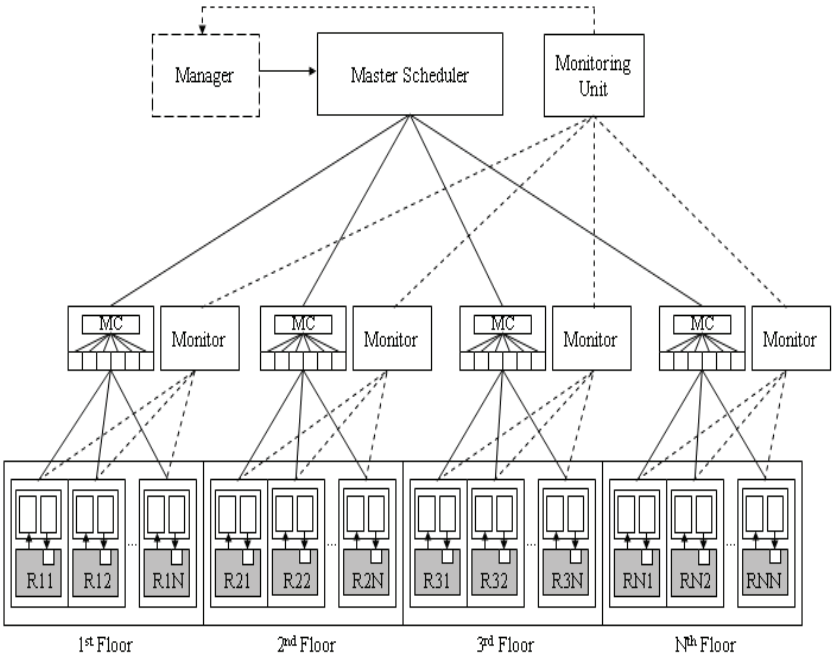


Figure 9. Overall structure of distributed temperature control model

manner and to distribute the schedule to control units when the time arrives. Kim [18] proposed a method for hierarchical scheduling in an environmental control. He used the system entity structure (SES) to organize the environmental control schedule in a hierarchical manner. This scheme is employed in the master scheduler to specify the overall control schedule in a hierarchically organized format, in which the control schedule for each room can be specified in multiple levels.

While the master scheduler maintains the master schedule, it periodically distributes the schedule to the master controllers. Upon receipt of the schedule, the master controller delivers it to subcontrollers, each of which controls its assigned room as scheduled. For example, the master scheduler dispatches a daily control schedule to each master controller, and then the master controller similarly dispatches different set points to each subcontroller according to this time schedule. The subcontroller is the main controller that maintains room temperature as specified in the schedule. Even though the subcontroller maintains room temperature, a guest can adjust room temperature at any time, and this adjustment overrides the regular schedule.

The floor unit is a real-time simulation model that provides an artificial environment for the control system. This floor unit model consists of multiple-room unit models, each of which is composed of a room model, a sensor model, and an actuator model. The room model calculates the current room temperature based on a given function, which is a simple linear function in this example, and provides the current room temperature upon request by the sensor model. The sensor model senses the room temperature and delivers it to the subcontroller every 10 seconds. The actuator model interprets commands from the subcontroller and delivers commands to the heater or the cooler in the room.

Figure 11 depicts the state transition diagram of the subcontroller model, which actually controls the room temperature. The subcontroller implements the event-based control scheme. The subcontroller only accepts sensing data during the *wait* phase, which is the time window (t seconds) for a valid input, and generates a command. Otherwise, it goes into the *error* phase and then generates an error message that goes to the master controller. When the master controller receives an error message from the subcontroller, it sends a *restart* signal along with a set point again. The master controller also provides a set point to the subcontroller at the beginning of every day.

First we built the model in DEVS-Java and checked its behavior to see if it worked correctly in terms of logical dynamics. Then the model was moved into the real-time DEVS/CORBA environment to check its timing behavior. In this example, a control schedule is provided to the master controller in a table format, which contains 10 different set points for each room. For the purpose of the example, we scaled down the time schedule for the master controller to distribute each set point every 10 minutes. This time

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

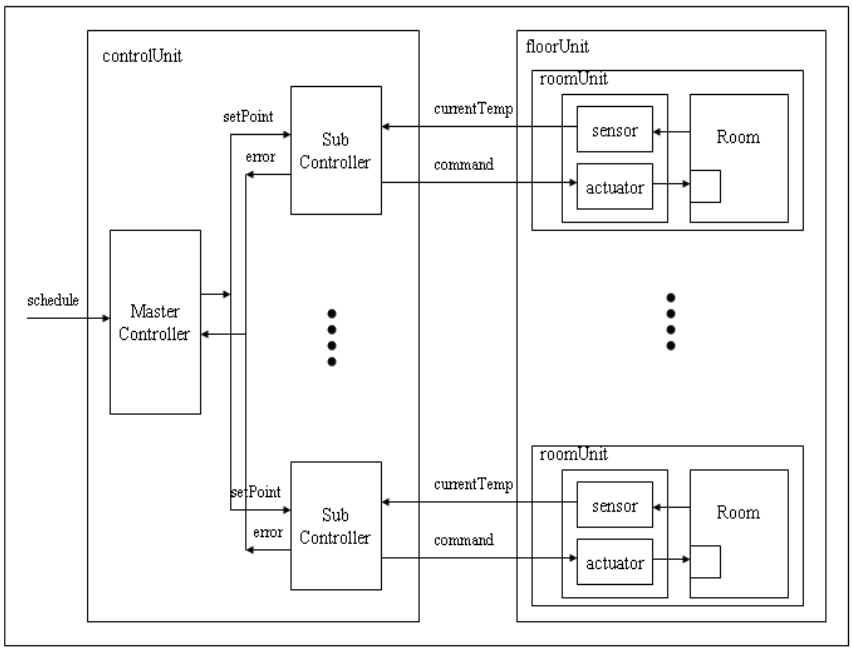


Figure 10. Minimized model architecture

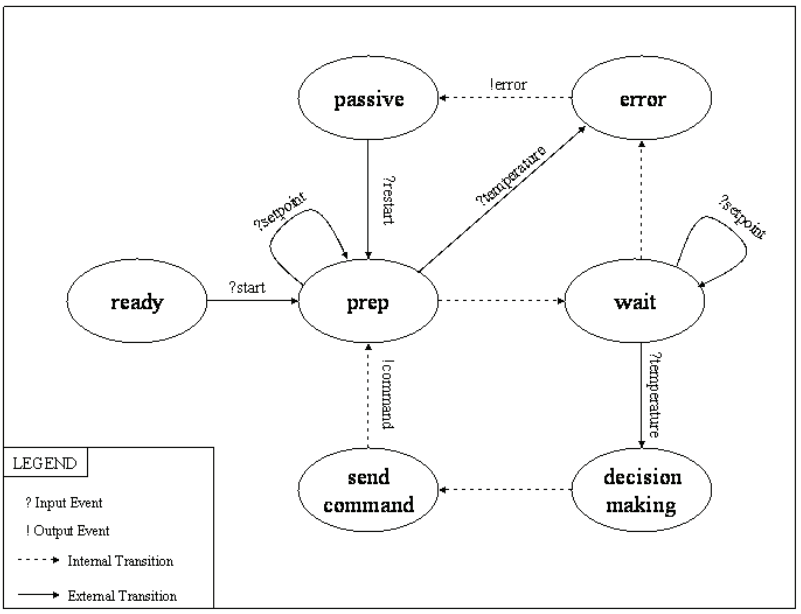


Figure 11. Subcontroller model

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

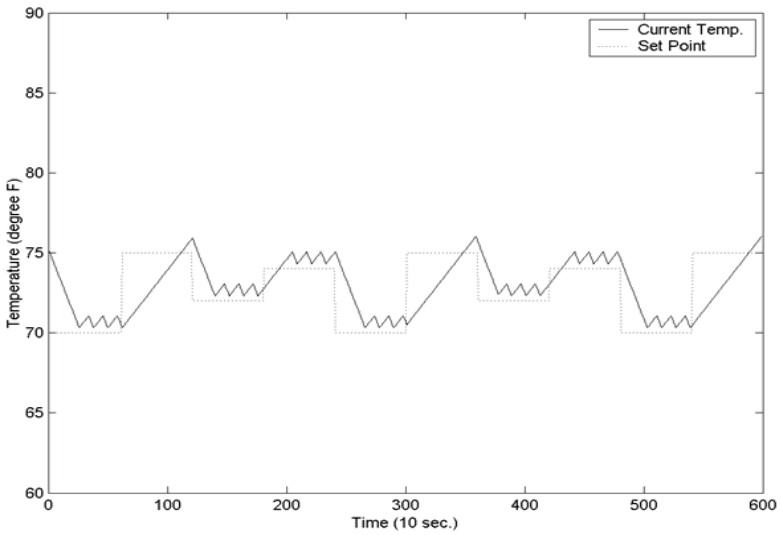


Figure 12. Result of temperature control without user interruption

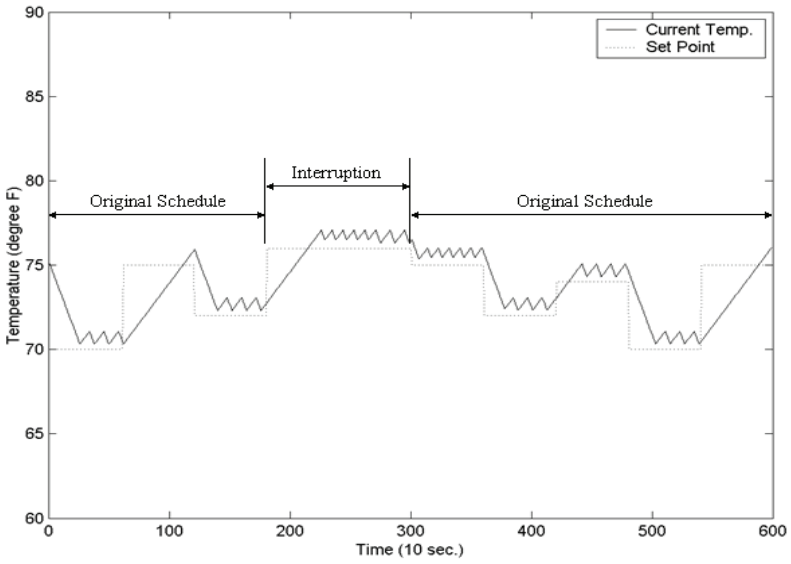


Figure 13. Result of temperature control with user interruption

schedule can be adjusted later on as needed. The model was executed in two ways: with and without user interruption. *User interruption* refers to the change of the set point by the user of the room. While running the model, we attached a monitoring window to each room to see room temperature changes in real time. One monitoring window is connected to the output port of each sensor model of the room unit so that the current room temperature can be monitored. Figure 12 shows the result of the temperature control of the first room without any interruption by the user, whereas Figure 13 shows the result with interruption. These results indicate that all the models are working correctly in terms of both logic and timing.

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

6. Conclusion

In this paper, a modeling and simulation framework, RT-DEVS/CORBA, was presented to support the development of distributed real-time systems. The framework supports model continuity for software development so that the same model can be reused with minimal change during different design stages—from model design to performance evaluation and even to final execution. To support model continuity, RTDEVS/CORBA adopted a layered architecture in which there is a separation of concerns underlying each layer. This layered architecture facilitates reuse of the software developed at different layers and makes it possible to choose different simulation and execution environments during different design stages while keeping the upper layer model unchanged.

To enable modeling and simulation in a distributed real-time environment, RTDEVS/CORBA has chosen ACE/TAO as the communication middleware, which supports distributed programming as well as real-time communication features such as real-time event service, real-time scheduling service, and QoS capabilities. Based on ACE/TAO, different implementation issues of RT-DEVS/CORBA such as the real-time simulator, time management techniques, and the real-time message delivery method have been studied and discussed in this paper.

7. References

[1] Zeigler, B. P., T. G. Kim, and H. Praehofer. 2000. *Theory of modeling and simulation*. 2d ed. New York: Academic Press.

[2] Zeigler, B. P., and J. Kim. 1993. Extending the DEVS-Scheme knowledge-based simulation environment for real-time event-based control. *IEEE Transactions on Robotics and Automation* 9 (3): 351-6.

[3] Laplante, Phillip A. 1997. *Real-time systems: Design and analysis*. 2d ed. Piscataway, NJ: IEEE Press.

[4] Krishna, C. M., and K. G. Shin. 1997. *Real-time systems*. New York: McGraw-Hill.

[5] Ghosh, K., K. Panesar, R. M. Fujimoto, and K. Schwan. 1994. PORTS: A Parallel, Optimistic, Real-Time Simulator. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pp. 24-31.

[6] Hong, J. S., and T. G. Kim. 1997. Real-time discrete event system specification formalism for seamless real-time software development. *Discrete Event Dynamic Systems: Theory and Applications* 7:355-75.

[7] Cho, S. M., and T. G. Kim. 1998. Real-time DEVS simulation: Concurrent, time-selective execution of combined RT-DEVS model and interactive environment. In *Proceedings of SCSC '98*, Reno, NV, pp. 410-5.

[8] Uslander, T. 1999. OPERA: A CORBA-based architecture enabling distributed real-time simulation. In *Proceedings of ISORC '99*, May.

[9] Lee, E. A. 1998. Modeling concurrent real-time processes using discrete events. UCB/ERL Memorandum M98/7, March, Berkeley, CA.

[10] Edwards, S. A. 1997. The specification and execution of heterogeneous synchronous reactive systems. Ph.D. diss., University of California, Berkeley.

[11] Kim, K. H. 1997. Object structures for real-time systems and simulators. *IEEE Computer*, August, 62-80.

[12] Kim, K. H., J. Liu, and M. H. Kim. 2000. Deadline handling in real-time distributed objects. In *Proceedings of ISORC 2000 (IEEE CS 3rd International Symposium on Object-Oriented Real-Time Distributed Computing)*, Newport Beach, CA, pp. 7-15.

[13] Galla, T. M., and R. Pallierer. 1999. Cluster simulation—Support for distributed development of hard real-time systems using TDMA-based communication. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, York, England.

[14] Cho, Y. K. 2001. RTDEVS/CORBA: A distributed object computing environment for simulation-based design of real-time discrete event systems. Ph.D. diss., University of Arizona, Tucson.

[15] Schmidt, D. C., D. L. Levine, and S. Mungee. 1998. The design of the TAO real-time object request broker. *Computer Communications* 21 (4): 294-324.

[16] Schmidt, D. C. 1994. ACE: An object-oriented framework for developing distributed application. In *Prodeedings of the 6th USENIX C++ Technical Conference*, Cambridge, MA.

[17] Kim, D., S. J. Buckley, and B. P. Zeigler. 1999. Distributed supply chain simulation in a DEVS/CORBA execution environment. In *Proceedings of the WSC*, Phoenix, AZ.

[18] Kim, T. G. 1990. Hierarchical scheduling in an intelligent environmental control system. *Journal of Intelligent and Robotic Systems* 3:183-93.

Young K. Cho *is a major in the Republic of Korea Air Force, Nonsan City, Choongnam, Korea.*

Xiaolin Hu *is a PhD student at the Arizona Center for Integrative Modeling and Simulation, Department of Electrical and Computer Engineering, University of Arizona, Tucson.*

Bernard P. Zeigler *is a professor at the Arizona Center for Integrative Modeling and Simulation, Department of Electrical and Computer Engineering, University of Arizona, Tucson.*

Downloaded from sim.sagepub.com at Bibliotheek TU Delft on October 3, 2014

Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

EMULATION WITH DSOL

Peter H.M. Jacobs
Alexander Verbraeck

Department Technology, Policy and Management
Delft University of Technology
Delft, THE NETHERLANDS

William Rengelink

TBA Nederland
Delft, THE NETHERLANDS

ABSTRACT

Manufacturing control systems are extremely hard to design and test. Testing Programmable Logic Controller (PLC) software in an on-line manufacturing setting can be costly, dangerous, and inefficient. The availability of a seamless transition between the real manufacturing process and a simulated manufacturing process on the one hand, and a real PLC and a soft PLC on the other hand might help to solve these problems. Using the Java-based object oriented simulation library DSOL (Distributed Simulation Object Library), a case study was conducted for a concrete floor manufacturer to see whether these problems could be overcome. The full simulation and hardware-in-the-loop tests with DSOL, with the Modbus middleware protocol, and with real and soft PLCs went fine, and showed the added value of the distributed, service-oriented paradigm on which DSOL is based.

1 INTRODUCTION

A case study conducted for TBA Nederland is presented in this paper. TBA is a simulation consulting firm active in the domain of logistic business process (re)engineering using simulation and *emulation* (TBA Nederland 2000). TBA's areas of expertise includes (re)design of manufacturing processes, airport capacity systems, i.e. luggage and cargo systems and container terminals.

The case presented in this paper concerns an emulation study conducted for Dycore b.v., a concrete floor manufacturer. The value of this case mainly results from the conditions under which it was conducted: the case study was done as a competition between a team of developers from TBA and a team from Delft University of Technology. TBA's team, which consisted of two senior engineers, used their de-facto simulation environment, eM-Plant, while the TU Delft team, which consisted of the TU Delft authors of this paper, used DSOL. DSOL is an open source, Java based, multi-formalism simulation suite (Jacobs, Lang, and Verbraeck 2002). The idea behind this, real life, real time competition, was for TBA to be able to assess to what extent DSOL is a serious alternative to eM-Plant and whether they might wish to use it in the future for emulation projects.

We introduce the case in section 1.1, then we discuss the importance of *emulation* in the design of control systems in section 1.2, and illustrate the advantages of using DSOL for emulation projects. We conclude the introduction of this paper with a requirement analysis.

A conceptual model of the case is presented in section 2. The emulation bottleneck, i.e. the communication between the realtime system and the simulation model, forms the topic of section 2.1. The specification of the model in DSOL and a comparison with the specification in em-Plant is presented in section 3. After presenting the experiments in section 3.4, we conclude this paper with conclusions on DSOL's applicability in the domain of emulation. Note, this paper has been read and its content has been endorsed by TBA Nederland and Dycore.

1.1 60m^3 of Concrete Floors on an Automated Guided Vehicle

Dycore is a concrete floor manufacturer. As such Dycore produces annually more than 3,000,000m^3 of concrete floors for the Dutch and global markets. Dycore employs more than 500 employees in their combined facilities. All their production plants are KOMO certified and comply to the NEN ISO 9001 standards (KOMO is a hall-mark of the SBK foundation (<http://www.komo.nl>) and NEN is the Dutch institute for normalization (<http://www.nen.nl>)).

The case presented in this paper deals with the production of *sheet piling floors* at the manufacturing plant in Breda, the Netherlands (see Figure 1).

The production of sheet piling floors is fully automated. All machines, sensors, conveyors, cranes and vehicles are controlled by a *programmable logic controller*, or PLC. The case deals with a subsystem of this production plant, called the *bewapeningsomloop* which is best translated as the *reinforcement gallery*. Dycore commissioned TBA to

Figure 1: Sheet Piling Floors

test a newly designed and developed programmable logic controller. The aim of this commissioning was to debug the engineered system in the lab - not on the factory floor, for a number of reasons. The first and most important one is that this plant works with a 2 shift production that should not be disturbed. A second reason, which results from the first reason, is that testing can only be done in a limited way during night and weekend hours, which are very costly. Finally the operators should be trained in an environment that does not disturb the operations.

1.2 The Importance of Emulation in the Design of Realtime Control

Testing the behavior of a PLC, which controls one or more devices that are part of a logistic system, is usually done by connecting the PLC to stand-alone versions of each individual device, called mock-ups (Schludermann, Kirchmair, and Vorderwinkler 2000, Schiess 2001). This approach to device testing is expensive and the test conditions are hard to reproduce. Above all these test are incomplete since the interaction of a device with other devices is ignored and the system as such cannot be tested as a whole (Schludermann, Kirchmair, and Vorderwinkler 2000, Schiess 2001).

Emulation is a *hardware-in-the-loop* approach that is designed to solve this incompleteness. Emulation implies that all *inputs* and *outputs* of a controller are connected to *simulated devices*. This enables better reproduction of test conditions and allows the tester to reproduce the interaction of various parts of the complete system (Schludermann, Kirchmair, and Vorderwinkler 2000). (Whorter, Baker, and Malan 1997) state that using the same model for system development, system testing using emulation and staff training, can reduce costs and plant set-up times.

1.3 Emulation with DSOL

We expected DSOL to perform better than traditional simulation environments for a number of reasons that are given below. One, DSOL clearly distinguishes a model from a simulator. This distinction supports the usage of one model for system development, system testing and training purposes, as the same model can be deployed for different purposes and with different simulators, e.g. a wall clock simulator for real operations and training, a stepwise simulator for testing, and a paced simulator for demonstrations. Two, the service oriented, open architecture underlying DSOL should make the deployment of an emulation model in a distributed, networked environment more straightforward. Three, the multi threaded, scalable characteristics of the Java programming language should make DSOL more effective in the performance-defiant domain of emulation. Four, the support for CAD drawings in combination with the inclusion of Java's 3D modeling should give DSOL advantage with respect to the straightforwardness of infrastructure modeling.

The value of this case for the validation of DSOL was is found in the opportunity provided support these claims in a real time, real life case. A specification of the requirements for the emulation model is given below.

1.4 Requirement Analysis

To help us understand the requirements for an emulation model of the reinforcement gallery, we will first briefly introduce the internal structure of a PLC. The major components of a PLC are its *CPU*, its *memory*, its *power supply*, its *inputs* and its *outputs*: where inputs provide a PLC with the ability to read signals from different input devices, e.g. sensors and buttons, and outputs provide a way for a PLC to control output devices, e.g. motors and cranes. Data exchange between a PLC and these physical devices is based on special *industrial protocols*. This leads to a first requirement for our emulation model. The most important two requirements from Dycore and TBA were the following:

Requirement 1. *The emulation model must ensure that no modification has to be made in the PLC or the PLC program for testing.*

Requirement 2. *The emulation model should support the industrial data exchange protocol used by Dycore's PLC.* Memory in a PLC can be distinguished into *system memory*

and *user memory*. System memory is used by a PLC for its internal process control system. The user memory contains a user program translated from a *ladder diagram* to a binary form. User memory is divided into blocks having special functions. Some parts of a memory are used for storing input and output status. The real status of an input is stored either as "*1*" or as "*0*" in a specific memory bit. The combination of 16 bits is called a *word*, and each input or output has one or more corresponding bits in memory. An example of two lamps is presented in Figure 2 to illustrate how memory is used to control specific devices. Other parts of memory are used to store variable contents for variables used in user program. For example, timer value, or counter value would be stored in this part of the memory (Matic 2001).

A PLC reads its inputs and sends its outputs on a regular basis. This time interval is called the *period* of the PLC. Since this period defines the accuracy of the control system and as such of the controlled devices, a third requirement is that

Requirement 3. *The emulation model should meet the realtime period of the PLC.*

To ensure that an emulation model meats this real time constraint, emulation requires an underlying real time operating system (Schludermann, Kirchmair, and Vorderwinkler 2000).

Requirement 4. *In situations where an emulation model is nevertheless deployed on standard, i.e. non real time, operating systems, e.g. Linux or Microsoft Windows, the emulation environment should report potential unacceptable backlog.*

Where both requirements focus on the usefulness of the emulation model, TBA presented a further set of requirements with respect to the usability of the testing environment.

Requirement 5. *The emulation model should animate all devices in realtime on top of the CAD layout which was well known to the controllers of the physical system.*

Requirement 6. *All simulated devices should be* controllable *at runtime through a graphical user interface provided by the emulation model. This implies clicking on a simulated device and stopping or resuming its operation, creating failures, pressing buttons, etc.*

2 CONCEPTUALIZATION

According to (Banks 1998) conceptualization implies the abstraction of a real system using a conceptual model, i.e. a series of mathematical and logical relations between objects. This conceptual model underlies both the DSOL specification presented in section 3 and TBA's eM-Plant specification. In this section a clear distinction is made between the *control system* and the *controlled system*. The control system, i.e. the PLC controls the controlled system, i.e the simulated devices in the emulation model. We start in section 2.1 with a conceptual model of the control system, i.e. the PLC. We continue in section 2.2 with the conceptual model of the controlled system, i.e. the DSOL emulation model. In section 2.3 we discuss the communication between the emulation environment and the real time PLC, and we conclude this section with conceptual models of the overall architecture.

2.1 The Conceptual Model of the Control System

To understand how the control system functions, it is necessary to reiterate what we stated in the introduction of this paper, i.e. that the inner works of a programmable logic control are based on changing bit values of internal memory addresses. This is illustrated in Figure 3. The memory of a PLC is divided in a number of *registers*, i.e. data elements containing 16 bits. A clear, but subjective, distinction is made between that part of the memory in which write operations take place, and that part which is read-only for external devices to support the consistency of the PLC. The registers that are read by external devices are called *Registers*, while those registers used for writing are called *InputRegisters*.

Some inputs and outputs use more than one bit to read or write a specific value. For example, consider an automated guided vehicle the speed of which is presented as a double value, i.e. a `double` value in Java. In this specific case, 32-bits are needed to store this speed value, and as such two *InputRegisters* are dedicated to the speed of a specific vehicle.

2.2 The Conceptual Model of the Controlled System

The conceptual model of the DSOL emulation model, i.e. the controlled system is presented, in this section. We introduce three types of devices (see Figure 4) to illustrate the emulation model.

- *Input devices* which *only send* data to the programmable logic controller. Examples of this type include a sensor, an emergency button and a GPS-device.
- *Output devices* which *only receive* data from the programmable logic control. Examples of this type include a lamp and a siren.
- *Combined devices* which *both send and receive* data to the programmable logic controller. Examples include a motor, the operation of which is controlled (or received), and which might send emergency events.

A number of remarks must be made with respect to the conceptual diagram presented in Figure 4. One, the terminology is rather confusing. Although the name `Interface`

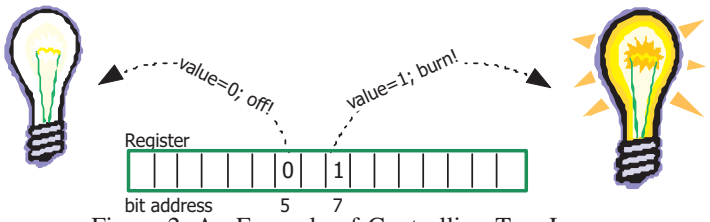


Figure 2: An Example of Controlling Two Lamps

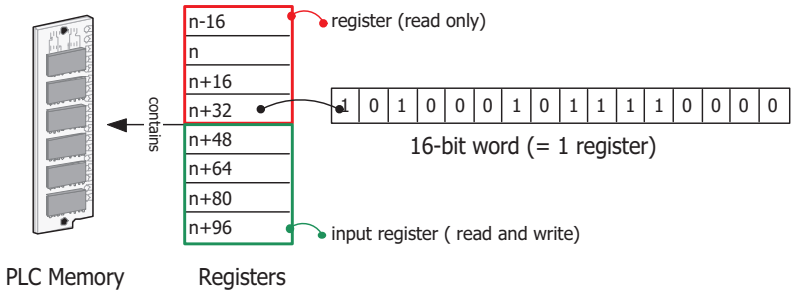


Figure 3: The Memory of a Programmable Logic Controller

might suggest that a Java interface is meant, it represents an abstract class specifying the interface between a simulated device and a PLC. Two, a `0..N` associative relation is presented between an interface and a specific device. This illustrates the fact that several simulated devices may share the same interface. Three, we see that combined devices are associated with both input and output interface. Four, the interface in Figure 4 is associated with the PLC. Having introduced conceptual models of the control system and the controlled system, a more detailed conceptual model of the communication between the control system and the controlled system is presented in the following section.

2.3 The Model-PLC Interface

The emulation model must conform to the PLCs industrial *Modbus* communication protocol to ensure that no modification has to be made on the PLC. Modbus is an application layer messaging protocol, positioned at level 7 of the OSI model, that provides client/server communication between devices connected on different types of buses or networks (Modicon 1996). Modbus can furthermore be accessed over a reserved system port 502 on the TCP/IP stack.

Modbus offers services specified by function codes, which are elements of request/reply protocol data units (Modicon 1996). A *Master-Slave* concept is applied in the field of programmable logic controllers to govern the lower level communication behavior on a network using a shared signal cable (Modicon 1996).

A number of basic public functions have been developed in the Modbus protocol for exchanging data that is typical for the field of automation. These functions are presented in Table 1.

Any TCP/IP based implementation of the protocol should extend the protocol data units with an IP specific header. As we will show in the next section, a detailed, reliable and above all high performance implementation of this protocol is crucial for using emulation for PLC testing purposes successfully.

The conceptual architecture for the emulation model is presented in Figure 5. On the left, the programmable logic controller (PLC) is attached to the TCP/IP network. On the server side of the network a Java implementation of the Modbus protocol ensures adequate communication with the PLC. To minimize the amount of communication over the network, this communication library is connected to a *shadow memory* of the PLC which is part of the emulation model. The shadow memory limits communication in the following ways.

- Although all registers are read periodically from the PLC, the shadow memory will only fire value change events whenever *input values*, i.e. values to be written into the memory of the PLC, are actually changed.
- The shadow memory only sends changed input registers values to the PLC.

Emulated components either write to this shadow memory, and thus to the PLC, or are asynchronously subscribed to changes on the memory addresses which control their behavior. A sensor is an example of such writing, i.e. of an input, component, while a crane is an example of a reading, i.e. of an output, component. The behavior of, and interaction between, components in the emulation model is time dependent and this requires a simulator.

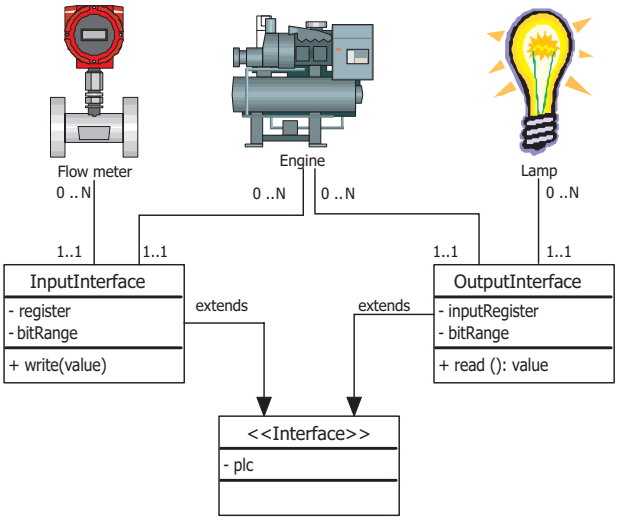


Figure 4: A Conceptual Model of Devices

Table 1: Basic Modbus Functions

Name	Type	Access	Visual representation
discrete input	single bit	read-only	
discrete output, i.e. coil	single bit	read-write	
input register	16-bit word	read-only	
output register	16-bit word	read-write	

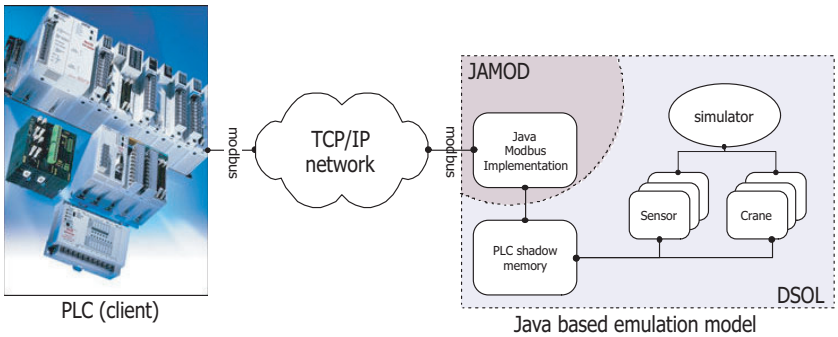


Figure 5: The Emulation Architecture

3 SPECIFICATION

The next activity in the design of the emulation model was to specify the model in DSOL. We begin in section 3.1 with the specification of the Modbus-DSOL communication. We continue with the specification of the simulated devices in DSOL in section 3.2.

3.1 Modbus Communication with DSOL

As argued in the introduction of this paper, great emphasis is placed on the usefulness of the simulation suite based on the ease with which developers can integrate or communicate with external libraries and services.

Although Modbus communication is required for this case to be successful, a Java implementation is clearly not considered among the tasks for a general purpose simulation

suite. A Google search on *"java Modbus"* presented an open source implementation of this protocol named *Jamod*. While the development of a dedicated eM-Plant-Modbus library took several ($\approx$ 3) weeks, the availability of this verified, validated and documented library allowed us to achieve DSOL-Modbus communication within hours. This we consider to be a scientific validation of the value of service oriented computing applied to a simulation suite.

The seamless integration between DSOL and Jamod is presented in Figure 6. The class diagram illustrates how the shadow memory, the input registry and DSOL's event library provide the communication between the PLC and the emulated devices.

The `ModbusMemoryImage` class specifies our implementation of the shadow memory. It implements Jamod's `ProcessImageImplementation` interface which embodies a set of operations for resolving and installing the basic functions presented in Table 1. The interface is implemented by our `ModbusInputRegister` class, which extends DSOL's `EventProducer` class. The `ModbusMemoryImage` further implements DSOL's `EventListener` interface and is asynchronously notified whenever the bit value of an input register is changed. The `ModbusOutputRegister` follows the same structure, but is, because of readability, not presented in this Figure.

3.2 The Specification of Emulation Components

A class diagram of one of the simulated devices is presented in Figure 7. For reasons of readability only this simple component, i.e. the sensor, is presented. All other input and output devices are specified in a similar manner.

The `Sensor` class extends the `Device` class. The fact that a sensor has no embedded knowledge of any communication protocol is illustrated in Figure 7. Since only semantic operations, e.g. the `isState()` operation, are implemented, this simulated device class could well be included in a more general purpose simulation library.

Specific bitwise Modbus memory updates are accomplished by the `SensorInterface` class. This class is asynchronously subscribed to state changes of the sensor to ensure a loosely coupled, efficient communication protocol. The principle of inheritance is applied with the creation of the `AbstractInterface` class. This abstract class uses the *ModbusInput* shown in Figure 6.

We expected a more elegant simulation model due to our ability to use Java's 3D library. Instead of defining hard coded relations which would inevitably result in emulated objects moving over predefined tracks, the DSOL model uses 3-dimensional bounds to see whether objects intersect. To illustrate this functionality, we present the `detect` method of a `Sensor` in Figure 8:

The current `view`, i.e. the volume representing the range of sight, of the potentially moving sensor is computed in line 235. This `view` is an instance of `Bounds`, which defines a convex, closed volume that is used for various intersection and culling operations. In line 240 the `intersects` method is invoked on this `view`. The `targets`, e.g. pallets or cars, are provided as an argument; these `targets` are resolved from a `context`.

3.3 The Specification of the DSOL-PLC Communication

(Schludermann, Kirchmair, and Vorderwinkler 2000) discuss time constraints in an emulation model. These constraints impose great challenges on the simulation environment and on the underlying operation system. To understand these challenges we illustrate the activity sequence of DSOL's simulator, i.e. the realtime clock.

Every time period the simulator sends the input part of the shadow memory to the PLC over the Modbus protocol. Then the simulator reads the output memory from the PLC and notifies subscribed listeners, i.e. Modbus outputs, in case the values have changed. These Modbus outputs block the simulator thread while they invoke mapped semantic operations on the device they control, e.g. `crane.stop()`. After completing this notification, the simulator fires an update animation event, and the CAD based graphical user interface will be redrawn.

The reinforcement welding equipment at Dycore requires a maximum time period of $30 \cdot 10^{-3}$ seconds. For the emulation to succeed, DSOL's emulation framework must guarantee the above sequence is completed within this period. The DSOL emulation model therefore was designed with two high-priority threads, i.e. the simulator thread and the communication thread, and one low priority animation thread. The loosely coupled relation between component behavior and animation furthermore ensured that while the refresh rate of the model was related to the period of the PLC ($\approx$ 35 Hz), the refresh rate of the animation could be slower ($\approx$ 5 Hz). The value of this distinction is that the increased priority of the communication thread allowed the required period of the emulation model to be reached (see Figure 9). Although the values presented in Figure 9 show that DSOL was in general well able to communicate every $30 \cdot 10^{-3}$ seconds with the PLC, there were occasions when a positive backlog occurred.

Because of Java's ability to spread tasks over multiple threads and to differentiate their priorities, DSOL outperformed eM-Plant by a wide margin in this task. Although we have not been given detailed information on the performance of TBA's eM-Plant model, the claim of DSOL substantially outperforming eM-Plant has been made by TBA's engineers. This achievement supports the value of interface based design. The openness of the simulation suite invited us to design a specific, performance aware `RealTimeClock` class which implements the `SimulatorInterface`.

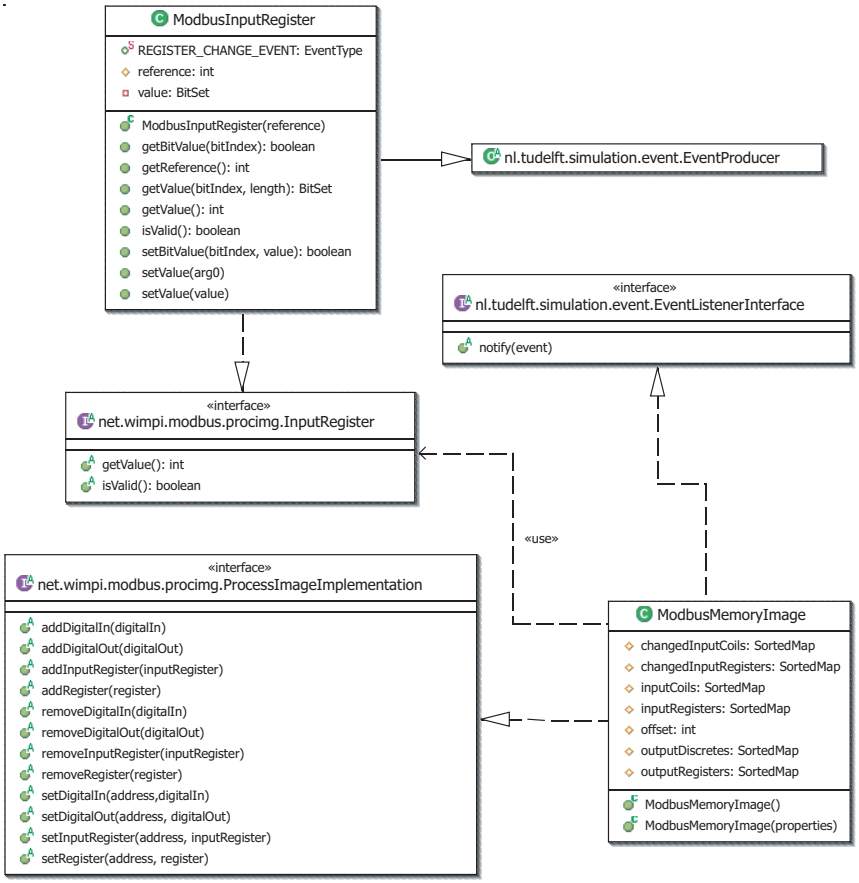


Figure 6: DSOL-Jamod Interdependence

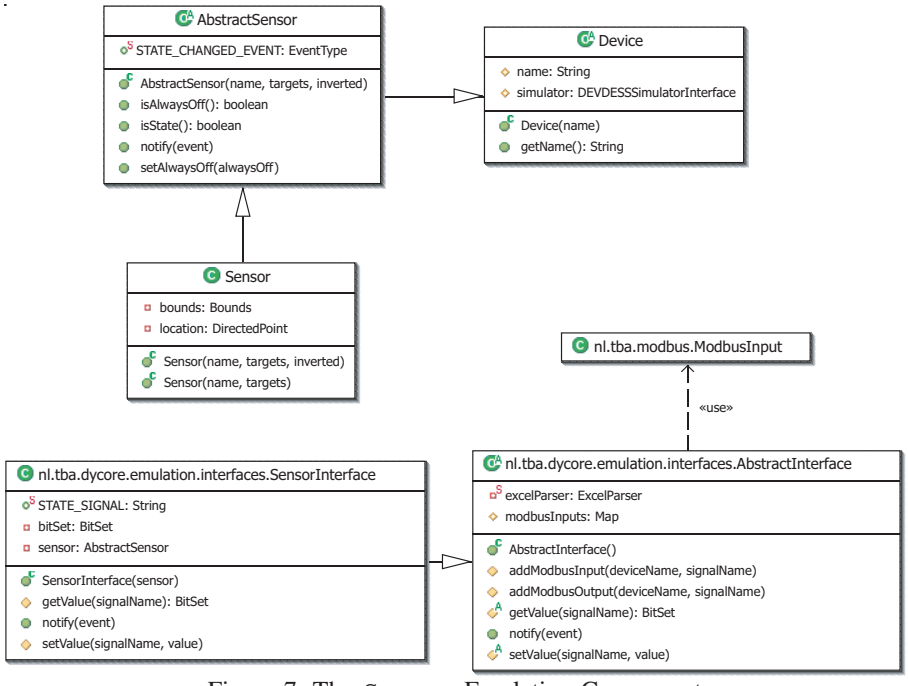


Figure 7: The `Sensor` Emulation Component

```
225 /**
226   * detects a locatable object
227   *
228   * @return whether the sensor has detected a Locatable
229   */
230 private boolean detect()
231 {
232   try
233   {
234     //We compute the space we can currently oversee.
235     Bounds view = BoundsUtil.transform(this.getBounds(), this
236         .getLocation());
237     for (Iterator i = his.targets().iterator(); i.hasNext();)
238     {
239       LocatableInterface locatable = (LocatableInterface) i.next();
240       if (view.intersect(BoundsUtil.transform(locatable.getBounds(),
241         locatable.getLocation())))
242       {
243         //Our view intersects with this target.
244         return true;
245       }
246     }
247   } catch (Exception exception)
248   {
249     Logger.warning(this, "detect", exception);
250   }
251   //Nothing detected
252   return false;
253 }
```

Figure 8: The `Detect()` Method

```
-------------------------------------------------------------------%
Tally: Backlog of ModbusConnectionThread                           %
N=8840 MAX=305.0 MIN=-35.0 AVG=-29.9495475113 STD=16.00013964397 %
-------------------------------------------------------------------%
```

Figure 9: The Measured Backlog (milliseconds) in the DSOL-PLC Communication

3.4 Experimentation

We present the graphical user interface of the Dycore emulation model in Figure 10. In this user interface the underlying Autocad files are rendered by an external, open source, geographical information system service named *Gisbeans* (Jacobs and Jacobs 2004). We consider the ease of using this external service, which is in no way related to the domain of simulation, to again act as a validation for the service oriented paradigm. The popup screen titled H2031 illustrates how devices can be individually and independently controlled.

4 DISCUSSION AND CONCLUSIONS

The most important question yet to be answered is to what extent this case supports our hypothesis that a service based simulation suite provides more effective emulation support. To answer this question we need to clarify the differences between the two specifications.

The first appearance of the service oriented paradigm underlying DSOL is the seamless integration of several off the shelf libraries and services. Poi was used to read/write Microsoft Excel files, Gisbeans was used to render CAD files and Jamod was used to communicate over the Modbus protocol. This is in clear contrast with the eM-Plant implementation developed by TBA. Requirements in eM-Plant are either natively supported or require dedicated, proprietary engineering.

The openness of the DSOL platform proven to be crucial for meeting the performance constraints of the case. The fact that tasks can easily be dispersed over several threads, each with a specific priority meant that the performance of DSOL's realtime clock outperforms eM-Plant substantially.

The loosely coupled relation between component behavior and animation ensured that while the refresh rate of the model was related to the period of the PLC ($\approx$ 35 Hz),

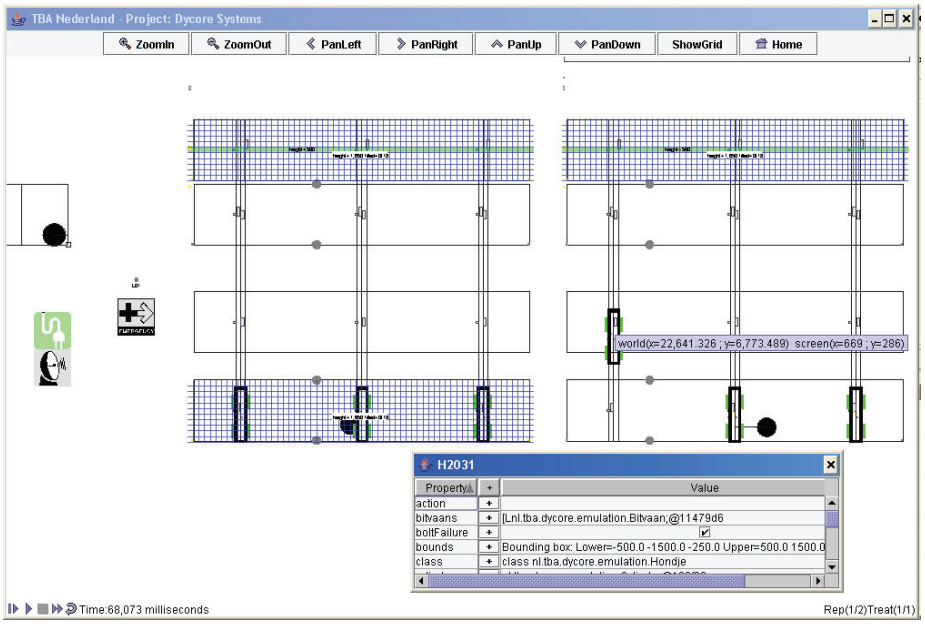


Figure 10: The Graphical User Interface

the refresh rate of the animation could be slower ($\approx$ 5 Hz). These measurements are conducted on a 1Ghz Pentium III, 512MB RAM system with DSOL 1.6 on JRE 1.4.2. The value of this distinction is that through the increased priority of the communication thread, the required period of the emulation model could be reached (see Figure 9). Although the values presented in Figure 9 show that DSOL was in general well able to communicate every $30 \cdot 10^{-3}$ seconds with the PLC, there were occasions were a positive backlog occurred. This justifies further research on the applicability of DSOL on a real-time operating system.

The inclusion of Java's 3D model resulted in a track-less infrastructure model. The sensors, conveyors and cranes can actually scan their neighborhood for pallets to be moved or lifted. Such a three-dimensional library is not available in the eM-Plant specification, which results in more constraining, dedicated relations between infrastructural objects if eM-Plant is used.

As these differences show, the service oriented paradigm underlying DSOL especially seems to show its added value whenever models require sophisticated domain specific challenges. In the comparison between the two projects several advantages of DSOL can be seen. For a large number of cases the currently used software still meets most of the requirements. But where traditional simulation environments are designed for a one purpose, one formalism and one target audience, this case clearly shows the added value to be gained from an interacting, open simulation suite.

5 OBTAINING THE SOFTWARE

DSOL is published under the General Public License. More information on the license can be found at <http://www.gnu.org/copyleft/gpl.html>. The DSOL project description can be found at <http://www.simulation.tudelft.nl> and the software can be downloaded from <http://sourceforge.net/projects/dsol/>.

REFERENCES

Banks, J. 1998. Principles of simulation. In *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*, ed. J. Banks, 3–31. New York: NY, USA: Wiley-Interscience.

Jacobs, J., and P. Jacobs. 2004. Gisbeans: a Java library for geographical information systems. Available via <http://gisbeans.sourceforge.net> [accesed October 21, 2004].

Jacobs, P., N. Lang, and A. Verbraeck. 2002. A distributed Java based discrete event simulation architecture. In *Proceedings of the 2002 Winter Simulation Conference*, ed. E. Yucesan, C.-H. Chen, J. Snowdon, and J. Charnes, 793–800. San Diego, CA, USA: Institute of Electrical and Electronics Engineers: ACM Press. Available via <http://www.informs-cs.org/wsc02papers/102.pdf> [accesed October 21, 2004].

Matic, N. 2001. *Introduction to plc controllers*. Belgrade, Serbia: mikroElektronika. Available via <http://www.mikroelektronika.co.yu/

english/product/books/PLCbook/plcbook.htm> [Accesed October 28, 2004].

Modicon 1996. Modbus protocol reference guide. Technical Report PI-MBUS-300, Modbus-IDA. Rev. J.

Schiess, C. 2001. Emulation: debug it in the lab — not on the floor. In *Proceedings of the 33rd conference on Winter simulation*, ed. M. Rohrer, D. Medeiros, and M. Grabau, 1463–1465. Arlington: VA, USA: Institute of Electrical and Electronics Engineers: ACM Press.

Schludermann, H., T. Kirchmair, and M. Vorderwinkler. 2000. Soft-commissioning: hardware-in-the-loop-based verification of controller software. In *Proceedings of the 32nd conference on Winter simulation*, ed. P. Fishwick, K. Kang, J. Joines, and R. Barton, 893–899. Orlando: FL, USA: Institute of Electrical and Electronics Engineers: ACM Press.

TBA Nederland 2000. Tba nederland specialised in simulation of factories, harbours, airports and railsystems. Accesed via <http://www.tbanederland.nl/default.asp> [accesed October 21,2004].

Whorter, S., B. Baker, and G. Malan. 1997. Simulation system for control software validation. In *Proceedings of the 1997 SCS Simulation Multiconference*. Atlanta: GA, USA.

AUTHOR BIOGRAPHIES

PETER H.M. JACOBS is a PhD. student at Delft University of Technology. His research focuses on the design of simulation and decision support services for the web-enabled era. His working experience within the iForce Ready Center, Sun Microsystems (Menlo Park, CA), and engineering education at Delft University of Technology founded his interest for this research. His e-mail address is <p.h.m.jacobs@tbm.tudelft.nl>.

ALEXANDER VERBRAECK is an associate professor in the Systems Engineering Group of the Faculty of Technology, Policy and Management of Delft University of Technology, and a part-time full professor in supply chain management at the R.H. Smith School of Business of the University of Maryland. He is a specialist in discrete event simulation for real-time control of complex transportation systems and for modeling business systems. His current research focus is on development of open and generic libraries of object oriented simulation building blocks in Java. Contact information: <a.verbraeck@tbm.tudelft.nl>.

WILLIAM RENGELINK (M.Sc. in Aerospace Engineering) is a senior consultant at TBA Nederland , a leading simulation consultancy company in the Netherlands and active for ports, airports and factories all over Europe. TBA is involved in a number of state-of-the- art automation projects concerning baggage handling at Amsterdam Airport Schiphol and the testing of the Control System in the Port of Rotterdam.His work is in the fields of simulation and emulation for logistic systems within airports and factories. He is involved in various international projects. His email address is <william@tba.nl>.

Lecture 10. Simulation and Gaming

Papers to study for the exam

- Alexander Verbraeck, Stijn-Pieter A. van Houten. From Simulation to Gaming: An Object-Oriented Supply Chain Training Library. In: M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, (Eds.). Proceedings of the 2005 Winter Simulation Conference. IEEE, 2005. pp. 2346-2354. Retrieved from: http://www.informs-sim.org/wsc05papers/294.pdf

- Rick van Krevelen, Martijn Warnier, Frances Brazier, Alexander Verbraeck and Thomas Corsi. Transparency, Consistency and Modularity of Strategic Reasoning: An Agent Architecture for Interactive Business Simulations. In: S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, (Eds.). Proceedings of the 2011 Winter Simulation Conference. IEEE, 2011, pp. 1617-1628. Retrieved from: http://www.informs-sim.org/wsc11papers/144.pdf

Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

FROM SIMULATION TO GAMING: AN OBJECT-ORIENTED SUPPLY CHAIN TRAINING LIBRARY

Alexander Verbraeck
Stijn-Pieter A. van Houten

Faculty of Technology, Policy and Managment
Jaffalaan 5
Delft University of Technology
Delft, 2628BX, THE NETHERLANDS

ABSTRACT

The development of web-enabled interactive training simulations is far from easy, especially when all models have to be developed from scratch for each training game. Actually, one would like to be able to reuse parts of existing, off-line simulation models in an interactive setting. The challenge is how to set-up simulation models or simulation libraries that are developed for off-line simulations in such a way that they can be reused for on-line situations, and adapted for different educational settings. Using a supply chain context as an example, this paper shows how libraries of simulation components can be applied both for off-line simulation studies and for on-line training. The paper also describes the other functionality that is needed to create a generally applicable component library for supply chain training simulations.

1 INTRODUCTION

Due to the dynamic nature of supply chains, simulation is a natural and important instrument for the analysis and design of supply chains and supply chain management. Models, particularly those that offer good insight through visualization and graphs, can help companies to structure and simplify their complex and dynamic supply chains. The simulation models can help to structure, transform, condense and visually display data in such a way that managers can quickly grasp a situation and act upon the presented information (Boyson et al. 2004). Many of the supply chain models are, however, created uniquely for one specific situation, and it is hard to reuse the developed model or parts of the model for other situations.

Another aspect that complicates the modeling of supply chains is that they are heavily distributed. For the management of supply chains, there is not one central organization that has the authority to make decisions on behalf of the entire supply chain. All actors take their own decisions, based on limited information, as the other organizations do not share all their data with others. This has a severe effect on the models that are created for supply chain decision making; many decisions have to be taken based on assumptions of other organization's information or behaviour.

Many 'traditional' supply chain models disregard the fact that suppy chain information is spread over multiple organizations and that modern supply chains are highly dynamic. Trade exchanges, short-term contracts, and spot buy markets make that supply chains are not static entities, but highly dynamic networks that work largely on a pull basis rather than on a push basis. Many traditional modeling applications, therefore, deliver increasingly limited value to companies struggling with these conditions.

This is not to say that modeling is not useful in today's fast-paced supply chains. On the contrary, Boyson et al. (2004) state that modeling is more important and more needed than ever before. Because of the nature of supply chain dynamics, managers often do not have insight into the ripple effects of their decisions. Effects also can easily get lost in the overwhelming flood of data that crosses the supply chain manager's desk daily, weekly, or monthly. A rapidly changing supply chain with a continuous change of partners leads to different sets of decisions than a stable chain with long-term contracts.

Most real world logistic problems have an inherent dynamic character; objects operate in parallel, affect each other and change over time. Descriptions of the real world are moreover inherently subjective, dependent on the vision of the focal actor involved in the problem. Many modeling techniques fail to describe and design this type of chaotic and dynamic problem situations. Especially in complex supply chain problems, with many actors involved, and lots of complex technology, it is extremely important to be able to show the *different* views of different actors, and to allow for interaction with the model to

study the effects of parameter changes as seen by these actors. This may require distributing a model over more computers.

When we want to prepare students or managers for taking better decisions in dynamic supply chains, we of course want realistic interactions in the models that are used in these teaching situations. For logistics students, it is extremely instructive to study complex systems by varying parameters in different parts of the system and look at the effects on the most important performance indicators in the model. Figure 1 shows part of a supply chain model as we developed it for teaching purposes in the simulation language Arena (Kelton et al. 2002). Students can vary all kinds of parameters for the organizations in the supply chain, and study the outcome of the model, both during the run and after the run. Indicators such as inventory levels, production times, cycle times, and costs can be studied in detail per supply chain actor, as illustrated in Figure 2.

The problem with these types of teaching or training models is that it is very time consuming to develop them, it is hard to incorporate the dynamics of the supply chain itself as the structure of the model is ususally fixed, and the interaction with the model is very limited. Each student or group of students is able to make the assignments by carrying out a number of 'what-if' studies, but we are not able to demonstrate the effects of decision making on a more continuous basis. In this paper we present a Java-based architecture of supply chain components that overcomes the above mentioned problems. The components can both be used in a stand-alone 'what-if' mode and in highly interactive supply chain training games. In developing the architecture, we tried to develop the component libraries in such a way that the 'mode of use' (demonstration, what-if, interactive, competitive game) is not visible in the components themselves, but rather in additional services that are used to deploy the supply chain library for a particular mode of use. These additional services for gaming are also briefly introduced in the paper.

2 A BASIC SUPPLY CHAIN ARCHITECTURE

As stated before, one of the major requirements of the supply chain architecture is to be applicable in multiple settings, ranging from classical simulation to on-line gaming. In these settings, many different problem contexts may be represented, such as inventory problems (Shapiro 2001, ch.11), aliances and global integration (Simchi-Levi et al. 2003, ch. 6 and ch.8; Archibugi et al. 1999), the increasing role of customers (Fredenhall & Hill 2001), the effects of e-business on supply chains (Poirier & Bauer 2000), or the effect of portals on supply chain management (Boyson et al. 2004). Specific cases may demonstrate the effects of decisions on the bullwhip effect in supply chains (Lee et al. 1997), the effects of postponement on inventory levels (Chopra & Meindl 2001;

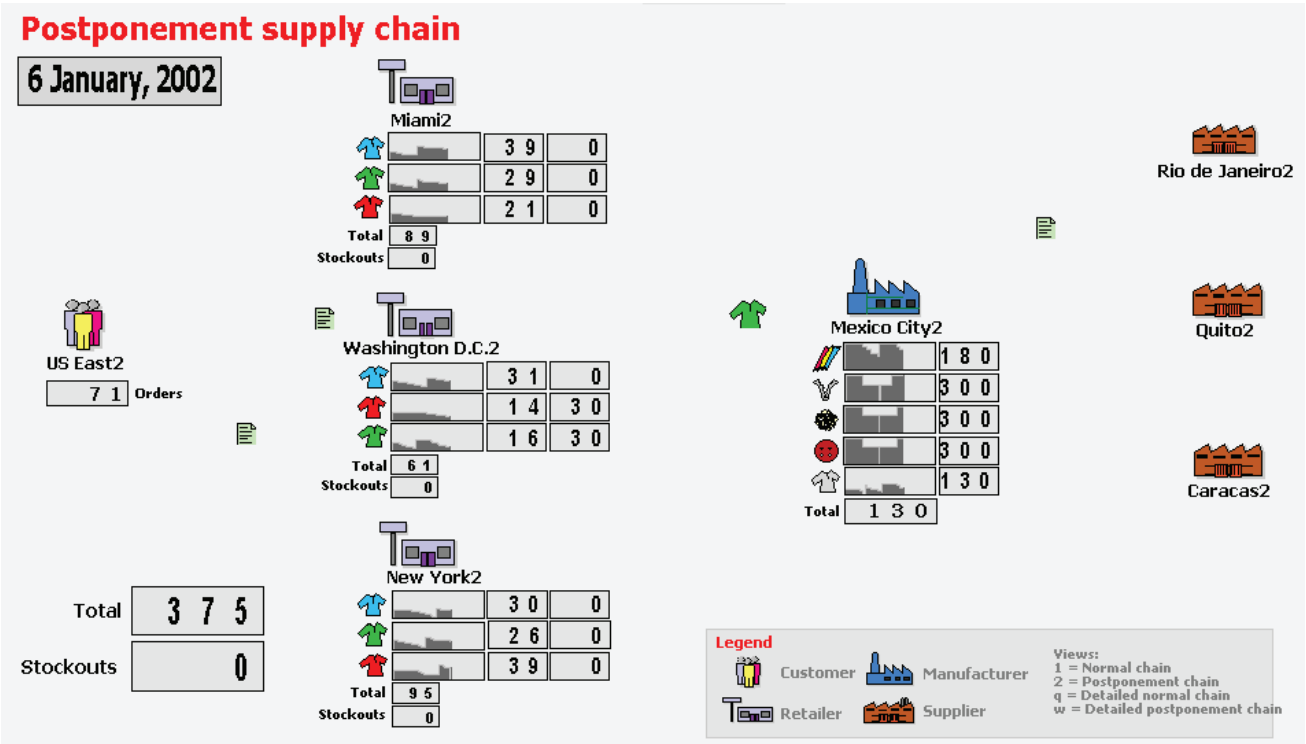


Figure 1:. Screen-Shot of a Supply Chain Teaching Game Developed in Arena

Hoek 2001), see also Figure 1, or the information that is needed for global sourcing and global sales (Waters 1999). Addressing all these different contexts asks for a set of supply chain and supply chain management building blocks that allow for flexible assembly.

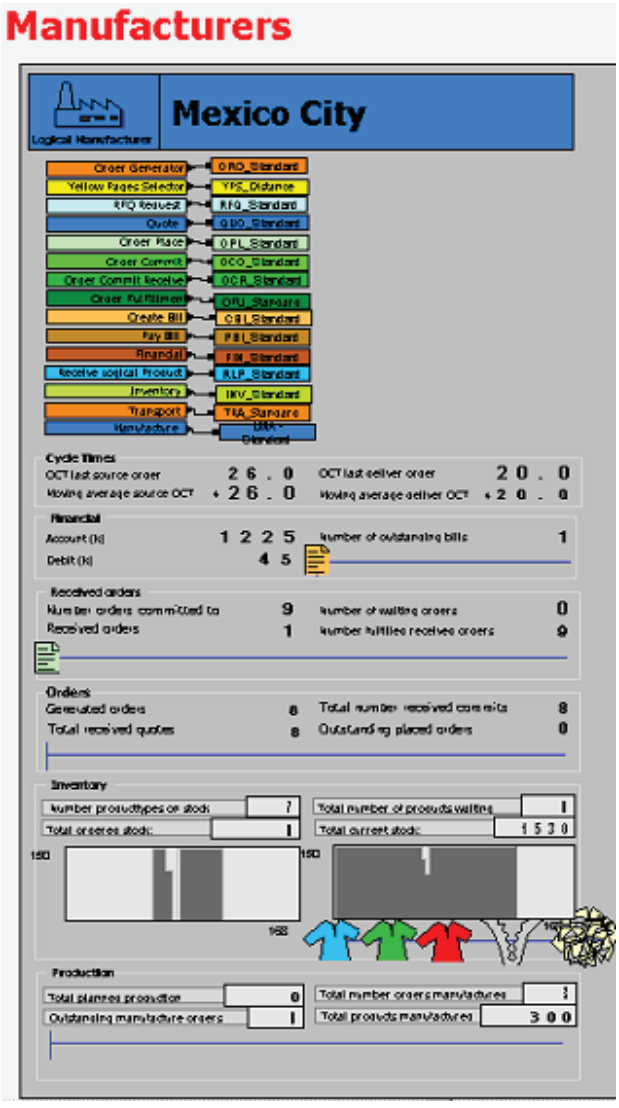


Figure 2: User Output of a SupplyChain Teaching Model

Supply chains consist of a series of companies that contribute to the production of the end product from raw material and add value to it in each step. The network structure consists of an arbitrary number of actors, which play different roles in the supply chain. In order to reach the needed flexibility, we chose for an object-oriented set-up of the supply chain library. To model the supply chain, an object model has been developed that can capture the supply chain generically. The object model focuses on the operational level; the actors within the supply chain, their roles, and the flows present between supply chain partners. Previous attempts have been made to model supply chains, but to capture the supply chain with its complexity and dynamics, the object model needs to consider the entire supply chain. This model is constructed with the goal to give a thorough description of supply chains, and secondly to form a framework for the development of a simulation library. With this simulation library, an arbitrary supply chain can be modeled and analyzed dynamically. Visualization can be put on top of the simulation to provide insight in the supply chain. The object model is made to define all relevant things in a supply chain by recognizing them as objects. All objects have a certain behavior, state and identity (Zobrist & Leonard 1997). Objects can interact using prescribed methods, and their behavior is dependent on their internal state, which can change. This state is described by the present data, which is stored in an object's attributes.

2.1 Requirements

The object model will be used as a framework to make a simulation library that will be used to specify different supply chain models. The products for which the simulation library is developed are analysis models, teaching models and models that will be used as the engine in supply chain portals. The object model should be suitable to describe any given supply chain for teaching models, analysis models, or decision support models.

Models for teaching purposes have different demands to the representation of supply chains than a real-life simulation model as used in the portal model and analysis models. For real-life situations and decision support, a realistic model is most important. This can imply that a company that wants to get insight in the working of its chain can visualize the flows of information and products in its supply chain The simulation of real-life supply chains will need real-life data, which is often difficult to obtain.

In a teaching environment more emphasis is placed on the outcome of the model. Concepts to be taught need to be clear to students. The visualization must emphasize the findings from their books. There is no need for a time-consuming search for real-life data. The network structure of the chain visualized however, can be extracted from real-life. This means that the object model must provide a generic model, which is suitable for both purposes.

The requirements to the object model are separated in different fields: requirements to the translation into a simulation library and requirements for the modeling of a variety of supply chains for analysis and for teaching purposes (Corver 2001).

2.1.1 Requirements for Modeling and Simulation

R1. Clear Description. To be able to make a one-to-one translation of the object model into a simulation library, the

object model must provide a clear description. This description needs the right amount of detail in order to be able to capture the supply chain dynamics and to provide a picture of the supply chain but without giving an overload of data.

R2. Right Level of Detail. A supply chain is difficult to comprehend because of its complexity. With a structured model it is possible to get better insight in the supply chain. The dynamics in the supply chain networks and relationships is a major part of this complexity. The object model must be able to capture this dynamics from a supply chain. In order to do this, it's necessary to understand the flows of information and products between actors. To get a controllable model, choices have to be made about the amount of detail. With a limited amount of detail the functioning of a supply chain must be described. There should also be enough detail to provide a description of a variety of supply chains. If the description is too detailed, the translation in a simulation library will be difficult and without enough details the modeling of supply chains in not possible.

R3. Clear Structure. The object model must give a description that provides the builder of the simulation library with a clear structure. This structure allows the one-to-one translation of the object model into a simulation library.

2.1.2 Requirements for Teaching and Training

R4. Potential Link to Real Supply Chain Data. Simulation models can be fed with real-life information in order to be able to display the current situation. To achieve this, a link will be made to the information systems and databases of the individual actors. In this connection also the desired data required in the supply chain model needs to be extracted. The object model should be prepared to handle these information systems. The object model must clearly state which information must be extracted from the information system. This information will consist of the objects, attributes and methods stated in the object model.

In a situation with dummy data, the simulation model must be able to function without extracting data from a real-life information system. For the object model there should not be a difference between the situation with or without a real-life information system.

R5. Calculate Supply Chain Performance Indicators. The object model must enable gathering of enough data to provide statistics. The object model must define the input data that is necessary to calculate performance measures. Using this data, model users can compare different scenarios, but also compare their organization to other companies in the same line of business. These performance measures should be taken over the entire supply chain but a drill down function will allow focusing on smaller sections of the chain. For displaying information to the users, choices should be made which data and how much data should be visible. Too much information will distract the user from the core of the problem. Insufficient information makes it impossible to make well-balanced decisions. The object model, however, should be able to provide all the desired information for different situations.

R6. Flexible Scope. The scope of the supply chain object model is to consider the entire supply chain if needed. When studying supply chain literature, one finds that some authors only look at the focal company or the focal company with some external links to first tier suppliers and customers. The object model must have the possibility for modeling the entire chain, but if desired, just a small part can be singled out. The reason for taking the entire chain into account is because the effect of events can go far beyond the first tier customer or supplier. For instance when the whole chain is taken into consideration the effects become clear of a transport strike on the delivery of end products to end-customers.

R7. Multi-Actor Visualization and Performance Indicators. In order to visualize the whole chain, different actors must be recognized. If only part of the supply chain is visualized, the focal actor and its suppliers and customers are described. Now it's important to look beyond these actors. In the supply chain literature the most commonly mentioned actors are: suppliers, manufacturers, distribution centers, retailers and transporters. These actors all have impact on the performance of the supply chain. The object model must be able to store the information about these actors.

R8. Represent Relevant Objects. To provide insight in the supply chain only relevant objects in the physical supply chain need to be visible. This includes the network structure and the different flows. Even though the object model should be prepared for real-life information, it is not desirable to model the actual information systems. The main reason is that for the visualization, the modeling of information systems will add nothing for the user of the display/control panel. The origin and the destination of the information flow are of interest, not the manner in which data is received. Training how to use an ERP system is usually offered as a separate course. The internal functioning of actors, including the processing of information, is also not relevant for the visualization.

R9. Allow Capturing of a Variety of Supply Chains. Most supply chains look like a tree where the branches and roots are the network of suppliers and customers. The focal company will deal with a number of suppliers and customers. Each of these links will have it's own suppliers and customers. The network structures will differ per product

or actor. It should be prevented that every industry needs its own template. A pharmaceutical product will require a different supply chain then for instance a car. Both supply chains should fit into the supply chain object model. The objects should be described on a generic and high level. Making the objects generic, will enable their use for different models. The aspects of a supply chain that can be recognized in every supply chain need to be described, not the specific instances for one supply chain.

R10. Extensibility. For specific situations, a supply chain might have to be described in more detail, so the object model must allow for extensions. An example is to allow for a more detailed description of actors. An example is that some internal processes in one version of the object model have not been taken into account. The black box, as the internal processes are now implemented, can be changed into a description with more detail but this has not been the primary goal of the object model. In preparation to extensions, several objects are present in the object model as placeholders for future extensions, such as advanced transportation modeling.

2.2 The Architecture and Object Model

The architecture has been set-up as a layered architecture, which captures the essence of supply chain operations and supply chain management. The first layer is formed by a set of definitions of *actors* that are able to *communicate* through *messages*.

The second layer is formed by a set of supply-chain specific actor definitions, such as `SupplyChainActor`, `Trader`, `Retailer`, `Manufacturer`, `Customer`, and `Supplier`. These actors exchange supply-chain specific messages such as `RequestForQuote`, `Quote`, `Order`, `Bill`, and `Payment`. When orders are accepted, the actors also exchange instances of a `Product` in the form of a `Shipment`. The business logic of the actors is implemented in a very flexible way. When receiving messages of a certain type, these messages are handled by instances of a message-type specific `Handler`. Each handler contains a parameterizable set of business logic that in turn can send out replies or other buisiness messages through the communication logic that is present in the actor layer. Handlers can be added or removed during the simulation run, which allows for flexible business logic. An example of a handler for handling an incoming `Bill` is the `BillTimeOutHandler`, which sends out an extra bill with a fine when the original bill has not been paid on time.

In addition, the second layer of the supply chain libraries contains reference implementations for financial management, demand generation, inventory management, production, and transportation.

After carrying out several projects with the library, it turned out that there was one very important component missing, which is not normally present in supply chain simulation libraries; the database of work on hand, promises, and history of information exchange. We dubbed this object the `ContentStore`. This object is close to the data that one could retrieve from an ERP system in a supply chain organization. The supply chain library we developed, keeps all messages that have been sent or received in the course of the simulation run in the `ContentStore`, thereby allowing the business logic to retrieve historical and status information for making decisions. For large simulations, an extension of the `ContentStore` has been defined, the so-called `LeanContentStore`, which deletes the references to messages after full delivery and payment of bills. If we would not do this for large simulations, memory would fill up fast. A possible extension that we have not yet developed would be the `DatabaseContentStore`, which would store the historical information on all exchanges with other actors in an actual database. After implementing the concept of a `ContentStore`, the implementation of business logic became much easier, because the references to all relevant information that is needed to handle incoming requests or orders, are present.

The third layer is the layer that extends the supply chain library for a certain *mode of use*. One mode of use is educational use by individual students. Individual users work with the supply chain models to be confronted with a certain aspect in supply chain management. Examples are models for experiencing the bullwhip effect, models for make-to-order supply chains versus make-to-stock supply chains, and models for comparing a normal strategy to a postponement strategy. These models can be used in pure *demonstration mode*, i.e. a teacher shows the models to a group, or in an *interactive training mode*, i.e. a student or a student team uses the model to answer some questions by carrying out a number of experiments involving different parameter settings. The layered architecture of the supply chain library is such that the individual training models should not lead to extra object classes. For all organizations (actors) in the model, several handlers are already available. If extra business logic is needed for the interactive mode (linking the handler to a user interface), the new handlers can easily replace the existing handlers in the use specific layer.

Another mode of use is the *gaming mode of use*, where some actors are controlled by the computer, and other actors are controlled by the students. Actually, actors are usually hybrid, where the students control some behavior of the organization while other business logic is carried out automatically by the algorithms in the handlers.

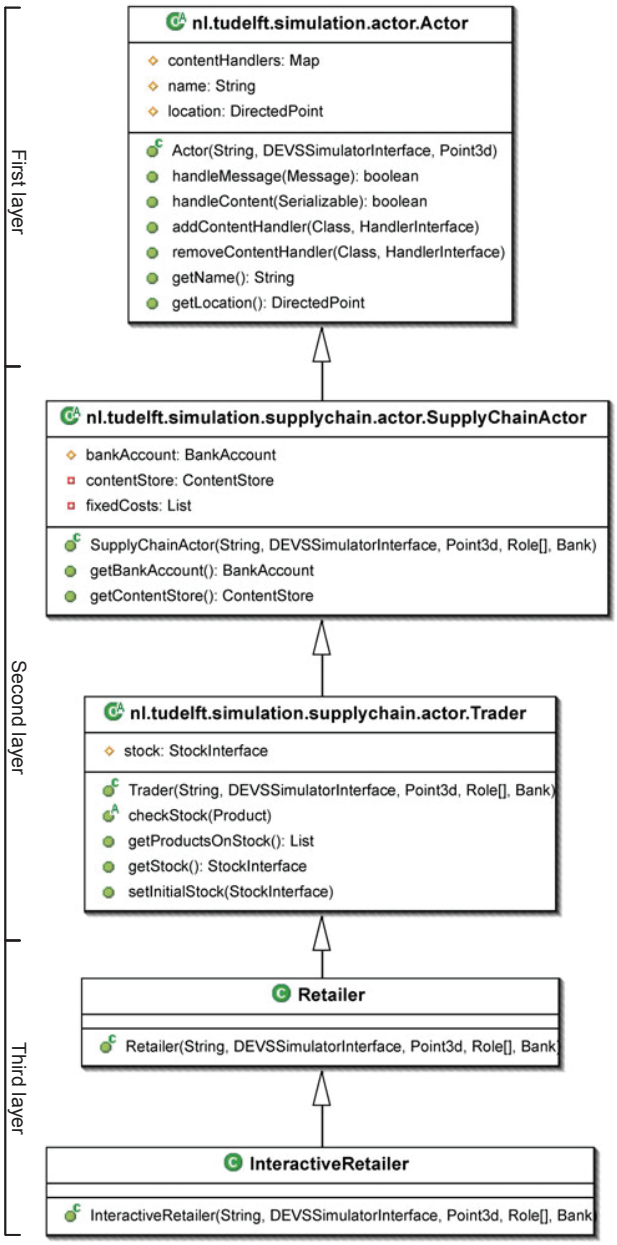


Figure 3: Example Hierarchy for the Actor Objects

It was an interesting question whether an interactive gaming mode of use could be easily designed on top of the supply chain layer. Because of the clear triggers of the business logic by messages, and the bundling of the business logic in the handlers, interactive and hybrid behavior was very easy to incorporate in the architecture. For those business messages where the students would have to make decisions themselves, the business logic is such that the relevant information for making the decision is first retrieved, for instance from the `ContentStore`. Instead of making calculations, making decisions, and send out new business messages as is the case in the fully automated handlers, the information is readied for display on the screen in the case of the interactive training mode. The player can react to the information, submit the answer in terms of the same variables that are calculated in the case of the automated handlers (e.g. price, order size, promised delivery dates), and based on that information, the `Actor` logic sends out the appropriate messages to one or more other actors, be it fully automated, hybrid, or fully human controlled actors. For an interactive mode of use, the implementation of the third layer would involve some specific handlers that are able to retrieve and prepare the information for the users, and process the fields that are based on the choices of the students, but again there is no need to change the core objects for the mode of use.

Concluding we can say that by staying close to a multi-actor, message-based supply chain architecture, we are able to use this architecture for multiple modes of use in training, without having to make time-consuming extensions to the existing architecture.

3 A REFERENCE IMPLEMENTATION IN JAVA

In order to test the architecture, an implementation was made in Java, on top of the Distributed Simulation Object Library DSOL, see Jacobs et al. (2002) for more information on DSOL. One of the main reasons for choosing for DSOL is that it is ready for distributed, web-based simulation, which is especially important for the individual training mode of use, e.g. portaled access to simulation models, and for the interactive gaming mode of use, where different players will have to interact with the model of the supply chain, or where the model of the supply chain is really distributed over the computers of the players, close to the structure of a real supply chain.

The three layer structure was implemented in three different projects in Java, using the DSOL environment as the simulation kernel. One project, called 'actor' is the implementation of the actor and messaging functionality. In Figure 3, an illustration is given to demonstrate how the `Actor` class is extended into an `InteractiveRetailer` class using the architectural choices as described in the previous section. The abstract `Actor` class is centered around communication (with `Message` objects), where each `Message` object has a certain `Content`. One is able to add or remove Handlers to or from `Actor` objects (a handler is defined by an object that implements the `HandlerInterface`) through the `addContentHandler` and `RemoveContentHandler` methods. Actors have a name and a geographic location (defined by a `LocationInterface`), so they are named, locatable objects that are able to communicate.

On the second layer of the supply chain libraries, the `Actor` class is extended into a `SupplyChainActor` class. The `SupplyChainActor` is an actor with a `BankAccount`.

and a `ContentStore`, the ERP-like data structure as explained in the previous section. The `BankAccount` illustrates that one of the core elements in trading is the exchange of money related to the exchange of products or services. The `SupplyChainActor` is still an abstract class, as we don't want users to directly instantiate this class, as the `SupplyChainActor` itself is still too abstract to carry any meaning in a supply chain teaching implementation.

Still in the second layer, the `SupplyChainActor` class is instantiated into a `Trader` class. A `Trader` is a `SupplyChainActor` that really owns products at a certain moment in time. In other words, the `Trader` has an inventory, represented by the `StockInterface`.

In the second layer, we also find recognizable reference implementations of actors that can really be recognized in the supply chain, for example the `Retailer`, `Manufacturer`, `Customer`, and `Supplier`. In the implementation, we could really see that the `Retailer`, as shown in Figure 3, did not need any extra attributes or methods to make it a retailer. The characterization of the retailer is done by giving it a specific set of handlers, and by leaving open another set of handlers to give the retailer its specific behavior.

```
public boolean handleContent(final Serializable
content)
{
  if (super.handleContent(content))
  {
    Bill bill = (Bill) content;
    try
    {
      bill.getSender().getSimulator().
        scheduleEvent(
        bill.getFinalPaymentDate()
        - bill.getSender().getSimulatorTime()
        + this.maximumTimeOut, this, this,
        "checkPayment", new Object[]{bill});
    } catch (Exception exception)
    {
      Logger.severe(this, "handleContent",
        exception);
    }
    return true;
  }
  return false;
}
```

Figure 4: Example of an Implementation of a Handler

Figure 4 illustrates how the core method of a handler can be implemented. In Figure 4, we see the `handleContent` method of the `BillTimeOutHandler`. This handler reacts to an incoming `Bill`. It calls its `super.handleContent()` method, which actually takes care of trying to pay the bill. This message in addition checks on the final payment date plus a timeout whether the bill has *actually* been paid, and schedules an event at the appropriate time to check this. At the timeout date, special measures can be taken for paying the bill anyhow, when the finances were apparently not such that the bill could be paid. One could think of business logic for borrowing money from the bank, or taking the money from a savings account to pay the bill on the timeout date to avoid a fine for paying late.

The reference implementation of the `InteractiveRetailer` in Figure 3, also lacks extra attributes and methods. The specific constructor of this class adds all kinds of interactive handlers to the `InteractiveRetailer` in addition to automated handlers for other business messages.

Please note that the implementations of the `Retailer` and the `InteractiveRetailer` are example, reference implementations. Using these classes as an example, any programmer can implement different types of supply chain actors with a different business logic in a very short amount of time.

During implementation we found out that in addition to the external messages that are sent by the other actors, we also need internal messages that are for instance the result of inventory levels dropping below a certain, pre-set value. An example is the `InternalDemand` message that the actor sends to itself when it might have to start ordering new products, which is handled by a certain `InternalDemandHandler` handler class. We also observed that in interactive gaming, other information was needed for the human players than the information present in the standard `ContentStore`. Therefore, a `GameActorContentStore` was created that stores the additional information needed for human players. As this store implements the same interface by extending the existing `ContentStore`, no changes in any other part of the code had to be made to accommodate this extended ERP system for the human players in the interactive game.

4 EXPERIENCES WITH THE SUPPLY CHAIN LIBRARY

We tested the supply chain library for the two modes of use as indicated in section 3, the demonstration / individual training mode and the interactive gaming mode of use.

For the individual use mode, three models were designed and built; exploring make-to-stock versus make-to-order strategies, comparison of normal supply chains with postponement supply chains, and a demonstration of the bullwhip effect as a result of a sales promotion. All three models could be instantiated in the DSOL-based Java library by setting the parameters for the reference actor implementations, such as the `Retailer`, `Manufacturer`, `Customer`, and `Supplier`, and by allocating the appropriate handlers, again with a number of parameters. No structural addition of object classes was needed to extend the supply chain library to specific individual training assignments.

For the interactive gaming mode, the so-called 'Distributor Game' was developed, which places students in the role of the logistics manager of a distribution center, and shows all the operational decisions that have to be taken to manage the distribution firm in terms of sales, purchases, inventory management, and financial manage-

ment. The game was implemented by instantiating the interactive reference implementations of the mode-of-use layer for the interactive players, and the standard implementations from the supply chain layer for the fully automated actors. Of course, the game implementation had to be extended with many other services that were specific for the gaming mode of use and for the game itself, but these did not lead to severe changes in the architecture in any of the three layers. The 'Distributor Game' was actually implemented and successfully used in a class of 32 MBA students in Spring 2005 at the Supply Chain Management Center at the R.H. Smith School of Business of the University of Maryland. More information about the game can be found on <http://www.gscg.org>.

5 DISCUSSION AND CONCLUSIONS

When matching the implementation with the requirements that were posed in section 2.1, we can see that we were able to meet most of the requirements. The supply chain library offers a clear description of supply chain concepts (R1) at the right level of detail (R2), which was shown in the interactive Distributor Game, where it was very easy to map real-world organizations on the supply chain actors as defined in the library, and to map their business logic on the developed handlers (R8). In that sense, the structure is also very clear (R3). Input (R4) and output (R5) could be easily linked to the supply chain models through the open Java-based structure of the DSOL simulation library. In addition, the `ContentStore` object, which acted as a kind of mini ERP-system, provided us with past, present and future information about the organization for which we wanted to calculate output statistics and visualization (R7). The scope of the models is flexible (R6), as there is no rigid structure with which modelers have to comply when building their supply chain models. The whole range from small supply chains with two or three actors to thousands of interacting organizations in a global network is supported with this architecture (R9). There is no lock-in on certain types of organizations, or types of products in this architecture. All classes are extensible, and interfaces are used wherever appropriate to allow programmers to extend or adapt the concepts for their specific supply chain needs (R10). When reviewing the two projects in the two different modes of use (individual training and interactive gaming) we can say that we were truly able to extend an object-oriented supply chain simulation library into a training and gaming library with all the flexibility that is needed for application in multiple contexts.

The implementation in three layers provided us with a very flexible and extendible framework for supply chain modeling with several modes of use. The advantage of the architecture chosen is that the core supply chain classes do not need to be changed for different modes of use. The separation between communicating actors on the first layer, supply chain specific actor extensions and behavioral representations on the second level, and use of specific classes in the third layer resulted in an intuitive and easy to maintain set of supply chain libraries for multiple modes of use.

ACKNOWLEDGMENTS

The authors acknowledge support from the Supply Chain Management Center at the R.H. Smith School of Business of the University of Maryland, in particular prof. Sandor Boyson and prof. Thomas Corsi.

REFERENCES

Archibugi, D. J. Howells, and J. Michie. 1999. *Innovation policy in a global economy*, Cambridge, UK: Cambridge University Press.

Bowersox , D.J., D.J. Closs, M.B. Cooper. 2002. *Supply chain logistics management*. New York: McGraw-Hill.

Boyson, S., T.M. Corsi, M.E. Dresner, and L.H. Harrington. 1999. *Logistics and the extended enterprise*. New York: John Wiley.

Boyson, S., L.H. Harrington, T.M. Corsi. 2004. *In Real-time: Managing the new supply chain*. Greenwood Publishers / Praeger.

Chopra, S. and P. Meindl. 2001. *Supply chain management: Strategy, planning and operations*. New York: Prentice-Hall.

Churchman, C.W. 1971. *The design of inquiring systems*, New York: Basic Books.

Corver, A. 2001. Supply chain visualization: Simulation as a means to gain insight in the supply chain. M.Sc. thesis, Delft University of Technology.

Fredenhall, L. and E. Hill. 2001. *Basics of supply chain management*, New York: St. Lucie Press.

Hoek, R. van. 2001. The rediscovery of postponement: a literature review and directions for research", *Journal of Operations Management*, 19(2), pp. 161-184.

Jacobs, P.H.M., Lang, N.A., Verbraeck, A. 2002. D-SOL: A distributed Java based discrete event simulation architecture. In P*roceeding of Winter Simulation Conference*, ed. E. Yucesan, C.-H. Chen, J. L. Snowdon and J. M. Charnes, 793-800, San Diego, California. Available via `http://www.informs-sim.org/wsc02papers/102.pdf` [accessed april, 15, 2005].

Kelton, W.D., R.P. Sadowski, and D.A. Sadowski. 2002. *Simulation with Arena*, 2nd Edition. New York: McGraw-Hill.

Lee, H.L., V. Padmanabhan, S. Whang, 1997. The bullwhip effect in supply chains. *Sloan Management Review*, 38 (3), pp. 93-102.

Poirier, C.C. 1999. *Advanced supply chain management*. San Francisco, CA: Berrett-Koehler Publishers.

Poirier, C.C. and M.J. Bauer. 2000. E-Supply Chain – Using the internet to revolutionize your business. San Francisco, CA: Berrett-Koehler Publishers.

Shapiro, J.F. 2001. *Modeling the supply chain*. Pacific Grove, CA: Duxbury.

Simchi-Levi, D., P. Kaminsky, and E. Simchi-Levi. 2003. *Designing & managing the supply chain*, 2nd Edition. New Yrok: McGraw-Hill.

Sol, H.G. and P.W.G. Keen. 2005. Decision support next generation. (forthcoming).

Waters, D. (Ed.). 1999. *Global logistics and distribution planning*. 3rd Edition. Boca Raton: CRC Press.

Zeigler, B.P., H. Praehoffer, and T. G. Kim. 2000. *Theory of modeling and simulation, second edition: Integrating discrete event and continuous complex dynamic systems*. Elsevier Science, San Diego.

Zobrist, G.W. and J.V. Leonard (Eds.). 1997. *Object oriented simulation: reusability, adaptability, maintainability*. New York: IEEE Press.

AUTHOR BIOGRAPHIES

STIJN_PIETER A. VAN HOUTEN is a Ph.D. student at Delft University of Technology. His research is focused on gaming services for decision support environments.
His e-mail address is
s.p.a.vanhouten@tbm.tudelft.nl
and his webpage is
www.tbm.tudelft.nl/webstaf/stijnh.

ALEXANDER VERBRAECK is chair of the Systems Engineering Group of the Faculty of Technology, Policy and Management of Delft University of Technology, and a part-time full professor in supply chain management at the R.H. Smith School of Business of the University of Maryland. He is a specialist in discrete event simulation for real-time control of complex transportation systems and for modeling business systems. His current research focus is on development of generic libraries of object oriented simulation building blocks in C++ and Java.
His e-mail address is
a.verbraeck@tbm.tudelft.nl,
and his web page is
www.tbm.tudelft.nl/webstaf/alexandv.

Proceedings of the 2011 Winter Simulation Conference
S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, eds.

TRANSPARENCY, CONSISTENCY AND MODULARITY OF STRATEGIC REASONING: AN AGENT ARCHITECTURE FOR INTERACTIVE BUSINESS SIMULATIONS

Rick van Krevelen
Martijn Warnier
Frances Brazier

Systems Engineering Group
Delft University of Technology
P. O. Box 5015, 2600 GA
Delft, THE NETHERLANDS

Alexander Verbraeck

Systems Engineering Group
Delft University of Technology
and
R. H. Smith School of Business
University of Maryland

Thomas Corsi

R. H. Smith School of Business
University of Maryland
3359 Van Munching Hall
College Park, MD 20742, USA

ABSTRACT

Interactive business simulations are widely used to explore and compare business strategies from both practice and theory. In many business simulations however, educators and researchers lack support in observing how the simulated actors operationalize their strategies, in validating whether operations have been aligned with the strategy, and also in (re)configuring available player and opponent strategies based on new theoretical or practical insights. This paper specifies requirements for a novel business simulation architecture that facilitates transparency, consistency and modularity of strategic decision making by simulated actors in interactive business simulations. A system architecture is proposed that integrates three components: an extensible agent middleware, a distributed simulation engine and a modular reasoning framework. How the architecture fulfills the three requirements of strategic reasoning transparency, consistency and modularity is illustrated in a use case of a business simulation game for supply chain management education.

1 INTRODUCTION

Games and simulations have long been applied as a tool for practice and training, from the ancient war games to today's complex (online) game environments. Since business games emerged early in the 20th century they followed the automation trend from paper to mainframes to home computers and web based simulation (Faria et al. 2009). Business simulations and games today educate management students in a variety of business strategies and tactics covering areas such as sales, finances and marketing. Typical advantages of simulation apply here, including relatively low cost training environments, no danger to participants, and avoiding ethical issues when things go wrong (Geurts et al. 2007, Pasin and Giroux 2011). Educators apply simulation games as a constructivist learning approach for their experiential or problem-based learning pedagogy (Lainema 2009, Thavikulwat and Pillutla 2010). Management research increasingly validates economic and computational models with interactive simulation to study human behavior in a complex system context. However, besides their advantages some of their challenges are now becoming clear.

978-1-4577-2109-0/11/$26.00 ©2011 IEEE

One challenge of business simulations is their (lack of) transparency in linking business strategies and operations. Business games often claim to provide insight into business strategy and "prepare managers for the challenges they will face in the business arena" (van Houten 2007, p. 1), yet the simulated actors typically interact at the level of business operations rather than strategy. This means that participants in business simulations have to translate their envisioned strategies into whichever operations they see fit, possibly mediated by the usability of user interfaces or the overall time pressure.

Another challenge related to transparent strategy operationalization, is the lack of a disciplined relation between the chosen strategy and performed actions. Due to this opacity, strategic-operational consistency is difficult to verify in many business simulations, both interactive and not. For instance in the domain of Supply Chain Management, interactive simulations like the *Global Supply Chain Game* (GSCG) suite by Corsi et al. (2006) or the *RealGame* by Lainema and Hilmola (2005), although successful in engaging players with supply complexities, offer no clear means to verify that the selected strategy frames, guides, or constrains the player's operational decisions. The player is free to make operational decisions that may or may not be consistent with the selected strategy and consistency may suffer. A similar challenge is found in non-interactive supply chain simulations for research, such as the *Trading Agent Competition for Supply Chain Management* (Sadeh et al. 2003) and many agent-based computational economics (ACE) experiments. Strategies are often described at a conceptual level with little support for communication, standardization or verification among researchers of their agent-based implementations.
This lack of discipline confounds the direct linkage between the selected strategy and its impact on player performance in the simulation, as well as on game facilitation. Since players' operational decisions are not constrained by their selected strategies, educators, as game facilitators, are able to provide players with little support in understanding the impact of selected strategy on performance. Such guidance or scaffolding is critical for learning outcomes in the simulation-based exploratory educational setting (Bekebrede et al. 2005, Kirschner et al. 2006). The GSCG suite (Corsi et al. 2006) for instance does not support educators in ensuring that players actually follow the strategies they think or claim to follow. Without consistency between claimed strategies and actual performed operations, educators can not provide valid or appropriate feedback which make business games the effective learning tools they are claimed to be.

A third weakness in business simulations is the possibility to (re)design strategies composed of alternative, innovative tactics or policies. Some business simulations allow parametrization to provide some configurability of strategic reasoning, for instance van Houten and Verbraeck (2007) in the GSCG suite. However restructuring or completely replacing the automated actor strategies is typically not an option. Due to monolithic designs, business games often provide facilitators with a fixed reusable business rule-base and limited numbers of clients able to interact within the same simulation. Moreover, given the little regard for future maintenance and model redesign, business game development remains costly (Summers 2004, Tan et al. 2010). Modularity of strategic decision making may provide an answer, yet a robust and extensible architecture for large-scale business strategy simulation seems to be lacking. Considering the above issues, this paper investigates the question:

> How to design a system that facilitates transparency, consistency and modularity of strategic reasoning by simulated actors in interactive business simulations?

The next sections are organized as follows. First, to arrive at requirements for a new system architecture for interactive business simulations, the observed shortcomings and related work is discussed. Following this section, a system architecture design is described that fulfills these requirements. The architecture's supporting role in the domain of supply chain management and education is then illustrated in a use case. The paper concludes with a discussion on some opportunities and challenges that lie ahead for interactive business simulation architecture where strategies are operationalized transparently, consistently and extensibly.

2 SYSTEM REQUIREMENTS

Business simulations and games interactively simulate business environments with multiple strategic actors (corporations, banks, markets, consumers, etc.) in domains ranging from logistics operations to strategic management. In business games, players decide the actions of their designated actors so as to experience the effects of their envisioned strategies in competitive and dynamic business scenarios. While useful for engaging in business strategy, in this section some challenges are explored that lead to requirements for a novel system architecture for business simulation and games.

2.1 Transparent Strategy Operationalization

One primary concern in this paper is the apparent lack of transparency of how business strategies are operationalized within existing business simulations and games. Cornerstone aspects of business strategy such as sales, purchasing, transportation and logistics, have been researched extensively. However in capstone business simulations where the simulated overall business strategy incorporates a combination of these domain-specific policies, a transparent mapping between strategies and operations is commonly missing. Pasin and Giroux (2011) note that in many business games the facilitators must be well-informed to interpret the behavioral rules in order to provide appropriate feedback to individual player teams.

Comparing different mental models among strategic decision makers, Gary and Wood (2010) show that decision makers with more accurate mental models of the fit between different strategic options and the business environment, achieve higher performance outcomes (Gary and Wood 2010). It is therefore important that business simulation users (players and educators) observe valid causal relations that show how their simulated actors' operations actually led to the overall strategic performance. Yet there does not seem to be any capstone business simulation game that shows precisely how business (sub)strategies, concerning for instance the sales and stock policies, were translated into operations, such as sending orders and receiving shipments, and in turn led to the overall business performance.

Furthermore, empirical results in business game research suggest that performance and learning do not covary (Gosen and Washbush 2004). Pasin and Giroux (2011) thus started to measure learning not by evaluating simulation performance but rather analyzed each of the students' decisions to identify specific strategically 'wrong' decisions or 'mistakes' that their logistics management simulation game was intended to reduce. Measuring learning by tracking the number of strategic mistakes is new in management simulation games (Pasin and Giroux 2011) and requires transparency of these inconsistencies. There is therefore a clear need for support of transparently measuring strategic reasoning in business simulation platforms.

Requirement 1 *The architecture must support business strategies to be operationalized transparently.*

2.2 Consistent Strategy Operationalization

Transparency alone however does not guarantee a disciplined, consistent strategy operationalization. An important argument for supporting consistency comes from the education domains where business simulations and games are widely used as an educational tool (Wolfe 1993, Faria and Wellington 2004). Business games offer a platform for exploratory learning with little or no guidance, but come with a danger. Several education researchers critique the minimally guided learning approach based on evidence that "it may have negative results when students acquire misconceptions or incomplete or disorganized knowledge" (Kirschner et al. 2006, p. 84). Furthermore, there is hardly any empirical evidence of the cognitive efficacy of business games beyond the self-reports from participants (Anderson and Lawton 2009). Pasin and Giroux (2011) only found four studies that actually measure cognitive-based outcomes for management simulation games, and with mixed results (Pasin and Giroux 2011, p. 1247). Defending business games, Bekebrede et al. (2005) conclude that they "work best when they are embedded into a broader research, learning or intervention process in which a number of complementary methods and activities are used" (Bekebrede et al. 2005, p. 6).

Business games thus offer rich, dynamic experiences but players still require much guidance from experts in order to ensure the pedagogic value of game sessions and fill in the gap between applied theoretic strategies and in-game performance. Following constructivist principles from education theory, Lainema (2009) argues that "the black-box model should be opened. ...This would provide better contexts and assistance to aid the individual in making sense of the environment as it is encountered." (Lainema 2009, p. 65). This need for learner guidance in business games presents a requirement for supporting consistency of the player's actions with respect to their chosen strategies.

Requirement 2 *The architecture must support business strategies to be operationalized consistently.*

2.3 Modular Strategy Operationalization

To explore and evaluate business strategies for new business scenarios, the strategy operationalization should also be reconfigurable and extensible at a tactical level. In order to enhance or extend behaviors of simulated actors to reflect existing or new practices, the system users must be given the freedom to change the strategic behaviors and business scenarios completely, rather than being constrained to its original concepts and rule base. A modular approach to strategic decision-making offers several advantages. One advantage of modularity is the strategic flexibility it could offer to the user during a simulation run. Another advantage of modular strategic decision-making is the possibility to add unforeseen phenomena outside the globally shared simulated world, for instance trader cartels that are negotiated peripherally. A third advantage is the opportunity to delegate the simulated actor's tasks or roles such as sales, purchasing or storage to a team of agents and/or human specialists of arbitrary size. Although Wang et al. (2006) also propose a modular and open reasoning architecture of internal micro-agents, it has some disadvantages: its internal micro-agent architecture is restricted to hierarchical cooperation topologies, the micro-agents are conceptually limited to just representing (partial) reasoning solutions, and the architecture has not been linked to interactive and/or business simulations.

Similarly in agent-based computational economics (ACE) a framework for modular strategic reasoning is not common. For instance, the *Trading Agent Competition for Supply Chain Management* (TAC/SCM) by Sadeh et al. (2003) has been used over the last decade to experiment with and educate about production scheduling, purchasing and sales strategies within a fixed-size computer market scenario. Players enter custom programmed agents into a competition and study their behaviors and performance against various opponents before improving them, but no standard approach is taken other than the Java implementation language. TAC/SCM currently does not support nor allow human interaction, although the TAC community continues to extend the platform to host new setups including agent vs. human competitions (Nelson et al. 2009). Besides its monolithic implementation, the underlying agent platform of TAC/SCM is not very robust nor extensible to new scenarios and setups. This inspired Ketter et al. (2010) to implement their own framework when creating their *Power Trading Agent Competition* (PowerTAC) platform, which aims to extend current ACE methods used specifically to study wholesaler strategies in electrical power markets. However, for the purposes of this paper an extensible business strategy simulation architecture is required that is more topology and domain independent.

To reduce development costs, various solutions have been proposed including the GSCG suite enabling parametrization to support a range of business games for multiple educational settings (van Houten and Verbraeck 2007), the reconfigurability of the *IndustryMasters* game available at industrymasters.com, and a plug and play pathway approach by Tan et al. (2010) for operations management games development. Simulation model reuse and extensibility for supporting new technologies, scenarios and ways of thinking will however remain important concerns to the model users as well as the owners. Modularity in the strategic decision making architecture offers flexibility in design, model reuse, and can thus reduce costs.

Requirement 3 *The architecture must support modular strategic decision making.*

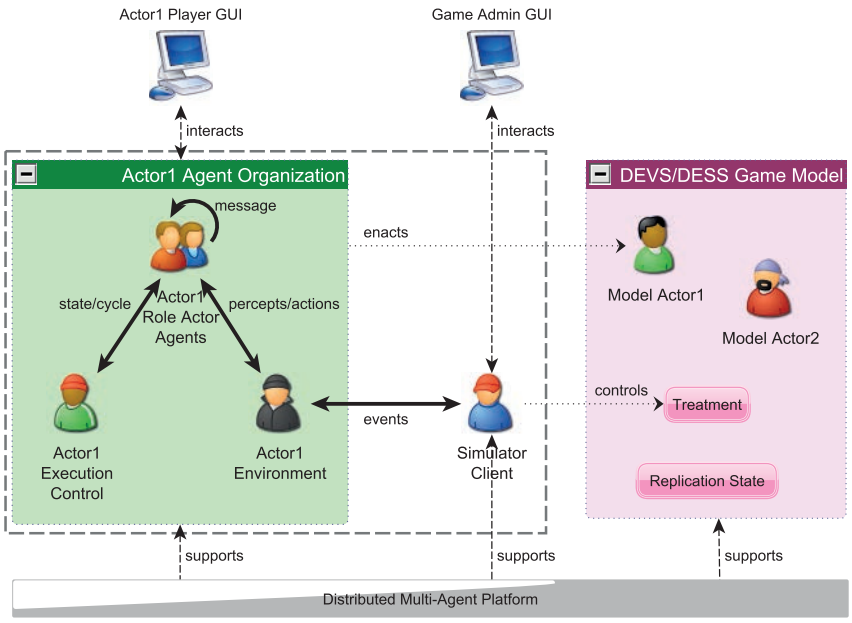


Figure 1: *Modular architecture for distributed interactive business games with local actor reasoning, integrating actor-agent organizations to coordinate a simulated actor's behavior (green area) and a DEVS/DESS game model simulation (purple area) via a distributed multi-agent platform (grey area).*

3 SYSTEM ARCHITECTURE

In order to meet the requirements of transparency, consistency and modularity as specified in the previous section, an architecture for distributed interactive business simulation is proposed that consists of three main components, as depicted in Figure 1. This section briefly discusses how the components separate the concerns of simulation, distribution, and reasoning, as well as their integration within the architecture.

3.1 Components

Successful modularity depends on a clear definition of the interfaces between components. Much in this regard has already been achieved in related research fields of simulation and distributed artificial intelligence in the form of simulation and reasoning formalisms. The concept of a multiple social agents messaging each other while perceiving and acting upon their environment helps to separate concerns for modular reasoning as well as the overall model simulation. The proposed architecture has the following components:

Extensible Agent Middleware Important properties of the agent middleware component, besides being able to support agents, are its robustness, distribution and extensibility. The middleware must be robust to common agent communication and operation failures. Distribution of agents and services across host processors is an essential prerequisite for the intended types of simulation setups. Finally, its extensibility must ensure that the other components can be maintained and modified without causing problems for the distributed setup itself.

Standard Simulator Engine The selected simulator engine must support distributed setups, enable continuous (real-time) simulation, and adhere to standard simulation formalisms specified by Zeigler et al. (2000): the Differential Equations System Specification (DESS) for continuous models, and the Discrete Event System Specification (DEVS) for event-based simulation. These formalisms enable validation of the simulation models on other platforms and simulators in a consistent manner.

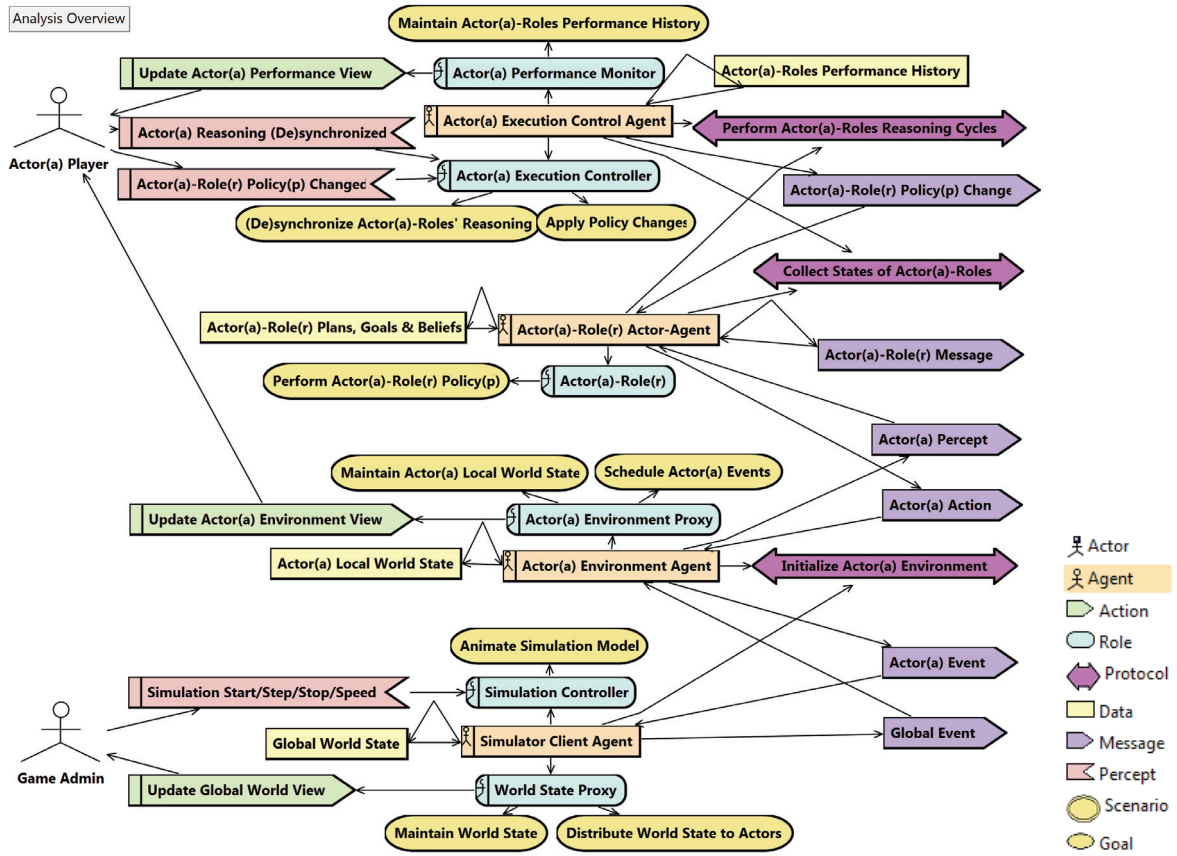


Figure 2: *Multi-agent system infrastructure supporting a distributed business simulation where a simulated actor's reasoning is delegated to (possibly several) actor-roles, each performed by an actor-agent.*

Modular Reasoning Framework The reasoning framework component localizes reasoning of the simulated actors and supports extensible multi-agent systems that implement the AgentSpeak(L) language by Rao (1996). AgentSpeak(L) is a formal language based on theory of mind and speech act theory where agents reason and communicate based on beliefs, desires and intentions (BDI). Formal reasoning enables validation of the localized reasoning in other agent reasoning frameworks. Enabling introspection and execution control, agents performing various simulated actor roles reason transparently and their behaviors or policies are modifiable and followed consistently.

3.2 Component Integration

To integrate all components, an infrastructure is designed that enables multiple agents to fulfill an extensible set of role policies that together form the strategy of a single simulated actor. This architecture assigns each simulated actor to its own separate reasoning agent organization, such that complexity can be shifted from the central simulation model to the peripheral reasoning engines. By abstracting the actor's behavior to a combination of policies rather than the complete set of individual actions, the overall simulation performance remains the same while actor strategies become clear in the simulation model itself. For instance, a trade actor in the global simulation might distinguish and delegate (and inspect) the sell, store, and purchase roles onto multiple coordinated agents, which then reason locally according to the specified role policies and only the resulting operations are sent back to the global simulation.

The integrating infrastructure's functionality and user-interaction is specified in Figure 2 using the Prometheus multi-agent system design methodology developed by Padgham and Winikoff (2002) and extended by Khallouf and Winikoff (2009). Prometheus applies and extends concepts from Agent-UML and the BDI reasoning framework to provide a goal-oriented description of a multi-agent system's functionalities. More information about the Prometheus methodology and design tool can be found at www.cs.rmit.edu.au/agents/prometheus/.

The infrastructure's analysis overview in Figure 2 shows the system interacting (via the user interfaces depicted in Figure 3) with two user types or Actors: the Game Admin and one Actor(a) Player for each simulated strategic actor $a \in A$ who delegates the actor's tactical behaviors $p \in P$ (sales, purchase, inventory, etc.) among their respective actor roles $r \in R_a$. The system itself provides six functionalities or system Roles (cyan) with relevant goals (yellow) which are performed by four types of agents (orange) through system input or Percepts (red) and output or Actions (green). In order to achieve their goals, agents can interact via messages (blue) or message Protocols (purple) and store required data (beige). The infrastructure's roles are as follows:

Simulation Controller Performed by the Simulator Client Agent, this system role adjusts the animation of the central simulation model according to perceived simulation *start*, *step*, *stop*, and *speed* commands from the Game Admin.

World State Proxy Also performed by the Simulator Client Agent, this system role (i) maintains a global world state to help initialize a strategic actor's environment, and (ii) distributes the current World state events to relevant strategic actor's environment.

Actor(a) Environment Proxy Performed by an Environment Agent, this system role (i) maintains a local world state for the respective strategic actor a, translating events into percepts for the delegated actor-roles, (ii) updates Actor(a) Player's environment view accordingly, and (iii) schedules any action taken by the delegated strategic actor-roles.

Actor(a) Execution Controller Performed by an Execution Control Agent, this system role (i) synchronizes (or desynchronizes) the reasoning cycles of the strategic actor's delegate role agents if commands from Actor(a) Player to do so are perceived, and (ii) applies perceived policy changes from Actor(a) Player for delegate roles in their respective Actor Agents.

Actor(a) Performance Monitor Also performed by an Execution Control Agent, this system role (i) maintains the performance history of all the strategic actor's delegated role agents R_a by collecting their current states from the respective Actor Agents, and (ii) updates the Actor(a) Player's performance view for the strategic actor based on its performance history emerging from the delegate agents' selected policy combination.

Actor(a)-Role(r) Performed by an Actor Agent, this system role performs (tactical/operational) policy p for actor role r as delegated by the (strategic) Actor(a) Player.

4 SYSTEM IMPLEMENTATION

To explore the feasibility of the architecture design, an implementation was made with a selection of open-source, Java-based (thus platform-independent) components: the AgentScape agent middleware, the DSOL based GSCG distributed business simulation suite, and the Jason agent reasoning framework.

4.1 AgentScape Agent Middleware

The open-source AgentScape middleware (Brazier et al. 2002, Overeinder and Brazier 2006) is selected for its robustness, scalability and extensibility. The middleware, used in large production environments as well as many research projects, is under constant development and was recently rebuilt to support more extensibility. More information about the AgentScape middleware can be found at www.agentscape.org.

4.2 DSOL and the Global Supply Chain Games Suite

The open-source *Distributed Simulation Object Library* (DSOL) by Jacobs et al. (2002), Jacobs (2005) implements and combines the DEVS and DESS simulation formalisms specified by Zeigler et al. (2000). DSOL replicates Java-based DEVS and/or DESS model simulations based on specified treatment parameters such as random seeds, warm-up time, etc. The *Distributor Game*, part of the *Global Supply Chain Games* (GSCG) suite (van Houten and Jacobs 2004, van Houten et al. 2005) mentioned in Section 1, is implemented with a DSOL-based supply chain training library (Verbraeck and van Houten 2005). More details about simulation with DSOL can be found at simulation.tudelft.nl, and about the GSCG suite at gscg.org.

4.3 Jason Multi-Agent System and Reasoning Framework

Jason is an open-source framework by Bordini et al. (2007) that implements the AgentSpeak(L) beliefs-desires-intentions (BDI) reasoning language by Rao (1996). In the Jason framework, multi-agent systems are composed of an environment, a group of agents, and possibly an execution controller to synchronize the agents' reasoning steps and inspect the agents' current states if so desired. Jason is also compatible with the Prometheus methodology (Padgham and Winikoff 2002, Khallouf and Winikoff 2009). Jason's agent, environment and execution control classes for introspection and policy modification can easily be extended to fit specific needs. More information about the Jason framework can be found at jason.sf.net.

4.4 Jason-DSOL Integration Using AgentScape

A new infrastructure of tier-agents was implemented on the AgentScape middleware, similar to Jason's standard tier-agents implemented for the Jade agent middleware (Bellifemine et al. 2001). Unlike Jade agents, AgentScape agents are completely mobile, enabling Jason agents to reason for the simulated actors in localized setups whilst distributed dynamically across remote clients. Performance of this Java implementation will be evaluated and compared in future work.

5 USE CASE

To illustrate the benefits of the proposed simulation architecture, this paper shows how it could support redesigning the Distributor Game, part of the business games suite by van Houten et al. (2005), which focuses on supply chain (or demand network) logistics. Aimed at educating the players in the field of business strategy, the current game-play in the suite's business games such as the *Distributor Game* typically revolves around operationalizing business strategy. After choosing their business strategies, players are assumed to execute them consistently, performing whichever operations they see fit the chosen strategy. Since the *Distributor Game*'s evaluation by van Houten (2007), this paper's authors confirmed through observation during follow-up sessions that, as students try to operationalize their strategy, it is hard for educators (a) to determine whether students follow their intended strategy consistently, (b) to determine exactly when they possibly changed it, and (c) to modify the challenges (presented for instance by demand and supply dynamics) that currently best fit the entire group's learning process. In short, the Distributor Game does not meet the requirements discussed in Section 2.

To meet with the requirements of transparent, consistent and modular strategic decision-making by the simulated actors, the game can be redesigned with the proposed architecture as follows.

1. To improve transparency, the simulation model is formalized as a DESS-in-DEVS system specification suited for the simulation service. Behavioral rules or policies governing simulated actors are formalized in the Prolog-based Jason reasoning language, ready for validation by experts on respective management theory. The educator is informed of all policy changes the players make during a simulation. This will show how their strategic decisions (and rationales) contributed to their overall performance in the simulation. Educators can now base their feedback on accurate information about how players operationalized their strategy.

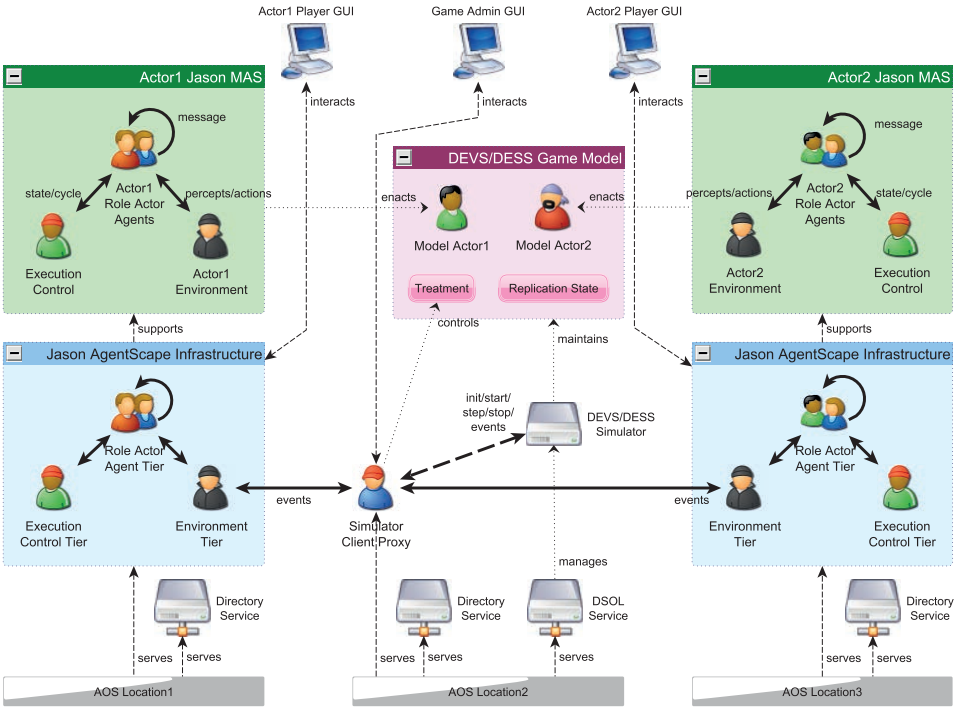


Figure 3: *Implementation of the agent architecture for interactive business games with localized actor reasoning. Agent organizations (green areas) composing a single actor's strategy are implemented in Jason and supported by mobile agents (blue areas). The game model simulation (purple area) is supported separately by a simulation service and client proxy agent. Agents and services run on the agent middleware.*

2. To improve consistency, strategies are delegated and performed (operationalized) by tactical Jason actor-role agents which execute any formalized strategy without mistake (although players' strategic choices for particular tactics may still be inappropriate). In the *Distributor Game* scenario, the demand and supply behaviors are controlled by the educator as game facilitator, whereas the distributor actor behaviors are controlled by the learners through their actor's delegate roles. Strategic mistakes can now be measured and applied by educators to provide appropriate feedback to help players improve and learn to make their policy changes systemic and consistent with their overall business strategy.
3. To improve modularity, actor strategies are decomposed and into (sub)strategies or tactics for relevant fields of expertise (purchase, sales, inventory, etc.). Each tactic is delegated to a separate tactical Jason actor-agent within an agent organization that is making decisions for one particular actor. Adding, removing or replacing these delegate agents reflects a change in overall actor strategy, for instance to mitigate or recover from some supply chain risk event observed by the player.

6 CONCLUSION

This paper proposes an architecture for distributed interactive business simulations to answer the main question of how to design a system that facilitates transparency, consistency and modularity of strategic decision making by simulated actors in interactive business simulations. The system's first requirement, that it supports business strategies to be operationalized transparently, is achieved through a formal reasoning language to specify how tactical policies operationalize parts of actor strategies, for instance purchase, sales, inventory, etc. The system's second requirement, that it supports business strategies to be operationalized consistently, is achieved through the ability to monitor and inspect the executed policies and their interactions.

The third system requirement, that it supports modularity in strategic decision making, is achieved with a multi-agent infrastructure enabling on-the-fly modification and (re)combination of actor (sub)strategies.

In future work the authors aim to verify that the transparency and consistency offered by this architecture actually contributes to and simplifies the validation of the business simulation itself and thus improves the players' mental models. Literature suggests that improving players' mental models will consequently improve their business performance. Furthermore, the modularity of the architecture is expected to contribute to the scalability of the business simulations that implement it. The effects of localization of reasoning on scalability deserve further investigation, as do performance effects of different configurations, for example with other simulation engines, reasoners, or individual actor implementations.

Transparency and consistency have been argued to be absolutely essential for business games to succeed as effective learning tools. Misconceptions due to lack of guidance can be prevented by providing correct insights about what causes might have led to the observed effects. That is, the game should provide a clear picture of how business strategies (and mistakes) actually led to the simulated actors' performance. The presented architecture facilitates precisely these aspects and thus enhances business game effectiveness.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for their comments. Furthermore, financial support from the Delft University of Technology and the Next Generation Infrastructures Foundation is also gratefully acknowledged for this research.

REFERENCES

Anderson, P. H., and L. Lawton. 2009, April. "Business Simulations and Cognitive Learning". *Simulation & Gaming* 40 (2): 193–216.

Bekebrede, G., I. S. Mayer, S. P. A. van Houten, R. T. H. Chin, and A. Verbraeck. 2005. "How serious are serious games? Some lessons from Infra-games". In *Proceedings DiGRA 2005 Conference Changing Views: Worlds in Play*, edited by S. de Castell and J. Jensen, 1–7: DiGRA.

Bellifemine, F., A. Poggi, and G. Rimassa. 2001, 7-9 July. "Developing Multi-agent Systems with JADE". In *ATAL'00: Proceedings 7th Int'l Workshop on Intelligent Agents: Agent Theories, Architectures and Languages*, edited by C. Castelfranchi and Y. Lespérance, Volume 1986 of *LNCS*, 42–47: Springer-Verlag.

Bordini, R. H., J. F. Hübner, and M. Wooldridge. 2007, October. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons.

Brazier, F. M. T., D. G. A. Mobach, B. J. Overeinder, S. van Splunter, M. van Steen, and N. J. E. Wijngaards. 2002, 27-31 May. "AgentScape: Middleware, Resource Management, and Services". In *SANE'02: Proceedings 3rd Int'l Conference on System Administration and Network Engineering*, 403–404.

Corsi, T. M., S. Boyson, A. Verbraeck, S. P. A. van Houten, C. Han, and J. R. MacDonald. 2006. "The Real-Time Global Supply Chain Game: New Educational Tool for Developing Supply Chain Management Professionals". *Transportation Journal* 45 (3): 61–73.

Faria, A. J., D. Hutchinson, W. J. Wellington, and S. Gold. 2009, August. "Developments in Business Gaming: A Review of the Past 40 Years". *Simulation & Gaming* 40 (4): 464–487.

Faria, A. J., and W. J. Wellington. 2004, June. "A Survey of Simulation Game Users, Former-Users, and Never-Users". *Simulation & Gaming* 35 (2): 178–207.

Gary, M. S., and R. E. Wood. 2010, June. "Mental models, decision rules, and performance heterogeneity". *Strategic Management J.* 32 (6): 569–594.

Geurts, J. L., R. D. Duke, and P. A. Vermeulen. 2007, December. "Policy Gaming for Strategy and Change". *Long Range Planning* 40 (6): 535–558.

Gosen, J., and J. Washbush. 2004, June. "A Review of Scholarship on Assessing Experiential Learning Effectiveness". *Simulation & Gaming* 35 (2): 270–293.

Jacobs, P. H. M. 2005, 15 November. *The DSOL simulation suite*. Ph. D. thesis, Delft University of Technology, Delft, The Netherlands.

Jacobs, P. H. M., N. A. Lang, and A. Verbraeck. 2002. "D-SOL: a distributed Java based discrete event simulation architecture". In *Proceedings of the 2002 Winter Simulation Conference*, edited by E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, 793–800. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Ketter, W., J. Collins, and C. Block. 2010, December. "Smart Grid Economics: Policy Guidance through Competitive Simulation". Technical Report ERS-2010-043-LIS, Erasmus Res. Inst. of Mgmt.

Khallouf, J., and M. Winikoff. 2009. "The goal-oriented design of agent systems: A refinement of Prometheus and its evaluation". *Int'l J. of Agent-Oriented Software Engineering* 3 (1): 88–112.

Kirschner, P. A., J. Sweller, and R. E. Clark. 2006, June. "Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching". *Educational Psychologist* 41 (2): 75–86.

Lainema, T. 2009, February. "Perspective Making – Constructivism as a Meaning-Making Structure for Simulation Gaming". *Simulation & Gaming* 40 (1): 48–67.

Lainema, T., and O.-P. Hilmola. 2005. "Learning More, Better and Faster? Computer-Based Simulation and Gaming of Production and Operations". *Int'l J. of Business Performance Management* 7 (1): 34–59.

Nelson, A., D. Nyabuti, J. Collins, W. Ketter, and M. Gini. 2009, 13 July. "Toward Human-Agent Competition in TAC SCM". In *TADA-09: IJCAI Workshop on Trading Agent Design and Analysis*, edited by E. H. Gerding: Association for Trading Agent Research / AAAI.

Overeinder, B. J., and F. M. T. Brazier. 2006. "Scalable Middleware Environment for Agent-Based Internet Applications". In *Applied Parallel Computing*, Volume 3732 of *LNCS*, 675–679. Berlin: Springer-Verlag.

Padgham, L., and M. Winikoff. 2002, 15 July. "Prometheus: A Methodology for Developing Intelligent Agents". In *AOSE 2002: 3rd Int'l Workshop on Agent-Oriented Software Engineering*, edited by F. Giunchiglia, J. Odell, and G. Weiss, Volume 2585 of *LNCS*, 174–185: Springer-Verlag.

Pasin, F., and H. Giroux. 2011, August. "The impact of a simulation game on operations management education". *Computers & Education* 57 (1): 1240–1254.

Rao, A. S. 1996. "AgentSpeak(L): BDI agents speak out in a logical computable language". In *MAAMAW'96: Proceedings 7th Eur. workshop on Modelling Autonomous Agents in a Multi-Agent World: Agents breaking away*, 42–55: Springer-Verlag.

Sadeh, N., R. Arunachalam, J. Eriksson, N. Finne, and S. Janson. 2003. "TAC-03: A Supply-Chain Trading Competition". *AI Magazine* 24 (1): 92–94.

Summers, G. J. 2004, June. "Today's Business Simulation Industry". *Simulation & Gaming* 35 (2): 208–241.

Tan, K. H., Y. K. Tse, and P. L. Chung. 2010. "A plug and play pathway approach for operations management games development". *Computers & Education* 55 (1): 109–117.

Thavikulwat, P., and S. Pillutla. 2010. "A constructivist approach to designing business simulations for strategic management". *Simulation & Gaming* 41 (2): 203–230.

van Houten, S. P. A. 2007, 6 November. *A suite for developing and using business games: supporting supply chain business games in a distributed context*. Ph. D. thesis, Delft University of Technology, Delft, The Netherlands.

van Houten, S. P. A., and P. H. M. Jacobs. 2004. "An architecture for distributed simulation games". In *Proceedings of the 2004 Winter Simulation Conference*, edited by R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, 2081–2086. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

van Houten, S. P. A., and A. Verbraeck. 2007. "Supporting parametrization of business games for multiple educational settings". In *Proceedings of the 2007 Winter Simulation Conference*, edited by S. G.

Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

van Houten, S. P. A., A. Verbraeck, S. Boyson, and T. Corsi. 2005. "Training for today's supply chains: An introduction to the Distributor Game". In *Proceedings of the 2005 Winter Simulation conference*, edited by M. Kuhl, N. Steiger, F. Armstrong, and J. Joines, 2338–2345. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Verbraeck, A., and S. P. A. van Houten. 2005. "From simulation to gaming: an object-oriented supply chain training library". In *Proceedings of the 2005 Winter Simulation Conference*, edited by M. Kuhl, N. Steiger, F. Armstrong, and J. Joines, 2346–2354. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Wang, M., M. Nowostawski, and M. Purvis. 2006, 26 July. "Declarative Agent Programming Support for a FIPA-Compliant Agent Platform". In *ProMAS 2005: 3rd Int'l Workshop on Programming Multi-Agent Systems—Revised and Invited Papers*, edited by R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, Number 3862 in LNCS, 252–266. Utrecht, The Netherlands: Springer-Verlag.

Wolfe, J. 1993, December. "A History of Business Teaching Games in English-Speaking and Post-Socialist Countries: The Origination and Diffusion of a Management Education and Development Technology". *Simulation & Gaming* 24 (4): 446–463.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation—Integrating Discrete Event and Continuous Complex Dynamic Systems*. 2nd ed. San Diego, CA: Academic Press, Inc.

AUTHOR BIOGRAPHIES

RICK VAN KREVELEN is a researcher and PhD candidate at the Systems Engineering Group of the Delft University of Technology, The Netherlands. He holds a Master's in Artificial Intelligence from the VU University Amsterdam. Drs. Van Krevelen researches Agent Technology for Business Games and Simulations. His e-mail address is d.w.f.vankrevelen@tudelft.nl.

MARTIJN WARNIER is an Assistant Professor at the Systems Engineering Group of the Delft University of Technology, The Netherlands. He holds a PhD in Computer Science from the Radboud University Nijmegen. Dr. Warnier's research concerns the design of large scale multi-agent based distributed systems. His e-mail address is m.e.warnier@tudelft.nl.

FRANCES BRAZIER holds the Engineering Systems Foundations Chair at the Delft University of Technology, where she chairs the Systems Engineering Group of the Faculty of Technology, Policy and Management. She is an expert in the design of large scale participatory systems: systems characterized by distribution of governance, autonomy, intelligence, engagement, and communication. Her e-mail address is f.m.brazier@tudelft.nl.

ALEXANDER VERBRAECK is a Full Professor of Systems and Simulation at the Systems Engineering Group of the Delft University of Technology, The Netherlands, and a part-time Research Professor at the R. H. Smith School of Business of the University of Maryland. Professor Verbraeck is a specialist in discrete event simulation for real-time control of complex transportation systems and for modeling business systems. His e-mail address is a.verbraeck@tudelft.nl.

THOMAS CORSI is the Michelle Smith Professor of Logistics at the Robert H. Smith School of Business of the University of Maryland. Professor Corsi is the Director of the Supply Chain Management Center. His e-mail address is tcorsi@rhsmith.umd.edu.

Lecture 11. Multi-Paradigm Simulation

Papers to study for the exam

- H. Vangheluwe, J. de de Lara, P.J. Mosterman. An introduction to multi-paradigm modelling and simulation. In: F. Barros and N. Giambiasi (eds.), Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems), Lisboa, Portugal, 2002. pp. 9-20.
  Retrieved from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.324.3465

- H. Vangheluwe, H. and J. de de Lara. "Meta-models are models too", In: E. Yücesan, C.-H. Chen, J.L. Snowdon, and J.M. Charnes (eds). Proceedings of the 2002 Winter Simulation Conference. New York: ACM Press. 2002. pp. 597-605.
  Retrieved from: http://www.informs-sim.org/wsc16papers/127.pdf

AN INTRODUCTION TO MULTI-PARADIGM MODELLING AND SIMULATION

Hans Vangheluwe

School of Computer Science
McGill University, Montréal
Québec, Canada
hv@cs.mcgill.ca

Juan de Lara

E.T.S. de Informática
Universidad Autonóma de Madrid
Madrid, Spain
Juan.Lara@ii.uam.es

Pieter J. Mosterman

Simulation and Real-Time Technologies
The MathWorks, Inc.
Natick, MA, USA
pieter_j_mosterman@mathworks.com

ABSTRACT

Modelling and simulation are becoming increasingly important enablers in the analysis and design of complex systems. To tackle problems of ever increasing complexity, modelling and simulation research is shifting from simulation techniques to modelling methodology and technology. In this article, the emerging field of Computer Automated Multi-Paradigm Modelling is presented. Multi-paradigm modelling adresses and integrates three orthogonal directions of research:

1. multi-formalism modelling, concerned with the coupling of and transformation between models described in different formalisms,
2. model abstraction, concerned with the relationship between models at different levels of abstraction, and
3. meta-modelling, concerned with the description (models of models) of classes of models, which allows formalism specification.

The article first introduces the general concepts of Modelling and Simulation theory, and explains how rigourous application thereof provides a sound basis for the meaningful exchange and re-use of knowledge about the behaviour of complex systems. The representation of models in diverse formalisms, at different levels of abstraction, and the (behaviour-conserving) transformation between the formalisms is demonstrated.

1 MODELLING AND SIMULATION

At a first glance, it is not easy to characterize modelling and simulation. Certainly, a variety of application domains such as fluid dynamics, energy systems, and logistics management make use of it in one form or another. Depending on the context, modelling and simulation is often seen as a sub-set of Systems Theory, Control Theory, Numerical Analysis, Computer Science, Artificial Intelligence, or Operations Research. Increasingly, modelling and simulation *integrates* all of the above disciplines. As a paradigm, it is a way of representing problems and thinking about them as much as a solution method. The problems cover the analysis and design of complex dynamical systems. In analysis, abstract models are built inductively from observations of a real system. In design, models deductively derived from a priori knowledge are used to build a system, satisfying certain design goals. Often, an iterative combination of analysis and design is needed to solve real problems. Though the focus of modelling and simulation is on the time-varying behaviour of dynamical systems, static systems (such as Entity-Relationship (ER) models, described in the Unified Modeling Language (UML) (Rumbaugh et al., 1999)) are a limit-case. Both physical (obeying conservation and constraint laws) and non-physical (informational, such as software) systems and their interactions are studied.

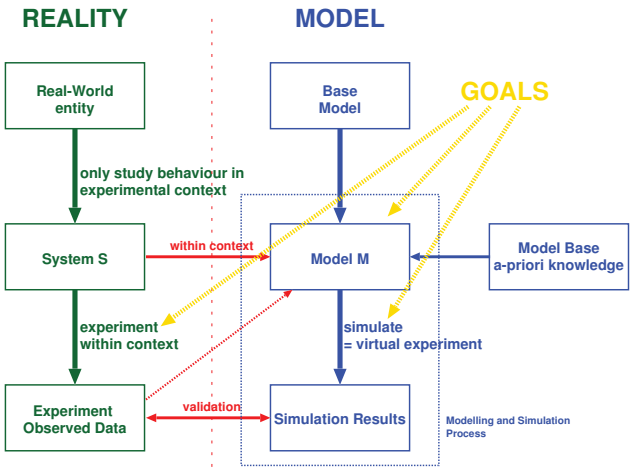


Figure 1: Modelling and Simulation concepts.

1.1 Concepts

Figure 1 presents modelling and simulation concepts as introduced by Zeigler (Zeigler, 1984b; Zeigler et al., 2000).

Real World Entity or object can exhibit widely varying behaviour depending on the context in which it is studied as well as the aspects of its behaviour which are under study.

Base Model is a hypothetical, abstract representation of the object's properties, in particular, its behaviour, which is valid in all possible contexts, and describes all the object's facets.

System is a well defined object in the Real World under specific conditions, only considering specific aspects of its behaviour.

Experimental Frame (EF) describes a limited set of circumstances under which a system (real or model) is to be observed or subjected to experimentation. As such, the Experimental Frame reflects the *objectives* of the experimenter who performs experiments on a real system or, through simulation, on a model.

(Lumped) Model (not to be confused with a lumped parameter model (Cellier, 1991)) is an abstract representation of a system within the context of a given Experimental Frame. Usually, certain properties of the system's structure and/or behaviour are reflected by the model (within a certain range of accuracy).

Experimentation is the act of carrying out an experiment. An experiment may interfere with system operation (influence its input and parameters) or it may not. As such, the experimentation environment may be seen as a system in its own right (which may itself be modelled in a lumped model). Also, experimentation involves observation. Observation yields *measurements*.

Simulation of a lumped model in a certain formalism (such as Petri Net, Differential Algebraic Equations (DAE) or Bond Graph) computes the dynamic input/output behaviour. Simulation may use symbolic as well as numerical techniques. Simulation, which mimics the real-world experiment, can be seen as *virtual experimentation*. Whereas the goal of modelling is to *meaningfully* describe a system, presenting information in an understandable, re-usable way, the aim of simulation is to be *fast and accurate*. Symbolic techniques are often favoured over numerical ones as they allow the generation of classes of solutions rather than just a single one (*e.g.,* $sin(x)$ as a solution to the harmonic equation as opposed to one single approximate trajectory solution). Furthermore, symbolic optimizations have a much larger impact than numerical ones thanks to their global nature. Crucial to the System–Experiment/Model–Virtual Experiment scheme is that there is a *homomorphic* relation between model and system: building a model of a real system and subsequently simulating its behaviour should yield the same results as performing a real experiment followed by observation and codifying the experimental results.

Verification is the process of checking the consistency of a simulation program with respect to the lumped model it is derived from.

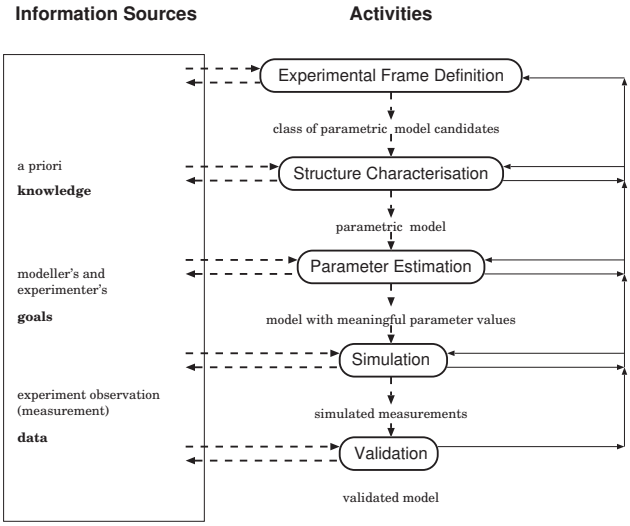


Figure 2: Model-based systems analysis.

Validation is the process of comparing experiment *measurements* with *simulation results* within the context of a certain Experimental Frame (Balci, 1997). When comparison shows differences, the formal model built may not correspond to the real system. A large number of matching *measurements* and *simulation results*, though increasing confidence, does *not prove* validity of the model however. For this reason, Popper has introduced the concept of *falsification* (Magee, 1985), the enterprise of trying to falsify or disprove a model. It is to be noted that the correspondence in generated behaviour between a system and a model will only hold within the limited *context* of the Experimental Frame. Consequently, when using models to exchange information between human or computer agents, a model must always be matched with an Experimental Frame before use. Conversely, a model should never be developed without simultaneously developing its Experimental Frame. This requirement has its repercussions on the design of model representation languages.

1.2 The Modelling and Simulation Process

The use of simulation has proven to be invaluable in the study of complex systems. The simulation activity is part of the larger model-based systems analysis enterprise. A framework for these activities is depicted in Figure 2. The Framework starts by identifying an Experimental Frame. As mentioned above, the frame represents the experimental conditions under which the modeller wants to investigate the system. As such, it reflects the modeller's goals and questions. Based on a frame, a class of matching models can be identified. Through structure characterization, the appropriate model structure is selected based on a priori knowledge and measurement data. Subsequently, during model calibration, parameter identification (or estimation)

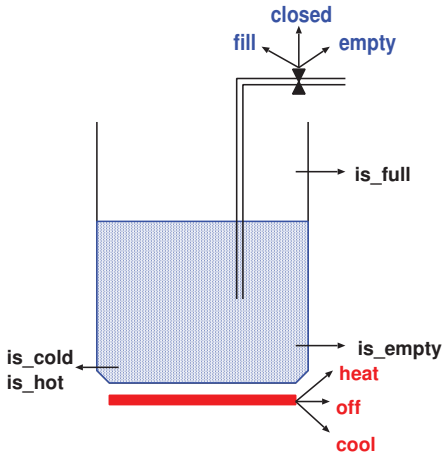


Figure 3: T, l controlled liquid in a vessel.

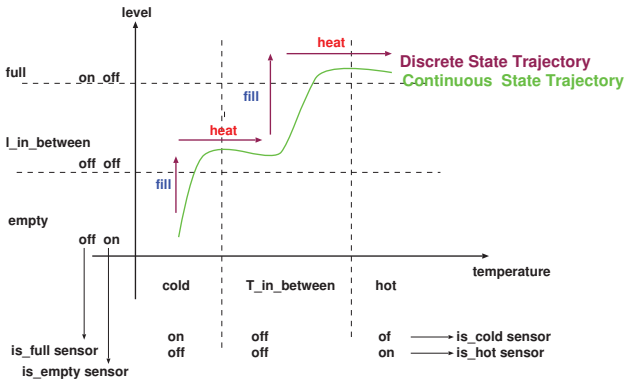


Figure 4: Trajectories.

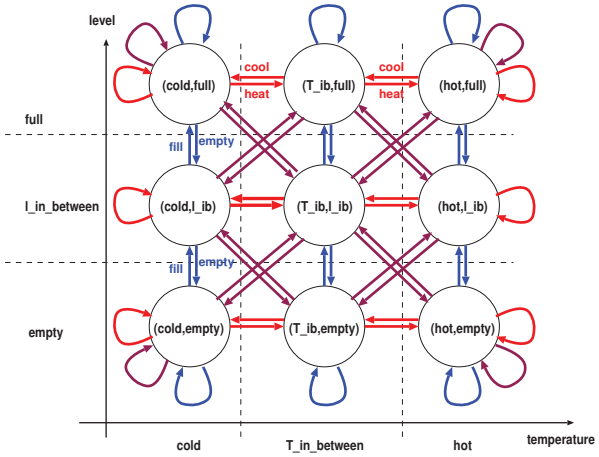


Figure 5: FSA formalism.

yields optimal parameter values for reproducing a set of measurement data. Using the identified model and parameters, simulation allows one to mimic the system behavior (virtual experimentation). The question remains however whether the model has predictive validity: is it capable not only of reproducing data which was used to choose the model and to identify parameters, but also of predicting new behavior ? In Figure 2, one notices how each step in the modelling process may introduce errors. As indicated by the feedback arrows, a model has to be corrected once falsified.

2 ABSTRACTION AND FORMALISM

Abstract models of system behaviour can be described at different levels of abstraction or detail as well as by means of different formalisms. The particular formalism and level of abstraction depends on the background and goals of the modeller as much as on the system modelled. As an example, a temperature and level controlled liquid in a vessel is considered. This is a simplified version of the system described in (Barros et al., 1998), where structural change is the main issue. On the one hand, the liquid can be heated or cooled. On the other hand, liquid can be added or removed. In this simple example phase changes are not considered. The system behaviour is completely described by the following Ordinary Differential Equation (ODE) model:

$$\begin{cases} \frac{dT}{dt} &= \frac{1}{l}[\frac{W}{c\rho A} - \phi T] \\ \frac{dl}{dt} &= \phi \text{ if } 0 < l < H \text{ else } 0 \\ is_low &= (l < l_{low}) \\ is_high &= (l > l_{high}) \\ is_cold &= (T < T_{cold}) \\ is_hot &= (T > T_{hot}) \end{cases}$$

The inputs are the filling (or emptying if negative) flow rate ϕ, and the rate W at which heat is added (or removed if negative). This system is parametrized by A, the cross-section surface of the vessel, H, its height, c, the specific heat of the liquid c, and ρ, the density of the liquid. The state of the system is characterized by variables T, the temperature and l, the level of the liquid. The system is observed through threshold output sensors $is_low, is_high, is_cold, is_hot$. Given input signals, parameters, and a physically meaningful initial condition (T_0, l_0), simulation of the behaviour yields a continuous state trajectory as depicted in Figure 4. By means of the binary (on/off) level and temperature sensors introduced in the differential equation model, the state-space may be discretized. The inputs can be abstracted to heater heat/cool/off and pump fill/empty/closed. At this level of abstraction, a Finite State Automaton representation (with 9 possible states) of the dynamics of the system as depicted in Figure 5 is most appropriate. Though at a much higher level of abstraction, this model is still able to capture the essence of the system's behaviour. In particular, there is a *behaviour morphism* between both models: model discretization (from ODE to FSA) followed by simulation yields the same result as simulation of the ODE followed by discretization.

2.1 Systems Theory

In general systems theory (Wymore, 1967), a causal (output is the consequence of given inputs), deterministic (a known input will lead to a unique output) system model *SYS* is defined. It is a template for a plethora of different formalims such as Ordinary Differential Equations, Finite State Automata, Difference Equations, Petri Nets, *etc*. Its general form is

$$SYS \equiv \langle \mathcal{T}, X, \Omega, Q, \delta, Y, \lambda \rangle$$

$\mathcal{T}$	time base
X	input set
$\omega : \mathcal{T} \to X$	input segment
Q	state set
$\delta : \Omega \times Q \to Q$	transition function
Y	output set
$\lambda : Q \to Y$	output function

$$\forall t_x \in [t_i, t_f]$$

$$\delta(\omega_{[t_i,t_f]}, q_i) = \delta(\omega_{[t_x,t_f]}, \delta(\omega_{[t_i,t_x]}, q_i))$$

The time base $\mathcal{T}$ is the formalisation of the independent variable time, also known as the indexing variable (Nance, 1981). The input set describes all possible allowed inputs (possibly a product set). An input represents input during a time-interval. The history of system behaviour is condensed into a *state*. The dynamics is described in a transition function which takes a current state, and applies an input segment $\omega \in \Omega$ to it to obtain a new state. The system may generate output. The output is obtained as a function λ of the state. State and transition function must obey the composition or transitivity property. This property, whereby a transition over a time interval $[t_i, t_f]$ can always be split into a composition of transitions over arbitrary sub-intervals, is the basis of all model simulators. As the output function is described separately, efficient simulators will only invoke it when the user desires to observe output. As *SYS* is a template for different formalisms, it is possible to describe both models of the vessel example. In the ODE case, the time base is continuous ($\mathbb{R}$). The transition function is written in integral form. Different numerical approximations of the integral can be used in the implementation of an abstract simulator.

$$SYS^{ODE}_{VESSEL} = \langle \mathcal{T}, X, \Omega, Q, \delta, Y, \lambda \rangle$$

$\mathcal{T} = \mathbb{R}$
$X = \mathbb{R} \times \mathbb{R} = \{(W, \phi)\}$
$\omega : \mathcal{T} \to X$
$Q = \mathbb{R}^+ \times \mathbb{R}^+ = \{(T, l)\}$
$\delta : \Omega \times Q \to Q$
$\delta(\omega_{[t_i,t_f]}, (T(t_i), l(t_i))) =$

$$(T(t_i) + \int_{t_i}^{t_f} \frac{1}{l(\alpha)} [\frac{W(\alpha)}{c\rho A} - \phi(\alpha) T(\alpha)] d\alpha,\ l(t_i) + \int_{t_i}^{t_f} \phi(\alpha) d\alpha)$$

$Y = \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} = \{(is_low, is_high, is_cold, is_hot)\}$
$\lambda : Q \to Y$

Q	$\mathcal{T}$: **Continuous**	$\mathcal{T}$: **Discrete**	$\mathcal{T}$: $\{NOW\}$
Continuous	DAE	Difference Eqns.	Algebraic Eqns.
Discrete	Discrete-event Naive Physics	FSA Petri Nets	Integer Eqns.

Table 1: Formalism classification.

$$\lambda(T, l) = ((l < l_{low}), (l > l_{high}), (T < T_{cold}), (T > T_{hot}))$$

At a higher level of abstraction, we have represented time as a discrete integer index. The transition function lists all possible state transformations.

$$SYS^{FSA}_{VESSEL} = \langle \mathcal{T}, X, \Omega, Q, \delta, Y, \lambda \rangle$$

$\mathcal{T} = \mathbb{N}$
$X = \{heat, cool, off\} \times \{fill, empty, closed\}$
$\omega : \mathcal{T} \to X$
$Q = \{cold, T_{between}, hot\} \times \{empty, l_{between}, full\}$
$\delta : \Omega \times Q \to Q$
$\delta((off, fill), (cold, empty)) = (cold, l_{between})$
$\delta((off, fill), (cold, l_{between})) = (cold, full))$
$\delta((off, fill), (cold, full))) = (cold, full))$
$\vdots$
$\delta((heat, fill), (hot, full))) = (hot, full))$
$Y = \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}$
$\lambda : Q \to Y$
$\lambda(T, l) = ((l = low), (l = high), (T = cold), (T = hot))$

Apart from the two causal, deterministic formalisms presented above, a host of other formalisms are in use. In particular, through time-scale or parameter abstraction, reality is often represented by means of *discrete-event models*. In these models, time evolves continuously ($\mathcal{T} = \mathbb{R}$), but the state of the system only changes at a finite number of points in time in a bounded time-interval. These times are called event-times. A number of discrete-event formalisms, called *world views* were constructed. World views range from Event Scheduling which expresses events and their consequences explicitly, focussing on the simulation efficiency, over Activity Scanning and the Three Phase Approach which emphasize changing conditions in the system, to Process Interaction which represents the system in terms of interacting processes (Balci, 1988). Zeigler (Zeigler, 1984a) developed the DEVS discrete-event system specification. It allows other discrete-event formalisms to be expressed in terms of DEVS.

2.2 Formalism Classification

Formalism classification based on the general system model structure in Table 1 shows how different instantiations of that structure lead to different formalisms. For continuous models, classification according to physical domains such

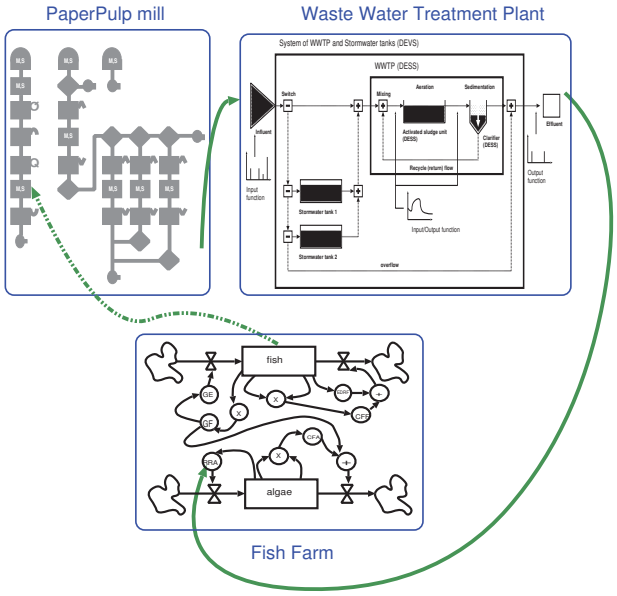


Figure 6: Complex system example.

as mechanical, electrical, and hydraulical, is meaningful. The variety of classifications leads to the insight that ultimately, one should picture a vast formalism-space and classify that space according to various criteria. Different criteria will lead to different equivalence classes. The *SYS* time base and state set allow for a high-level classification of formalisms. Whithin the discrete-event realm, the world views provide a further means of classification. Note how many languages and tools correspond to each single formalism.

Though quite generic, the formalisms presented do not describe *non-causal* models. When describing physical systems, non-causal models describe conservation laws and constraints without imposing a computational causality. Non-causal modelling formalisms and how they enable meaningful model re-use are described in (Cellier, 1991).

3 MULTI-FORMALISM MODELLING

Complex systems are characterized, not only by a large *number* of components, but also by the *diversity* of the components. One of the observations of the European Commission's ESPRIT Basic Research Working Group 8467 (Vangheluwe and Vansteenkiste, 1996) "Simulation for the Future: New Concepts, Tools and Applications" was that for the analysis and design of such complex systems, it is no longer sufficient to study the diverse components separately, using the specific formalisms these components were modelled in. Rather, it is necessary to answer questions about properties –most notably behaviour– of the *whole* system. Also, to communicate knowledge about these systems, one must deal with the diversity of the information.

3.1 A Complex System Example

To focus the attention, Figure 6 presents a complex system. The complexity lies in the diversity of the different components, both in abstraction level and in formalism used:

- A paper and pulp mill produces paper from trees with polluted water as a side-effect. This system is modelled as a process interaction discrete-event scheduling system (in particular, in GPSS (Gordon, 1996)).
- A Waste Water Treatment Plant (WWTP) purifies the polluted effluent from the mill. Some solid waste is taken to a landfill. The partially purified water flows into a lake. This system is modelled using Differential Algebraic Equations (DAEs) describing the biochemical reactions in the WWTP.
- A Fish Farm grows fish in the lake. The fish feed on algae which are highly sensitive to polluted water. The water is also used for a tree plantation which supplies the paper mill. This eco-system is modelled using the Forrester System Dynamics formalism. The dotted feedback arrow from the fish farm to the paper mill indicates the possible disastrous impact of poisoned fish on the productivity of workers in the mill.

It is obvious that decision-making for this system will require understanding of the behaviour of the overall system. Studying the individual components will not suffice. The complexity of this system and its model is due to

- the *number* of *interacting*, coupled, concurrent components. Complex behaviour is often a consequence of a large number of feedback loops;
- the variety of component *formalisms*. Often, a mix of software and hardware, continuous and discrete components occurs;
- the variety of *views*, at different levels of *abstraction*.

A model of a system such as the one described above may be valid (within a particular experimental context) at a certain *level of abstraction*. This level of abstraction, which may be different for each of the components, is determined by the available knowledge, the questions to be answered about the system's behaviour, the required accuracy of answers, *etc.* Orthogonal to the choice of model abstraction level is the selection of suitable *formalisms* in which the models are described. The choice of formalism is related to the abstraction level, the amount of data that can be obtained to calibrate the model, the availability of solvers/simulators for that formalism, as well as to the kind of questions which need to be answered.

3.2 Forrester System Dynamics

To fully explain the model, we now present the semantics of the Forrester System Dynamics (FSD) formalism. The FSD formalism (Cellier, 1991) describes the variation of material-like quantities or levels. The variation is determined by birth rates (*BR*) and death rates (*DR*). *BR* and *DR* are graphically represented as valves to the left and right respectively of boxes denoting the levels. Levels may influence each other by influencing each other's *BR* and *DR*.

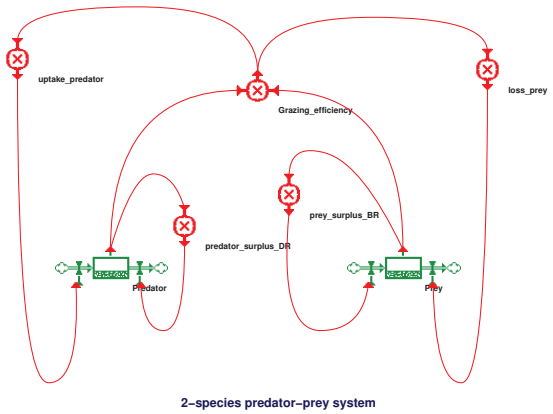


Figure 7: Predator-prey, FSD formalims.

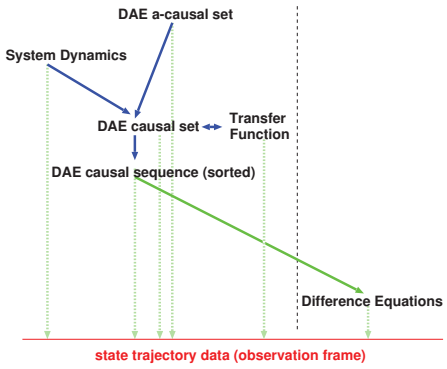


Figure 8: Formalism transformation.

The influences are given by algebraic functions. Figure 7 shows a typical interaction between a predator and a prey. The (product) interaction between predator and prey populations influences the predator's birth rate and the prey's death rate. The System Dynamics semantics is given by mapping each of the level/*BR*/*DR* combinations onto an Ordinary Differential Equation

$$\frac{d\ level}{dt} = BR - DR.$$

The operations such as product and sum are mapped onto the appropriate algebraic equations and couplings are mapped onto algebraic equalities.

Based on this relationship between the System Dynamics and the ODE formalisms, translation of any model in the first formalism to the same model described in the second formalism is possible. In practice, this implies that a compiler can translate any model in the first formalism onto a (behaviourally equivalent) model in the second formalism. In Figure 8, a small part of "formalism space" is depicted in the form of a Formalism Transformation Graph (FTG). The different formalisms described before are shown as nodes in the graph. The vertical striped line denotes the distinction between continuous models (on the left) and discrete models (on the right). The well known Difference Equations formalism is often implicitly used in numerical simulators: ODEs are discretised by means of a suitable numerical scheme and the resulting difference equations are iteratively solved. Suitable refers to the nature of the equations as well as to the accuracy requirements. The arrows denote a homomorphic relationship "can be mapped onto", implemented as a transformation between formalisms. The vertical, dotted lines denote the existence of a *solver* or *simulation kernel* which is capable of simulating a model, thus generating a trajectory. A trajectory is really a model of the system in the data formalism (time/value tuples). Obviously, the experimental frame of a trajectory is very limited. In a denotational sense, traversing the graph makes *semantics* of models in formalisms explicit: the meaning of a model/formalism is given by mapping it onto some known formalism (whose meaning may in turn be given by mapping it onto an even more basic formalism, *etc.*). Though a multi-step mapping may seem cumbersome, it can be perfectly and correctly performed by tools. The advantage of this approach is that the introduction of a *new formalism* only requires the description of the mapping onto the nearest formalism as well as the implementation of this mapping in the form of a model translator. It is often meaningful to introduce a new formalism for a specific application, encoding particular properties and constraints of the application. Often, translation involves some loss of information. This loss may be a blessing in disguise as it entails a reduction in complexity, hopefully leading to an increase in (simulation) performance. Usually, the aim of multi-step mapping is to eventually reach the trajectory level. Another major use for formalism transformation is the answering of particular questions about the system. Some questions can only be answered in the context of a particular formalism. In case of a FSD model for example, the visual inspection of the model can provide insight into influences. If the model is mapped onto a set of Algebraic and Ordinary Differential Equations, a dependency analysis may reveal algebraic cycles not apparent at the System Dynamics level. At this same level, one may check whether parts of the model are linear. If so, these parts may be solved symbolically by means of computer algebra. Also, transformation to the Laplace domain (*i.e.,* to a Transfer Function form) leads to a plethora of techniques for stability analysis. Finally, the transformation, through numerical simulation, to the data level, allows for quantitative analysis of problems posed in initial value form.

Note how the larger the number of intermediate formalisms becomes, the higher the possibility for optimization along the way will be.

Above all, the traversal described above is the basis for the meaningful coupling of models described in different formalisms. This will be discussed next.

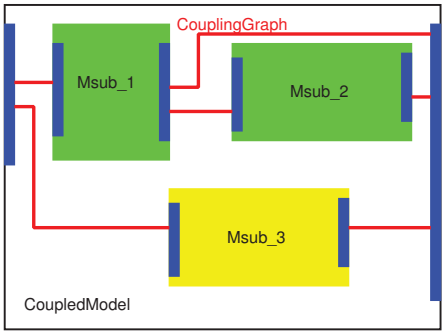


Figure 9: Multi-formalism coupled model.

3.3 Coupled Model Transformation

Figure 9 shows a multi-formalism coupled model (the different shades denote different formalisms). Models of this type are said to belong to the *network* or *coupled* formalism. When we observe a structured model at this level, we can only make meaningful assertions about its structure, its outside connections (its interface) and its components, not about its overall meaning or behaviour ! Formally, a coupled model has the form

$$CM \equiv \langle id, interface, S, C \rangle$$

The model is identified by a unique identifier *id* (for example, a name or reference). The $interface$ is a set of connectors or ports to the outside. Associated with the ports are allowed values as well as causality. Meaningful causalities are $\{in, out, inout\}$. The set S contains the sub-models (or at least their unique identifiers). The coupling information is contained in a graph structure C. For non-causal, continuous models, the graph is undirected. For causal models, the graph is directed. Obviously, a coupled model is only valid if types and causalities of connected ports are compatible. In certain cases, the graph may be annotated with extra information. In case of discrete-event models, a tie-breaking function is usually required to select between simultaneous events (Zeigler, 1984a).

If all the sub-models are described in the same formalism F, it may be possible to (recursively, bottom-up, if the sub-models are coupled models in their own right) replace the coupled model (at least conceptually) by one atomic model of type F. In this case, F is called *closed under coupling* (or under composition). The closure property has to be proven for each F. The property often holds by construction. In case of Differential Algebraic Equations (DAEs), connections $\{connect(port_i, port_j)\}$ are replaced by $port_i = port_j$ coupling equations. Together with the sub-models' equations, these form a DAE. In formalisms such as Bond Graphs, information about the physical nature of variables allows one to generate either the above type of equations in case of coupling of across variables (this corresponds to Kirchoff's voltage law in electricity) and an equation summing all connected values to zero for through variables (this corresponds to Kirchoff's current law in electricity). In discrete-event models, implementing closure involves the correct time-ordered *scheduling* of sub-model events. The most imminent event will always be processed first. The tie-breaking function is used to resolve conflicts due to simultaneous events (an artifact of the high level of abstraction).

If a coupled model consists of sub-models expressed in different formalisms, different approaches are possible:

- A *super-formalism* can be used which subsumes the different formalisms of the sub-models. The different sub-models are thus described in the same formalism. The Hybrid DAE (Vangheluwe, 2000) and DEVS&DESS (Zeigler et al., 2000) formalisms integrate continous and discrete modelling constructs. Meaningful super-formalisms which truly add expressiveness and reduce complexity are rare. Bond Graphs (Cellier, 1991) are a good example of the integration of different domains (mechanical, electrical, hydraulical).
- Another approach is to *transform* the different sub-models to one *common formalism*. Which formalism to transform to depends on the questions asked. The closest common formalism for the WWTP (DAE) and fish farm (System Dynamics) example is the DAE formalism. By transforming the fish farm model to a set of DAEs, and using the closure property of the DAE formalism, it becomes possible to answer questions about the overall model. Often, the target formalism is the trajectory level.
- In the *co-simulation approach*, each of the sub-models is simulated with a formalism-specific simulator. Interaction due to coupling is resolved at the trajectory level. Compared to transformation to a common formalim before simulation, this approach, though appealing from a software engineering point of view, discards a variety of useful information. Questions can *only* be answered at the trajectory level. Furthermore, there are obvious speed and numerical accuracy problems for continuous formalisms (Foster and Yelmgren, 1997). The approach is meaningful only for discrete-event formalisms. In this realm, it is the basis of the DoD High Level Architecture (HLA) for simulator interoperability.

The transformation to a common formalism mentioned above proceeds as follows:

1. Start from a coupled multi-formalism model. Check consistency of this model (*e.g.,* whether causalites and types of connected ports match).
2. Cluster all models described in the same formalism.
3. For each cluster, implement closure under coupling.
4. Look for the "best" common formalism in the Formalism Transformation Graph all the remaining different

formalisms can be transformed to. In the worst case, this will be the trajectory level in which case the approach falls back to co-simulation. Which common formalism is best depends on a quality metric which can take into account transformation speed, potential for optimization, *etc.* .

5. Transform all the sub-models to the common formalism.
6. Implement closure under coupling of the common formalism.

Certain questions about a system can only be answered in certain formalisms, necessitating model transformation to the appropriate formalism. A side-effect of mapping onto a common formalism is the great potential for optimization of the flattened model, as well as the reduced number of (optimized) simulation kernels needed. When models are used to *communicate* knowledge between humans or tools, the formalism used must be clearly and uniquely specified. As described further on, meta-modelling allows for explicit reasoning about often subtle differences between formalisms. This is achieved by explicitly modelling both syntax and semantics of formalisms.

3.4 The Formalism Transformation Graph

To describe which formalism transformation are possible, the Formalism Transformation Graph (FTG) is used (Vangheluwe, 2000). As an example, the transformations currently known to the authors are shown in Figure 10. The line at the bottom denotes the state trajectory formalism. In principle, this should be a single node, but a line was used for aesthetic purposes. The vertical dashed line denotes the crude division between continuous and discrete(-event) models. Vertical dashed arrows denote the existence of a simulator which maps an abstract model onto a state trajectory (given initial conditions, parameters, *etc.*). It is noted that iterative simulation can be seen as a special case of model transformation.

4 META-MODELLING

A proven method to achieve flexibility for a modelling language to support many formalisms is to model the language itself. This approach is demonstrated in, among others, the *domain modelling environment* (DOME) (Honeywell, 1999; Engstrom and Kruger, 2000) and the *multigraph architecture* (MGA) (Karsai et al., 2000). To illustrate this notion, consider the state transition diagram in Figure 11 (in the Deterministic Finite State Automata formalism). When in the ON *state*, a *transition* to the OFF state occurs when the *condition* $t > 2$ becomes true and this generates an alarm *action*. The state, transition, condition, and action elements are part of any state transition diagram and their dependencies can be modelled as shown in Figure 12 (in the Entity-Relationship formalism). This model specifies a family of state transition diagrams where each instantiation has states

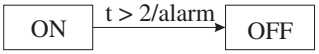


Figure 11: A state transition diagram model.

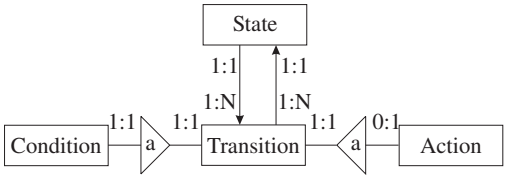


Figure 12: A model of state transition diagram models.

that are connected by transitions. Each state can have any number of outgoing transitions, indicated by the $1:1$ and $1:N$ cardinality on the downward arrow in the figure, *i.e.,* each state can have between 1 and N transitions and each transition has to exit between 1 and 1 states (*i.e.,* it has to be connected to one and only one state), where N represents any number. Each transition can enter only one state and each state may have any number of entering transitions indicated by the cardinality on the upward arrow. The transitions between states have two attributes, one condition that allows the state transition to be taken and one optional (indicated by the $0:1$ cardinality) action. The model in Figure 12 can be used to specify different state transition diagram formalisms, *e.g.,* the action attribute can be made mandatory for each transition by changing the cardinality from $0:1$ to $1:1$. Furthermore, actions can be associated with states as well.

Such a model of the modelling language is called a *meta*-model. It prescribes the possible mathematical structures (formalisms) that can be expressed in the modelling language and can be tailored to specific needs of particular domains. From the meta-model specification, the modelling language, graphical or textual, can then be instantiated automatically. This requires the meta-model modelling formalism to be sufficiently rich and support the constructs needed to define a modelling language. To allow for easy extension, the meta-model modelling formalism can be modelled by a *meta-meta*-model. This meta-meta-model specification captures the basic elements that can be used to design a meta-model modelling formalism. In case new concepts and structures are required, these can be conveniently modelled at a meta-meta-level.

For example, the meta-model in Figure 12 is limited to the family of state transition diagrams. This restriction can be removed by modelling the model of state transition diagrams in Figure 12 by the meta-meta-model in Figure 13. It contains an abstract representation of the mechanisms that are part of the state transition diagrams meta-model, *i.e.,* entities (states, transitions), attributes (actions and conditions), and relations between them. This meta-meta-model groups entities and relations by an *object* model component and each of them optionally has any number of attributes. It also shows that each relation connects to one entity as

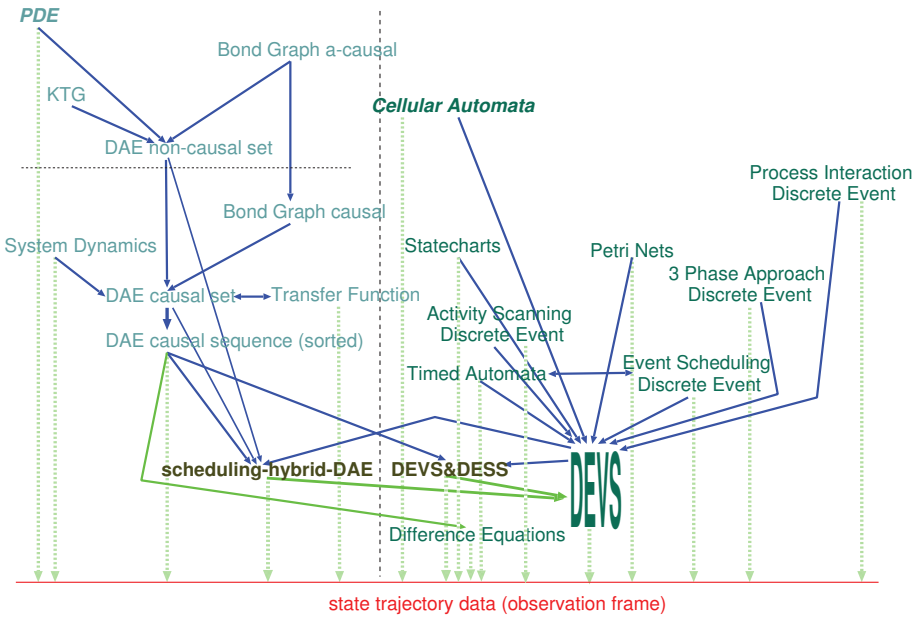


Figure 10: Formalism Transformation Graph.

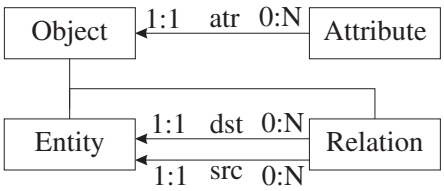


Figure 13: A state transition diagram meta-meta-model.

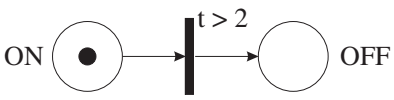


Figure 14: A Petri net model.

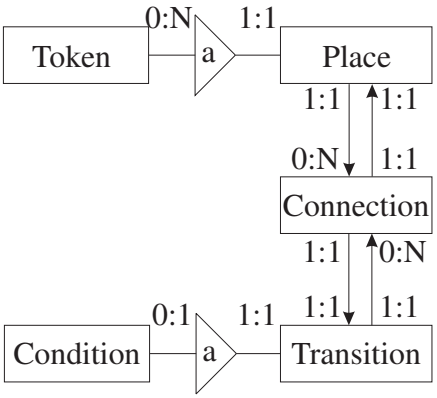


Figure 15: A model of Petri net models.

its source (marked src) and one entity as its destination (marked dst), and, therefore, directed links can be used. The meta-meta-model specification language has to consist of entities, attributes to specify the cardinality and relations to specify the three types, (i) source relations (src), (ii) destination relations (dst), and (iii) attribute relations (atr). Given the meta-meta-model in Figure 13, a broader family of meta-models can be described. For example, a Petri net as illustrated in Figure 14 consists of *places* (shown as transparent circles) and *transitions* (shown as solid rectangles). Each place has *connections* to transitions and each transition to places. A transition may have a condition and when this condition is true and all its input places, *i.e.,* places that are the source to the transition, contain a *token* (shown as a black dot inside a place), it may fire (the corresponding transition may be executed). The Petri nets meta-model shown in Figure 15 specifies places and transitions that can connect to one another. Furthermore, the tokens are specified as optional attributes of a place and conditions as an optional attribute of a transition. The family of Petri nets that can be instantiated from this meta-model allows places with multiple tokens. However, in some modelling paradigms, only one token per place is allowed, which can be conveniently changed in the meta-model by specifying 0 : 1 cardinality, and, consequently, constraining the family of Petri nets. Note that this represents both a static and dynamic constraint. When the Petri net is initialized, it can be ensured by the model editor that each place has been assigned at most one token. However, it is still necessary to dynamically check whether this constraint is violated during execution. To be able to fully specify modelling formalisms, a meta-level formalism is often extended with a textual *constraint language* complementary to the often graphical base. This language can be used to rule out semantically incorrect models and greatly reduce the family

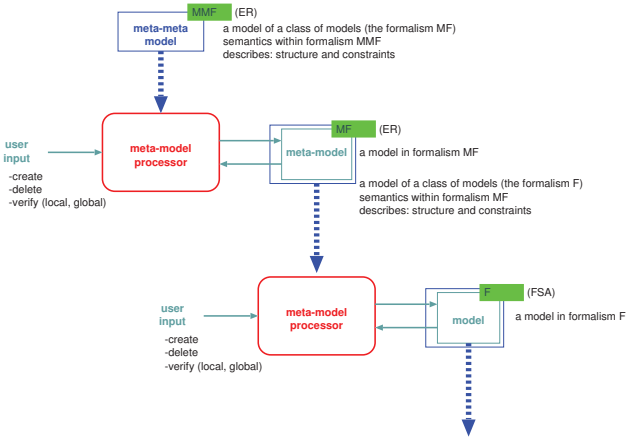


Figure 16: Meta-$\cdots$ models.

of models that can be modelled (Nordstrom, 1999). For example, a model of a family of Petri nets could include the constraint that there should never be more than ten tokens in the net (to model a resources limit) by an additional specification `Token.allInstances->size < 10`.

The outlined meta-modelling approach leads to the layered structure in Table 2. The model row represents a particular model such as the state transition diagram in Figure 11. At a meta-level, the meta-model row represents a class of models (or formalism), *e.g.,* the model of the family of state transition diagrams in Figure 12. This meta-model has meaning within a formalism which is specified by a meta-meta-model. This could be the model in Figure 13.

The apparent advantage of this approach is the tremendous flexibility that can be achieved.

This is manifested in the ease with which new formalisms can be designed. By adapting the model of a modelling formalism, and automatically generating a prototype of the modelling environment, design choices can be rapidly evaluated.

Figure 16 depicts how meta-models are models too. An FSA model is constructed within the FSA formalism. In the meta-modelling approach, the FSA formalism is explicitly modelled. The FSA formalism's meta-model has meaning within the Entity-relationship formalism.

4.1 Meta-modelling Model Transformation

Upto now, only model *syntax* has been considered in meta-modelling. It is however above all necessary to describe model *semantics*. At the core of this is the specification of model transformation.

Some of the manipulations we are interested in are:

- Formalism transformation: Given a model in a certain formalism, these transformations convert it into a model, but expressed in another formalism. For Modelling and Simulation, possible transformations are found in the Formalism Transformation Graph.

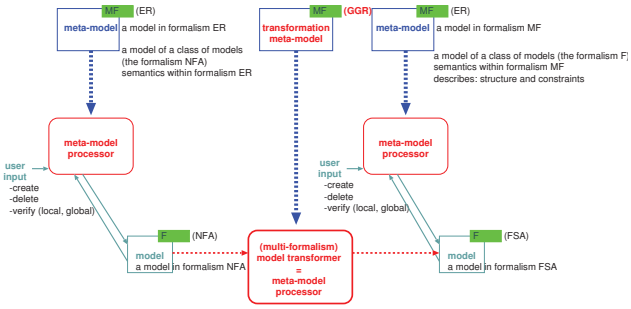


Figure 17: Model transformation meta-specification.

- Model optimization: These transformations do not change the formalism in which the model is expressed. Their application results in a reduction of model complexity.
- Code Generation: These transformations produce a textual representation of the model (subject to syntactic constraints) suitable for interpretation by a simulator.
- Simulator specification: These specifications give the operational semantics of the model.

All these tasks depend on the formalisms of interest. However, since models determined by some meta-model can always be described as graphs (subject to the constraints given by the meta-model), these tasks can be performed by a generic graph-transformation algorithm. Therefore it makes sense to combine meta-modelling and *graph-grammars* (Dorr, 1995) in a unifying framework. Meta-models determine the classes of graphs that are allowed on the LHS and RHS of a graph-grammar rule. Furthermore, the rules themselves, and the grammars, can be viewed as models in the graph-grammar (GGR) formalism, which itself is described in a meta-model. This is depicted in Figure 17. Graph Grammars are composed of rules, each mapping a graph on the left-hand side (LHS) to a graph on the right-hand side (RHS). A graph-grammar is applied to an input graph (called host graph) in order to perform a transformation. When a match is found between the LHS of a rule and a part of the host graph, this subgraph is replaced by the RHS of the rule. Rules may also have a condition that must be satisfied for the rule to be applied, as well as actions to be performed when the rule is executed. A rewriting system iteratively applies matching rules in the grammar to the graph, until no more rules are applicable.

On the one hand, the use of a model (in the form of a graph grammar) of graph transformations has some advantages over an implicit representation (embedding the transformation computation in a program) (Blonstein et al., 1996):

- it is an abstract, declarative, high level representation, and
- the theoretical foundations of graph rewriting systems may assist in proving correctness and convergence properties of the transformation tool.

Level	Description	Example
meta-meta-model	Model used to specify modelling languages	`Relation hasDestination Entity`
meta-model	Model used to specify models, an instantiation of a meta-meta-model	`State connectsTo Transition`
model	The description of an object in a specific formalism	`when t > 2 transition from ON to OFF`

Table 2: Three layer meta-modelling structure.

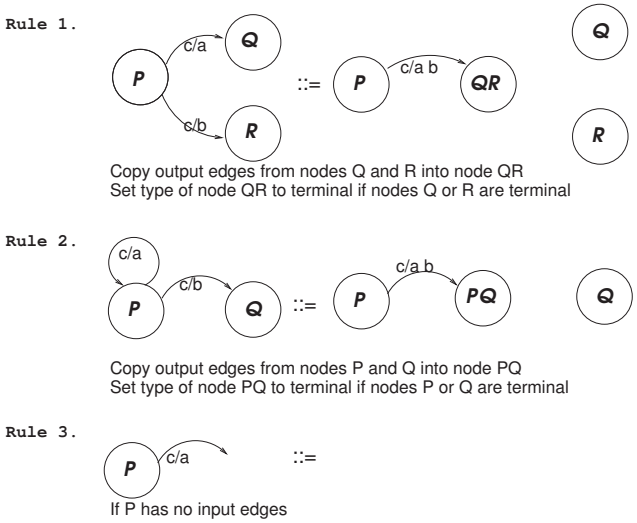


Figure 18: NFA to DFA Graph Grammar.

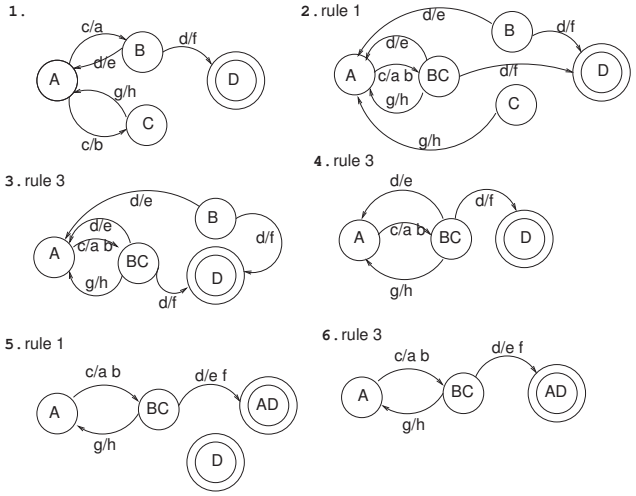


Figure 19: NFA to FSA Transformation.

On the other hand, the use of graph grammars is constrained by efficiency. In the most general case, subgraph isomorphism testing is NP-complete. However, the use of small subgraphs on the left hand side of graph grammar rules, as well as using node labels and edge labels can greatly reduce the search space.

As an example, Figure 18 gives the rules for transforming a non-deterministic finite state automaton (NFA) into an equivalent deterministic finite state automaton (DFA). Figure 19 shows the application of this graph grammar to a simple NFA.

It should be noted that the *operational semantics* of a formalism may be expressed in the form of a graph grammar model. This, by encoding the formalism's transition function. As such, graph grammar application corresponds to simulation. We can thus regard the graph grammar as an *executable specification*. This approach is desirable for its generality, since it can be applied to a wide class of formalisms. There is, however, a tradeoff made between generality and efficiency. As a general rule, customized, hand-coded, formalism-specific simulation algorithms are more efficient. The approach of relying on graph grammars is expensive due to the nature of the graph matching algorithm.

However, there are other motivations, both theoretical and practical:

- Explicitly defining the operational semantics of any formalism should be part of the design of a simulator. The specification is then the basis for any implementation.
- The specification and its interpretation provide a framework (a *reference implementation*) for verifying and testing different (hand-crafted, efficient) implementations.
- It provides a portable simulator, since it is more abstract than a hand-coded implementation.
- It allows for reasoning about the described systems. For example, it allows for the definition of general algorithms for bisimulation.

5 CONCLUSIONS

We have presented a comprehensive overview of multi-paradigm Modelling and Simulation. The focus was on the concept of multi-formalism modelling of complex systems, and its relation to the semantics of models. Furthermore, meta-modelling was presented as a means of dealing with multiple formalisms. Currently,

the first two authors are developing AToM^3, a meta-modelling environment. AToM^3 uses graph grammars to specify (at a meta-level), the transformations in the Formalism Transformation Graph. AToM^3 can be found at `http://moncs.cs.mcgill.ca/MSDL/research/projects/AToM3`.

ACKNOWLEDGEMENT

Prof. Vangheluwe gratefully acknowledges partial support for this work by a National Sciences and Engineering Research Council of Canada (NSERC) Individual Research Grant.

REFERENCES

Balci, O. (1988). The implementation of four conceptual frameworks for simulation modeling in high-level languages. In Abrams, M., Haigh, P., and Comfort, J., editors, *Proceedings of the 1988 Winter Simulation Conference*, pages 287–295. Society for Computer Simulation International (SCS).

Balci, O. (1997). Principles of simulation model validation, verification, and testing. *Transactions of the Society for Computer Simulation International*, 14(1):3–12. Special Issue: Principles of Simulation.

Barros, F. J., Zeigler, B. P., and Fishwick, P. A. (1998). Multimodels and dynamic structure models: an integration of DSDE/DEVS and OOPM. In Medeiros, D., Watson, E., and M.S., M., editors, *Proceedings of the 1998 Winter Simulation Conference*, pages 413–419. Society for Computer Simulation International (SCS).

Blonstein, D., Fahmy, H., and Grbavec, A. (1996). Issues in the practical use of graph rewriting.

Cellier, F. E. (1991). *Continuous System Modeling*. Springer-Verlag, New York.

Dorr, H. (1995). Efficient graph rewriting and its implementation.

Engstrom, E. and Kruger, J. (2000). A meta-modeler's job is never done: Building and evolving domain-specific tools with DOME. In Varga, A., editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 83–88. IEEE Computer Society Press. Anchorage, Alaska.

Foster, L. and Yelmgren, K. (1997). Accuracy in DoD High Level Architecture Federations. In Obaidat, M. S. and Illgen, J., editors, *Summer Computer Simulation Conference (SCSC'97)*, pages 451–460. Society for Computer Simulation International (SCS). Arlington, Virginia.

Gordon, G. (1996). *System Simulation*. Prentice Hall of India, second edition.

Honeywell (1999). DOME guide. http://www.htc.honeywell.com/dome/, Honeywell Technology Center, Honeywell. version 5.2.1.

Karsai, G., Nordstrom, G., Ledeczi, A., and Sztipanovits, J. (2000). Specifying graphical modeling systems using constraint-based metamodels. In Varga, A., editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 89–94. IEEE Computer Society Press. Anchorage, Alaska.

Magee, B. (1985). *Popper*. Fontana Press (An Imprint of HarperCollins Publishers), London.

Nance, R. E. (1981). The time and state relationships in simulation modeling. *Communications of the ACM*, 24(4):173–179.

Nordstrom, G. G. (1999). *Metamodeling – Rapid Design and Evoluion of Domain-Specific Modeling Environments*. PhD dissertation, Vanderbilt University, Electrical Engineering.

Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley.

Vangheluwe, H. L. (2000). DEVS as a common denominator for multi-formalism hybrid systems modelling. In Varga, A., editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 129–134. IEEE Computer Society Press. Anchorage, Alaska.

Vangheluwe, H. L., Kerckhoffs, E. J., and Vansteenkiste, G. C. (2001). Computer automated modelling of complex systems. In Kerckhoffs, E. J. and Snorek, M., editors, 15^{th} *European Simulation Multi-conference (ESM)*, pages 7–18. Society for Computer Simulation International (SCS). Prague, Czech Republic.

Vangheluwe, H. L. and Vansteenkiste, G. C. (1996). SiE: Simulation for the Future. *Simulation*, 66(5):331 – 335.

Wymore, A. W. (1967). *A Mathematical Theory of Systems Engineering – the Elements*. Wiley Series on Systems Engineering and Analysis. Wiley.

Zeigler, B. P. (1984a). *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, London.

Zeigler, B. P. (1984b). *Theory of Modelling and Simulation*. Robert E. Krieger, Malabar, Florida.

Zeigler, B. P., Praehofer, H., and Kim, T. G. (2000). *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, second edition.

Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

META-MODELS ARE MODELS TOO

Hans Vangheluwe

School of Computer Science
McGill University
Montréal, Québec H3A 2A7, CANADA

Juan de Lara

ETS Informática
Universidad Autonóma de Madrid
Madrid, SPAIN

ABSTRACT

This article introduces *multi-formalism* modelling and *meta-modelling* to facilitate computer assisted modelling and simulation of complex systems. To aid in the automatic generation of multi-formalism modelling and simulation tools, formalisms are modelled in their own right, at a meta-level, within an appropriate formalism. This approach is implemented in the interactive tool ATOM^3 (A Tool for Multi-formalism Meta-Modelling). This tool is used to describe formalisms commonly used in the simulation of dynamical systems, as well as to generate custom tools to process (create, edit, simulate, …) models expressed in the corresponding formalism. ATOM^3 relies on graph rewriting techniques to perform the transformations (modelled as graph grammars) between formalisms as well as for other tasks, such as code generation or simulator specification.

The Finite State Automata (FSA) formalism is used to demonstrate the concepts of meta-modelling as well as model transformation (in particular, simulation of FSA models).

The issue of a neutral model exchange and re-use format is addressed in the context of meta-modelling. Core XML is proposed as a standard external format. Thanks to the power of the meta-modelling approach, DTD, XMLSchema, and XSLT specifications may be replaced by models, externally represented in core XML, in appropriate formalisms (Entity Relationship for syntax and Graph Grammar for transformation respectively).

1 INTRODUCTION

Modelling complex systems is a difficult task, because such systems often have components and aspects whose structure as well as behaviour cannot be described in a single comprehensive formalism. Examples of commonly used formalisms are Differential-Algebraic Equations (DAEs), Bond Graphs, Petri Nets, discrete event system specification (DEVS), Entity-Relationship diagrams, and State Charts. Several approaches to modelling complex systems are possible:

- A single super-formalism may be constructed which subsumes all the formalisms needed in the system description. This is not possible nor meaningful in most cases, although there are some examples of formalisms which span several domains (*e.g.,* Bond Graphs for the mechanical, hydraulic and electrical domains.)
- Each system component may be modelled using the most appropriate formalism and tool. To investigate the overall behaviour of the system, co-simulation can be used. In this approach, each component model is simulated with a formalism-specific simulator. Interaction due to component coupling is resolved at the trajectory (simulation data) level. The co-simulation engine orchestrates the flow of input/output data in a data-flow fashion. In this approach, questions about the overall system can only be answered at the level of input/output (state trajectory) level. It is no longer possible to answer higher-level questions which could be answered within the formalisms of the individual components. Furthermore, there are speed and numerical accuracy problems for continuous formalisms, in particular if one attempts to support computationally non-causal models. The co-simulation approach is meaningful mostly for discrete-event formalisms. It is the basis of the DoD High Level Architecture (HLA) `<hla.dmso.mil>` for simulator interoperability.
- In multi-formalism modelling, as in co-simulation, each system component may be modelled using the most appropriate formalism and tool. However, a single formalism is identified into which each of the component models may be symbolically transformed (Vangheluwe 2000). Obviously, the system properties which we wish to investigate

must be invariant under the transformations. The formalism to transform depends on the question to be answered about the system. The Formalism Transformation Graph (see Figure 1) suggests DEVS (Zeigler, Praehofer, and Kim 2000) as a universal common modelling formalism for simulation purposes (generating input/output trajectories).

It is easily seen how multi-formalism modelling subsumes both the super-formalism approach and the co-simulation approach.

Although the model transformation approach is conceptually appealing, there remains the difficulty of interconnecting a plethora of different tools, each designed for a particular formalism. Also, it is desirable to have problem-specific formalisms and tools. The time needed to develop these is usually prohibitive. This is why we introduce meta-modelling, whereby the different formalisms themselves, as well as the transformations between them are modelled. This pre-empts the problem of tool incompatibility. Ideally, a meta-modelling environment must be able to generate customized tools for models in various formalisms provided the formalisms are described at the meta-model level. When these tools rely on a common data structure to internally represent the models, transformation between formalisms is reduced to the transformation of these data structures.

In this article, we present ATOM^3, a tool which implements the ideas presented above. Using Finite State Automata as an example, we demonstrate the use of meta-modelling to specify and automatically generate an FSA modelling and simulation tool. To allow tool-neutral exchange and re-use of models, XML is proposed as an external model storage format. As meta-models are models too, there is no need for a dedicated syntax such as DTD or XMLSchema to describe formalism syntax.

2 MULTI-FORMALISM MODELLING

Complex systems are characterized not only by a large number of components, but above all by the diversity of these components (and the feedback interaction between them). For the analysis and design of such complex systems, it is not sufficient to study the individual components in isolation. Properties of the system must be assessed by looking at the *whole* multi-formalism system.

In Figure 1, a part of the "formalism space" is depicted in the form of a *Formalism Transformation Graph* (FTG). The different formalisms are shown as nodes in the graph. The arrows denote a behaviour-preserving homomorphic relationship "can be mapped onto", using transformations between formalisms. The vertical dashed line is a division between continuous and discrete formalisms. The vertical, dotted arrows denote the existence of a solver (simulation kernel) capable of simulating a model.

3 META-MODELLING

As stated in the previous section, one of the characteristics of complex systems is the diversity of their components. Consequently, it is often desirable to model the different components using different modelling formalisms. This is certainly the case, when inter-disciplinary teams collaborate on the development of a single system. A proven method to achieve the required flexibility for a modelling language that supports many formalisms and modelling paradigms is to model the modelling language itself (Honeywell 1999). Such a model of the modelling language is called a meta-model. It describes the possible structures which can be expressed in the language. A meta-model can easily be tailored to specific needs of particular domains. This requires the meta-model modelling formalism to be rich enough to support the constructs needed to define a modelling language. Taking the methodology one step further, the meta-modelling formalism itself may be modelled by means of a meta-meta-model. This meta-meta-model specification captures the basic elements needed to design a formalism. Table 1 depicts the levels considered in our meta-modelling approach. Formalisms such as Entity-Relationship diagrams are often used for meta-modelling. To be able to fully specify modelling formalisms, the meta-level formalism may have to be extended with the ability to express constraints (limiting the number of meaningful models). For example, when modelling a Determinsitic Finite State Automaton, different transitions leaving a given state must have different labels. This cannot be expressed within Entity-Relationship diagrams alone. Expressing constraints is most elegantly done by adding a constraint language to the meta-modelling formalism. Whereas the meta-modelling formalism frequently uses a graphical notation, constraints are concisely expressed in textual form. For this purpose, some systems (Gray, Bapty, and Neema 2000), including ATOM^3 use the Object Constraint Language OCL `<www.omg.org>` used in the UML.

Figure 2 depicts the structure we propose for a meta-modelling environment. ATOM^3 was initialized using a hand-coded Entity-Relationship meta-meta-model. As the Entity-Relationship formalism can be described in an Entity-Relationship model, the environment could be bootstrapped.

3.1 Meta-Modelling FSA in ATOM^3

ATOM^3 is a tool written in Python `<www.python.org>` which uses and implements the meta-modelling concepts presented above.

The main component of ATOM^3 is the Kernel. This module is responsible for loading, saving, creating and manipulating models, as well as for generating code. By default, a meta-meta-model is loaded when ATOM^3 is invoked. This meta-meta-model allows us to model meta-

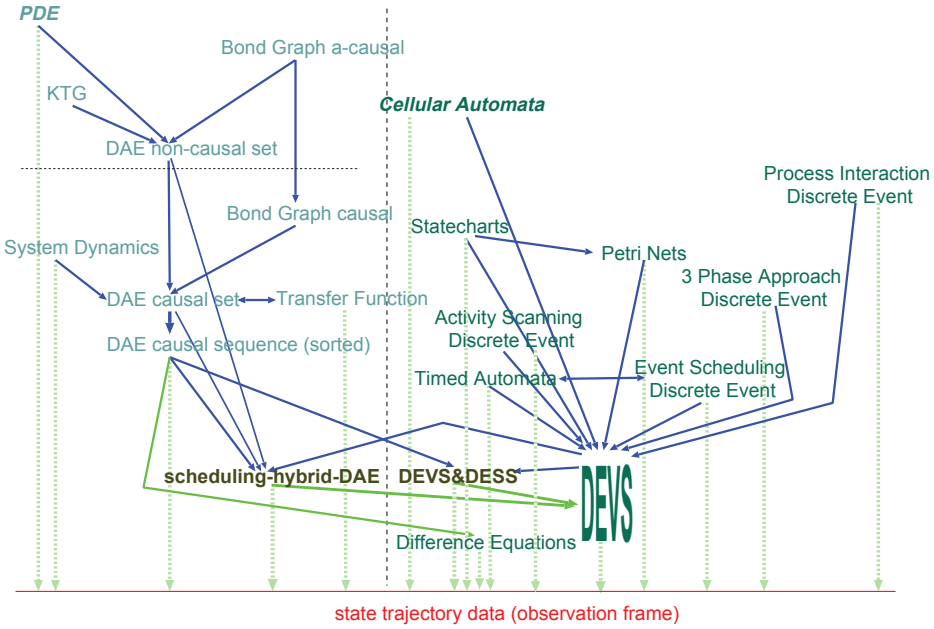


Figure 1: Formalism Transformation Graph

Table 1: Meta-Modelling Levels

Level	Description	Example
Meta-Meta-Model	Model used to specify modelling languages	Entity-Relationship Diagrams, UML class Diagrams.
Meta-Model	Model used to specify simulation models	Finite State Automata, Ordinary differential equations (ODE).
Model	The description of an object in a certain formalism	$f'(x) = -\sin x, f(0) = 0$ (in the ODE formalism)

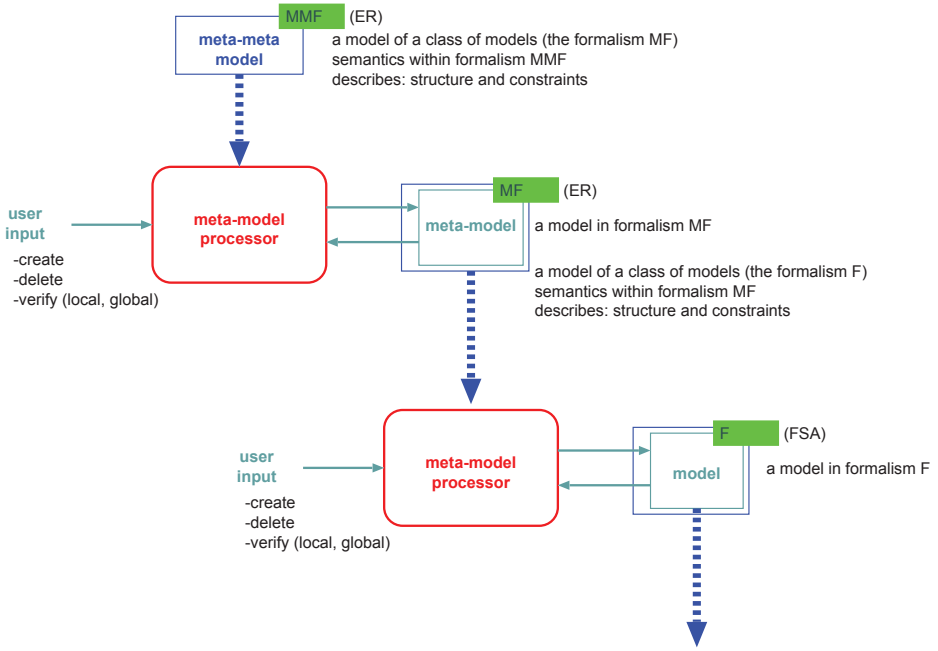


Figure 2: Proposed Working Scheme for a Meta-Modelling Environment

models (modelling formalisms) using a graphical notation. For the moment, the Entity-Relationship formalism extended with constraints is available at the meta-meta-level. When modelling at the meta-meta-level, the entities which may appear in a model must be specified together with their attributes. We will refer to this as the semantic information. For example, to define the Deterministic Finite Automaton Formalism, it is necessary to define both States and Transitions. Furthermore, for States we need to add the attribute name and type (initial, terminal or regular). For Transitions, we need to specify the condition that triggers it. This is shown in Figure 3. Note how the "current" entity is present to facilitate model simulation.

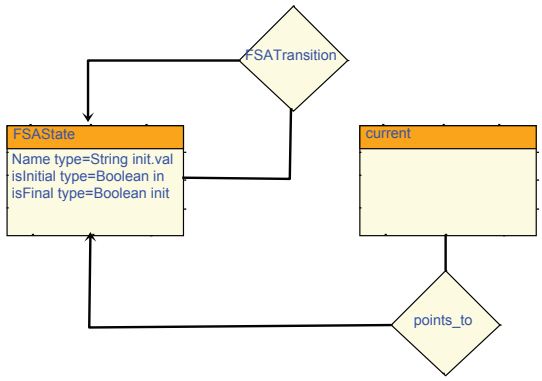


Figure 3: FSA Meta-Model (ER+Constraints)

In general, in ATOM^3 we have two kinds of attributes: *regular* and *generative*. Regular attributes are used to identify characteristics of the current entity. Generative attributes are used to generate new attributes at a lower meta-level. The generated attributes may be generative in their own right. Both types of attributes may contain data or code for pre and post conditions. Thus, in our approach, we can have an arbitrary number of meta-levels as, starting at one level, it is possible to produce a generative attribute at the lower meta-level and so on. The meta-chain ends when a model has no more generative attributes. Attributes can be associated with individual model entities as well as with a model as a whole.

Many modelling formalisms support some form of coupled or network models. In this case, we need to connect entities and to specify restrictions on these connections. In our finite automaton example, States can be connected to Transitions, although this is not mandatory. Transitions can also be connected to States, although there may be States without incoming Transitions. In ATOM^3, in principle, all objects can be connected to all other objects as well as to themselves. Usually, a meta-meta-model is used to specify/generate constraints on these connections. Using an Entity-Relationship meta-meta-model, we can specify *cardinality* constraints in the relationships. These relationships will generate constraints on object connection at the lower meta-level.

The above definitions are used by the Kernel to generate the Abstract Syntax Graph nodes. These nodes are Python classes generated using the information at the meta-meta-level. In the meta-meta-model, it is also possible to specify the graphical appearance of each entity of the lower meta-level. This appearance is, in fact, a special kind of generative attribute. For example, for the Deterministic Finite State Automaton, we have chosen to represent States as circles with the state's name inside the circle, and Transitions as arrows with the condition on top. That is, we can specify how some semantic attributes are displayed graphically. We must also specify connectors, that is, places where we can link the graphical entities. For example, in Transitions we have specified connectors on both extremes of the arc and in States on 4 symmetric points around the circle. Further on, connection between entities is restricted by the specified semantic constraints. For example, a Transition must be connected to two States. The meta-meta-model generates a Python class for each graphical entity. Thus, semantic and graphical information are separated, although, to be able to access the semantic attributes' values both types of classes (semantic and graphical) have a link to each other.

Using the FSA meta-model information, ATOM^3 allows editing of syntactically correct FSA models. As an example, an FSA recognizing even binary numbers (the input is a sequence of 0s and 1s) is shown in Figure 4. Actually, this figure already shows the first step of a graph grammar specified FSA simulator (asking the user for the input sequence).

3.2 Graph Grammars

Both for modelling and simulation purposes it is necessary to transform (structure as well as attributes) of models. The Graph Grammars formalism (Dorr 1995) allows us to *model* model transformations. In analogy to string grammars, graph grammars can be used to describe graph transformations, or to generate sets of valid graphs. Graph grammars are composed of rules, each mapping a graph on the left-hand side to a graph on the right-hand side. When a match is found between the left-hand side of a rule and a part of an input graph, this subgraph is replaced by the right-hand side of the rule. Rules may also have a condition that must be satisfied in order for the rule to be applied, as well as actions to be performed when the rule is executed. A rewriting system iteratively applies matching rules in the grammar to the graph, until no more rules are applicable.

The use of a model (in the form of a graph grammar) of graph transformations has some noteworthy advantages over an implicit representation (embedding the transformation computation in a program) (Blonstein, Fahmy, and

Grbavec 1996). The main advantages using a model of graph transformations can be summarized as follows:

- It is an abstract, declarative, high level representation.
- The theoretical foundations of graph rewriting systems may assist in proving correctness and convergence properties of the transformation tool.

On the other hand, the use of graph grammars is constrained by efficiency. In the most general case, subgraph isomorphism testing is NP-complete. However, the use of small subgraphs on the left hand side of graph grammar rules, as well as using node labels and edge labels can greatly reduce the search space.

Since we store simulation models as graphs, it is possible to express the transformations shown in the FTG as well as simulation-transformations at a meta-level, as graph grammars.

In a graph grammar, entities are labelled with numbers. In our case, entities are both states and transitions. If two nodes in a left hand side and a right hand side have the same number, that means that the node must not disappear when the rule is executed. If a number appears in a LHS but not in a RHS, that means that the node must be removed when applying the rule. If a number appears in a RHS but not in a LHS, that means that the node must be created if the rule is applied.

For subnode matching purposes, we should specify the value of the attributes of the nodes in the LHS that will produce a matching. In all the rules in our case, all the attributes have the value of $\langle ANY \rangle$, which means that any value will produce a matching. It is also needed to specify the value of the attributes once the rule has been applied and the LHS has been replaced by the RHS. This is done by specifying attributes in the RHS nodes of the rule. If no value is specified, and the node is not a new node (the label appears in the LHS), by default it will keep its values. This is denoted by $\langle COPIED \rangle$. It is also possible to calculate new values for attributes, and we certainly must do this if a new node is generated when replacing the LHS by the RHS.

Here, we describe the operational semantics of the FSA formalism by means of a Graph Grammar model. Executing the Graph Grammar on a particular FSA model and input segment will result in a simulation trace. The graph grammar is depicted in Figure 5. The user constructs an FSA model in AToM^3 as shown in the even binary number recognizer model of Figure 4. As an initial action, the user is prompted to provide an input segment ("010" in this case). Rule 1 identifies the initial state of the model, creates a "current" node and lets it point to the initial state. Subsequently, Rules 2 and 3 "match" a transition pattern (between different states or in a self-loop respectively) and the current input and re-write the model appropriately. The simulation continues until no more rules match. This process is depicted in Figure 6 for the even binary number recognizer FSA example with "010" as input segment.

4 XML REPRESENTATION

By default, AToM^3 saves models as Python scripts. This removes the need to parse the model as this is taken care of by the Python interpreter. Formalism-specific syntax checks are done upon reading. They are based on information from the (ER + constraints) meta-model of the formalism the model being loaded is expressed in.

For model exchange and re-use purposes (possibly over the WWW), a neutral format is required. XML seems the most viable candidate. XML is a base-language for expressing arbitrary, structured data in text form. It consists of several modules: core syntax, meta-syntax, linking, style-bindings, …Of these only the core syntax is common to all XML applications. Applications can choose to omit the other modules if they don't need them. All attributes in XML are by default string valued, although the meta-syntax is able to restrict that.

As depicted in Figure 7, it is sufficient to hand-craft a lexical analyser and parser for "core XML" syntax as described by Bert Bos in <http://www.w3.org/XML/9707/XML-in-C>.

The core XML grammar in pseudo-EBNF is given below:

```
document: prolog element misc*;
prolog: VERSION? ENCODING? misc*;
misc: COMMENT | attribute_decl;
attribute_decl: ATTDEF NAME attribute+ ENDDEF;
element: START attribute* empty_or_content;
empty_or_content:
 SLASH CLOSE | CLOSE content END NAME? CLOSE;
content: (DATA | misc | element)*;
attribute: NAME (EQ VALUE)?;
```

Literals are appropriately defined in the lexical specification <http://www.w3.org/XML/9707/scanner.l>. The full Bison specification is available at <http://www.w3.org/XML/9707/parser.y>

The core XML syntax accepts any nested combination of <tag> ...</tag> or < ... /> constructs. Rather than using DTDs or XMLSchema to specify formalism-specific valid syntax, we propose a more expressive ER+constraints model. Note how such a meta-model (of formalism F) is a model in its own right and can be saved in XML format. One a model's syntax has been checked, it can be transformed (simplified, simulated, converted into another formalism, …). Transformations are again modelled, most likely in the Graph Grammar formalism. Transformation models can again be saved and shared in XML format. Note how transformation specifications such as XSLT are

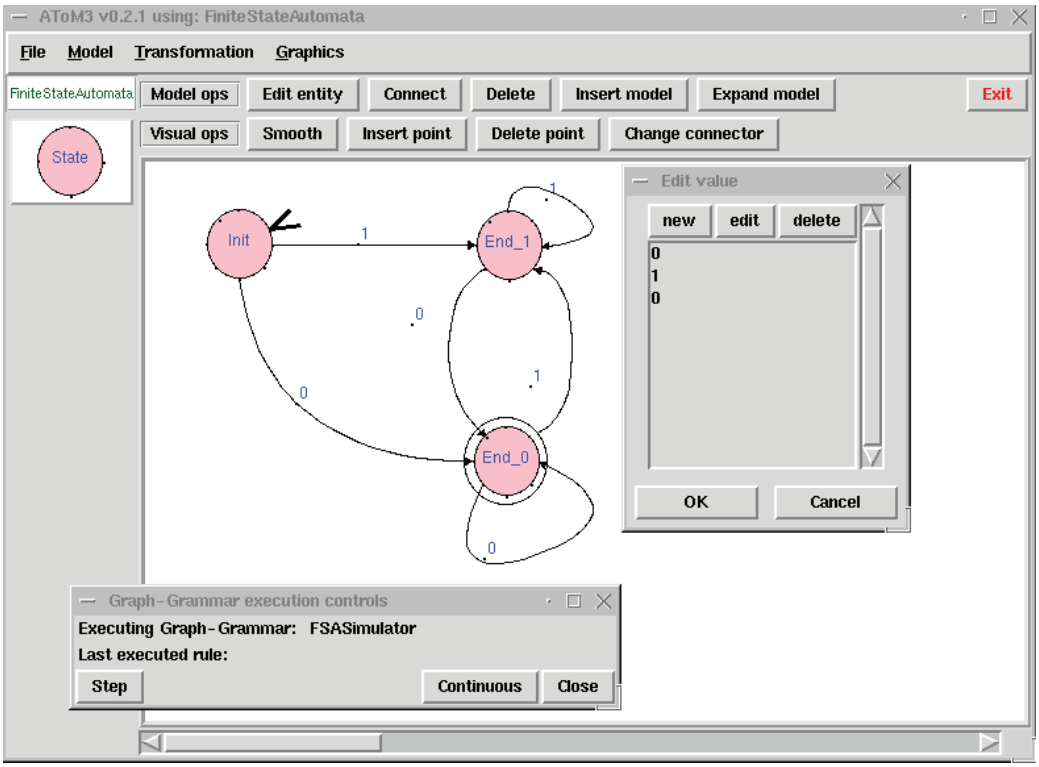


Figure 4: FSA Model, Binary Digit Sequence Input

easily expressed in this framework. Ultimately, we need to save the model under study in the neutral XML format. Output generation is nothing but another transformation and can as such again be explicitly modelled. The whole scenario is depicted in Figure 7. The approach can easily be extended to read and write custom modelling languages such as Modelica (Elmqvist et al. 1999).

5 CONCLUSIONS

In this article, we have presented an approach for modelling complex systems. Our approach is based on Meta-Modelling and Multi-Formalism modelling, and is implemented in the tool ATOM^3. This code-generating tool, developed in Python, relies on graph grammars and meta-modelling techniques and supports hierarchical modelling.

The advantages of using such an automated tool for generating customized model-processing tools are clear: instead of building the whole application from scratch, it is only necessary to specify – in a graphical manner – the kinds of models we will deal with. The processing of such models can be expressed by means of graph grammars, at the meta-level. Our approach is also highly applicable if we want to work with a slight variation of some formalism, where we only have to specify the meta-model for the new formalism and a transformation into a "known" formalism (one that already has a simulator available, for example). We then obtain a tool to model in the new formalism, and are able to convert models in this formalism into the other for further processing.

A side effect of our code-generating approach is that some parts of the tool have been built using code generated by itself (bootstrapped): one of the first implemented features of AToM^3 was the capability to generate code, and extra features were added using code thus generated.

As there is a need for a neutral model exchange format, we have presented XML and its integration in the meta-modelling framework.

ATOM^3 is being used to build small projects in a Modelling and Simulation course at the School of Computer Science at McGill University.

ACKNOWLEDGMENT

Prof. Vangheluwe gratefully acknowledges partial support for this work by a National Sciences and Engineering Research Council of Canada (NSERC) Individual Research Grant.

REFERENCES

Blonstein, D., H. Fahmy, and A. Grbavec. 1996. Issues in the practical use of graph rewriting. *Lecture Notes in Computer Science* 1073:38–55.

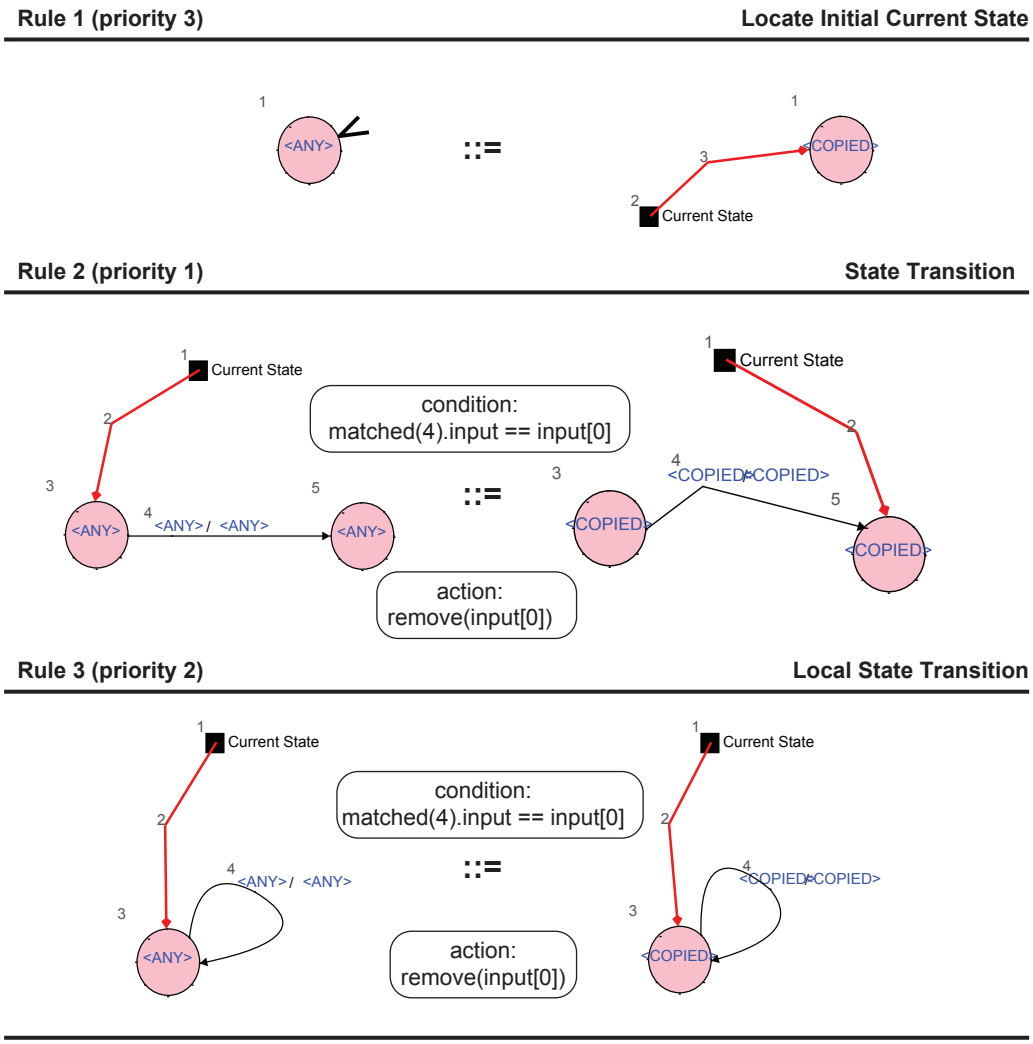


Figure 5: FSA Simulator Model (GG)

Dorr, H. 1995. Efficient graph rewriting and its implementation. *Lecture Notes in Computer Science* 922.

Elmqvist, H., B. Bachmann, F. Boudard, J. Broenink, D. Brück, T. Ernst, R. Franke, P. Fritzson, A. Jeandel, P. Grozman, K. Juslin, D. Kågedal, M. Klose, N. Loubère, S.-E. Mattson, P. Mosterman, H. Nilsson, M. Otter, P. Sahlin, A. Schneider, H. Tummescheit, and H. Vangheluwe. 1999. Modelica – a unified object-oriented language for physical systems modeling: Tutorial and rationale. Report, The Modelica Design Group. `<http://www.modelica.org/>`.

Gray, J., T. Bapty, and S. Neema. 2000. Aspectifying constraints in model-integrated computing. In *OOPSLA Workshop on Advanced Separation of Concerns*. Minneapolis, MN.

Honeywell 1999. DOME guide. `<http://www.htc.honeywell.com/dome/>`. Honeywell Technology Center, Honeywell. version 5.2.1.

Vangheluwe, H. L. 2000. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *IEEE International Symposium on Computer-Aided Control System Design*, ed. A. Varga, 129–134: IEEE Computer Society Press. Anchorage, Alaska.

Vangheluwe, H. L., J. de Lara, and P. J. Mosterman. 2002. An introduction to multi-paradigm modelling and simulation. In *Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems)*, ed. F. Barros and N. Giambiasi, 9 – 20. Lisboa, Portugal.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modelling and simulation: Integrating discrete event and continuous complex dynamic systems*. Second ed. Academic Press.

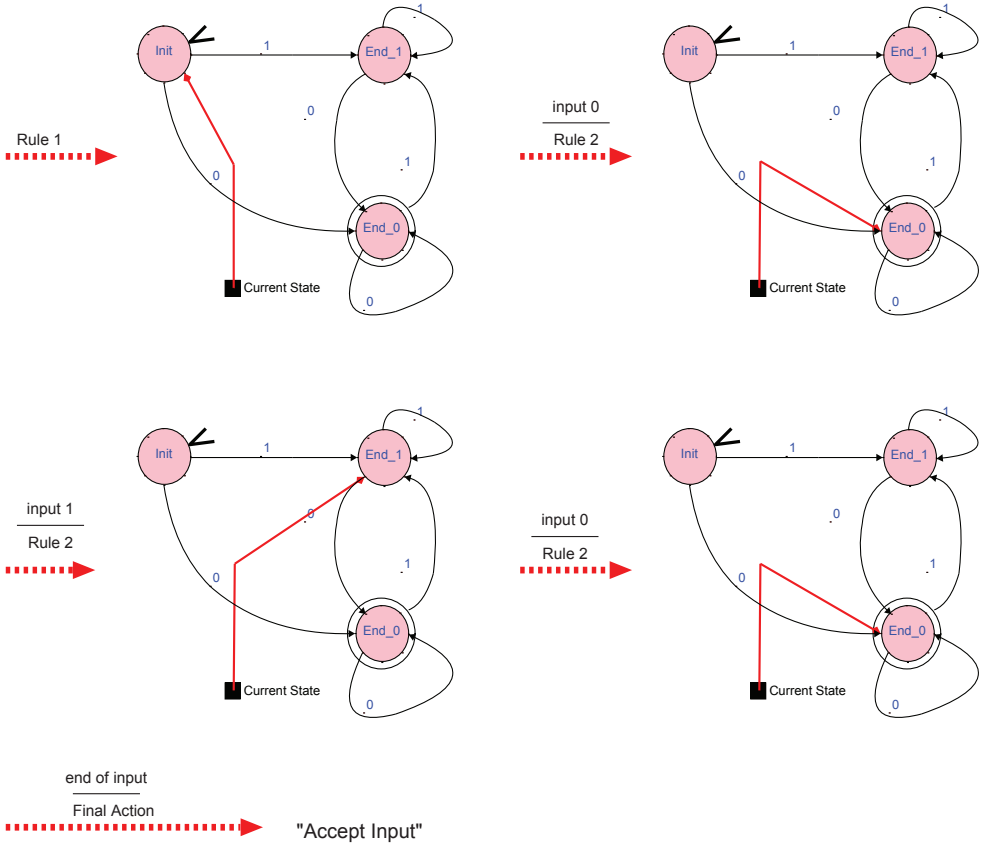


Figure 6: Simulation Steps (Rule Matches Due to Input)

AUTHOR BIOGRAPHIES

HANS VANGHELUWE, D.Sc. is an Assistant Professor in the School of Computer Science at McGill University, Montréal, Canada where he teaches Modelling and Simulation, as well as Software Design. He also heads the Modelling, Simulation and Design Lab (MSDL). He has been the Principal Investigator of a number of research projects on the development of a multi-formalism theory for Modelling and Simulation. Some of this work has led to the WEST++ tool, which was commercialised for use in the design and optimization of Waste Water Treatment Plants. He was the co-founder and coordinator of the European Union's ESPRIT Basic Research Working Group 8467 "Simulation in Europe", and a founding member of the Modelica Design Team. His current research is focused on the development of the AToM^3 tool for Computer Aided Multi-Paradigm Modelling (CAMPaM) and simulation. His e-mail address is `<hv@cs.mcgill.ca>`, and his web page is `<www.cs.mcgill.ca/~hv>`.

JUAN DE LARA is an Assistant Professor at the Universidad Autonóma (UAM) de Madrid in Spain, where he teaches software engineering, as well as modelling and simulation. His research interests include Web Based Simulation, Meta-Modelling, distance learning, and social agents. He recieved his PhD in June 2000 at UAM. He graduated in 1994 with a Top of Class Award as a Technical Engineer in Computer Science. In 1996 he received the honour of Higher Engineer in Computer Science. During 2001, as a post-doctoral researcher in the MSDL, he created the AToM^3 prototype. His e-mail address is `<Juan.Lara@ii.uam.es>`, and his web page is `<www.ii.uam.es/~jlara>`.

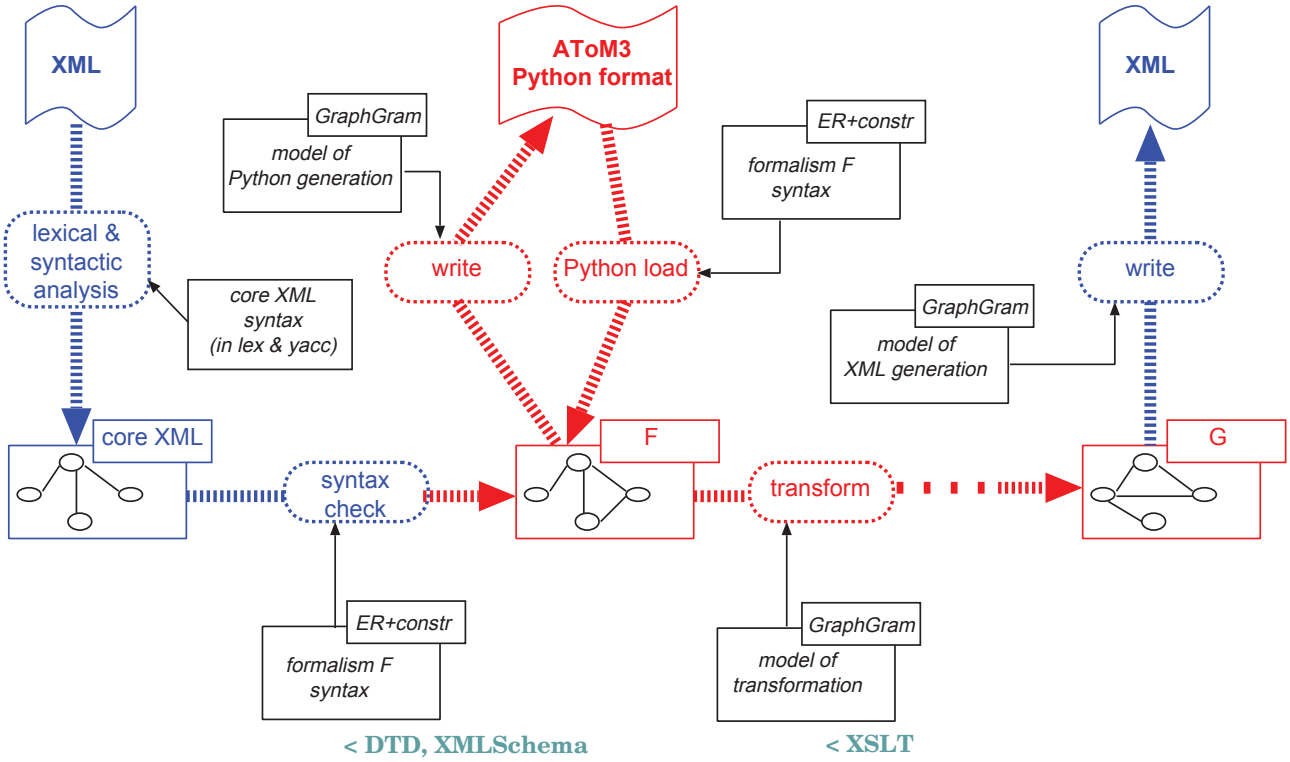


Figure 7: XML and Meta-Modelling

Lecture 12. Multi-Resolution Simulation

Papers to study for the exam

- Y. Yilmaz, A. Lim, S. Bowen, & T. Ören. Requirements and Design Principles for Multisimulation With Multiresolution, Multistage Multimodels. In: S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, eds. Proceedings of the 2007 Winter Simulation Conference. IEEE, 2007. pp. 823-832.
  Retrieved from: http://www.informs-sim.org/wsc07papers/097.pdf

- Mamadou D. Seck, H. Job Honig. Multi-perspective modelling of complex phenomena. Comput Math Organ Theory (2012) 18:128–144. DOI: 10.1007/s10588-012-9119-9
  Open access: https://link.springer.com/content/pdf/10.1007%2Fs10588-012-9119-9.pdf

Proceedings of the 2007 Winter Simulation Conference
S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, eds.

REQUIREMENTS AND DESIGN PRINCIPLES FOR MULTISIMULATION WITH MULTIRESOLUTION, MULTISTAGE MULTIMODELS

Levent Yilmaz
Alvin Lim
Simon Bowen

Computer Science and Software Engineering
107 Dunstan Hall, Auburn University
Auburn, A.L. 36849, U.S.A.

Tuncer Ören

M&SNet: Ottawa Center of the MISS
SITE, University of Ottawa
Ottawa, ON K1N 6N5, CANADA

ABSTRACT

The significance of simulation modeling at multiple levels, scales, and perspectives is well recognized. However, existing proposals for developing such models are often application specific. The position advocated in this paper is that generic design principles for specifying and realizing multiresolution, multistage models are still lacking. Requirements for simulation environments that facilitate multiresolution multistage model specification are introduced. A multimodel specification formalism based on graph of models is suggested along with design precepts to enable flexible dynamic model updating. The notion of multisimulation is introduced to enable exploratory simulation using various types of multimodels.

1 INTRODUCTION

Multilevel and multistage modeling is highly interdisciplinary, with progress occurring independently across fields (Davis 2000; Takahashi et al. 2002). Yet, disparate efforts involving the design and development of various types of multimodels result in incoherent solutions that lack a common unifying framework. This proposal is partially based on the rationale depicted with a taxonomy (Yilmaz and Ören 2004) of discrete-event based multimodel methodology that is equally applicable for continuous and discontinuous-change models expressed in ordinary differential equations and hence explores the viability of realization of a unified multimodel formalism. The premise is that, by developing a generalized multimodel formalism that can be configured with alternative strategies one can instantiate various types of multimodels within the same model development framework. Such a strategy is in sharp contrast with the current trend in developing formalisms for different modeling needs, where distinct solutions for multiple representations of scale, resolution, and staging are defined. More importantly, the availability of ensemble of models encapsulated within a multimodel will facilitate exploratory experimentation with computational models, the lack of which is a significant limitation in supporting discovery-based science. The following characteristics of simulations exacerbate the design challenges for modeling and simulating entities at multiple levels of resolution.

- For most realistic problems, the nature of the problem changes as the simulation unfolds. Initial parameters, as well as models can be irrelevant under emergent conditions. Relevant models need to be identified and instantiated to continue exploration. Manual exploration is not cost effective and realistic within a large problem state space (Yilmaz and Tolk 2006).
- Our knowledge about the problem being studied may not be captured by any single model or experiment (Bankes 1999). Instead, the available knowledge is viewed as being contained in the collection of all possible modeling experiments that are plausible given what is known and what is learned.
- Dealing with uncertainty is paramount to analyzing complex evolving phenomena (Davis 1988). Adaptivity in simulations and scenarios is necessary to deal with emergent conditions for evolving systems in a flexible manner.
- As simulations of complex phenomena are more and more used to aid intuition, dynamic run-time simulation composition with exploratory simulation will help identify (re)solution strategies that are flexible, adaptive, and robust (Yilmaz and Tolk 2006).
- In symbiotic simulation (Fujimoto et al. 2002) as well the problems advocated by the NSF DDDAS program, information about the actual performance and accuracy characteristics of the system are acquired during actual simulation rather than before. Run-time model and simulation updating

1-4244-1306-0/07/$25.00 ©2007 IEEE

can be used to advance the symbiotic online simulation field, where models, as well as parameter spaces may evolve.

- In a recent IEEE Spectrum article (Guizzo 2004) on UrbanSim, urban planners look decades ahead, experts conclude as follows: "…. ideally, stakeholders should be able to see computer-generated streetscape animations of **different** scenarios—what it would be like to walk in the neighborhood or commute to work, for example. It would be easier then for people to visualize the effects of **alternative** development policies and projects…."

As a result, multifaceted, multiparadigm (Zeigler and Ören 1986; Fishwick and Zeigler 1992), multiresolution modeling (Davis 2000) and cross resolution consistency management in multiresolution simulations (Reynolds et al. 1997) are explored to layout observations and the conceptual foundations to better understand the challenges in developing advanced simulation modeling infrastructures that can deal with uncertainty and multilevel phenomena. However, there are a number of limitations of existing proposals:

1. The lack of fundamental requirements and design principles results in application specific realization of such models;
2. The lack of a unified and coherent framework that enables development of various types of multimodels results in inconsistent and ad hoc development;
3. The lack of dynamic model and simulation updating and branching limits the potential of run-time exploration using simulation.

The purpose of this paper is to (1) reiterate the rationale for using novel simulation modeling such as multimodels, and multiresolution modeling, (2) present a comprehensive taxonomy of multimodels, and (3) layout fundamental requirements in developing simulation modeling infrastructures that enable multimodel design. A graph-based multimodel specification formalism is presented along with its basic syntax and semantics to facilitate developing simulators for such multimodels.

The rest of the paper is organized as follows. In section 2 we overview existing work on multimodels, with special emphasis on limitations in dealing with the type of problems advocated in this section. Section 3 presents a comprehensive taxonomy of multimodels. Note however, this paper focuses on designing **M**ulti**R**esolution **M**ulti-**S**tage **M**ultimodels (MRMSM) and the use multisimulation to generate their behavior. In section 4 we elaborate on requirements for MRMSM. A model specification formalism is advocated in section to improve dynamic model and simulation updating, which is a critical and essential issue in MRMSM design. Section 6 elaborates on a selected subset of MRMSMs to suggest how models need to be designed to satisfy the requirements set forth in section 4. Section 7 introduces the multisimulation concept. Finally, section 8 concludes by summarizing the contribution of the paper.

2 BACKGROUND

Several projects have explored the issues involved in the construction of multiresolution models. Multiresolution modeling (MRM), originally proposed by Davis and Bigelow (2002) was concerned with resolving the representational and conceptual differences among models that are joined to satisfy a common objective. Davis (2000) builds his exploratory analysis methodology on MRM, which involves building a single model, a family of models, or both to describe the same phenomena at different levels of resolution (Davis and Bigelow 2002). One of the limitations of this approach as it pertains to type of problems depicted above is that models are statically defined at the design time. The level of granularity at which resolution levels are defined in the methodology is in terms of functions and their parameters. Updating the functional decomposition model at run-time by adding new branches or extending the depth of the tree is not part of the methodology. Furthermore, if there are multiple models that can be coupled to inputs of an aggregate model to obtain a high-resolution model, different models can be defined only through design time coupling. The analyst then selects a particular high-resolution model to experiment. While it is possible to instantiate each one of the models to perform multiple simulation experiments, one has to foresee in advance what high-resolution models are plausible during the design time. Extensibility and adaptivity is not engineered into the simulation. Hence, simultaneous exploration with multiple models based on run-time decisions is not considered.

Multi-resolution simulation entities need to be capable of simultaneously operating at different levels, while maintaining consistency at each level of abstraction. Challenges involved in assuring consistency across levels of resolutions are explored in (Reynolds et al. 1997). It has been argued that designing cross-resolution models requires principled design strategies that take potential discrepancies into account (Reynolds et al. 1997; Davis 2000). While the observations presented in (Reynolds et al. 1997) are generally applicable, the presented design strategy is overly simplistic. The design involves a Multi-resolution Entity (MRE) that encapsulates a specific number of entities, each with its own state information. Dynamically updating entities in a given resolution level is not addressed in the MRE design scheme. However, exploring Course of Actions (COAs) with different models require seamless update of individual components at selected resolution levels. Extension of the levels of resolution of an MRE is also not addressed. The tight coupling between the resolution levels makes extensibility difficult within the MRE design scheme. The notion of using a consistency enforcer for

Table 1: Taxonomy of Multimodels (Yilmaz and Oren 2004)

Based on	Additional Criteria				Type of multimodel (MM) (Synonyms are represented within parentheses)
Structure of submodels	**Number** of submodels active at a given time			Only one	Single aspect MM (Sequential MM)
				2 or more	Multiaspect MM
	Variability of structure (variability of number of submodels)	**Static**			Static-structure MM
		Dynamic (Dynamic-structure MM) (Variable-structure MM)	Number of submodels	Extensible	Extensible MM
				Depends on model's stage	Multistage MM
			Alterations of submodels	No	Non-mutational MM
				Yes	Mutational MM
					Evolutionary MM
Behavior (activation) of submodels	**Nature of knowledge** to activate submodels	Constraint-driven			Constraint-driven MM (Adaptive MM)
		Pattern-directed (Pattern-directed MM) (Metamorphic MM)	Sub-model selection is cyclic	No	Acyclic MM
				Yes	Cyclic MM
		Goal-directed			Goal-directed MM (Exploratory MM)
	Location of knowledge to activate submodels	Within the MM (Internal activation of submodels)			Active MM (Internally activated MM)
		Outside the MM (External activation of submodels)			Passive MM (Externally activated MM)

consistency management across resolution levels is a useful attribute of the MRE design strategy.

3 MULTIMODELS

Many real-world phenomena can not be modeled by one single model; rather, they require the use of a set of complementary models that together are able to describe the whole process (Ören 1987, 1991; Zeigler et al. 2000; Yilmaz and Oren 2004). Basic definitions and brief explanations of the envisioned multimodel types shown in Table 1 follow:

A multimodel is a modular model that subsumes multiple submodels that together constitute the behavior of a complex multi-phased process. A multimodel encapsulates several aspects of reality (i.e., submodels) in one model. In single aspect models only one aspect of reality can exist at a given time (to be represented by an appropriate sub-

model) and transitions can occur from one submodel to another one under monitored conditions. As shown in Table 1, there are various design decisions in multimodel design: *Based on the completeness of submodels*, there are two cases: One can know all (2 or more of) the submodels at the beginning i.e., at modeling stage. Yet, there can be emergent conditions where the need for additional submodel(s). *Based on the number of active submodels*, one needs to consider two cases: (1) Only one submodel is active at a given time or (2) two or more submodels are active at a given time. Simultaneous existence of two or more model components would facilitate simulation of multiple aspects of the phenomena under study. Multiresolution models are considered in this paper as a specific case of multiaspect multimodels. Multistage multimodels are viewed as dynamic structure models, in which the number of submodels depends on the stage of the problem.

Based on the location of information necessary for the activation of submodels there are two cases: the necessary information can be (1) within the submodels or (2) it can be external to submodels. The transitions between submodels can be goal-directed (goal directing the submodel transition rule and goal-directed submodel transition mechanism should be specified) or pattern-directed. Nature of information necessary for the activation of submodel(s) entails the selection conditions of a submodel, where there are also two cases: (1) we know the laws of affecting model transition. *Pattern-directed activation* entails a meta-pattern to guide (1) selection of known submodels and (2) request of new submodels corresponding to an interruption of the simulation gaming using the recommender.

4 REQUIREMENTS FOR MRMSM DESIGN

MRMSM encapsulates a set of components/submodels that represent a system or phenomena at different levels of abstraction possibly at multiple phases. Since such an entity can simultaneously operate at multiple levels of resolution, a number of submodels can be active at a time. Shifts in the phase of a problem trigger model updating by changing the active set of entities. For the sake of simplicity we assume that the number of entities needed in a MRMS model is known in advance.

Requirement 1- An MRMSM should be decoupled from the knowledge regarding how its submodels are composed, created, and represented.

This requirement is critical to facilitate that the MRMSM can be configured with one of multiple families of submodels representing an entity at different levels of resolution for a specific stage of the problem. Furthermore, this requirement imposes the constraint that a family of submodels is designed to work together, and that the MRMSM should be independent of how these submodel families are created and composed. This requirement enables isolating implementation of submodels from the MRMSM.

Requirement 2 - The effects of concurrent interactions at multiple levels of resolution must be combined consistently (Reynolds et al. 1997).

MRMSMs constitute multiple submodels that co-exist during simulation. That is, for instance, a low-resolution and high-resolution entity within the model may simultaneously interact with other external entities at corresponding levels of resolution. Consistency problems between different resolution levels might emerge if shared resources are not used mutually exclusively. Concurrent access to such resources needs to be controlled and coordinated.

Requirement 3 – The state of entities at different levels of resolution within a MRMSM should be consistent.

In an MRMSM the state of the aggregated low-resolution entities should be updated as the state of the corresponding high-resolution entities change. This requires definition of multiple one-to-many dependencies between entities so that when one object changes state, its dependents are notified and updated automatically.

Requirement 4 – In an MRMSM the entities should be allowed to alter their behavior when their internal states are changed.

MRMSM behavior depends on the state and/or stage of the problem, and it must change its behavior at run-time depending on that state/problem phase. Therefore, an MRMSM should enable localization of state-specific behavior and partition behavior for different stages. MRMSM should enable the definition of family of related protocols, encapsulate each one, and make them interchangeable to facilitate varying the protocols independently of the entities that are configured with them.

Requirement 5: The constraints regarding when and under what conditions (1) the consistency of the elements of families of submodels in a multiresolution model be enforced and (2) a shift in the stage of the problem be triggered must be independent of the MRMSM.

Corresponding to the time path of the change of a problem should be a time path of the appropriate submodel families. But, the question is what should be the sequence of this shift pattern of models of family of? Or should there be trigger mechanisms indicating when a shift should occur? This requirement suggests that the constraints be separated from the MRMS model and not be intertwined with the en-

tities to facilitate reuse of the submodel families in a different context with different constraints

Requirement 6 - At the time of the model update, the new set of low-resolution and high-resolution entities and their dependent components must be dynamically loaded and linked into the run-time environment of the simulation.

This requires new model and simulator decoupling strategies that avoid persistent connections to facilitate extensibility. Also, the intricate details of complex model instantiation process should be as independent of the MRMSM as possible to enable flexible update.

Requirement 7 - The consistency of entities undergoing (re)placement needs to be preserved.

The event scheduling and simulation protocol need to be restricted or regulated to facilitate interleaving of entity replacement activities with the simulation events.

Requirement 8 - The state of an MRMSM must be constructed, or at least continue from a specific state after an update operation.

This requires externalization through abstraction, state saving, transmission, and reconstruction after the update operation.

Requirement 9 – An MRMSM should have a mechanism for changing the structure and behavior of the model dynamically. In other words, an MRMS model should support its modification a priori.

This requirement suggests that MRMSMs need to provide facilities that establish a self-representation of the model, offer means by which this representation can be updated, and assure that the manipulations to the self-representation influence the behavior of the model.

Requirement 10 – An MRMSM should provide flexibility in terms of defining behavioral resolution to facilitate analysis that are independent of an implementation and programming language specific constructs.

Component-based (Zeigler et al. 2000 – System Entity Structure), function-based (Davis 2000), and attribute-based (Reynolds et al. 1997) approaches to resolution refinement are course-grain approaches. However, flexible what-if analysis requires analysts to select the behavioral resolution at arbitrary levels of granularity without relying on implementation specific constructs. That is, a high-level *behavioral formalism* that enables *fine-grain stepwise refinement* is needed to specify the behavior of an entity at multiple levels of resolution. This facilitates fine-tuned control of a simulation at run-time at different levels of resolution. By having capabilities to control the scope of update, analysts can perform what-if analyses at arbitrary levels.

5 MULTIMODEL SPECIFICATION

While there exist efforts in conceptualizing multiresolution modeling to characterize system behavior at different levels of aggregation and abstraction, a unified and uniform model specification formalism that expedites the realization of the above requirements is missing. The question is what particular model design specification approach is conducive to handling the above requirements. For instance, most of the requirements above need facilities to explicitly specify the (1) transition conditions between model families, (2) cross resolution consistency management, and (3) actions needed for dynamic model replacement and seamless configuration update.

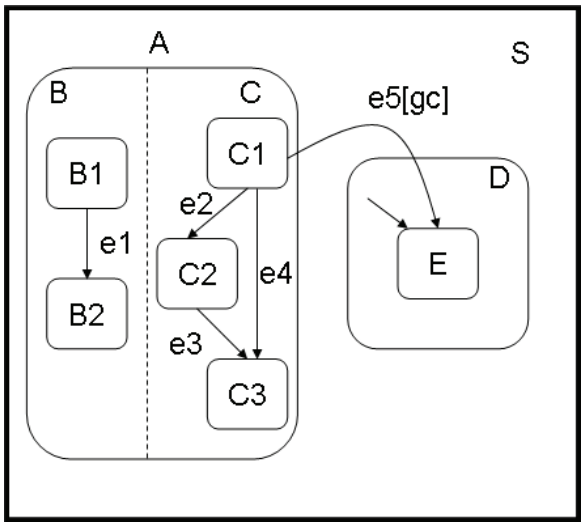


Figure 1: Multimodel Structure

Figure 1 depicts the structure of a multimodel *S*, which has submodels *A* and *D*. At first glance, the diagram can be interpreted as a statechart. However, in our context each node designates a submodel or model component, whereas arcs denote transitions between models. Each transition indicates the event that trigger the activation of a target node, the conditions under which it is activated, and the actions taken to complete the change in the configuration. Submodel A, in Figure 1, can be a multiresolution model with two concurrent levels, each one of which is represented by submodels *B* and *C*. Note that the level of decomposition can be arbitrary by extending the hierarchy. Similarly, submodels *B* and *C* can represent two different aspects of the same subsystem represented by model A. The submodel *D* and alternative configurations of models within *A*

facilitate representing distinct stages of the problem. The simulation of a multimodel is based on the interpretation of the model reachability structure depicted in Figure 2. The structure is an *and-or graph* that has two types of nodes. The *and-nodes* require every descendant node to be active. On the other hand, only one descendant node of an or-node is active, while the rest are dormant. In Figure 2, the model S is represented as an *or-node*, as either *A* or *D* is active at any point in time during the simulation. The submodel *A*, however, is an *and-node*, as both *C* and *D* can be active. The submodels *B* and *C* are *or-nodes*. This requires either *B1* or *B2* (basic nodes of *B*) to be latent within submodel B. Similarly, at most one of the submodels of C can be active at any point of time.

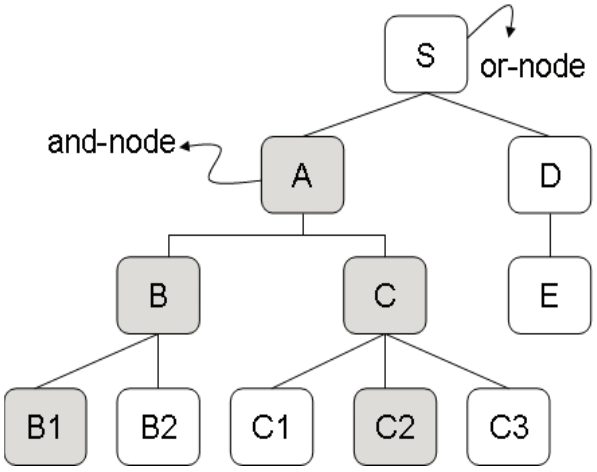


Figure 2: Multimodel And-Or Graph

Definition: For every pair of nodes, their **scope** is the lowest common ancestor, e.g., *scope (B1, E)= S.*

Definition: A pair of nodes is **orthogonal** if their scope is an or-node.

Definition: The **configuration** of a multimodel S is a minimal subtree *CFG(S)* that satisfies

- $S \in CFG(S)$
- If $A \in CFG(S)$ and A is an or-node then *CFG(S)* contains exactly one of the submodels of A.
- If $A \in CFG(S)$ and A is an and-node then *CFG(S)* contains all of the submodels of A.

The **basic configuration** is the set of basic nodes in a configuration, which uniquely identifies a multi-model configuration. Simulating a multimodel requires the definition of its state and semantics of the transitions that facilitate the generation of the its behavior. The state of a multimodel consists of the following elements.

- CFG - A configuration with respect to the multimodel under consideration.
- TR - History information – last visited submodel.
- CND- Assignment of all atomic conditions.
- TM – Set of all scheduled timeout events at the current state.
- E – The subset of atomic events.

A transition takes place to change the configuration of a multimodel to switch to a new stage, insert new set of submodels, update the level of resolution and fidelity. A transition, (*M1, <eventExpr, condExpr, actions>, M2*), from model M1 to M2 is triggered or enabled when

- $M1 \in X.CFG$
- $eventExpr \in (X.E \cup X.TM)$
- $X.CND \prec condExpr$

Once a transition is enabled, its execution results in the change of the configuration of the multimodel. To specify the change, we need to introduce the semantics of a transition. Given a transition $t = (R, < E, C, A >, P)$ at a status *X,* let $S = scope(R, P)$. The set of nodes that are exited as a result of the transition is defined as $EXIT(t) = X.CFG(S) - \{S\}$. The set of nodes that depict the submodels that are activated is defined as $ENTRANCE(t) = CFG(S, P)$. This set of nodes refers to the configuration emerging from S (excluding S) and then including P and then expanding by initial node labels. When an enabled transition transforms the configuration of a multimodel the following steps take place.

- A new construction is created by replacing the source submodel M1 with the target submodel, M2.
- The entrance conditions associated with M1 (e.g., *in(M1)*) become false, whereas the entrance conditions associated with M2 (e.g., *in(M2)*) become true.
- The events *exit(M1)* and *enter(M2)* are added to the new status.
- The actions specified in the transition are executed. These actions need to address the requirements listed in section 4.

Designing multimodels based on the above formalism in terms of communicating and interacting objects and classes that are customized to deal with the requirements listed in section 4 is a significant challenge. The next section shows how the principled use of a number of design patterns may

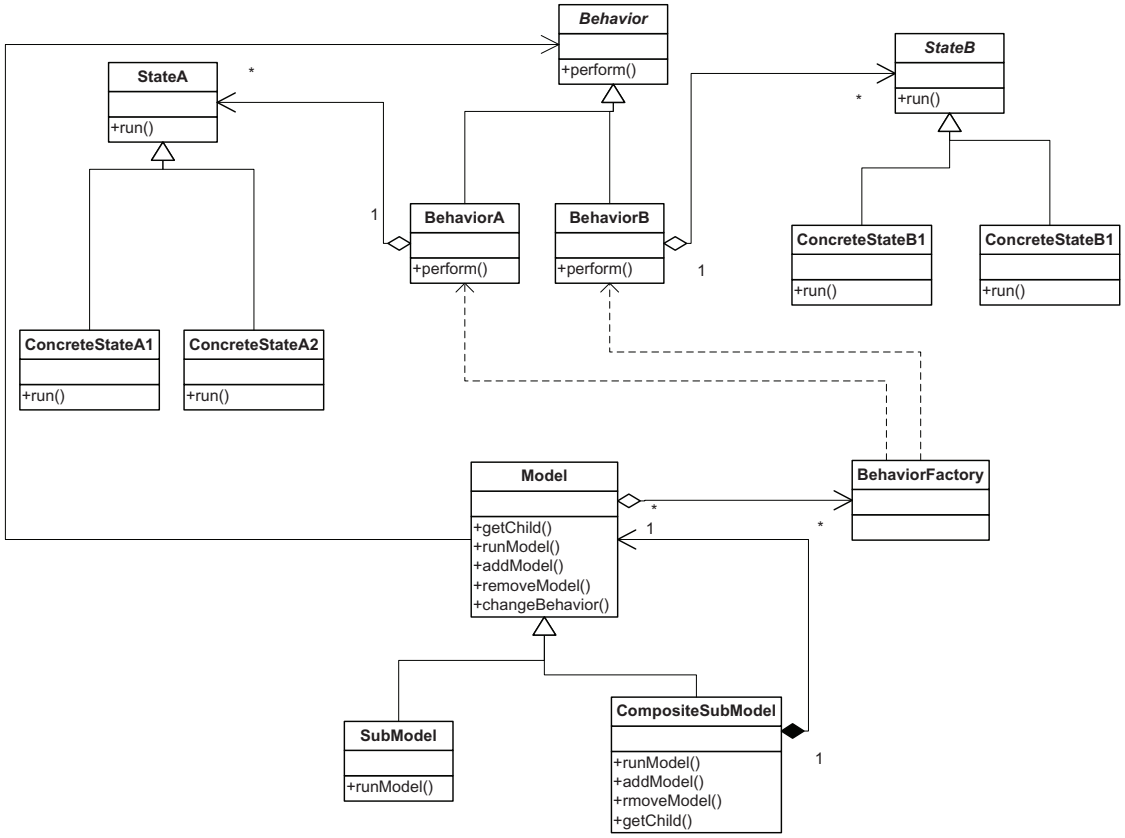


Figure 3: A Partial and Preliminary Design Strategy for Multimodels

facilitate the realization of a selected requirement with the above formalization in mind

6 DESIGN PRINCIPLES FOR MRMSM

The specification formalism discussed in the previous section requires composing objects into tree structures to represent part-whole hierarchies. Furthermore, each submodel needs to be treated uniformly regardless of whether it is a composition (e.g., multiresolution) or an individual submodel. Multiresolution or multistage models can be elements of a coarse granular models, which operate at multiple levels of resolutions and stages themselves. Hence, it is critical to make sure that a multimodel simulator that uses the strategy discussed in section 5 ignores the differences between compositions of submodels and submodels or individual components. The design structure between *Model*, *CompositeSubModel*, and *Submodel* aims to achieve this objective. Specifically, the *Model* defines the interface for objects in the composition, which realize the default behavior. The *Submodel* represent those nodes in the composition that have no children nodes. The *CompositeSub model* defines behavior for submodels that have multiple levels of resolution or aspects.

A significant requirement as discussed in the previous section is to *decouple MRMS from the knowledge regarding how its submodels are composed, created, and represented*. The *BehaviorFactory* component facilitates the creation of families of related or dependent submodels or model components without explicitly specifying the concrete realizations. For instance, referring to Figure 1, the structure and behavioral representation of submodel *B* can be encapsulated within the *BehaviorB* component of Figure 3. As a result, the MRMS model (i.e., *Model*) is independent of how submodel *B* is instantiated. This information is encapsulated within *BehaviorB*. That makes it possible configure the MRMS model with one of several submodel families. Furthermore, since the related set of submodels and their components are designed to be used together, the decoupling of MRMS from *BehaviorB* enables enforcing this constraint. Finally, this configuration makes it easy to exchange submodel families easier.

The existence of *State* components associated with *Behavior* entities aims to allow a submodel to alter its behavior when its internal configuration changes (i.e., *sub-*

model C starts using *C2* as opposed to *C1* after the configuration of the multimodel is updated). As a result, the submodel will appear to change its subcomponents. That is, the submodels behavior depends on the configuration of the multi-model, and the submodel needs to change its behavior at run-time depending on the new configuration. The consequence of this design strategy is that configuration-specific behavior and partitions behavior for different configurations. This makes the model transition concept presented in section 5 explicit.

7 MULTISIMULATION WITH MRMSM

Multisimulation (or multisim, for short) is simulation of several aspects of reality in a study (Ören 2001). It includes simulation with multimodels, simulation with multiaspect models, and simulation with multistage models. *Simulation with multimodels* allows computational experimentation with several aspects of reality; however, each aspect and the transition from one aspect to another one are considered separately. (In special cases, multimodels can be metamorphic models or evolutionary models). *Simulation with multiaspect models* (or multiaspect simulation) allows computational experimentation with more than one aspect of reality simultaneously. This type of multisimulation is a novel way to perceive and experiment with several aspects of reality as well as exploring conditions affecting transitions. While exploring the transitions, one can also analyze the effects of encouraging and hindering transition conditions. *Simulation with multistage models* allows branching of a simulation study into several simulation studies; each branch allowing to experiment with a new model under similar or novel scenarios.

Requirement 11 – Multisimulation with multimodels, multiaspect models or multistage models needs mechanisms to decide when and under what conditions to replace existing models with a successor or alternative.

Staging considers branching to other simulation studies in response to scenario or phase change during experimentation. Graphs of model families may facilitate derivation of feasible sequence of models that can be invoked or staged. More specifically, a graph of model families can be used to specify alternative staging decisions. Each node in the graph depicts a model, whereas edges denote transition or switching from one model to another. Figure 4 depicts the components of the abstract architecture of a possible multisimulation engine.

A meta-simulator is a scheduler that generates staged composition of models by traversing the model stage graph and coordinates their simulation and staging within distinct simulation frames. Each frame simulates a distinct subset of models derived from the model stage graph. Note however, not all staged compositions are feasible or useful. Hence, the meta-simulator needs to consult with the model recommender before model staging to determine if emergent trigger or transition condition in the simulation is consistent with the precondition of the model to be staged. More than one model in a family can qualify for staging; in such cases separate simulation frames need to be instantiated to accommodate and explore plausible scenarios. Given a collection of models (or more generally, family of models), a stage graph can be generated automatically by an optimistic approach that connects every available node (model) to every other node within the domain of problem. The edges in a model stage graph denote plausible transitions between models as the problem shifts from one stage to another.

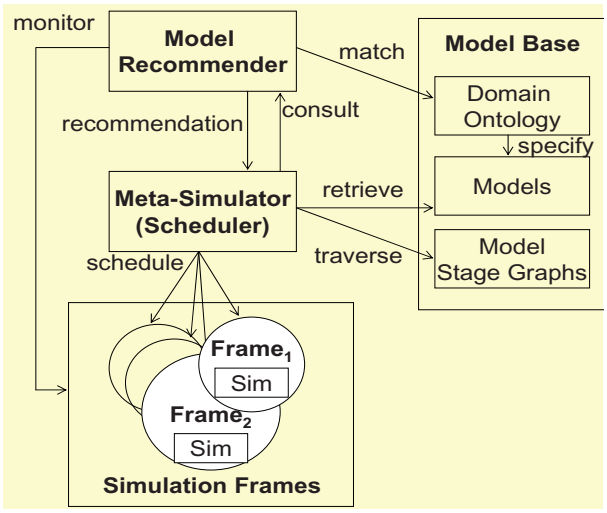


Figure 4: Components of the Multisimulation Engine

Model recommendation in multisimulation can simply be considered as the exploration of the model staging space that can be computed by reachability analysis of the graph. There are two modes for the usage: (1) Offline enumeration of paths using the graph and performing a staged simulation of each model in sequence one after the other, unless a model staging operation becomes infeasible due to conflict between the transition condition and the precondition of the successor model and (2) run-time generation of potential feasible paths as the simulation unfolds. In both cases, an online model recommender plays a key role to qualify a successor model. The first case requires derivation of sequence of models using a traversal algorithm. The edges relate families of models. Therefore, the actual concrete models, the preconditions of which satisfy the transition condition need to be qualified, since transition to some of these model components may be infeasible due to conflict between a candidate model and inferred situation. Identifying such infeasible sequences is computationally intractable; otherwise, it would have been possible to de-

termine if the conjunction of two predicates is a tautology by using a polynomial time algorithm.

Experience in component-based simulation paradigm, however, indicates that for most model components preconditions are simple. Hence, it is possible to eliminate some models that violate the transition condition. For the remaining possible transitions it is possible to select one of the three strategies: (1) omit all difficult qualification conditions, (2) decide on an edge by edge basis which specific models of a model family to include, and (3) include all difficult edges. Omitting all difficult associations between transitions and model preconditions is conservative. This strategy excludes all infeasible models. The cost is the exclusion of some feasible edges. Hand-selecting those associations between transition conditions and models facilitate inclusion of feasible models. Yet, the costs involved with this level of accuracy are the potential human-error and effort needed to filter out infeasible models. Choosing to include all difficult associations is liberal, in that it ensures inclusion of all feasible models. The cost is the inclusion of some infeasible models, hence the inclusion of some undesirable staged compositions that enforce models to be simulated even when their qualification conditions are violated. Nevertheless, it is possible to screen out such models using an online model recommender.

Candidate models and associated simulations can be maintained by *focus points*. A focus point manages branch points in the simulation frame stack. Suppose that a goal instance (i.e., stage transition condition) is at the top of the stack. If only a single model qualifies for exploration, then it is pushed onto the stack. Yet, if more than one model matches the condition, a simulation focus point is generated to manage newly created simulation branching (discontinuity) points, each one of which have their own contexts. When a path is exhausted, the closest focus point selects the next available model to instantiate the simulation frame or return to the context that generated the focus point. As simulation games are explored, a network of focus points is generated. Determining which focus point should be active at any given time is the responsibility of the meta-scheduler. When more than one model is qualified, then scheduler needs to decide which one to instantiate. Control rules can inform its decision. Three steps involve in deploying a new simulation frame in such cases: matching, activation, and preference. The matching step should both syntactically and semantically satisfy the request. The activation step involves running a dynamic set of rules that further test the applicability of models with respect to contextual constraints. Finally, the preference steps involve running a different set of rules to impose an activation ordering among the active frames.

8 CONCLUSIONS

The purpose of this paper is to (1) reiterate the rationale for using novel simulation modeling such as multimodels, and multiresolution modeling, (2) present a comprehensive taxonomy of multimodels, and (3) layout fundamental requirements in developing simulation modeling infrastructures that enable multi-model design. The paper presented a set of such requirements along with preliminary design precepts that have potential to realize emergent challenges. Furthermore, a graph-based multimodel specification formalism is presented along with its basic syntax and semantics to facilitate developing simulators for such multimodels. The notion of multisimulation is introduced to enable exploratory simulation using various types of multimodels.

REFERENCES

Bankes, S. 1999. Tools and techniques for developing policies for complex and uncertain systems. In *Proc. of the National Academy of Sciences, Colloquium*, Vol. 99, suppl. 3, pp7263-7266.

Davis, K. P. 2000. Exploratory analysis enabled by multiresolution, multiperspective modeling. In *Proceedings of the 2000 Winter Simulation Conference*. ed. Jeffrey A. Joines, Russell R. Barton, K. Kang, and Paul A. Fishwick., 2000. Available from RAND as RP-925.

Davis, K. P and Bigelow, H. J. 2002. *Motivated metamodels: Synthesis of cause-effect reasoning and statistical modeling*, The RAND Corporation, Santa Monica, CA.

Davis, K. P. 1988. *The role of uncertainty in assessing the NATO-PACT central region balance, RAND N-2839,* The RAND Corporation, Santa Monica, CA, 1988. (also available in General Accounting Office, NATO-Warsaw Pact Conventional Force Balance, GAO/NSIAD-89-23B, 1988.

Fishwick, P. and Zeigler P. B. 1992. A multimodel methodology for qualitative model engineering. *ACM Transactions on Modeling and Simulation*, 2(1) pp. 52-81.

Fujimoto, R., D. Lunceford, E. Page, A. and Uhrmacher L. 2002. Technical report of the Dagstuhl-seminar on grand challenges for modelling and simulation.Available via <http://www.informatik.uni-rostock.de/~lin/GC/index.html> [last accessed March 14, 2007].

Guizzo E. (2004). "Software and the city," *IEEE Spectrum*, Available via <http://www.spectrum.ieee.org/WEBONLY/wonews/dec03/1203urb.html> [last accessed March 14, 2007].

Reynolds, F. P., Natrajan, A., Srinivasan S. 1997. Consistency maintenance in multiresolution simulation. *ACM Transactions on Modeling and Simulation*, 7(3) pp. 368-392.

Ören, T.I. 1987. Model update: A model specification formalism with a generalized view of discontinuity," In *Proceedings of the Summer Computer Simulation Conference*, Montreal, Quebec, Canada, 1987 July 27-30, pp. 689-694

Ören, T.I. 1991. Dynamic templates and semantic rules for simulation advisors and certifiers," In *Knowledge-Based Simulation: Methodology and Application*, P.A. Fishwick and R.B. Modjeski (Eds). Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 53-76.

Ören, T.I. (2001 – Invited contribution). Towards a Modelling Formalism for Conflict Management. In: Discrete Event Modeling and Simulation: A Tapestry of Systems and AI-based Theories and Methodologies. H.S. Sarjoughian and F.E. Cellier (eds.), Springer-Verlag, New York, pp. 93-106.

Takahashi, K., Yugi, K., Hahimoto, K., Yamada, Y., Pickett, C., and Tomita, M. 2002. Computational challenges in cell simulation. *IEEE Intelligent Systems* 17:64-71.

Yilmaz, L. and Ören, T.I. 2004. Dynamic model updating in simulation with multimodels: A taxonomy and a generic agent-based architecture. In *Proceedings of SCSC 2004 - Summer Computer Simulation Conference*, July 25-29, 2004, San Jose, CA., pp. 3-8.

Yilmaz, L. and Tolk A. 2006. Engineering ab initio dynamic interoperability and composability via agent-mediated introspective meta-simulation. In *Proceedings of the 2006 Winter Simulation Conference*, pp. 1075-1082. Monterey, CA. December 3-6 2006.

Zeigler, P. B. and Ören, T.I. 1986. Multifaceted. multiparadigm modeling perspectives: Tools for the 90's. In *Proceedings of the 1986 Winter Simulation Conference*. pp. 708-712.

Zeigler, B. P., Praehofer, H. Kim, G. T. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, Academic Press.

AUTHOR BIOGRAPHIES

LEVENT YILMAZ is assistant professor of Computer Science and Software Engineering in the College of Engineering at Auburn University. Dr. Yilmaz earned his Ph.D. and M.S. degrees from Virginia Tech. His research interests are on advancing the theory and methodology of simulation modeling, agent-directed simulation to explore dynamics of socio-technical systems, organizations, and human/team behavior, and education in simulation modeling. Dr. Yilmaz is a member of ACM, IEEE Computer Society, Society for Computer Simulation International, and Upsilon Pi Epsilon.
URL: <http://www.eng.auburn.edu/~yilmaz>

ALVIN LIM is associate professor of Computer Science and Software Engineering in the College of Engineering at Auburn University. Dr. Lim received his Ph.D. from University of Wisconsin-Madison. His research interest are self-organizing networks, wireless mobile networks, high performance networks, mobile computing and databases, reliable and dynamically reconfigurable distributed systems, complex software development, parallel processing, and performance measurements and analysis. In self-organizing networks, he is developing new techniques and protocols that allow network nodes to form spontaneous networks, assemble the network themselves, dynamically adapt to device failure and degradation, manage movement of network nodes, and react to changes in task and network requirements. In wireless mobile networks, he is investigating adaptive communication protocols for large-scale mobile information systems and the performance implications of these protocols.
URL: <http://www.eng.auburn.edu/~lim>

SIMON BOWEN is a graduate student at Auburn University. He earned his B.S at Auburn and will be getting his M.S. in Computer Science in 2007. He is currently doing research with Dr. Yilmaz and Dr. Lim in multi-resolution multi-stage modeling and design. His research interests are in software design and architecture and simulation modeling.

TUNCER ÖREN is a professor emeritus of Computer Science at the University of Ottawa. His current **research** activities include (1) advanced M&S methodologies such as: multimodels, multisimulation, and emergence; (2) agent directed simulation; (3) cognitive simulation (including simulation of human behavior by fuzzy agents, agents with dynamic personality and emotions, agents with perception, anticipation, and understanding abilities); and (4) reliability and quality assurance in M&S and user/system interfaces. He has **contributed** in Ethics in simulation (http://www.scs.org/ethics/), M&S Body of Knowledge, and multilingual M&S dictionaries http://www.site.uottawa.ca/~oren/SCS_MSNet/simDic.htm . Currently he is a member of the Board of Directors, AVP for Ethics, and Founding Director of the McLeod Modeling and Simulation Network of the Society for Modeling and Simulation International (SCS). He has over 350 **publications,** mostly on M&S (some translated in Chinese and German) and has been active in over 350 **conferences** and seminars held in 29 countries. He is **recognized** by IBM Canada as a Pioneer of Computing in Canada (2005).
URL: <http://www.site.uottawa.ca/~oren/>

Comput Math Organ Theory (2012) 18:128–144
DOI 10.1007/s10588-012-9119-9

Multi-perspective modelling of complex phenomena

Mamadou D. Seck · H. Job Honig

Published online: 28 April 2012
© The Author(s) 2012. This article is published with open access at Springerlink.com

Abstract This conceptual paper discusses the limitations of a single-perspective hierarchical approach to modelling and proposes multi-perspective modelling as a way to overcome them. As it turns out, multi-perspective modelling is primarily a new methodology, using existing modelling techniques but extending the modelling hierarchy with a new epistemological level which integrates the different perspectives. The methodology will be presented in some detail, and its use will be demonstrated by analyzing an example taken from a socio-political context.

Keywords Multi-perspective modelling · Complex systems modelling · Aspect models · Bridge models

1 Introduction

As evidenced from the growing number of publications on social simulation and rising research interests, modelling and simulation is increasingly applied to the study of social phenomena. The domain of 'social systems' deals with higher levels of objective complexity (Bailly and Longo 2003) in comparison with the artificial and physical systems which have traditionally benefited the most from modelling and simulation as a method of inquiry. Hence, the '*complexity*' paradigm emerged to deal with the new class of systems. In this paper we will look at multi-perspective modelling to address the problems that come with modelling complex systems.

M.D. Seck (✉) · H.J. Honig
Department of Technology, Policy and Management, Delft University of Technology, Jaffalaan 5, 2628 BX Delft, The Netherlands
e-mail: m.d.seck@tudelft.nl

H.J. Honig
e-mail: h.j.honig@tudelft.nl

The remainder of this paper will first set the stage by providing a background on modelling complex phenomena. Following that, it will introduce multi-perspective modelling as a way to overcome some of the problems of traditional reductionist modelling, and show where it fits into the modelling hierarchies as defined by Zeigler (Zeigler et al. 2000) and Klir (1985). We will interpret perspectives as (theoretical) positions from which we look at the world, providing both a deliniation of the system and a way to decompose it, a definition general enough to encompass levels in the sense of Lane (2006). The paper continues with a formalization of multi-perspective models. Following that, an example taken from a socio-political context will be analyzed within the extended framework; the example shows how different levels can be integrated in a single model. The paper ends with a discussion of the results and directions for further work.

2 Modelling complex phenomena

A common way to understand systems is to impose on them a hierarchical structure by isolating parts and defining relations between them; parts are further aggregated into higher level parts in a way that aims to minimize the relations between formed composite parts (Wimsatt 2000). For example, from a structural perspective we may decompose a technical system like the International Space Station (ISS) into a hierarchy of parts and subparts. The same could be done from a functional perspective. This approach has proved to be tremendously successful. In fact, it was so successful that it has led people to claim that hierarchy is the '*architecture of complexity*' (Simon 1962) and apply it to non-technical systems as well. Simon furthermore argues that by decomposing systems in a hierarchical way, our depiction of reality does not suffer too much, and hence he refers to systems that can be modeled this way as '*nearly-decomposable*'. According to this vision, it is possible and even desirable to analyze all systems like 'Chinese boxes', where an entity contains smaller entities, with the latter smaller entities containing even smaller entities, and so forth. From Simon onwards, hierarchy theory developed as a sub-field of systems theory and has had a broad influence in modelling (Pattee 1973). Hence, current modelling and simulation approaches inherited a strong reductionist world view. For instance, the DEVS formalism (Zeigler et al. 2000) uses atomic and composite models to represent systems.

As non-technical systems usually have higher complexity than technical systems, they are often studied at different levels, each level corresponding to a specific time and space scale. This led Lane to point out that different types of hierarchy exist (Lane 2006), of which the *level hierarchy* accounts for this case. Sometimes, the levels correspond to different scientific domains, like for example sociology and psychology, where one level corresponds to aggregates composed of units that are (at least partly) described by the lower level. Similarly, in a medical context the levels could be the body, the organs, the tissues making up these organs, and the individual cells making up the tissues. Again, each level consists of elements described at the lower level, but not fully defined by that level.

Yet another type of system occurs when a given element can be part of different hierarchies at the same time. For example, in a social system the unit of behavior

could be the individual, who can at the same time be part of a family, a company, a neighborhood, a religious group. Each of these different ecosystems can be described by its own (level) hierarchy, but all these hierarchies share the same low-level element, the individual in this case. This situation is referred to as a *tangled hierarchy* (Lane 2006).

When studying such systems, one has to set boundaries; one has to select amongst the different ecosystems or levels the parts that will be modeled. By doing so one adopts, at least implicitly, a theoretical perspective. The concept of a partitioning frame (Winther 2006) posits the existence of a *de facto* theoretical unit guiding the investigation and decomposition of a given system of interest. It is often neglected that when decomposing a system from a single perspective, the resulting hierarchical decomposition on a system is a simplifying abstraction that comes at a cost in terms of model expressiveness and richness (Agazzi 1991; Levins 2006), and that, by not explicitly recognizing the chosen perspective, one tends to overlook the limitations imposed by it. This may be of little consequence when one is modelling engineered systems like the aforementioned ISS, because such systems are a result of a design process that adopted the very same principles and hence resulted in systems that are clearly hierarchical, with a well-defined whole/part structure. In such cases perspectives have clear boundaries and their interactions are few, so usually the constraints they pose will be acceptable.

In contrast, modelling social systems as simple hierarchies will fail to answer all questions one might pose or lead to answers that are inaccurate. This is because the system's decomposition has necessarily lost relations between elements occurring in the real world; relations outside the partitioning frame, which will not be brought back to life when the behavior of the reconstructed system is studied: the latter system can only be a straightforward composition of lower level behaviours.

3 Overcoming the problems of single-perspective reductionism

The problem of modelling complex systems as a compositional hierarchy has already been posed in various ways. For instance, Lane showed the qualitative difference between inclusion and level hierarchies (Lane 2006). As an example, in systems biology, Uhrmacher discussed the difficulty of representing complex multi-level biological systems (Uhrmacher et al. 2007) using hierarchical modeling. By adding an organizational level hierarchy to the classic compositional hierarchy of DEVS (through ports and variables that apply throughout the composed model), her approach tackles some of the problems of reductionism; it sticks, however, to the unique decomposition paradigm. Dalle and colleagues (Dalle et al. 2008) proposed to enrich hierarchical modeling by introducing shared components in DEVS hierarchical models. However, Dalle's approach is not meant to model complex systems (e.g., social systems) that exhibit the tangled hierarchies that are common to such systems (Lane 2006).

Prevalent modeling approaches in social simulation are also not immune from the issue stated here. To tackle complexity, modularity is commonly seen as a desirable feature, and complex models are built by coupling sub-models which are each responsible for part of the total system's behavior. To understand such coupled models,

the property of being 'closed under coupling' is a very important one. It states that any composite model is equivalent to a corresponding atomic ('flattened') model expressed in the same formalism: no 'magic' results from composing lower level models (Zeigler et al. 2000). The standard methodology when creating a complex model is thus to decompose it into smaller and smaller parts until reaching a given unit of behavior generation. This decomposition effort is only possible through the adoption of an implicit or explicit theoretical perspective. Thus, ultimately, such models cannot be other than inclusion hierarchies, like 'Chinese boxes'.

A core tenet of the *complexity paradigm* is the departure from the above type of reductionism (Morin 1990). It emphasizes that, by analyzing a (natural) system into (artificial) parts, significant properties of the system will be irremediably lost (Mikulecky 2001). We understand complexity as a relational property between the observed system and the observer who partitions the system in a *de facto* imperfect way. The practical problem thus becomes not reductionism *per se*, but rather our misguided assumption that the parts of a system form an objective reality, instead of being the result of a certain abstraction or theoretical perspective imposed on the system. A parallel can be drawn with the role of theories, as in the following quote taken from (Popper 1959): "*Theories are nets cast to catch what we call 'the world': to rationalize, explain, and to master it. We endeavor to make the mesh finer and finer*."

4 Multi-perspective modelling

Before introducing a way to make the "mesh finer and finer" without weaving a new net, we will have a quick look at the modelling relation as defined by Rosen (2000), Mikulecky (2001). This modelling relation represents a bridge between two worlds: the natural world in which we live and which we try to understand, and the 'inner (mental) world' which represents our perceptions of the former (see also Rosen 1991, 2000). In the modelling relation, our perceptions of the world are called the *formal system*, because when performing the modelling task we somehow formalize the outside world. It is important to note, however, that whenever we conceptualize anything at all, we are defining such a formal system, no matter how imprecise (Mikulecky 2001). In contrast one may refer to the natural world as the *natural system*. The relationships between both worlds are depicted in Fig. 1, which contains both Rosen's and more specific terms that apply to modelling and simulation. In this figure, the conceptualization step is represented as *encoding*; it results in a formal system. The inverse step is shown as *decoding*; it interprets the formal system and tries to explain or predict the natural world on the basis of our understanding of the formal system. In simulation modelling, the encoding step corresponds to the construction of a simulation model; the decoding step corresponds to analysis and validation. Whereas the natural system evolves through *causality*, the formal system evolves through *implication* by following the rules that were defined during the formalization step. In simulation terms, the phenomena occurring in the natural system are modeled as events taking place in the model, causing the latter to evolve along a path that reflects the natural system's evolution.

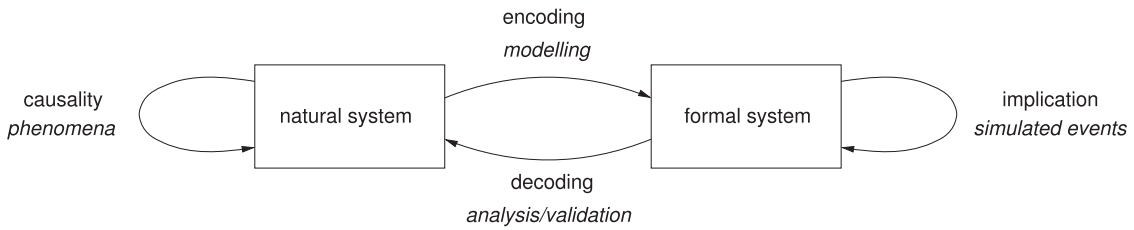


Fig. 1 Modelling simple phenomena

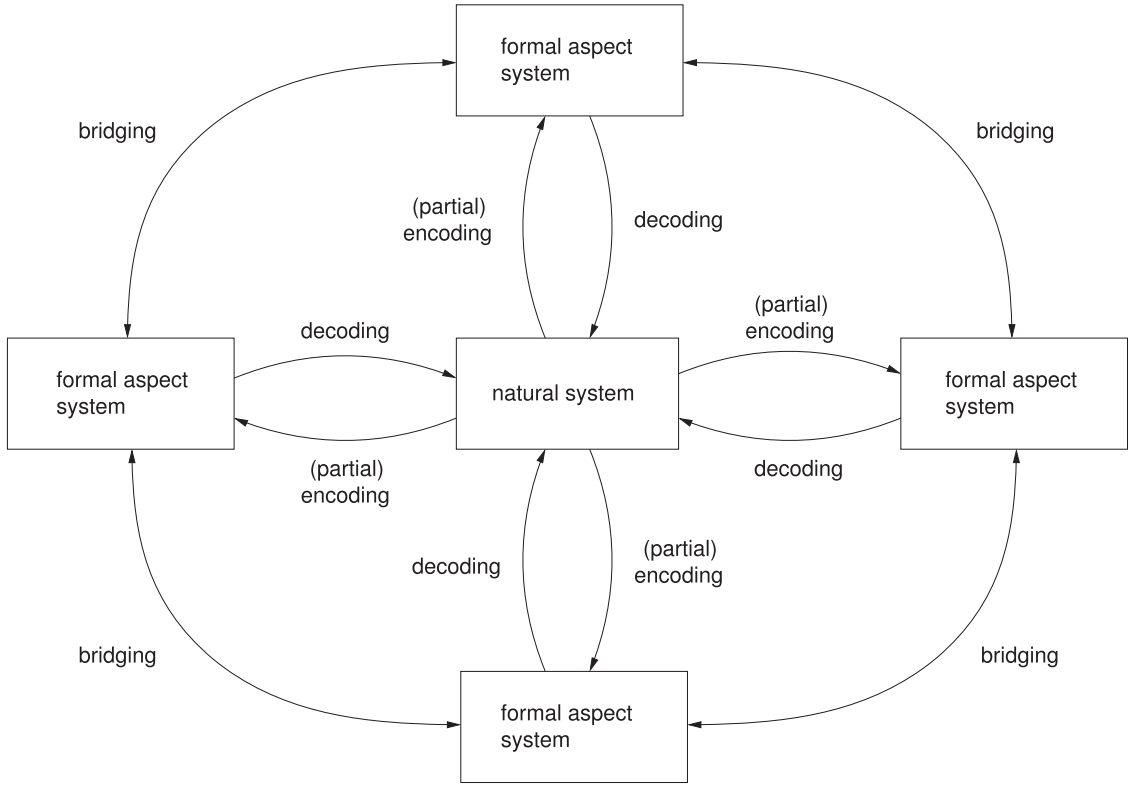


Fig. 2 Modelling complex phenomena through multiple perspectives

Now if the phenomena we try to model are complex, a reductionist formal system can only be partially successful in describing the natural system (Agazzi 1991; Mikulecky 2001). By describing a natural system as a collection of perspectives, though, where each perspective is associated with a unique formal system (having a unique decomposition) as shown in Fig. 2, we can model a system in an inherently 'richer' way by having multiple non-isomorphic decompositions that may influence each other. Such multi-perspective models can indeed capture the tangledness of the systems that result when we observe the world from different perspectives. As Morin puts it (Morin 1990), "we must found the idea of a complex system on a non-hierarchical concept of the whole" (Morin 1990). In a similar way, Levins (2006) proposes the robustness methodology, which, in a sort of triangulation, invites to analyze and model systems with multiple conceptually independent tools, thus improving accuracy of the models by relating the outcomes obtained from different perspectives.

The relation between complexity and multiple perspectives has been acknowledged by various authors. Kaufmann has stated that the number of possible theo-

retical perspectives and the resulting alternative decompositions of a system are an indication of its complexity (Kauffman 1970). Wimsatt's notions of descriptive and interactional system complexity further strengthen this view (Wimsatt 1972). His notion of *descriptive complexity* refers to the property of a system allowing various alternative and non-isomorphic decompositions, while *interactional complexity* refers to the amount of cross-cutting relations (with the action of defining them indicated as *bridging* in Fig. 2) between those alternative system decompositions (Wimsatt 1972). Wimsatt provides some examples at both extremes of descriptive and interactional complexity: a piece of granite decomposed into subregions of roughly constant chemical composition and crystalline form, under the perspectives of electrical conductivity, thermal conductivity, and density, is descriptively simple. If the system is decomposed according to each of these perspectives, the resulting models will be isomorphic. At the other extreme, a differentiated multi-cellular organism decomposed under anatomical, physiological, biochemical, etc., perspectives is descriptively and interactionally complex; the resulting model structures are non-isomorphic. Descriptive and interactional simplicity often characterize engineered systems, while descriptive and interactional complexities characterize evolved systems.

5 Connecting aspects

Compared to single-perspective modelling, building multi-perspective models requires additional effort: the management of alternative system perspectives as well as the identification of the bridges between them. These tasks are key to realizing the additional power of multi-perspective modelling. With aspect models covering different perspectives of the same natural system, the central question becomes how to connect the resulting non-isomorphic aspect models.

Different perspectives may correspond to the levels of a level hierarchy (Lane 2006), but they might also come from different theoretical perspectives at the same level. Either way, relations result from two components in the two decompositions sharing a common referent, like a parameter in one perspective being an observable variable in another perspective, or an atomic model in the first perspective having an aggregation/disaggregation relation with a coupled model in the second perspective, such that the atomic model constrains the structure of the coupled model, and the coupled model's components' outputs are used in the transition functions of the atomic model.

As an example, consider a single natural system and two formal systems (A and B) that each give a partial description (explanation) of the phenomena in the natural system. Now assume that in A we have a notion p, that might be implicit or explicit. Explicit means that we have defined p in our theory, but we are not able to predict its value inside A. Since we cannot predict it inside A, we can only fix its value or make it a parameter of the model.[1] Implicit means that p is not even present in the model, but

[1]The existing theory doesn't say anything about how the parameter may obtain its value; such knowledge is simply not present within the perspective. It does say, however, what are the results of changes in p's value for the aspect being modeled.

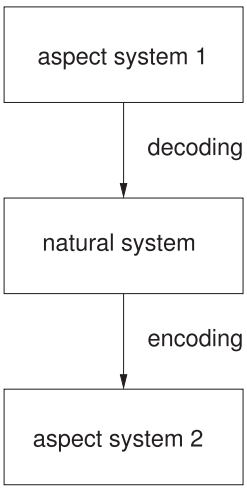


Fig. 3 Bridging involves decoding and encoding steps

its effects are still accounted for when computing A's outputs. Now if we have such a p, we might be able to predict its value using outputs from model B representing another perspective. Or, if there are no such outputs, we might know that there exist variables inside B that could be used to derive a value for p in A. In that case we could change B to add an observable variable β and create a *bridge model* that uses β to compute a value for p. Doing so will improve the quality of A's prediction of the value α, which can be validated. The term bridge model was chosen to indicate that they represent a theory about the relations between values in different aspect models. Bridge models are *not* a theory about the system under consideration, as such theories belong to the domain of the individual aspect models. Hence, bridge models cannot have inputs other than those that connect to some aspect model's output, nor outputs other than those that connect to some aspect model's input(s).

In order to identify common variables and relations we have to identify the terms p in A that we know to have a relationship with observable variables β in B. If p is implicit, it has to be made explicit first; if it is already explicit, it has to be made an input of A. Similarly for β, which might have been implicit at first, but has to be made into an output of B. The relation between the new input(s) and output(s) will usually be defined empirically. Note that in determining the variables and relations (which must have referents in the natural system whose behavior we can observe) we have to go to a decode/encode cycle as defined by the modelling relation (see Fig. 3, which represent the part of Fig. 2 that corresponds to a single bridge model). This decode/encode cycle avoids a problem noted by Agazzi: "…, a single term, used in two different disciplinary contexts, receives, within the two, different meanings and refers to different types of objects" (Agazzi 1991). Still, finding references and creating the bridge models involves transdisciplinary experts that are able to identify which properties of objects in one aspect model control (or are controlled by) other properties of other objects in another aspect model. Alternatively, statistical methods and data mining may be employed to find predictive relationships between variables belonging to different domains (Zwick 2004; Klir 1975, 1976). Whereas the former method would result in explanatory bridge models, the latter results in just predictive models.

source system	What are the inputs of the system, what are the (observed) variables.
data system	What is the input/output relation of the system
generative system	What states and state transitions explain the system's behaviour
structure system	What are the system's components and how are they connected
multi–aspect system	What are the relevant perspectives and how do they interrelate
metasystem	How does the system evolve over time

Fig. 4 Extended modelling hierarchy that includes aspect-models

6 Extending the modelling hierarchy

To clarify the position defended in this paper and to see how multi-perspective modelling relates to traditional simulation modelling, we place our discussion in the context of a very general yet powerful system concept that Klir and Zeigler independently proposed in the late 70's and the early 80's as "*epistemological system categories*" and "*levels of system specification*" respectively. These two concepts are essential components of the integrative system frameworks introduced by these authors and presented in detail in Klir (1985), Zeigler et al. (2000). The epistemological categories of a system framework are the classes of system it can represent. System categories are also called epistemological levels, because at each level, new knowledge is gained on the system of interest, which was not known at the lower levels. An overview of the various epistemological levels is given in Fig. 4. Note that this figure already includes the level of multi-aspect systems that will be introduced later.

In the context of an inductive system framework (Klir 1985), epistemological categories emerge from examples observed in reality. New examples or areas of interest that cannot be fully supported in the existing categories should lead to the definition of new categories. In this section, the category of '*multi-aspect systems*' will be added to Zeigler and Klir's frameworks. Aspect systems were already mentioned in Fig. 2, and are in fact also an explicit part of Zeigler's System Entity Structures (SES). However, when building models with DEVS, one aspect is chosen as the modeller's perspective; the other aspects are '*pruned*'. Hence, aspect models are no longer part of the epistemological levels defined by Zeigler's framework (Zeigler et al. 2000). To understand how aspect systems relate to the other categories, we will first describe the levels in some detail.

Basically, a system is "*conceived as a set of variables together with a relation recognized among their state sets*" (Klir 1985). System categories are implicitly defined by the type of relations that can be defined between their variables. The most basic system category defined by Klir is the '*source system*'; this level of system specification is referred to in Zeigler's framework as the *observation frame*. This category determines a system's boundary by defining its input and output variables, with their state and support sets. Thus defined, the source system embodies both the object of study as well as the pragmatic orientation and the theoretical perspective selected by the investigator, and can be radically different depending on the choices made.

Once the source system in known and defined, supplementing it with data brings about the next system category: the *data system*. This level consists in a collection of input/output pairs indexed by the support set, which, in modelling and simulation, is always *time*. Building a data system is a routine activity in modelling and simulation, as it is used to derive model parameters and because this is the only way of performing quantitative validation of simulation experiments.

The core of modelling and simulation occurs at the next category, that of '*generative systems*'. A simulation model can actually be defined as a data generation device. Time invariant functions are specified using the predefined observational variables and new internal variables which operationalize a theoretical language. Zeigler rightfully calls this level the *state transition level*. All simulation modelling paradigms and languages provide concepts and constructs to specify the state transition mechanism governing the dynamics of the system. The specification of a generative system depends on the source system that was previously defined for it. In cellular automata, each cell is a generative system (a finite state automaton) characterizing the relation between input variables (the state of neighboring cells) and one output variable (the state of the cell itself) over time. In agent based modelling, the modeller chooses a certain level of granularity, and defines (through behavioral rules) a mechanism that amounts to a generative system.

The next system category is the *structure system* level. This level specifies how systems defined in lower categories (generally generative systems) can be integrated in a compositional fashion. Cellular automata, coupled DEVS models, multi-agent models, and system dynamic models are all structure systems. They all specify relations between lower level model components.

Whereas most modelling stops at this level, Klir defines yet higher levels, which he calls *metasystems*, *meta-metasystems*, etc. (Klir 1976). The latter system categories are used to model evolving systems by specifying transitions between structure systems.

The proposal we make is to add a level that specifically addresses the limitations of adopting a single perspective. The new level is intended to be used to integrate different perspectives into the same model, and would allow us to model problems that, due to their complexity, cannot be solved within a unique theoretical framework (for example, a model that needs to integrate psychological and sociological perspectives in the same model). It should be noted here that a organizational level of the problem domain may be modeled by means of a perspective, but perspectives are not limited to represent organizational levels; they can also used to represent alternative decompositions corresponding to different theoretical or pragmatical perspectives at a comparable level of organization. In Yilmaz and Ören's taxonomy of multi-models (Yilmaz and Ören 2005), the proposed multi-perspective models fit in the class of multi-aspect model. Simulating such systems would amount to simulating the different perspectives in parallel, and allowing them to influence each other. While Vangheluwe (Vangheluwe and de Lara 2002) provides a platform for meta-modelling, he doesn't define an epistemological basis for the interaction between the different models. Nevertheless, his platform might prove very useful for building multi-perspective models using the methodology outlined in this paper.

The new category of *multi-aspect system* specifies how generative or structure systems defined in different perspectives may be integrated into a multi-perspective

system specification. To this end, two types of models will be distinguished: *aspect models* and *bridge models*. An aspect model is a model of a system according to a given perspective; it is a regular generative or structure model, in the sense that it has well defined observable input and output variables, and it is modeled according to a theory relevant to the problem being investigated. A multi-aspect system contains a finite number of such aspect models and defines the relations between them. To integrate the aspect models, bridge models are introduced. A bridge model maps outputs from one or more aspect models to some input in another aspect model. Bridge models play the same role as do the *bridge principles* of Nagel (1961) and Hempel (1965) in theory reduction; as bridge models reason about the relations between different theoretical or pragmatic frameworks, they are indeed at the meta-level when compared to the lower levels. This has important implications for the methodology of building such models: one could say that aspect models specify all of the pre-theorized knowledge in the multi-perspective model, while bridge models represent the not-yet theorized part of the models. As will be shown in the section about methodology, bridge models can only be understood through decoding and encoding steps which relate the models through their common referent (which is part of the natural system being modeled). This extra decoding/encoding step is what distinguishes this layer from the preceding one, the structure system.

7 Formalization of multi-perspective models

Our formalization of multi-perspective models will be derived from the standard formalization of coupled models in DEVS (Zeigler et al. 2000), which defines a coupled model as a tuple containing a set of input events and a set of output events, two sets containing subcomponents and their names, three sets containing external input couplings, external output couplings, and internal couplings. In multi-perspective models, the subcomponents will be separated in two distinct sets: aspect models and bridge models.

The different aspect models to be connected in a multi-perspective model are regular DEVS models, where the interaction between the different sub-models is done through bridge models. Hence, the dynamics of each aspect are modeled as before, and connection points must be defined to allow the various models to influence each other where necessary. Sometimes the different aspect models can be connected directly, the output of one aspect model providing an input for another aspect model, but usually this will not be the case and some transformation has to take place. These transformations (see also Fig. 2) are taken care of by the bridge models, which are DEVS models as well. Bridge models are communication paths between aspect models; their inputs are connected to the outputs of aspect models, while their outputs influence the aspect models though the latter's inputs.

As will be shown in their formalization, multi-perspective models do not introduce any new syntax to the definition of DEVS. They merely add a semantic distinction between two types of models (the aforementioned aspect models and bridge models). Making this distinction allows us to specify methodological constraints to how the models are used and how they can be interconnected. The resulting multi-perspective

models still have the important property of being closed under coupling, which means that current simulation platforms can deal with them.

The constraints on how models can be connected are not intended to limit the modeller in any way. Instead, they are meant to provide guidelines for modellers to define proper interfaces between the models that implement the various theoretical perspectives, and as such they are used to support the proposed methodology. This aspect will be highlighted in the next section.

We can now define a multi-perspective model as follows:

$$MPM = \langle X, Y, A, B, EIC, EOC, IC \rangle$$

where:

$X =$ the set of inputs of the multi-perspective model,
$Y =$ the set of outputs of the multi-perspective model,
$A =$ the set of aspect models, each one being a regular DEVS model,
$B =$ the set of bridge models, each one being a regular DEVS model,
$EIC = \{(ei, i) \mid ei \in X, i \in \bigcup_{s.X \mid s \in A}\}$ (i.e., the set of external input couplings),
$EOC = \{(o, eo) \mid o \in \bigcup_{s.Y \mid s \in A}, eo \in Y\}$ (i.e., the set of external output couplings),
$IC = \{(o, i) \mid i \in \bigcup_{s.X \mid s \in A \cup B}, o \in \bigcup_{s.Y \mid s \in A \cup B}\}$ (i.e., the set of internal couplings).

The couplings between models are subject to the following constraints:

– any aspect model's input can only be connected to a single source:

$$\forall (x, i), (y, i) \in EIC \cup IC : x = y$$

– any bridge model's input can only be connected to a single source:

$$\forall (x, i), (y, i) \in IC : x = y$$

– any output of the multi-perspective model can only be connected to a single source:

$$\forall (x, o), (y, o) \in EOC : x = y$$

– all aspect models' inputs must be connected either an external input or to a bridge model's output:

$$\forall a \in A,\ i \in a.X : \exists (_, i) \in EIC \vee \exists (o_b, i) \in IC \mid b \in B \wedge o_b \in b.Y$$

– all bridge models' inputs must be connected to an aspect model's output:

$$\forall b \in B,\ i \in b.X : \exists (o_a, i) \in IC \mid a \in A \wedge o_a \in a.Y$$

– all outputs of the multi-perspective model must be connected to an aspect model's output:

$$\forall o \in Y : \exists (o_a, o) \in EOC \mid a \in A \wedge o_a \in a.Y$$

Most of the above constraints are the result of one not being able to obtain a value by connecting two outputs. Some others are the result of the constraints posed in Sect. 5, which states that bridge models are theories about relations between aspect models, and should not have external inputs or outputs of their own.

While the formalization given above assumes regular DEVS models as its component models, it should be realized that modelling system from different perspectives

might well result in the need to share components between perspectives. This would be possible by basing the formalization upon extended DEVS (Dalle et al. 2008), instead of regular DEVS.

As the reader will note, nothing was said about the time base of the individual aspect models. This is in line with DEVS, where time is implicit as and assumed to be consistent between the sub-models in a hierarchical model. The same restriction holds for the different aspect models as well. Since the literature on complex systems clearly states that the various perspectives might be defined using different time scales, further research will need to define some extension of the formalism that allows for the mapping of different time bases.

8 Applying multi-perspective modelling to a socio-political problem

An example model is presented in this section to clarify the ideas discussed previously. Let us consider analyzing a conflict opposing two communities (ethnic, political or religious) with a third actor, an armed policing force playing a peace keeping role. The goal of the study could be, for example, to compare different policing tactics in order to discover the one that is most effective in reducing the total count of violent actions.

The first question faced when designing a model for this conflict would be to decide the appropriate level of description. A modeller with social mechanist or methodological individualist tendencies would consider describing the system at the lowest level where individuals interact. A sociological realist would be satisfied with modelling laws occurring at the societal level. In between these two, there could be many possible levels of description. Practical questions such as the granularity of data, the availability of theories, or the availability of computational power can help decide this issue. Literature search can reveal that collective action literature is clearly relevant to this problem. For example, a theoretical framework based on diffusion theory was proposed in Myers and Oliver (2008) to predict the size and severity of protest and riot events. This theory offers an explanation at the aggregate level. At the individual level, we could make the hypothesis that stress and emotions are the important drivers of behavior in this context. Emotional appraisal theory (Lazarus 1991) is relevant to model how agents interpret their environments and react to it.

From this short analysis, one can see that the conflict may be considered from two distinct and non-isomorphic perspectives which in this case correspond to two distinct levels of description. The two levels use distinct underlying theories. One of the theories can be more complete than the other to describe some aspects of the problem. For example, the low level model accounts for a peace keeping (police) force, which is not present in the collective action model. The alternative decompositions can be seen in the SES ontology depicted on Fig. 5.

From the *aggregate collective action* perspective, the conflict system consists of two communities. Each community is further decomposed into two collective actions which they can use as means of expression. These collective actions are ‘peaceful protest’ and ‘violence’. Each of those collective actions is modeled with the Opposing Forces Diffusion model (OFD) as an underlying theory (Myers and Oliver 2008).

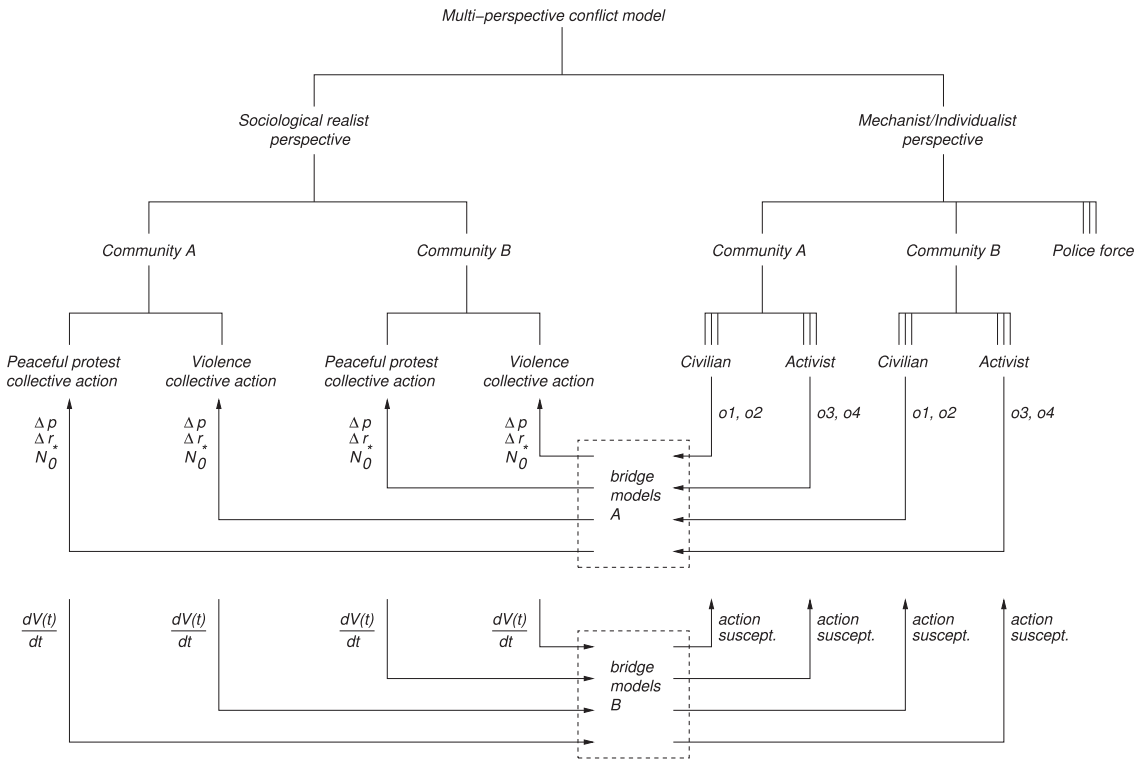


Fig. 5 A multi-perspective conflict model

In each of the collective action models, a provocation and a repression rate of adoption are defined as input variables, along with the initial proportion of adopters of those modes of expression. The OFD model is a theoretical and explicative model. Its theoretical grounding and straightforward parameter interpretation make it a good candidate for simulation. But, being an explicative model, the OFD model parameters are estimated a posteriori in face of available data; in comparison, a predictive model is supposed to generate the data on the basis of these parameters. In addition, we want the parameters to change dynamically as a result of interactions and interpretation by agents, like the influence of the policing force (which, in this perspective, is part of the environment). As we will see below, bridge models will dynamically provide the required parameters on the basis of information obtained from the other perspective.

In Myers and Oliver (2008), *provocation* (P) and *repression* (R) are formalized as two similar logistic functions representing the proportions of adopters of the two competing ideologies. The intensity of collective actions depends on the size of the difference between P and R, when $P > R$ as in the following equations:

$$P^*(t) = \frac{1}{1 + \frac{1-N_0^*}{N_0^*} e^{-pt}}$$

$$R^*(t) = \frac{1}{1 + \frac{1-N_0^*}{N_0^*} e^{-rt}}$$

Table 1 Observation frame for collective action OFD

Name	Type	Range	Meaning
N_0^*	input	0..1	initial proportion of adopters
Δp	input	0..1	provocation rate of adoption
Δr	input	0..1	repression rate of adoption
$\frac{dV(t)}{dt}$	output	0..1	event probability at time t

Table 2 Observation frame for Civilian

Name	Type	Range	Meaning
action susceptibility	input	0..1	willingness to take action
actions (o1-04)	output	$S_C \times S_A \times S_P$	agent actions

where N_0^* represents the initial proportion of adopters in the population and p and r represent the provocation and repression ideologies' infectiousness, respectively. Event probability at any instant is obtained by:

$$\frac{dV(t)}{dt} = P^*(t) - R^*(t)$$

In the *mechanistic* perspective, the conflict system is decomposed into four subcomponents: the two communities, the policing force, and the terrain through which they interact. Each community is composed of *Civilian* agents, and *Activist* agents. The Civilian agent represents an individual of the general population in one community, who can be in states such as: 'inactive', 'demonstrating', and 'rioting'. The Activist agent represents a more active individual of the population who can be in states such as: 'inactive', 'provocation', 'violence'. The transitions between these states are governed by the emotions elicited by an emotional appraisal module in each agent. The policing force is composed of *Reconnaissance patrol* agents, *Combat patrol* agents, and *Crowd control* agents, each with their own specified behavior, derived from miliary rules of engagement. The terrain is modeled as a lattice where each cell has parameters reflecting the symbolic value of each location for each community, the visibility it allows for the policing force agents, the density of populations from the communities, etc. It also contains state variables being updated depending on what is currently occurring in each location. The behavior of the agents is determined by local interactions and by a parameter related to the global state of the conflict ('action susceptibility' in Table 2), which is again derived from the other perspective through a set of bridge models.

Based on the decomposition given in Fig. 5, an observation frame can be built for this system. Tables 1 and 2 show the observation frames for the collective action OFD model and a civilian agent respectively. Data can be collected about the conflict to set initial parameters for the model. The initial parameters of the OFD model could be estimated with numerical regression techniques on historical data as shown in Myers and Oliver (2008). Such data on conflicts are available in databases such as

the Upsala Dataset on Armed Conflicts (Data on Armed Conflict n.d.). After that, an appropriate formalism is chosen to specify the models. For this example, all models were specified in the DEVS formalism.

Bridge models define the relations between the aspect models. Adapting provocation (p) and repression (r) rates as a result of micro-level outcomes requires hypotheses regarding how, for example, a violence act will have a future effect on the adoption of the violent provocation ideology in both the source population and the target population. The collective action literature has numerous references on the effect of prior actions on future collective behavior. One clear fact is that policing style influences the evolution of conflicts. For example, G. Marx analyzed different police behaviours and their effect on crowds (Marx 1970). Newman and Lynch show the role of vengeance as an ideology fueling violence in feuding societies (Newman and Lynch 1987). Assumptions on the influences between peaceful and violent ideologies within a community or between opposing communities can be diverse. The specification of these assumptions can be done in the bridge models.

Outputs from the low level models with relevance to the collective action models can be listed as follows:

(o1) Effective crowd control: a peaceful demonstration did not escalate to rioting,
(o2) Rioting: a peaceful demonstration escalated to rioting,
(o3) Failed violence act: the police force was able to stop violence act,
(o4) Successful violence act: activist successfully perpetrated a violence act.

The above values are the inputs to a set of bridge models A represented in Fig. 5; they are mapped to a certain effect (null, positive, or negative) on the Peaceful Protest and Violence collective actions through their provocation and repression rates p and r. For example, we assumed that a successful (o4) violence act by an activist group affiliated to Community 1 has the following effects:

- strengthens provocation ideology for peaceful demonstrations in Community 1,
- does not have any effect on the repression ideology for peaceful demonstration in Community 1,
- strengthens provocation ideology for violent behavior in Community 1,
- does not have any effect on the repression ideology for violent behavior in Community 1,
- does not have any effect on the provocation ideology for peaceful demonstration in Community 2,
- strengthens repression ideology for peaceful demonstrations in Community 2,
- strengthens provocation ideology for violent behavior in Community 2,
- does not have any effect on the repression ideology for violent behavior in Community 2.

Whenever the bridge model calculates new values for p and r, a new value of N_0^* is derived from these as follows:

$$N_0^* = \frac{1}{1 + \frac{\frac{1-N}{N}e^{-pt} - \frac{1-N}{N}e^{-rt}}{e^{-pt} - e^{-rt}}}$$

Once the bridge models are built, the overall model representing the conflict according to two perspectives (also corresponding to levels of organizations) is ob-

tained. In our case, the resulting model is a coupled DEVS model containing two aspect models, each represented as a classical DEVS model, and two bridge models, also DEVS models, responsible of encoding and decoding values from one perspective to another. An earlier account of this work focusing on the models themselves can be found in (forthcoming).

9 Conclusions and further research

This paper has shown some of the limitations of single perspective hierarchical modelling and has proposed multi-perspective modelling as a solution, with motivations taken from system theory, philosophy of science, and modelling and simulation practice.

Multi-aspect systems have been shown to form a specific system category, in between structure systems and meta-systems. It is argued that the syntax of the current generative modelling formalisms is expressive enough to create multi-perspective models. Consequently, a formalization was provided based upon DEVS coupled models, specifying Aspect and Bridge models with constraints managing their interactions. These constraints are chiefly methodological. This is in line with our proposal itself, which is *de facto* a methodological extension of an existing modelling paradigm. The extension is to clarify the way in which multiple perspectives can interact in a meaningful way in an integrated model.

A first definition of a corresponding methodology is given, inspired by Klir's systems approach, Rosen's modelling relation, and the logical empiricists' bridge principle concept. The methodology explains why notions from one modelling perspective must be decoded through the natural system and re-encoded into another perspective's formal system. An example application of the methodology on conflict simulation is given.

Future work includes further investigation on the nature of bridge models and their relation with data mining and statistical techniques. Applicability of the approach to modelling styles other than DEVS will also be studied.

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

Agazzi E (1991) Reductionism as negation of the scientific spirit. In: The problem of reductionism in science. Kluwer Academic, Dordrecht, pp 1–29

Bailly F, Longo G (2003) Objective and epistemic complexity in biology (towards a "conceptual organization"). In: International conference on theoretical neurobiology. National Brain Research Center, New Delhi

Bunge M (1979) Ontology II, treatise of basic philosophy, vol 4. Reidel, Dordrecht

Dalle O, Zeigler BP, Wainer GA (2008) Extending DEVS to support multiple occurrence in component-based simulation. In: Mason S, Hill R, Mönch L, Rose O, Jefferson T, Fowler J (eds) Proceedings of the 40th conference on winter simulation. IEEE, Miami, pp 933–941

Data on Armed Conflict. (n.d.) Department of Peace and Conflict Research, Uppsala University. Available from http://www.prio.no/CSCW/Datasets/Armed-Conflict/
Hempel CG (1965) Aspects of scientific explanation; and other essays in the philosophy of science. Free Press, New York
Kauffman SA (1970) Articulation of parts explanation in biology and the rational search for them. In: Psa: proceedings of the biennial meeting of the philosophy of science association. University of Chicago Press, Chicago, pp 257–272
Klir GJ (1975) On the representation of activity arrays. Int J Gen Syst 2:149–168
Klir GJ (1976) Identification of generative structures in empirical data. Int J Gen Syst 3:89–104
Klir GJ (1985) Architecture of systems complexity. Saunders, New York
Lane D (2006) Hierarchy, complexity, society. In: Pumain D (ed) Hierarchy in natural and social sciences. Springer, Berlin, pp 81–119
Lazarus R (1991) Emotion and adaptation. Oxford University Press, New York
Levins R (2006) Strategies of abstraction. Biol Philos 21(5):741–755
Marx GT (1970) Civil disorder and the agents of social control. J Soc Issues 26(1):19–57
Mikulecky DC (2001) The emergence of complexity: science coming of age or science growing ol? Comput Chem 25(4):341–348
Morin E (1990) Introduction à la pensée complexe. Seuil, Paris
Myers DJ, Oliver PE (2008) The opposing forces diffusion model: the initiation and repression of collective violence. Dyn Asymmetric Confl 1(2):164–189
Nagel E (1961) The structure of science: problems in the logic of scientific explanation. Routledge, London
Newman GR, Lynch MJ (1987) From feuding to terrorism: the ideology of vengeance. Contemp Crises 11:223–242
Pattee HH (ed) (1973) Hierarchy theory; the challenge of complex systems. Braziller, New York
Popper KR (1959) The logic of scientific discovery. Hutchinson, New York
Rosen R (1991) Life itself: a comprehensive inquiry into the nature, origin, and fabrication of life. Columbia University Press, New York
Rosen R (2000) Essays on life itself. Columbia University Press, New York
Simon HA (1962) The architecture of complexity. In: Proceedings of the American philosophical society, vol 106. American Philosophical Society, Philadelphia, pp 467–482
Uhrmacher AM, Ewald R, John M, Maus C, Jeschke M, Biermann S (2007) Combining micro and macro-modeling in DEVS for computational biology. In: Henderson S, Biller B, Hsieh M-H, Tew J, Barton R (eds) Proceedings of the 39th conference on Winter simulation. IEEE Press, New York, pp 871–880
Vangheluwe H, de Lara J (2002) Meta-models are models too. In: Yücesan E, Chen C-H, Charnes J (eds) Proceedings of the 2002 winter simulation conference. IEEE Press, San Diego
Wimsatt WC (1972) Complexity and organization. In: PSA-1972. Boston studies in the philosophy of science, vol 20. Reidel, Dordrecht, pp 67–86
Wimsatt WC (2000) Emergence as non-aggregativity and the biases of reductionisms. Found Sci 5(3):269–297
Winther RG (2006) Parts and theories in compositional biology. Biol Philos 21(4):471–499
Yilmaz L, Ören T (2005) Discrete-event multimodels and their agent-supported activation and update. In: Proceedings of the agent-directed simulation symposium of the spring simulation multiconference (SMC'05), San Diego, CA, pp 63–72
Zeigler BP, Praehofer H, Kim TG (2000) Theory of modeling and simulation. Academic Press, San Diego
Zwick M (2004) An overview of reconstructability analysis. Kybernetes 33(5/6):877–905

Mamadou D. Seck received his Ph.D. degree from the Paul Cezanne University of Marseille and his M.S. and M.Eng. Degrees from Polytech' Marseille, France. He is currently an Assistant Professor in the Systems Engineering section at the Technology, Policy, and Management department of Delft University of Technology. His research interests include modeling and simulation formalisms, dynamic data driven simulation, human behavior representation and social simulation, and agent directed simulation.

H. Job Honig is assistant professor; he has been a member of the Systems Engineering group at TPM, TU Delft for eight years. Before that, he has worked in the field of Natural Language Processing and in several Software Engineering research projects. His current research interests include: (philosophical) foundations of modeling and simulation, meta-modeling, equation based discrete event modeling, multi-perspective modeling, as well as the design and implementation of modeling languages for those.

Lecture 13. Simulation Languages 1

Papers to study for the exam

- Richard E. Nance. Simulation Programming Languages: An Abridged History. In: C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman (Eds.). Proceedings of the 1995 Winter Simulation Conference. IEEE, 1995. pp. 1307-1313
  Retrieved from: http://www.informs-sim.org/wsc95papers/1995_0199.pdf

- Ole-Johan Dahl and Kristen Nygaard. SIMULA - An ALGOL-based Simulation Language. Communications of the ACM, Vol 9, No 9, 1966. pp. 671-678
  Retrieved from:
  http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.384&rep=rep1&type=pdf

Proceedings of the 1995 Winter Simulation Conference
ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman

SIMULATION PROGRAMMING LANGUAGES: AN ABRIDGED HISTORY

Richard E. Nance

Systems Research Center and Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061-0251, U.S.A.

ABSTRACT

Knowing history can be protective; we have all heard that those who do not are doomed to repeat it. Considering one well regarded expert's estimate of 137 simulation programming languages (SPLs) created by 1981, many perhaps have already duplicated the numerous mistakes of their predecessors. History can also be informative, instructive and entertaining as hopefully this abridged and differently focused approach can illustrate. Questions concerning the causes for so many SPLs, the remarkably similar parallel developments, and the role of the SPLs versus programming languages in general might admit to historical answers. At the least, sharing speculations could prove enlightening and amusing.

1 INTRODUCTION

This abridged history of discrete event simulation programming languages is taken from the more complete version presented at the History of Programming Languages II Conference in Cambridge, Massachusetts 20-23 April 1993. A preprint appeared in *A C M SIGPLAN Notices 28* (3): March 1993, pp. 149-175.

Discrete event simulation programming languages (SPLs) must meet a minimum of six requirements: (1) generation of random numbers to represent uncertainty, (2) process transformers, to permit other than uniform random variates to be used, (3) list processing capability, so that objects can be created, manipulated, and deleted, (4) statistical analysis routines, to provide the descriptive summary of model behavior, (5) report generation, to provide the presentation of potentially large reams of data in an effective way for decision making, and (6) a timing executive or time flow mechanism. Clearly these requirements could be met by a general purpose language (GPL), but developments of SPLs progressed with the conviction that model complexity could be better accommodated and modeling conveniences better provided with a language reflecting the problem-solving domain. Nevertheless, packages of simulation-specific routines in numerous GPLs are significant; some have emerged to a very popular form in use historically and today.

Figure 1 shows the chronological development of simulation programming languages from both a vertical and horizontal perspective. The vertical perspective follows the division among world views, the "WELTANSICHT" of Lackner (1962, p.3). The "event scheduling," "activity scan," and "process interaction" categorizations are now termed the "classical" because they were used very early by Lackner (1964) and Kiviat (1967; 1969) as the basis for distinguishing and contrasting SPLs. Overstreet (1986, p. 171) captured the distinctions best by using the concept of locality:

> *Event scheduling* provides locality of time: each event routine describes related actions that may all occur in a single instant.
>
> *Activity scanning* provides locality of state: each activity routine describes all actions that must occur because a particular model state is reached.
>
> *Process interaction* provides locality of object: each process routine describes the entire action sequence of a particular model object.

The horizontal perspective in the SPL chronology describes the organization of the remainder of this paper. The five periods chosen, although somewhat arbitrarily, reflect remarkably similar developments occurring in the SPLs of that period. In fact, the earliest SPLs have continued, either in extended or modified forms or as generational successors. These five periods are labeled:

1955-60:	The Period of Search,
1961-65:	The Advent,
1966-70:	The Formative Period,
1971-78:	The Expansion Period,
1979-86:	The Period of Consolidation and Regeneration.

Since some time is needed to develop a historical perspective, no attempt is made to include developments beyond 1986.

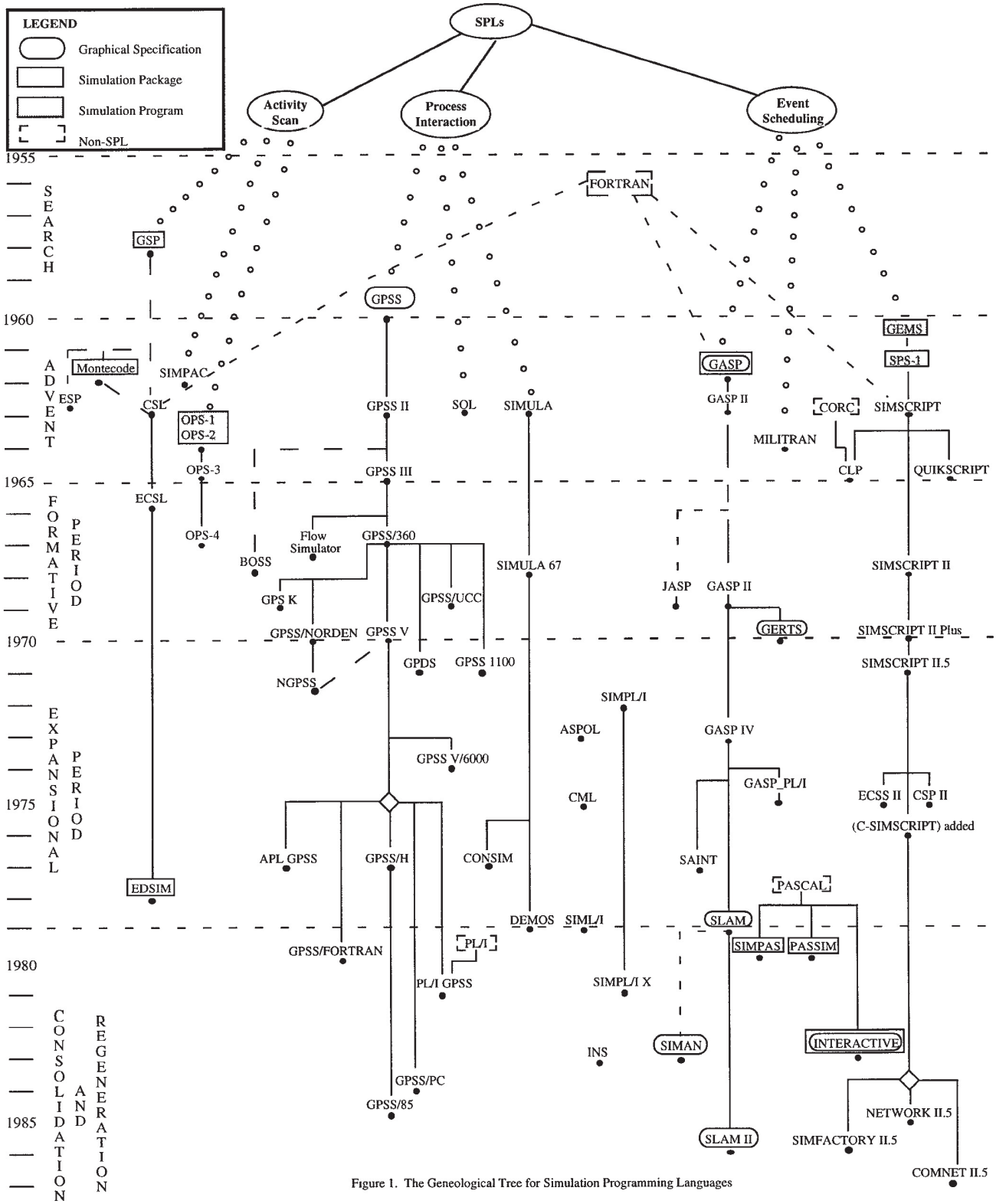


Figure 1. The Geneological Tree for Simulation Programming Languages

2 THE PERIOD OF SEARCH (1955-1960)

This period is marked by efforts to discover not only concepts of model representation but the representational needs for doing simulation analysis. Early groups at General Electric, UCLA, and elsewhere were cited in Conway (1967, p. 219). The General Simulation Program (GSP) of K.D. Tocher and D.G. Owen (Tocher, 1960) is considered the first "language effort." Tocher describes his identification and reuse of routines needed for each simulation application as a "simulation structure." The claim that such a structure would enable "automatic programming of simulations" (Tocher 1960, p. 51) seems a little inflated; however, the same claim was made for assemblers about five years earlier.

3 THE ADVENT (1961-1965)

The forerunners of SPLs currently in use appeared during the period of 1961-1965. GPSS was developed on various IBM computers and the name changed to the General Purpose Simulation System (from General Purpose System Simulator). Originating in the problem domain of communication systems, the original block semantics were ideally suited for queuing models. An interesting exercise demonstrating the importance of graphical output for on-line validation occurred in 1965 with an IBM 2250 interactive display terminal tied to a GPSS model (Reitman 1992).

At the Norwegian Computing Center the language SIMULA was begun in 1963 by Ole-Johan Dahl and Kristen Nygaard. The language was developed for the Univac 1107 and represented an extension to Algol 60, the most popular GPL in Europe. The early language version proceeded through four stages: (1) a discrete event network concept, (2) basing of the language on Algol 60, (3) modifications and extensions of the Univac Algol 60 compiler as the process concept is introduced, and (4) the implementation of the SIMULA I compiler. More follows concerning the impact of SIMULA, particularly the successor version, on programming languages in general.

The event scheduling SPL SIMSCRIPT appeared in 1963. Harry Markowitz, later to receive a Nobel Prize for his work in portfolio theory, provided the major conceptual guidance. The RAND corporation developed the language under sponsorship by the US Air Force, and Bernard Hausner was the sole programmer. Herbert Karr authored the SIMSCRIPT manual (Markowitz 1963, p.iii). Both syntactically and organizationally, SIMSCRIPT was heavily influenced by FORTRAN. The entire FORTRAN language was included as a subset. Later SIMSCRIPT I.5 was developed as a proprietary version by CACI (Karr 1965). A major difference was that SIMSCRIPT I.5 was compiled into assembly language rather than FORTRAN.

The Control and Simulation Language (CSL), representing the activity scan world view, appeared in 1963 as a joint venture of Esso Petroleum Company Ltd. and IBM United Kingdom, Ltd. The creators, John Buxton and John Laski, readily acknowledged the contributions of Tocher and his colleagues at United Steel Companies (Buxton and Laski 1962, p. 198). A second version of CSL, labeled C.S.L. 2 by Clementson (1966), is attributed to P. Blunden, P. Grant, and G. Parncutt of IBM UK (IBMUK 1965). CSL draws heavily on FORTRAN for the implementation, but the conceptual basis is clearly influenced by GSP. A simpler two-phase executive is used in CSL, in contrast with the three-phase executive adopted in GSP. The language became a favorite for program generation techniques based on the entity cycle or activity cycle diagrams, which were successors to the wheel charts originally developed by Tocher (Tocher 1966).

GASP (General Activity Simulation Program), developed by Philip J. Kiviat at the Applied Research Laboratory of the United States Steel Corporation, was begun in 1961, originally in ALGOL. Early on the decision was made to base the GASP simulator on FORTRAN II. Originally designed to bridge the gap between operating personnel and computer programmers, GASP, like GPSS, utilized flow chart symbols considered familiar to engineers. The set of FORTRAN II subroutines that made up GASP included a random number generator from the resident library. The GASP executive controlled the timing following an event scheduling view, and debugging help was provided in a special subroutine.

OPS-3, while relatively unknown to current day users, was an SPL ahead of its time. It represented an innovative technical effort at MIT to build simulation capability on the time sharing system CTSS. Principal developers were Greenberger, Jones, Morris, and Ness; however many others contributed in various ways (Greenberger 1965). The OPS-3 *system* was intended to be a multipurpose, open-ended, modular, compatible support for creative researchers that were not necessarily computing experts. Additions to OPS-3 could be made in a variety of GPLs. Clearly, the influence of SOL and SIMSCRIPT are present in the time flow mechanism, but the power of interactive model execution was unique to the language. The manifestation of the timing control was more akin to the three-phase method in Tocher's GSP than to any other form at the time.

Numerous other SPLs appeared during this advent period. Notable, although not a discrete event language, was DYNAMO. Developed at MIT for systems dynamic modeling, DYNAMO influenced several of the developers of discrete event SPLs (Kiviat 1991). Significant in the effect of DYNAMO was graphical output, strong typing, and extensive error detection capability. Other languages of note were SIMPAC, with a fixed-time-increment timing routine, developed at the System Development Corporation (Bennett 1962); SOL (Simulation Oriented Language), created by Knuth and McNeley as an extension to ALGOL and structured

much like GPSS; and MILITRAN, produced by the Systems Research Group of the Office of Naval Research (Systems Research Group, Inc. 1964).

4 THE FORMATIVE PERIOD (1966-1970)

The period from 1966-1970 reflected the conceptual clarification of languages. Concepts, possibly considered as secondary during the trials of bringing a SPL to implemented form, were reviewed and refined to promote a more consistent representation of a world view and to clarify its presentation to users. Rapid hardware advancements forced some languages, notably GPSS, to undergo major revisions. GPSS II and III, both of which emerged during the advent period, were replaced by GPSS/360, an extended version of the language. Other vendors also began to produce look-a-likes: RCA with its flow simulator (Greenberg 1972, p. 8), from Honeywell in 1969 (GPSK 1969) and GPSS/UCC. As a measure of the change in the language from GPSS III to GPSS/360, the number of block types increased from 36 to 44 and set operations were expanded by the introduction of groups. A HELP block opening the simulator to routines in other languages was also provided.

Major changes occurred in SIMULA, and the version SIMULA 67 emerged as a principal player at least on the conceptual level among programming languages. The concept of "classes of objects was added; subclassing and class concatenation to enable inheritance were provided; and direct, qualified references for object manipulation was created. Standardization emerged with the SIMULA 67 Common Base Language (Nuggard and Dahl 1981).

SIMSCRIPT II, also dependent somewhat on SIMSCRIPT I.5 for its basic concepts of entity, attribute, and set, clearly represented a major advancement in SPLs. An expressed goal of SIMSCRIPT II was to be self-documenting. In fact, the language was written to be used on five levels, and with its free-form, English-like mode of communication, and "forgiving" compiler, the user received major considerations in the language design. Markowitz (1979) indicated that the language was initially intended to be used for two additional levels, one that dealt explicitly with database entities and yet another to provide a language useful for writing other languages. (Markowitz 1979, p. 29).

ECSL, developed for Courtaulds Ltd. by the originators of CSL, became a popular language in the UK. ECSL departed from the heavy emphasis on FORTRAN of CSL, and Clementson (1966, p. 215) with the expressed intent to embrace users who were non-programmers relied heavily on the entity-cycle diagrams for model input. ECSL was the target language for CAPS (Computer Aided Programming System), the first *interactive program generator* (Mathewson 1975).

The careful reader has perhaps noted that GASP II appears twice in the genealogical tree of Figure 1. A preliminary description of the revised version appears in manual form in 1967 (Pritsker 1967) while a listing of FORTRAN subprograms designated as a GASP II compilation is dated "3/13/63." This early revision is due to Kiviat alone.

OPS-4 is primarily the doctoral research of Malcolm M. Jones (1967). Based on PL/I, OPS-4 encouraged incremental construction in test of model components. All three world views were supported in the language. Extensive debugging and tracing capabilities and on-line diagnostic explanations were available. Model execution could be interrupted and attribute values redefined.

New languages introduced during this period included BOSS (Burroughs Operational Systems Simulator) in 1967, Q-GERT in the early 1970s, and a number of others described in the hallmark workshop on simulation programming languages chaired by Buxton (1968).

5 THE EXPANSION PERIOD (1971-1978)

Major advances in GPSS during this period came from outside IBM as the success of the language caused others to extend either its capabilities or the host hardware environment. GPSS/NORDEN was an interactive, visual, on-line environment programmed in COBOL. This version permitted user interaction through a CRT terminal and interruption by the user to redefine certain standard numerical attributes and then resume execution. A second version NGPSS (NORDEN GPSS) is described as a superset of GPSS/360 and GPSS V for both batch and interactive execution. Limited database capability was provided.

GPSS V/6000, developed by Northwestern University for Control Data Corporation, was completed in 1975. Complete compatibility with the IBM product was claimed. GPDS (General Purpose Discrete Simulator) was developed as a program product of Xerox Data Systems for Sigma Computers. Similarly, a UNIVAC product labeled GPSS 1100 is referenced with a 1971 date (UNIVAC 1971a, 1971b). A major addition was GPSS/H in 1975, created by James O. Henriksen and produced in a compiled version. Released by Wolverine Software Corporation, GPSS/H has become the principal version of the language since that time.

This period of expansion for SIMULA took the form of a pure system description language called DELTA (Holbaek-Hanssen 1977). The DELTA project sought to implement system specification for simulation execution through a series of transformation from a high level user perspective through an intermediate form called BETA, culminating with an executable language called GAMMA.

A major departure from its roots was taken by SIMSCRIPT with the adoption of a process interaction world view in the mid-1970s. During this period, the capability for combined (discrete event and continuous) modeling was added to the language. This expansion period marked the beginnings of the breakdown in conceptual distinctions among SPLs.

Notable changes occurred to GASP during this period, primarily through the work of Pritsker and his graduate students at Purdue. GASP IV became available in 1974 (Pritsker 1974). GASP IV provided combined modeling, and differentiated between state events and time events as a means for representing both the condition concept of activity scan and the temporal change concept of event scheduling. A transaction flow world view was incorporated into GASP during 1975-1976 (Washam 1976a; 1976b). Pritsker and Young (Pritsker 1975) produced a PL/I version of the language, a phenomenon that occurred repeatedly during this period (the mapping of an SPL onto PL/I).

Interactive program generators were a major domain of activity in the U.K. Although initially beginning with Programming By Questionnaire in the U.S., CAPS-ECSL, DRAFT/FORTRAN, DRAFT/GASP and others sought to simplify the modeling task through the entity cycle (or activity cycle) diagram as a means for constructing the basic outline of the model. Mappings to different languages enabled a user to select the particular implementation with which he or she was most comfortable. A related, but very different approach, was taken by George Heidorn in the early 1970s in his work to build a natural language interface to GPSS (Heidorn 1976). In addition to GASP noted above, other mappings to PL/I included SIMPL/I, a PL/I preprocessor (SIMPL/I 1972), PL/I GPSS (Metz 1981) and SIMPL, a descendant of OPS-4 (Jones 1971a; 1971b). Slightly later SIML/I appeared in 1979, as a language for computer system modeling (MacDougall 1979).

6 CONSOLIDATION AND REGENERATION (1979-1986)

This final period in the chronology might be characterized as one where predominant SPLs extended their implementation to many computers and microprocessors while keeping the basic language capabilities relatively static. However, two major descendants of GASP appeared to play major roles: SLAM II and SIMAN. The Simulation Language for Alternative Modeling (SLAM), produced by Pritsker and Associates, Inc., sought to provide multiple modeling perspectives and combined modeling capabilities (Pritsker 1979). SLAM is a FORTRAN preprocessor, whereas its predecessors were packages. The major structure of the SLAM design still follows that of GASP; in fact, many of the identical subroutine and function names in GASP IV are repeated in SLAM.

SIMAN is derived from SIMulation ANalysis. Originally couched in a manufacturing systems application domain, SIMAN possesses a general modeling capability found in SPLs such as GASP IV. C. Dennis Pegden developed SIMAN as a one person faculty project in about a two year period. SIMAN is claimed to be the first major simulation language executable on the IBM PC and designed to run under MS-DOS constraints (Pegden 1991). The marketing of SIMAN has recognized the major advantage of output animation with a companion product called CINEMA.

Almost as history repeating itself, the emergence of PASCAL, yet another popular GPL, stimulated the appearance of a number of simulation packages based on the language. SIMPAS (Bryant 1980; 1981) is an event scheduling package that captures all the requirements identified for simulation modeling. SIMPAS, implemented as a preprocessor, was designed to be highly portable yet complete in its provision of services. In contrast, PASSIM (Uyeno 1980) provided less services, requiring more knowledge of PASCAL by the modeler. Yet another example was INTERACTIVE, described as a network simulation language using graphical symbols for model representation (Lakshmanan 1983).

INSIGHT was developed by Roberts (1983a) to model health care problems and for general simulation use. INSIGHT adopts the transaction flow world view and offers a graphical model representation that must be translated manually into INSIGHT statements. A FORTRAN preprocessor, INSIGHT provides the usual process transformers and some assistance in output analysis.

7 CONCLUDING SUMMARY

Why do the simulation programming languages exhibit remarkably similar periods of development? Perhaps it is the intense commercial competition, more intense than that for GPLs, which explains this fact. Perhaps many of the ideas and concepts, recognized eventually as important for general purpose use, in having their birth in SPLs created the early conceptual clashes which are inevitable for new ideas. Consider the numerous concepts and techniques that either originated with SPLs or gained visibility through their usage in a language:

- the process concept,
- object definition as a record data type,
- definition and manipulation of sets of objects,
- implementation of the abstract data type concept,
- quasi-parallel processing using the co-routine concept,
- delayed binding with run-time value assignment,
- English-like syntactic statements to promote self-documentation,
- error detection and correction in compilation,
- dynamic storage allocation and reclaim, and
- tailored report generation capabilities.

The importance of simulation in its influence on programming language development and concepts cannot be denied. The claim of Sammet (1969, p.650) that her justification for categorizing simulation languages as a specialization warranting limiting description is, "their usage is unique and presently does not appear to represent or supply much carry-over into other fields." Clearly the above list serves to dispel that misperception.

Now as we see simulation models at the core of analysis and training research, described as "virtual environments" or in entertainment applications using "virtual reality" are we entering a new period of change? Is the significance of modeling and simulation on its effect in sciences and business about to be universally recognized? Is Licklider's (19670 prediction in 1967 about to be realized:

> *In their static form, computer-program models are documents. They preserve and carry information just as documents printed in natural language do, and they can be read and understood by recipients who know the modeling language. In their dynamic form, however, computer-program models appeal to the recipient's understanding directly through his perception of dynamic behavior. That model of appeal is beyond the reach of ordinary documents. When we have learned how to take good advantage of it, it may - indeed, I believe it will - be the greatest boon to scientific and technical communication, and to the teaching and learning of science and technology, since the invention of writing on a flat surface.*

REFERENCES

Bennett, R. P., P. R. Cooley, S. W. Hovey, C. A. Kribs, and M. R. Lackner. 1962. *Simpac User's Manual*, Report #TM-602/000/00, System Development Corporation, Santa Monica, California.

Buxton, J. N., ed. 1968. Simulation programming languages. In *Proceedings of the IFIP Working Conference on Simulation Programming Languages*. North-Holland Publishing Company.

Buxton, J. N. and J. G. Laski. 1962. Control and simulation language. *The Computer Journal*, 5:194-199.

Clementson, A. T. 1966. Extended control and simulation language. *The Computer Journal*, 9:215-220.

Conway, R. W., W. L. Maxwell, and L. W. Miller. 1967. *Theory of Scheduling*. New York: Addison-Wesley Publishing Company.

Delfosse, C. M. 1976. *Continuous Simulation and Combined Simulation in SIMSCRIPT II.5*. CACI. Arlington, Virginia.

Greenberger, M., M. M. Jones, J. H. Morris, and D. N. Ness. 1965. *On Dash Line Computation and Simulation: The OPS-3 System*. Massachusetts: MIT Press.

Heidorn. 1976. Automatic programming through natural language dialogue, a survey. *IBM Journal of Research and Development*, 20:302-313.

Henriksen, J. O. 1976. Building a better GPSS: a 3:1 enhancement. In *Proceedings of the 1975 Winter Simulation Conference*, 465-469. New Jersey: AFIPS Press.

Holbaek-Hanssen, E., P. Händlykken, and K. Nygaard. 1977. System description and the DELTA language. *DELTA Project Report No.* 4. Second Printing. Norwegian Computing Center.

IBMUK. 1965. *CSL Reference Manual*. IBM United Kingdom Ltd.

Jones, M. M. 1967. Incremental simulation on a time-shared computer. Unpublished Ph.D. dissertation, Alfred P. Sloane School of Management, Massachusetts Institute of Technology.

Jones, M. M. and R. C. Thurber. 1971a. The SIMPL Primer.

Jones, M. M. and R. C. Thurber. 1971b. SIMPL Reference Manual.

Kiviat, P. J. 1967. Digital Computer Simulation: Modeling Concepts, RAND Memo RM-5378-PR, RAND Corporation. Santa Monica, California.

Kiviat, P. J. 1969. Digital Computer Simulation: Computer Programming Languages, RAND Memo RM-5883-PR. RAND Corporation. Santa Monica, California.

Kiviat, P. J. 1991b. Personal Communication.

Knuth, D. E. and J. L. McNeley. 1964a. SOL - a symbolic language for general purpose system simulation. *IEEE Transactions on Electronic Computers* EC-13:401-408.

Knuth, D. E. and J. L. McNeley. 1964b. A formal definition of SOL. *IEEE Transactions on Electronic Computers* EC-13:4, 409-414.

Lackner, M. R. 1962. Toward a general simulation capability. In *Proceedings of the SJCC*, 1-14. San Francisco, California.

Lackner, M. R. 1964. Digital simulation and system theory. System Development Corporation, SDC SP-12. Santa Monica, California.

Lakshmanan, R. 1983. Design and implementation of a PASCAL based interactive network simulation language for microcomputers. Unpublished Ph.D. dissertation, Oakland University, Rochester, Michigan.

Licklider, J. C. R. 1967. Interactive dynamic modeling. In *Prospects for Simulation and Simulators of Dynamic Systems*, eds. G. Shapiro and M. Rogers. New York: Spartan Books.

MacDougall, M. H. 1979. The simulation language SIML/I. In *Proceedings of the National Computer Conference*, 39-44.

Markowitz, H. M. 1979. SIMSCRIPT: past, present, and some thoughts about the future. *Current Issues in Computer Simulation*, eds. N. R. Adam and Ali Dogramaci, 27-60. New York: Academic Press.

Markowitz, H. M., B. Hausner, and H. W. Karr. 1963. *SIMSCRIPT: A Simulation programming language*. The RAND Corporation. Santa Monica, California.

Mathewson, S. C. 1975. Interactive simulation program generators. In *Proceedings of the European Computing Conference on Interactive Systems*, 423-439. Brunel University, U.K.

Metz, W. C. 1981. Discrete event simulation using PL/I based general and special purpose simulation lan-

guages. In *Proceedings of the Winter Simulation Conference*, eds. T. I. Oren, C. M. Delfosse, C. M. Shub, 45-52.

Overstreet, C. M. and R. E. Nance. 1986. World view based discrete event model simplification. *Modeling and Simulation Methodology in the Artificial Intelligence Era*, eds. M. S. Elzas, T. É. Oren, and B. P. Zeigler, 165-179. North Holland.

Pegden, C. D. 1991. Personal Communication.

Pegden, C. D., R. E. Shannon, and R. P. Sadowski. 1990. *Introduction to Simulation Using SIMAN*. McGraw-Hill.

Pritsker, A. A. B. 1967. *GASP II User's Manual.* Arizona State University.

Pritsker, A. A. B. 1974. *The GASP IV Simulation Language*. New York: John Wiley and Sons.

Pritsker, A. A. B. and C. D. Pegden. 1979. *Introduction to Simulation and SLAM*. New York: John Wiley and Sons.

Pritsker, A. A. B. 1990. *Papers, Experiences, Perspectives*. Indiana: Systems Publishing Company.

Pugh, A. L., III. 1963. *DYNAMO User's Manual.* The MIT Press, 2nd Ed.

RCA. 1967. *Flow Simulator.* RCA publication #70-05-008.

RCA. 1967. *Flow Simulator Information Manual.* RCA publication #70-35-503.

Reifer, D. J. 1973. Simulation Language Survey. Hughes Aircraft Corporation, Interdepartmental Correspondence, Ref. 2726.54/109.

Reitman, J. 1992. How the hardware and software world of 1967 conspired (interacted?) to produce the first in the series of winter simulation conferences. In *Proceedings of the 1992 Winter Simulation Conference*, eds. J. J. Swain, R. C. Crain, and J. R. Wilson, 52-58.

SIMPL/I. 1972. SIMPL/I (Simulation Language Based on PL/I). *Program Reference Manual.* IBM publication #SH19-5060-0.

Systems Research Group, Inc. 1964. *Systems Research Group, Inc. MILITRAN Programming Manual.* Prepared for the Office of Naval Research, Washington, DC.

Tocher, K. D. 1966. Some techniques of model building. In *Proceedings IBM Scientific Computing Symposium on Simulation Models and Gaming*, 119-155. White Plains, New York.

Tocher, K. D. and D. G. Owen. 1960. The automatic programming of simulations. In *Proceedings of the Second International Conference on Operational Research*, eds., J. Banbury and J. Maitland, 50-68.

UNIVAC 1100. 1971a. *UNIVAC 1100 Series General Purpose Simulator (GPSS 1100) Programmer Reference.* UP-7883.

UNIVAC 1100. 1971b. *UNIVAC 1100 General Purpose Systems Simulator II Reference Manual.* UP 4129.

Uyeno, D. H. and W. Vaessen. 1980. PASSIM: a discrete-event simulation package for PASCAL. *Simulation* 35:183-190.

Washam, W. B. 1976a. GASPPI: GASP IV with process interaction capabilities. Unpublished Master's thesis, Purdue University, West Lafayette, Indiana.

Washam, W. B. and A. A. B. Pritsker. 1976b. Putting process interaction capability into GASP IV, ORSA/TIMS Joint National Meeting, Philadelphia.

AUTHOR BIOGRAPHY

RICHARD E. NANCE is the RADM John Adolphus Dahlgren Professor of Computer Science and the Director of the Systems Research Center at Virginia Polytechnic Institute and State University. He received B.S. and M.S. degrees from N.C. State University in 1962 and 1966, and the Ph.D. degree from Purdue University in 1968. He has served on the faculties of Southern Methodist University and Virginia Tech, where he was Department Head of Computer Science, 1973-79. Dr. Nance has held research appointments at the Naval Surface Weapons Center and at the Imperial College of Science and Technology (UK). Within ACM, he has chaired two special interest groups: Information Retrieval (SIGIR), 1970-71 and Simulation (SIGSIM), 1983-85. He has served as Chair of the External Activities Board and several ACM committees. The author of over 100 papers on discrete event simulation, performance modeling and evaluation, computer networks, and software engineering. Dr. Nance has served on the Editorial Panel of *Communications ACM* for research contributions in simulation and statistical computing, 1985-89, as Area Editor for Computational Structures and Techniques of *Operations Research*, 1978-82, and as Department Editor for Simulation, Automation, and Information Systems of *IIE Transactions*, 1976-81. He served as Area Editor for Simulation, 1987-89 and as a member of the Advisory Board, 1989-92, *ORSA Journal on Computing*. He is the founding Editor-in-Chief of the ACM *Transactions on Modeling and Computer Simulation*. He served as Program Chair for the 1990 Winter Simulation Conference. Dr. Nance received a Distinguished Service Award from the TIMS College on Simulation in 1987. He is a member of Sigma Xi, Alpha Pi Mu, Upsilon Pi Epsilon, ACM, IIE, ORSA, and TIMS.

SIMULA—an ALGOL-Based Simulation Language

OLE-JOHAN DAHL AND KRISTEN NYGAARD
Norwegian Computing Center, Oslo, Norway

This paper is an introduction to SIMULA, a programming language designed to provide a systems analyst with unified concepts which facilitate the concise description of discrete event systems. A system description also serves as a source language simulation program. SIMULA is an extension of ALGOL 60 in which the most important new concept is that of quasi-parallel processing.

1. Introduction

SIMULA (SIMUlation LAnguage) is a language designed to facilitate formal description of the layout and rules of operation of systems with discrete events (changes of state). The language is a true extension of ALGOL 60 [1], i.e., it contains ALGOL 60 as a subset. As a programming language, apart from simulation, SIMULA has extensive list processing facilities and introduces an extended co-routine concept in a high-level language.

The principal features of SIMULA are defined below. The syntax rules given here are slightly simplified versions of the actual rules, since our intention is merely to introduce what we feel are the most important ideas of the language. For a complete definition of SIMULA, see [2].

A SIMULA compiler has been in operation on the UNIVAC 1107 computer since January 1965. The compiler translates a SIMULA system description into an object code simulation program for the system described. The compiler has now been used for the analysis of a great number of problems. Compiling and execution speeds are comparable to those of ALGOL and FORTRAN programs of corresponding complexity.

SIMULA has been designed and implemented by the authors at the Norwegian Computing Center under a contract with the Univac Division of Sperry Rand Corporation.

1.1 Design Objectives

Simulation is now a widely used tool for analysis of a variety of phenomena: nerve networks, communication systems, traffic flow, production systems, administrative systems, social systems, etc. Because of the necessary list processing, complex data structures and program sequencing demands, simulation programs are comparatively difficult to write in machine language or in ALGOL or FORTRAN. This alone calls for the introduction of simulation languages.

However, still more important is the need for a set of basic concepts in terms of which it is possible to approach, understand and describe all the apparently very different phenomena listed above. A simulation language should be built around such a set of basic concepts and allow a formal description which may generate a computer program. The language should point out similarities and differences between systems and force the research worker to consider all relevant aspects of the systems. System descriptions should be easy to read and print and hence useful for communication.

The need for the inclusion of algorithmic procedures as parts of a discrete event system description makes it natural to let a simulation language contain an algorithmic language as a subset. Efforts then should be made, not only to ensure that the simulation language is a logical extension of the algorithmic language, but also to achieve an increase in the power of the algorithmic language as such.

1.2 Basic Concepts

An ALGOL program (block) specifies a sequence of operations on data local to the program, as well as the structure of the data themselves. SIMULA extends ALGOL to include the notion of a collection of such programs, called "processes," conceptually operating in parallel. The processes perform their operations in groups called "active phases" or "events." Between any two consecutive active phases of one process any number of active phases of other processes may occur. The sequence of operations in the system

The paper was presented as the basis of a lecture given at a Seminar on Simulation Languages at Centro Nazional Universitario di Calcolo Elettronico, Pisa, Italy, May, 1966. An earlier, less comprehensive version was presented at a NATO "Conference on the Role of Simulation in Operations Research," Hamburg, Germany, September, 1965.

as a whole thus becomes a sequence of active phases of the processes present in the system. This mode of operation will be called "quasi-parallel."

The following are the main extensions to ALGOL, which are introduced:

(1) Means of describing processes, generating processes dynamically, and referencing existing ones.
(2) Means of delimiting and sequencing active phases of processes, with or without reference to the concept of "system time."
(3) Means of making data local to processes accessible from other processes.
(4) An ordered set mechanism convenient for queueing and stacking processes.

The SIMULA concepts are introduced partly by extensions to the ALGOL 60 syntax, partly through system procedures. Among the latter are a number of procedures for random drawing from mathematically or empirically defined distribution functions and for accumulating system time integrals and histograms.

2. Processes

The process concept is intended as an aid for decomposing a discrete event system into components, which are separately describable. In general, a process has two aspects: it is a data carrier and it will execute actions. The items of data carried by a process are described by a sequence of specifications and declarations; the actions are described by a sequence of statements called its operation rule. The two aspects are combined into a block called a *process block.*

The description of a process is called an *activity* declaration. The concept of an "activity," which is a class of processes described by the same declaration, is distinguished from the concept of a "process," which is one dynamic instance of an activity declaration. The declaration is syntactically similar to that of a procedure.

⟨activity identifier⟩ ::= ⟨identifier⟩
⟨activity body⟩ ::= ⟨statement⟩
⟨activity declaration⟩ ::= **activity** ⟨activity identifier⟩
⟨formal parameter part⟩; ⟨specification part⟩ ⟨activity body⟩

A process block is the activity body itself, if the latter takes the form of an ⟨unlabelled block⟩; otherwise it is an implied block enclosing the activity body. The parameters and the items declared local to the process block are called the *attributes* of a process; its operation rule is the list of statements within the process block.

Unlike procedures, which are dynamically nested, the relationship between processes is a symmetric one. They operate in a quasi-parallel fashion, as defined below.

A discrete event system will be viewed as a collection of processes, whose actions and interactions comprise the behavior of the system. Processes will enter and leave the system as results of actions within the system itself.

2.1 PROCESS REFERENCES

Processes can be referenced individually. Physically, a process reference is a pointer to an area of memory containing the data local to the process and some additional information defining its current state of execution. However, for reasons stated in the Section 2.2 process references are always indirect, through items called "elements." Formally a reference to a process is the value of an expression of type **element.**

⟨element expression⟩ ::= **none** | ⟨variable⟩ | ⟨function designator⟩ |
⟨process designator⟩ | ⟨activity identifier⟩
⟨process designator⟩ ::= **new** ⟨activity identifier⟩
⟨actual parameter part⟩

element values can be stored and retrieved by assignments and references to **element** variables and by other means.

A process is generated by evaluating a *process designator.* Its value is a reference to the generated process. A process will remain part of the system *as long as it can be referenced* through a computable **element** expression. (The deletion is automatic and is effected through a combination of the "reference count" and "garbage collection" techniques; see [6].)

Example.

```
element Pat;
activity secretary (redhaired, thumbs);
        Boolean redhaired; integer thumbs;
begin - - - end;
Pat := new secretary (true, 10);
```

The value of the variable Pat is now a reference to the generated secretary process. If another value is later assigned to Pat, and if there is no other way of referencing this secretary, she is deleted.

The symbol **new** serves to distinguish between different uses of activity identifiers. As described in Section 4 an ⟨activity identifier⟩ may be used as an **element** expression, without the symbol **new**, within a "connection block."

The language contains a mechanism for making the attributes of a process accessible from the outside, i.e., from within other processes. This is called remote accessing. A process is thus a *referenceable data structure.*

It is worth noticing the similarity between a process whose activity body is a dummy statement, and the record concept recently proposed by C. A. R. Hoare and N. Wirth [3]. Since processes can reference one another through local **element** variables, they have the same list forming capabilities. The ordered set mechanism described in the next section has been added for convenience. The mechanism for "remote accessing" in SIMULA differs from the one proposed in [3].

2.2 ELEMENTS AND SETS

The element and set concepts[1] serve to facilitate and standardize the manipulation of queues and other linear

[1] Many of the ideas presented in this section were inspired by the SLIP system [5].

lists of processes. A set is an ordered sequence of *elements*, which are objects having a standard format as shown in Figure 1. An element pointing to a process is a marker representing that process. There may be more than one

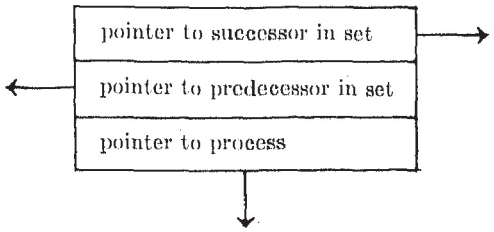


FIG. 1. Element in a set. In addition to the information shown, an element contains a "reference count," which implicitly defines its "lifetime" in the system.

element in the system pointing to the same process. A process is said to be member of a set if there is an element in the set pointing to it.

A value of type **element**, except the value **none**, is a pointer to an element. We may equivalently say that the value is the element itself, and we shall do so in order to shorten the following discussion.

Two of the three components of an element (Figure 1) are themselves **element** values. They are accessed through basic **element** procedures. If X is an element in a set, *suc*(X) is the successor element and *pred*(X) is the predecessor. These pointers between elements of a set are updated implicitly by system procedures such as *include*(X) *in:*(S), *precede*(X)*by:*(Y), and *remove*(X), where X and Y are elements and S is a set. If an element X currently has no set membership, the values of *suc*(X) and *pred*(X) are both **none**.

The third component of an element, the process pointer, remains fixed. It is not itself an **element** value, and is therefore not explicitly accessible as the value of an expression. There is, however, a corresponding **element** procedure "*proc*," which generates an element. The value of *proc*(X) is a new element with the same process pointer as X and no set membership. As shown above, a ⟨process designator⟩ is another "generating" expression, which actually generates both a process and an element referring to it. The latter has no set membership and is the value of the expression.

The process pointer of an element is used implicitly by a number of system procedures and special statements, which operate on the process referenced by a specified element. The element concept and the technique of referencing processes indirectly yield the following desirable language properties.

(1) Ordered sets of processes can be manipulated by means of efficient standard procedures.

(2) When a process is referenced through an element in a set, its successor and predecessor in the set are immediately accessible.

(3) Any given process can be member of an unlimited number of sets at the same time.

(4) The members of a set can be processes of different classes.

A set has one permanent dummy element called the *set head*, which conforms to the standard element format, except that it refers to no process. Together with the ordinary elements of a set it forms a circular list of elements (Figure 2). An "empty" set has a set head which is its own successor and predecessor.

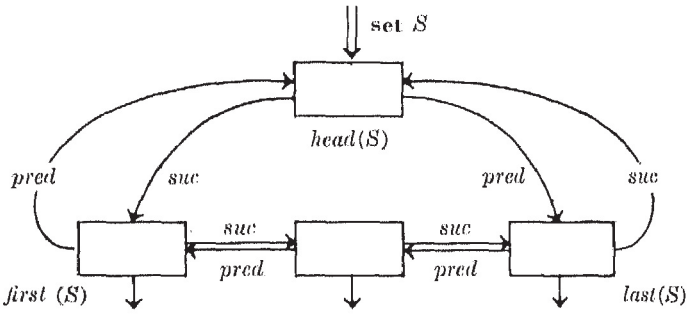


FIG. 2. Set with 3 members

A set designator

⟨set designator⟩ ::= ⟨variable⟩

has a value of type **set**, which is, physically, a pointer to a set head. This pointer remains fixed during the scope of the set designator. The set head is generated as the result of the set declaration, at which time the set is empty. At the end of the scope of a set each element loses its set membership. The head of a set S can be referenced as an element through the function designator "*head*(S)." The "first" and "last" elements of the set are defined as

$$first(S) = suc(head(S))$$

$$last(S) = pred(head(S)).$$

The members of a set S may be scanned successively by means of the following **for** clause:

for X := *first*(S), *suc*(X) **while** *exist*(X) **do**

or, in order to avoid the execution of the controlled statement if the set is empty:

X := *head*(S); **for** X := *suc*(X) **while** *exist*(X) **do**

The Boolean expression "*exist*(X)" has the value **true** if X refers to a process, otherwise **false**. The set head is the only element of a set which does not refer to a process thereby signalling the "end" of the set in either direction.

2.3 The Sequence Control

The actions of a process are grouped together in active phases, separated by periods of inactivity. Only one process is active executing actions at any one time. An inactive period of a process is caused by a deactivating statement executed by that process. (Deactivating statements are special cases of the sequencing statements described in Section 2.6.) The statement terminates the current active phase, and control leaves the process. Usually a *reactivation point* is defined for the process, whose function is similar to that of the "router" of a co-routine, as defined by M. E. Conway [4]. At the time of the next active phase of the process, control re-enters the process at the place where it most recently left, as specified by the reactivation point. (A deactivating statement may be given inside any sub-block or procedure called by the process, therefore the reactivation point must also include information relating to the status of a "block stack" belonging to the process, as used in most Algol implementations.)

The reactivation point concept allows the programmer to string together actions occurring at widely different times into a logically coherent sequence. He can think of a process as possessing its own *"local" sequence control*, identical to the main control during active phases, and represented by the reactivation point during inactive periods. From this local point of view a deactivating statement allows other processes to be active during its execution.

An active phase of a process is called an *event*. It has an associated *system time*, which remains constant during the execution of the event. The system times form a non-decreasing sequence of **real** numbers. A deactivating statement may invoke an inactive period of a definite or indefinite length. As an example of the former, consider the following activity declaration.

```
activity report(dt); real dt;
begin L: write (-----); hold(dt); go to L end;
```

The statement "*hold*(*dt*)" is a call to a system procedure, which represents an inactive period of length *dt* in system time. It follows that a report process will give output with regular system time intervals. Several report processes could operate in quasi-parallel, e.g., **new** *report*(7) and **new** *report*(30) giving "weekly" and "monthly" reports, provided that the system time is being given in units of one day.

2.4 The Sequencing Set (SQS)

An event can be scheduled to happen either immediately or at some later time. The event is the next active phase of some specified process. A process for which an event has been scheduled but not completed has an associated *event notice* representing the event. There can be at most one event notice associated with a given process.

An event notice contains a reference to the associated process (through an element) and a **real** number, called its time reference. The event notices are members of the *sequencing set* (SQS), which is ordered according to non-decreasing time references, as shown in Figure 3. The time reference of an event notice is accessed through the **real** function *evtime*(*X*), where *X* is an **element** expression referencing the associated process. The sequencing set is not, strictly speaking, a "set" in the sense of Section 2.2.

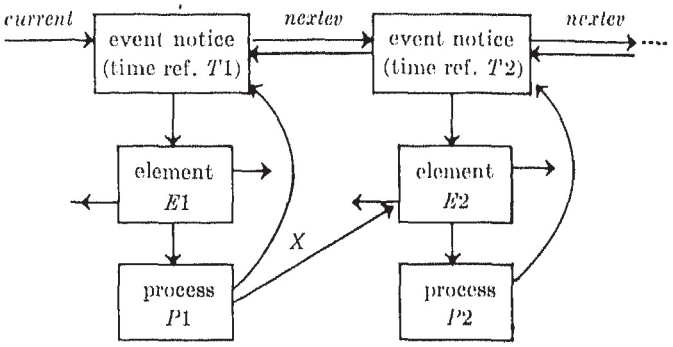


Fig. 3. The sequencing set. *P*1 is the currently active process, *P*2 is suspended, "current" is equal to *E*1 and "time" is equal to *T*1. *P*1 has a local **element** variable *X*, to which has been assigned the value "nextev(current)". "evtime(X)" is equal to *T*2, which is greater than or equal to *T*1.

The event notice at the "lower end" of the SQS is called the *current* event notice. Its associated process is the currently active one, and its time reference is regarded as the current value of the "system time." The active process may be referenced by the **element** procedure "*current*," and there is a **real** procedure "*time*" equivalent to "*evtime*(*current*)."

When the current active phase is completed, the current event notice is deleted. Its successor in the SQS becomes the current event notice, and control enters the associated process at its reactivation point.

2.5 States

A process can be in one of four possible *states*: active, suspended, passive, and terminated. As simulation proceeds, the states of processes will change.

(1) *Active.* The currently active process can, by *sequencing statements*, alter the states of processes, including its own.

(2) *Suspended.* A suspended process has an associated event notice and a reactivation point. Unless a change of state is caused by another process, the next active phase of this process will start when the event notice becomes the current one.

(3) *Passive.* A passive process has a reactivation point, but no associated event notice. It will remain passive until a change of state is caused by another process. At

the time of generation a process is passive, and its reactivation point is in front of the first statement of its operation rule.

(4) *Terminated.* A process becomes terminated whenever control passes through the final **end** of its operation rule or exit from the process is made by a **go to** statement. Such a process has no reactivation point and no event notice. The state of a terminated process can not be altered by ordinary sequencing statements.

A process will remain part of the system at least as long as it has an associated event notice. It can be referenced through system procedures such as "*current*" if active, or "*nextev*(*X*)" if suspended (where *X* refers to an active or suspended process). A passive or terminated one will remain (as a data structure) as long as it can be referenced through an **element** expression.

2.6 Sequencing Statements

Sequencing statements are statements operating on the SQS, thereby altering the states of processes. A sequencing statement may delete an event notice and/or schedule an event by generating an event notice and include it in the SQS.

The basic sequencing statements are:

(1) *cancel* (⟨element expression⟩), which deletes the event notice, if any, associated with the referenced process,

(2) *terminate* (⟨element expression⟩), which in addition deletes the reactivation point, if any,

(3) scheduling statements, which are described below.

Deletion of the current event notice causes the current active phase to terminate. Any sequencing statement, except "terminate" will in that case define a reactivation point for the process at the end of the statement. The statement "*terminate*(*current*)" is equivalent to leaving the process through its final **end**. The statement "*cancel* (*current*)" represents an inactive period of indefinite length. The process becomes passive.

A scheduling statement usually generates an event notice for a specified process and includes it in the SQS. The process is referenced through an element expression, whose value is stored in the event notice.

In addition, the statement will specify explicitly either (1) the time reference of the event notice, or (2) its position in the SQS. The information not specified is determined according to rules consistent with the ordering criterion of the SQS.

The basic scheduling statements are special syntactic constructions.

⟨activator⟩ ::= **activate** | **reactivate**
⟨simple timing clause⟩ ::= **at** ⟨arithmetic expression⟩ |
delay ⟨arithmetic expression⟩
⟨timing clause⟩ ::= ⟨simple timing clause⟩ |
⟨simple timing clause⟩ **prior**
⟨scheduling clause⟩ ::= ⟨empty⟩ | ⟨timing clause⟩ |
before ⟨element expression⟩ | **after** ⟨element expression⟩
⟨scheduling statement⟩ ::= ⟨activator⟩ ⟨element expression⟩
⟨scheduling clause⟩

The activator **activate** will cause the generation of an event notice only if the referenced process is passive, whereas **reactivate** in addition will delete the event notice associated with an active or suspended process and "re-schedule" the event.

A timing clause specifies the time reference of the generated event notice, either "*time*" or the value of the expression, whichever is greater, and this determines its position in the SQS. The event notice is normally placed behind all others with the same time reference, but it can also be placed in front of these event notices by adding the symbol **prior**.

The statement "*hold*(*T*)" already mentioned is equivalent to:

reactivate *current* **delay** *T* (or "**at** *time* + *T*")

Another option is to specify the position of the event notice relative to (**before**, or **after**) the event notice associated with a second process. It gets the same time reference as the latter. If the second process is passive or terminated, no scheduling takes place.

The scheduling clauses "**delay** 0 **prior**", "**at** *time* **prior**", "**at** 0 **prior**" and "**before** *current*" are equivalent. The generated event notice is placed in front of the current event notice, whereby the currently active process becomes suspended. The scheduled event becomes the current event. This is called "direct scheduling". Since direct scheduling is used frequently in practice, an empty (omitted) scheduling clause is understood to be an abbreviation for direct scheduling.

The statement "**activate** ⟨element expression⟩" is therefore somewhat similar to a procedure call, in the sense that actions are invoked as a "subroutine" to the calling program. Notice, however, that the event notice associated with the calling process may well be deleted or rescheduled before control returns.

Two processes *X* and *Y*, calling each other by, respectively, **reactivate** *Y* and **reactivate** *X*, would function as co-routines in the sense of [4]. They could communicate through nonlocal variables.

2.7 Parameters and Nonlocals

The quasi-parallel operation of processes is not compatible with the simple stack structure of an Algol program at run time. In order to make possible an efficient data storage allocation scheme the activity declaration is subject to certain restrictions as compared to the procedure declaration.

(1) Only ⟨type⟩ and ⟨type⟩ **array** parameters are permitted. The former are transmitted by value, the latter by location. The activity declaration has no ⟨value part⟩.

(2) Activity declarations are only accepted in the block heads of certain special blocks called Simula blocks; this is to permit unrestricted reference to nonlocals within processes.

2.8 Main Program

A Simula block

⟨Simula block⟩ ::= **SIMULA** ⟨unlabelled block⟩

is itself a statement, which can be part of an otherwise ordinary Algol program. Simula concepts are not available outside a Simula block. The SQS is local to it and all activity declarations describing a discrete event system are given in the Simula block head.

The Simula block functions dynamically as a process called the "main program," and at the same time as an outer block to all processes except itself. It is always present in the system. Upon entry into a Simula block the main program is the only process in the system. It is active, and system time is zero. When exit is made out of the Simula block the simulation is terminated.

The initialization of the system is usually done by the main program. It may suspend itself during the simulation proper, or may take an active part in it.

3. An Example

The following program is a skeleton Simula description of a classical "job shop" system. The shop consists of a given number (*nmg*) of machine groups. The *j*th group contains a number of identical machines, given by the initial value of "*available* [*j*]." They operate in parallel and have a common queue (*que*[*j*]) of orders waiting. The orders are treated on a "first come first served" basis at each group.

An order defines its own path through the shop (*mgroup*[1], ⋯ , *mgroup*[*n*]) and its processing time at each group of the path (*ptime*[1], ⋯ , *ptime*[*n*]). The arrival pattern of orders and their contents are defined by an input stream. The data describing an order are the number *n* of steps in its schedule, its arrival time *T*, and the schedule itself represented by the components of the arrays *mgroup* and *ptime*. The end of the arrival pattern is signalled by a zero *n*, where the associated *T* is the system time at which the simulation is to be terminated.

The system might be formulated as a collection of machine processes acting on orders (through remote accessing). However, in this case, the simplest program is obtained by looking at the system entirely from the point of view of an order, since the latter contains all the information relevant to the routing and processing. The main program acts as a steering routine timing the order arrivals.

The following program contains only statements essential to the behavior of the system. No data are collected, and no output is given.

```
Line
 1. begin integer nmg; read(nmg);
 2. SIMULA begin integer array available [1:nmg];
      set array que [1:nmg];
 3. activity order(n); integer n;
 4. begin integer i, mg; integer array mgroup[1:n];
      array ptime[1:n];
 5.   read(mgroup, ptime);
 6.   for i:= 1 step 1 until n do
 7.   begin mg := mgroup [i];
 8.   if available[mg] = 0 then
 9.   begin wait(que[mg]); remove(current) end
10.   else available[mg] := available[mg] −1;
11.   hold(ptime[i]);
12.   if empty(que[mg]) then available[mg] := available[mg] +1
13.   else activate first (que[mg])
14.   end path through shop
15.   end order;
16. integer n; real T;
17. read(available);
18. next: read(n, T); reactivate current at T;
19. if n > 0 then begin activate new order(n); go to next end
20. end SIMULA end program
```

Comments

Line 2. All sets are empty initially. The machines are all available (cf. line 17).

Line 3. The activity declaration (lines 3–15) describes the structure and behavior pattern of an order.

Line 5. An order reads its own schedule during its first active phase.

Line 6. The controlled variable counts the number of steps in the schedule. The following compound statement (lines 7–14) describes the behavior of an order at each machine group.

Line 9. If there is no available machine in the current machine group (*mg*), the order includes itself in the appropriate queue and goes passive. The system procedure *wait*(*S*) is equivalent to

include(*current*, *S*); *cancel*(*current*).

The order becomes "*last*(*que*[*mg*])". The local sequence control stays at this statement until the order has become "*first*(*que*[*mg*])", and is activated by another order leaving the machine group (line 13). The next statement removes the order from *que*[*mg*].

Line 10. The order allocates an available machine to itself.

Line 11. The hold statement represents the processing time.

Line 12. If the queue is now empty, the machine is made available.

Line 13. If not, this order activates the one in front of the queue. The latter removes itself from the queue (line 9) and becomes suspended (line 11). Control then returns to the former order, which proceeds to the next machine group or leaves the shop.

Line 15. At this point the order is a member of no set, and the only reference to it is from the current event notice. At the end of the active phase the latter is deleted and the order therefore leaves the system.

Line 17. This is the first statement of the Main Program, and therefore it is the first statement of the first event, taking place at system time zero. The sizes of the machine groups are read.

Line 18. The arrival of the next order is timed according to the value read for *T*. The Main Program suspends itself until system time *T*.

Line 19. The **element** expression "**new** order(*n*)" generates an order. Its first active phase is scheduled directly, whereby the *n*-step schedule is read (line 5). At the end of this active phase (line 9 or 11) control returns to the Main Program. If *n* is nonpositive, the simulation is terminated (at system time *T*).

4. Remote Accessing

Remote access to items local to a process is made possible through the *connection* mechanism. A connection

statement may have the form:

inspect ⟨element expression⟩ **when** A_1 **do** S_1
when A_2 **do** S_2
⋮
when A_n **do** S_n
otherwise S

where A_i are activity identifiers and S_i and S are statements. If the referenced process belongs to the activity A_i, S_i is executed and the other statements are skipped. The **otherwise** clause is optional. The symbol "**inspect**" may be replaced by "**extract**", which will remove the referenced element from its set, if it has set membership.

S_i acts as if it is immediately enclosed in a block containing exactly the same local items as the process block of the referenced process. Thereby the attributes of the process are accessible within S_i through their local names. S_i is called a *connection block*, and the referenced element and process are said to be *connected*. Within a connection block the relevant activity identifier functions as an **element** procedure, whose value is the connected element.

A connection clause (**when** A **do** S), or otherwise clause (**otherwise** S) within the lexicographic scope of more than one connection statement is part of the one with the smallest scope. An otherwise clause terminates the scope of the corresponding connection statement. (Evidently the scope of a connection statement can also be terminated by any symbol "**end**", "**else**" or ";" which is not a part of the statement.)

The attributes of a process may include procedures, switches and labels. By going to a label local to a connected process it is possible to override its reactivation point and to "revive" a terminated one. [This situation is explained further in 2.]

The connection mechanism has been designed with the following objectives in mind:

(1) To make the language "self-protecting" in the sense that only meaningful references to attributes can be made in a program acceptable to the compiler. The user is forced to check the class of the referenced process.

(2) To give easy and rapid access to the attributes once this check has been made. Interaction between processes often requires remote access to several attributes of the same process, occasionally interspersed with scheduling statements.

(3) To preserve the locality of attribute identifiers. The same identifier can be used for different purposes within different activities. The semantic contents of an attribute identifier within a connection block are defined by the activity identifier of the connection clause.

The next version of the language, now being implemented, will be extended by "**external**" declarations (and specifications) for remote accessing. An **external** declaration declares a nonlocal name for a local quantity (in contrast to **own**, which declares a local name for a nonlocal quantity). Remote access to a quantity local to a given process and declared **external** will be written "⟨element expression⟩.⟨identifier⟩." **external** identifiers will belong to the SIMULA block, and must be unique. It follows that the semantic contents of such an identifier is known to the compiler anywhere within the SIMULA block, and that a simple run-time consistency check can be devised for the process reference.

5. Second Example

The program below is a description of a simple epidemic model. A contagious, nonlethal disease is spreading through a *population* of a given fixed size. (Words used as identifiers in the program are italicized here.) Certain countermeasures are taken by a public health organization.

An individual infection has a given *incubation* period, during which the subject is noncontagious and has no *symptoms*, followed by a contagious period of a given *length*. Each *day* of the latter period the subject may seek *treatment* and get cured. The (unconditional) probability of doing so is *probtreat*[*day*] (*day* = 1, 2, ⋯, *length*). Each sick person has an expected number of *contacts* per day. At one such contact the probability of infecting an uninfected person is *prinf*. A person previously cured is immune. If untreated the infection ceases spontaneously after the given period.

Only *sick persons* appear as processes in the system. When cured they leave the system. The very first infection is generated by the Main Program. A subject infected by another person is included as an element of a set belonging to the latter, called his *environment*. Any subject is a member of at most one such set. It will be seen that the subjects initially form a simple tree structure. As subjects are cured they are removed from the tree, which will therefore disintegrate into smaller tree structures. The latter will grow independently, disintegrate further, and so forth. As the number of *uninfected* persons decreases the growth of the contagion slows down, until finally it dies out.

The public countermeasures are represented by *treatment* processes. A patient is removed from the environment set to which he belongs, if any. If he has visible symptoms, he is cured (terminated). In addition his environment is searched and each member is subjected to a full treatment, which may cause another environment to be searched, etc. A patient displaying no symptoms is given "mass treatment" (cheap pill) which has a certain probability, *probmass*, of success. His environment is not searched. In the present model, treatments act instantaneously in system time. The simulation ends after *simperiod* units of time.

The following program shows only the system description (SIMULA block) itself, and no output is given. The following quantities are declared and given values in outer blocks:

integer *population, length, contacts, u1, u2, u3, u4*;
real *incubation, prinf, probmass, simperiod*;
array *probtreat*[1: *length*];

The variables *u1*, *u2*, *u3*, *u4* represent different "streams" of pseudo-random numbers.

```
1.  SIMULA begin integer uninfected;
2.    activity sick person;
3.    begin integer day; Boolean symtoms; set environment;
4.      uninfected := uninfected-1; symptoms := false;
5.      hold(incubation); symptoms := true;
6.      for day := 1 step 1 until length do
7.      begin if draw(probtreat[day], u1)
          then activate new treatment (current);
8.        infect(Poisson(contacts, u2), environment);
9.        hold(1) end
10.     end sick person;
11.   procedure infect(n, S); value n; integer n; set S;
12.   begin integer i;
13.     for i := 1 step 1 until n do
14.     if draw(prinf×uninfected/population, u3) then
15.     begin include(new sick person, S);
          activate last(S) end
16.   end infect;
17.   activity treatment(patient); element patient;
18.   begin element X;
19.     extract patient when sick person do
20.     if symptoms then
21.     begin terminate(patient);
22.       for X := first(environment) while exist(X) do
23.       activate new treatment(X) end
24.     else if draw(probmass, u4) then terminate (patient)
25.   end treatment;
26.   uninfected := population;
27.   activate new sick person; hold(simperiod)
28. end SIMULA
```

Comments

Line 4. A new sick person has entered the system: the number of persons that have not been in touch with the disease, must therefore be reduced by one.

Line 5. He executes no action relevant to the disease during the incubation period.

Line 6. The controlled statement (lines 7–9) describes his behavior pattern each day of the contagious phase.

Line 7. *"draw"* is a **Boolean** system procedure, which makes a random drawing. The probability of getting **true** as the result is given by the first parameter. If the activate statement is executed the sick person is terminated immediately (line 21).

Line 8. Through the *"infect"* procedure (lines 11–16) he tries to infect a number of other persons, given as a random drawing from the Poisson distribution with mean *"contacts."* *"Poisson"* is an **integer** valued system procedure.

Line 9. Further actions are postponed till the next day. System time is measured in days in this example.

Line 10. If the sick person is never treated, the local sequence control will finally leave the operation rule through **end**, whereby he becomes terminated (cured). He also leaves the system.

Line 14. When a person chosen at random is contacted, the probability of infecting him is *"prinf"* multiplied by the probability that he is previously uninfected. If the value of *draw* is **false** nothing is done, and therefore the identity of the contacted person is of no concern.

Line 15. A sick person is generated and included in *S*, which is the *"environment"* set of the person calling the procedure (line 8). The former becomes *last(S)*. His first active phase is scheduled directly, whereby the number of uninfected persons is updated (line 4) before the next contact is made.

Line 17. The **element** parameter refers to a sick person process.

Line 19. The patient is removed from the set (*environment*) of which it is a member (if any) and, since it is a *"sick person"* process, it is *connected*, and the following statement is executed. This statement (lines 20–24) is a *connection block*, in which the attributes of the connected sick person are accessible.

Line 20. *"symptoms"* is an attribute of the connected sick person (cf. line 3).

Line 21. The sick person process is terminated; i.e., the event notice and reactivation point belonging to it are both deleted.

Line 22. *"environment"* is an attribute of the connected sick person. The **for** statement amounts to a scanning of this set, because the controlled statement (line 23) removes the current element *X* from the set (line 19 of the new *treatment* process).

Line 24. A patient without symptoms is given mass treatment with probability *"probmass"* of success.

Line 25. The treatment process leaves the system, and so does the sick person, if he has become terminated.

Line 26. The Main Program starts here. Initially the entire population is uninfected.

Line 27. The first infection is generated, after which the Main Program suspends itself for the simulation period.

Acknowledgments. The SIMULA compiler presently operating has been implemented by the authors as an extension of the UNIVAC 1107 ALGOL compiler, under a contract with the UNIVAC Division of Sperry Rand Corporation. The ALGOL compiler was constructed at the Case Institute of Technology by a team originally headed by Joseph Speroni (now at UNIVAC International). The input-output and backing store facilities, together with facilities for string handling and other valuable extensions to ALGOL 60, have been carried over to the SIMULA system. The authors are indebted to Nicholas Hubacker (UNIVAC International) for help in finding the way through the ALGOL compiler.

The authors wish to express their sincere thanks to those who have contributed to the development of the SIMULA project, in particular to: Bernard Hausner, of Rand Corp., whose experience with SIMSCRIPT had no little impact at an early stage; Ken Walter, of Case Institute of Technology, who boosted the project during a few summer months; Björn Myhrhaug, whose contribution to the implementation has been indispensable; and Sigurd Kubosch, who is rapidly becoming indispensable. Dag Belsnes and Sigurd Kubosch have given valuable comments and criticism concerning the form of the present paper. The authors also wish to thank the Programming Languages editor of this journal for his assistance.

RECEIVED OCTOBER, 1965; REVISED MAY, 1966

REFERENCES

1. NAUR, P. (Ed.). Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6* (Jan. 1963), 1–17.
2. DAHL, O.-J., AND NYGAARD, K. SIMULA—a language for programming and description of discrete event systems, introduction and user's manual. Norwegian Computing Center, Forskningsveien 1 B, Oslo 3, Norway, May 1965.
3. HOARE, C. A. R., AND WIRTH, N. A. Contribution to the development of ALGOL 60. *Comm. ACM 9* (June, 1966), 413–432.
4. CONWAY, M. E. Design of a separable transition-diagram compiler. *Comm. ACM 6* (July 1963), 396.
5. WEIZENBAUM, J. Symmetric list processor. *Comm. ACM 6* (Sept. 1963), 524–544.
6. WEIZENBAUM, J. Letter to the editor. *Comm. ACM 7* (Jan. 1964), 38.

Lecture 14. Simulation Languages 2

Papers to study for the exam

- T.W. Tewoldeberhan, A. Verbraeck and V. Hlupic. Implementing a discrete-event simulation software selection methodology for supporting decision making at Accenture. Journal of the Operational Research Society (2010) 61, 1446-1458.

Journal of the Operational Research Society (2010) 61, 1446–1458 © 2010 Operational Research Society Ltd. All rights reserved. 0160-5682/10

www.palgrave-journals.com/jors/

Implementing a discrete-event simulation software selection methodology for supporting decision making at Accenture

TW Tewoldeberhan[1], A Verbraeck[2] and V Hlupic[3]*

[1]*Oracle Corporation, Chicago, USA;* [2]*Delft University of Technology, Delft, The Netherlands; and* [3]*University of Westminster, United Kingdom*

For large international companies with their own simulation team, it is often hard to make a decision related to selection of new discrete-event simulation software. This paper presents a comprehensive discrete-event simulation software selection methodology that has been successfully used for decision making at Accenture consulting company. Accenture already used a simulation tool at the start of the project, but wanted to find out whether the current tool used still was the most appropriate one for its needs, and to evaluate the latest discrete-event simulation tools. The developed methodology consists of two phases: phase 1 quickly reduces the long list to a short list of packages, and phase 2 matches the requirements of the company with the features of the simulation package in detail. Successful application of the proposed methodology indicates its possible application for decision making in other large organisations, provided that the study is performed by a third party to avoid risks of influencing the outcome of the selection process.

Journal of the Operational Research Society (2010) **61,** 1446–1458. doi:10.1057/jors.2009.119
Published online 14 October 2009

Keywords: simulation; software selection; methodology

Introduction

Currently the market offers a variety of discrete-event simulation software packages. Some are less expensive than others, some are generic and can be used in a wide variety of application areas whereas others are more specific, and some have powerful features for modelling whereas others provide only basic features. Modelling approaches and strategies are different for different packages. There are many properties that make each discrete-event simulation package different. All discrete-event simulation packages have their particular strengths and weaknesses. This makes decision making related to the selection and purchase of a simulation package difficult. Buying the appropriate discrete-event simulation package is important and can save a lot of money. On the other hand, a selection of inadequate package can result in delayed and disrupted projects, and a loss of time and resources.

The need for a structured and standardised approach to simulation software selection is evident: 'Many organisations choose a tool or set of tools without establishing formal evaluation criteria or thoroughly examining tools. Instead, they frequently base their decision on highly visible attributes such as documentation or look and feel, rather than on quality and support of a specific method. It is now time to find ways to consistently, objectively evaluate a tool's utility and appropriateness' (Chikofsky *et al*, 1992). A survey by Christy and Watson (1983) discovered that a majority of the companies select simulation software that is championed by someone in the organisation or recommended by someone external to the organisation, and a minority of companies used a structured approach with a checklist of evaluation criteria. In addition, the large number of discrete-event simulation software packages available currently in the market can create confusion and be overwhelming for a newcomer to the field (Banks and Gibson, 1997), (Banks and Carson, 2004).

This paper presents a comprehensive discrete-event simulation software selection methodology successfully implemented at Accenture. Accenture is the one of the world's leading management and technology services organisations. Accenture has a large team of 40 simulation experts, who operate on a worldwide basis in many types of projects for a wide range of customers. Before the start of this project, the Accenture simulation team used two different simulation packages, but they wanted to standardise on one package. The team applies simulation techniques to many different problem domains. The main ones are helping clients in defining strategies, designing business processes, analysing performance and extracting customer's experience. For the team, having one generic discrete-event package instead of several packages increases model reusability, staff interchangeability and reduces model development time, training costs, and

**Correspondence: V Hlupic, University of Westminster, Westminster Business School, 35 Marylebone Road, London NW1 5LS, United Kingdom.*
E-mail: hlupicv@wmin.ac.uk

purchasing costs. The outcome of this research provided the basis for making a decision on which discrete-event simulation package Accenture will use for the next few years.

This paper is structured as follows. First, the background of the project is described, introducing the state of the field and presenting selection methods available from the scientific community. Second, the methodology that was developed and implemented at Accenture is described. Third, the results obtained by applying the methodology are presented and discussed. Finally, conclusions and issues for further research are addressed.

Research background

Due to the problems associated with decision making related to simulation software selection, any research in this area is of interest for academics and simulation practitioners. Many researchers have carried out surveys on available packages for different purposes. However, there are only a limited number of papers that describe methods to perform an evaluation of discrete-event simulation packages. Hlupic (1997) developed a software tool (SimSelect) that selects simulation software given the required features. Hlupic *et al* (1999) created a comprehensive framework for simulation software evaluation, and Nikoukaran *et al* (1999) have modified this framework and produced its hierarchical representation. Most recently, Bosilj-Vuksic *et al* (2007) have adapted the original framework by Hlupic *et al* (1999) to the area of business process modelling. Examples of other studies related to simulation software evaluation and selection are provided below.

Banks (1991) as well as Banks and Carson (2004) suggested a scoring procedure for simulation packages being evaluated that includes assigning a score between zero and 10 for each criteria and normalising the results using weighting of the criteria. Pidd (1989, 2004) provided some general advice regarding the selection of discrete simulation software. Proposed guidelines for simulation software selection include, in general, the development of a preliminary model of application, consideration of available resources and future applications, examination of the available software, and asking the vendors for assistance if possible.

Lyons *et al* (2000) compared and evaluated three discrete-event simulation packages by modelling the operations of a small enterprise. They have produced a simple list of criteria that was used for evaluation of a simulator, simulation language-based package and simple queuing simulation application. Sahay and Gupta (2003) developed a comprehensive list of software selection criteria in the context of a supply chain solution. Their study was focused on decision-support systems software in general rather than on simulation software. Feinstein and Cannon (2002) have proposed a framework for evaluation of simulation models, whilst Shanklin *et al* (2001) described the evaluation of two process-simulation software packages used in modelling industrial, biotechnology-related processes. Balci (2001) has proposed a methodology for certification of modelling and simulation applications using the analytic hierarchy process to evaluate criteria (indicators). Tobias and Hofmann (2004) compared and evaluated four programming libraries for agent-based computer simulation of social interventions. They have developed a simple list of criteria and a rating system that was used for evaluation and selection of the most suitable software.

Earlier relevant studies on simulation software evaluation and selection include, for example, a discussion on factors to consider before selecting manufacturing simulation software by Deaver (1987). The author identified a need to thoroughly analyse system requirements before selecting simulation software, as simulation packages vary widely in capability. Some of the factors that should be considered before simulation software selection include: identification of potential simulation users, consideration of future training for employees, determination of types of systems to be simulated, analysing the resources currently available, and consideration of the amount of time that is to be dedicated to simulation. Ekere and Hannam (1989) evaluated four simulation languages using a simple Likert scale, whilst Mackulak *et al* (1994) conducted a survey of simulation practitioners in order to ascertain the important features of industrial simulation environments.

Furthermore, Holder (1990) proposed a structured approach to the selection of simulation software. This approach suggests that software selection should commence with a consideration of the available resources within the organisation, and a determination of the simulation objectives. This should produce a reduced list of software products that are to be evaluated prior to selection. The study was concluded with the suggestion that vendors should be asked to assist both in the evaluation of software features and in test modelling.

In addition, Davis and Williams (1994) illustrated the evaluation and selection of simulation software using the analytic hierarchy process method. They claimed that a potential user faces the problem of selecting the package that best matches the requirements at some preferred cost. A simple three-step method for the selection of simulation software was proposed by Bovone *et al* (1989). The purpose of this method was to obtain the weights that can express the importance of software evaluation criteria with regard to the simulation objectives. Separate software evaluation tables should be constructed both for the conceptual design (initial modelling stage), and for the detailed design (detailed modelling). On the basis of an evaluation of several simulation packages using this method, it was concluded that no product was superior to the others with regard to both software purposes (conceptual and detailed design). Hlupic and Paul (1996) propose a six-step methodology that addresses activities to undertake and factors to consider during the process of simulation software selection. Although the issues included in the methodology are useful, this methodology does not provide an algorithm for ranking evaluated packages.

The main findings drawn from the research studies analysed reveal that although many studies have recognised the

Table 1 A summary of the main contributions of the relevant literature on simulation software selection

Authors	*Main contribution to simulation software evaluation literature*
Hlupic (1997)	Developed a software tool (SimSelect) that selects simulation software given the required features
Hlupic *et al* (1999)	Created a comprehensive framework for simulation software evaluation with more than 300 criteria
Nikoukaran *et al* (1999)	Modified Hlupic *et al* (1999) framework and produced its hierarchical representation
Bosilj-Vuksic *et al* (2007)	Adapted the original framework by Hlupic *et al* (1999) to the area of business process modelling
Banks (1991), Banks and Carson (2004)	Suggested a scoring procedure for simulation packages being evaluated that includes assigning a score between zero and 10 for each criteria and normalising the results using weighting of the criteria
Pidd (1989, 2004)	Provided general guidance regarding the selection of discrete simulation software
Lyons *et al* (2000)	Produced a simple list of criteria, then compared and evaluated three discrete-event simulation packages by modelling the operations of a small enterprise
Feinstein and Cannon (2002)	Proposed a framework for evaluation of simulation models
Sahay and Gupta (2003)	Developed a comprehensive list of software selection criteria in the context of a supply chain solution
Shanklin *et al* (2001)	Described the evaluation of two process-simulation software packages used in modelling industrial, biotechnology-related processes
Tobias and Hofmann (2004)	Developed a simple list of criteria and a rating system, compared and evaluated four programming libraries for agent-based computer simulation of social interventions
Balci (2001)	Proposed a methodology for certification of modelling and simulation applications using the analytic hierarchy process
Deaver (1987)	Provided a discussion on factors to consider before selecting manufacturing simulation software
Ekere and Hannam (1989)	Evaluated four simulation languages using a simple Likert scale
Mackulak *et al* (1994)	Conducted a survey of simulation practitioners in order to ascertain the important features of industrial simulation environments
Holder (1990)	Proposed a structured approach to the selection of simulation software
Davis and Williams (1994)	Illustrated the evaluation and selection of simulation software using the analytic hierarchy process method
Bovone *et al* (1989)	Proposed a simple three-step method for the selection of simulation software
Hlupic and Paul (1996)	Proposed a six-step methodology that addresses activities to undertake and factors to consider during the process of simulation software selection

importance of software selection, none of them propose a structured methodology implemented in practice. It is apparent that there is a limited comparative analysis of alternative simulation software tools and even less is empirically based (Lyons *et al*, 2000). Furthermore, due to the intensive use of simulation in various environments, and the costs involved in purchasing simulation software, hardware and training, further research in simulation software evaluation and selection seems to be needed and applicable. The need for having an efficient selection method for discrete-event simulation packages is also increasing as the simulation application domain broadens (Hlupic and de Vreede, 2005; Robinson, 2003; Shannon, 1992) and as the number and type of discrete-event simulation packages increases (Kelton, 2006; Swain, 1999). Companies and institutions that use simulation do some research for their own use, and they use different methodologies and approaches. However, projects and published results on how to effectively conduct evaluation and selection process of discrete-event simulation software in practice are limited.

Two of the most elaborate evaluation frameworks are described in Hlupic *et al* (1999) and in Nikoukaran *et al* (1999). These frameworks are structured, and pay attention to a rich set of criteria on which simulation packages can be compared. It is, however, difficult to base a decision for a large multinational company on these criteria, as lists of criteria provide only a comparison, without weighting and without a method to determine the relative weights, and the weight differences between parts of the simulation team. Table 1 shows a summary of the main contributions of the relevant literature on simulation software selection. An applied structured methodology for evaluation and selection of simulation software developed in this research is described in the following section.

Selection of simulation sofware: a methodology

This section describes in detail simulation software selection methodology developed in this research, in two distinctive phases. The first subsection describes phase 1 of the methodology, related to creating a list of required software features, identifying the overall vision for the simulation team, extraction and weighting of criteria, as well as evaluation and ranking of specific simulation packages. In the

second subsection, phase 2 of the methodology is described involving detailed evaluation of simulating packages using a case study approach, conducting experiments, performing sensitivity analysis and final ranking of software.

The first step taken for development of simulation software selection methodology to support decision making was to ascertain the requirement specifications for this methodology. During this process, various sources were used: a review of relevant literature as described in Section 2, practical experiences obtained in various simulation project and software selection activities, and a consultation with the team of Accenture simulation experts. This has resulted in the following requirements:

- The methodology should support decision-making process.
- The methodology should be objective and should not be biased to a particular software package.
- The methodology should be repeatable in different environments.
- The methodology should be structured.
- The methodology should be implementable.
- The methodology should be simple to follow.
- The methodology should allow all stakeholders to participate in the selection process.
- The methodology should enable parallel execution of some activities as much as possible.
- It should be possible to automate at least some aspects of the methodology, and
- The scoring algorithm used in the methodology should not yield results that are sensitive to small changes in scores and weights.

Using the specified requirements, a two-phase selection methodology has been developed, using action research approach (Burns, 2007). All research participants (researchers and consultants at Accenture) were fully involved in an interactive, iterative process of research decisions related to the phases of methodology development, evaluation criteria chosen used for software selection, weighting of criteria as well as the practical implementation of the proposed methodology. Each important step in development of methodology and its practical application was discussed in detail among all stakeholders (researchers and consultants at Accenture) until consensus was reached and practical decision was made.

Figure 1 shows the general overview of the proposed methodology.

To evaluate simulation packages, and to select the best one for a large company, is a time consuming task unless an efficient methodology is used. Usually, choosing from a list of alternatives requires detailed knowledge of the selection criteria, and on the score of the alternatives on these selection criteria. If there are many alternatives and if the criteria list is long, the evaluation becomes a challenging task. The proposed methodology can facilitate this task efficiently. The methodology was designed to be fast and as objective as possible,

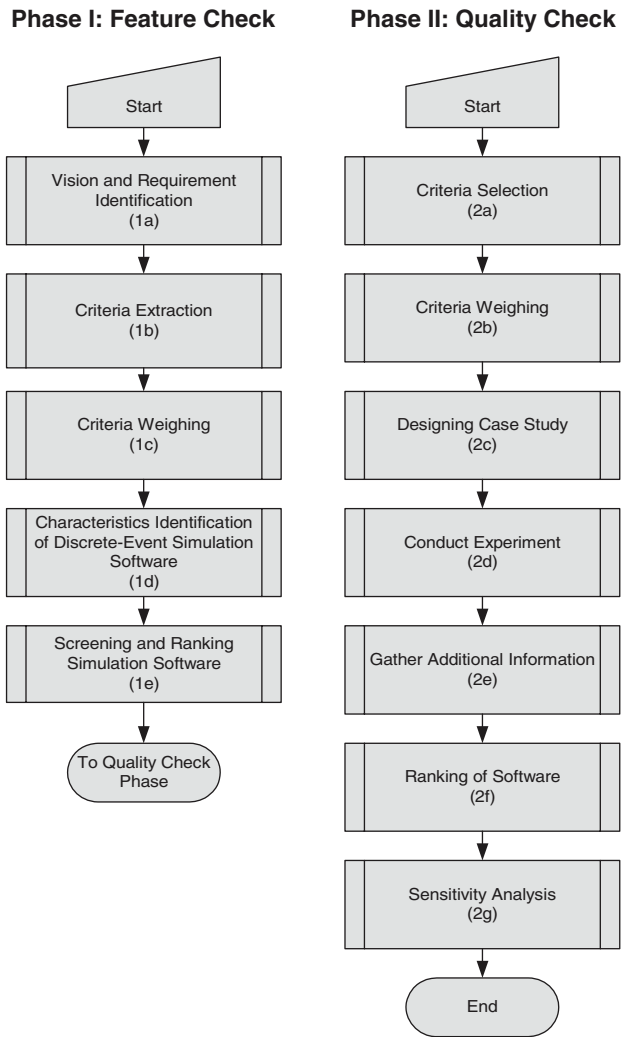


Figure 1 Two-phase methodology for evaluation and selection of discrete-event simulation software.

and it should respond to the specific needs of the simulation team that wants to acquire a new simulation tool.

In the first phase, simulation packages are selected based on the existence of the most important features and criteria. In the second phase, detailed evaluation and analysis are done for packages that satisfy the requirements of the first phase. During the two phases, numerous interactions take place between the simulation team, management team, selection team, and simulation package vendors. The selection team is defined here as the analysts who are responsible for carrying out the research and making the final recommendations for selection. They can be employees of the company or external consultants. The interactions between the team members are indicated in Figure 2. In addition, internal customers of simulation should be included in the selection process as they are valuable resources for specifying the relevant criteria and features that needs to be considered. The vendors have only a limited interaction with company management and

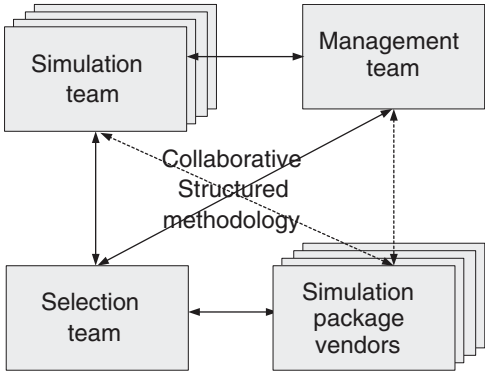


Figure 2 Interactions during the Selection Process.

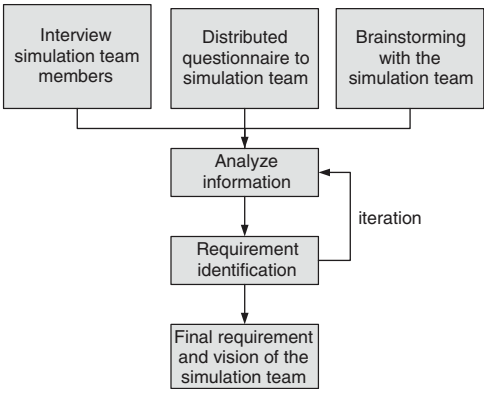


Figure 3 Vision and requirement identification.

the simulation team, and most of their interaction with the selection team.

Phase 1: feature check

In phase 1, a list of required features is created, and a wide list of discrete-event simulation software packages are checked for availability of these required features. To accomplish this 'feature-check' phase, the following steps are taken: vision and requirement identification, criteria extraction, criteria weighting, characteristics of discrete-event simulation software identification, and screening and ranking the simulation software. Those packages that satisfy the first phase are transferred to the second phase. The detailed activities of phase 1 is as follows.

Vision and requirement identification (1a) Identifying the overall vision of the simulation team for the near future is necessary for identifying the functional requirement for the simulation packages. The vision covers items such as: current and near future application area of discrete-event simulation, type of product/service, types of customers, business process or work routine, business objective, and similar aspects. A questionnaire is one of the main methods to be used to extract the necessary information from the team. In other cases where the selection is carried out internally within a company, other methods such as workshops or a brainstorming session may be used, preferably with the support of the group collaboration tools (Hlupic and de Vreede, 2005) (see Figure 3). The exact purpose of the questionnaire is to get the following items: future goal and plan of the simulation team regarding discrete-event simulation studies, main fields on simulation in which the team is involved, current use of discrete event simulation package in the team as well as near future use, features the team experts need from a simulation package, constraints the team expert have in the current simulation packages they use, and a criteria list for evaluation of packages with a ranking. In addition, interviews and library research can be used. Results that are found to be inconsistent during the analysis are discussed again with the team. The criteria list is, then, categorised using the Hlupic *et al* (1999) and Nikoukaran *et al* (1999) frameworks.

Criteria extraction (1b) Based on the vision identified and additional sources, criteria are extracted by the selection team, in close co-operation with the simulation team. The additional sources for input are external experts on simulation, the company's clients, internal project findings and reports, and literature. Criteria are extracted from the vision by asking questions such as: '*What are the main features of a simulation package that make it applicable to a wide range of problems?*', '*What functionalities are often used during an engagement*?', '*How does the team address a client's problem?*', '*How does the team use the simulation packages?*', and '*What are the main objectives of the simulation projects carried out?*'.

For the additional inputs, literature, research projects and clients are consulted. From literature and research projects, the importance of certain criteria for particular application areas can be extracted and discussed with the simulation team and management. The reason behind using this approach is that different features of simulation packages are required for different application areas. For example, optimisation might be considered very important for supply chain management whereas external connectivity might be considered to be more important for real-time simulation. The initial list of criteria should be discussed with the simulation team for additional feedback and consistency.

Criteria weighting (1c) After the extraction of the criteria, weights based on the level of importance are given to each criterion. In addition, to increase the efficiency of the selection, hard (criteria that is essential and must be satisfied by any means) and soft (optional) criteria are identified. To cope with the various locations of the simulation team members—even when the simulation team is based in one location, the team members will be external with clients on various projects—questionnaires are used to weigh the criteria. The result is then analysed and presented to the team for additional feedback. E-mail is ideal to update the team with the latest progress.

For giving weights to the criteria, each member input from the simulation team is considered. Weighting is done by first defining the scale. A five-level scale is defined using scales 1–5. Five indicates very important and one indicates least important. After weight definition, members of the team give their personal weight to each of the criteria. For each criterion, the average weight is then calculated from the weights obtained from the members. Afterwards, standard deviations and averages are analysed and compared with the raw data. The results of the analysis and additional input gathered from other sources (eg, literatures, clients) are then presented to the team members. Finally results are discussed to decide if they need to be adjusted. Modification of the weights is then done based on the feedback of the team.

In addition to assigning weights to the criteria, identification of hard (essential) and soft (optional) criteria is also made. Hard criteria are obtained from the weights. Those criteria that are very important (close to 5) are considered to be hard criteria. In this way, hard and soft criteria are separated. The hard criteria obtained are then discussed with the group so as to be sure there is nothing to be added or removed. The team should be well aware that a simulation package that does not satisfy just one of the exit criteria is immediately taken out of consideration.

Characteristics identification of discrete-event simulation software (1d) For the evaluation processes, discrete-event simulation packages available in the market are collected. The sources of information used are conference proceedings, research papers, vendor websites, input from the simulation team, and simulation practitioners. As the focus is on discrete-event simulation packages, packages that are, for example, used specifically for continuous simulation and Monte Carlo simulations are not considered. However, packages that have discrete-event simulation capability as well as other capability are considered as possible candidates.

After the final list of discrete-event simulation packages is prepared, the characteristics of the packages are identified. Different ways can be used to extract the characteristics. One of the ways is consulting the vendors, experimenting with the demo version of the packages, referring to research papers that describe the experiences with simulation packages, and consulting simulation experts (outside and inside the company). In most of the cases, however, vendors can best be consulted using questionnaires. The questionnaire is prepared to address general issues and specific questions regarding the packages. The general questions address issues such as modelling approach, simulation software class, simulation type, and application area.

Specific questions about the features of the packages, supplied by the vendors, are categorised based on a framework. The criteria categories are:

- *Model development*: questions that are related to model development and modelling approach. This includes features like model-building tools, reusability of libraries, coding aspects, conditional routings, queuing policies, and other related aspects.
- *Input modes*: This includes input modes such as interactive mode, batch mode, from external files (spreadsheets, database, text files, etc), and random number generation.
- *Testing and efficiency*: The questions in this category include debugging features and error control.
- *Execution*: The questions in this category covers features such as multiple replications, automatic batch run, warm up period, and reset capability.
- *Animation*: The questions in this category cover animation development features, animation running features, display features, and icon development.
- *Output*: The questions in this category include features used for displaying outputs either in terms of numbers or business graphics. It also includes capability to communicate with external packages.
- *User*: The questions in this category include cost, compatibility of packages with different operating systems and hardware.

The questionnaire should be designed to gather as much data as possible. As the data obtained from vendors can be incorrect, validation of the data is done by using the other methods listed above that is, experimenting with the package, reading research papers, consulting simulation experts, and checking vendors website. For the experimentation, demo and full versions of the packages are used. Experiments are done by building a small model and trying to address each criterion (and feature) in the criteria list, by checking most of the commands available, reading help files about the features of the package, running and debugging small models and demo models, and viewing and experimenting with already existing demo models provided by the vendor or by other users. Another method used for extracting the characteristics of the packages is library research. Different research papers, conference proceedings, and vendor websites are used.

Screening and ranking of simulation software (1e) For ranking packages, the method of Multi-Criteria Decision Analysis (MCDA), which is Simple Multi-Attribute Rating Technique (SMART) (Lootsma, 1999) (Bots and Hulshof, 2000), was used. Other approaches such as Multiplicative Analytic Hierarchy Process and Additive Analytic Hierarchy Process could also be used, but the project team decided to focus on SMART method for the purpose of simplicity.

Once the characteristics of the packages are identified, screening and ranking is performed. Screening is done by using the hard (essential) criteria that are obtained in the previous steps. The packages that don't satisfy one or more of the hard criteria are removed from the potential list. Those that satisfy all the hard criteria are kept for further ranking. Testing is done by checking whether the packages satisfy the requirement set by each criterion.

After the packages are filtered (by using the hard criteria), further ranking is done on the remaining packages (using soft criteria) to select a maximum of 10 best packages. The number 10 was chosen to restrain the solution domain, yet being representative. For ranking, the following procedure is used:

- *Define score*: if a package satisfies a particular criterion, it scores 1 otherwise 0.
- Give score to each package for all the criteria using 1 and 0.
- By following a simple MCDM method called SMART, criteria weights were multiplied by the scores of each package for all the criteria.
- The sum of the product of weight and score for each package is compared.
- The first 10 packages that have the highest scores are selected for further investigation.

Two points need to be mentioned here. First of all, the ranking is done by only checking whether a package contains a particular feature or not. It is not based on the quality of a feature. The quality of each feature is checked in the next phase of the analysis. Second, the aim of the ranking is only to filter the best 10 packages in the list. This, however, doesn't necessarily mean that the one on the top of the list is going to be the best of the overall evaluation process. The best package for the company is known once the quality of each feature is tested for all the remaining packages in the second phase of research.

Phase 2: quality check

In phase 2, discrete-event simulation software packages are evaluated for their quality. To accomplish this quality-check phase, the following steps are taken: Criteria selection, criteria weighting, designing a case study, conduct experiments, gather additional information, ranking of software, sensitivity analysis. The detailed activities of phase 2 are described hereafter.

Criteria selection (2a) To evaluate the packages quality wise, some criteria have to be selected. The criteria address features that are important to analyse the quality of the package, considering the use that the simulation team plans to make of the simulation package. Features that need to be evaluated are selected by brainstorming with team members, selection team judgments and information already available from the first phase of the project.

Criteria weighting (2b) The criteria obtained are weighed by following the method mentioned in the Criteria weighting (1c) step in the first phase.

Designing case study (2c) The purpose of the case study is to help evaluate the features of interest by carrying out a small simulation study using each of the packages that remained after the first phase. The case study is prepared in such a way that it addresses most of the criteria in the list of the first step of phase 2. It does not require a specific modelling approach—as we do not want to introduce a bias to a certain modelling approach. The case study should be small but fairly complicated, and it should represent a typical problem that is representative for the type of simulation study carried out by the simulation team.

The case study can best be constructed from scratch; when taking a previous project as case study, there can easily be a bias towards the currently used simulation package—in positive or negative sense depending on the success of the project. After choosing a typical case, it needs to be extended to address the criteria that need to be evaluated. The case study evolves by using feedback from the team members and, if necessary, outside experts. As the case study is on paper, it is again easy to involve the simulation team members in the selection process because the case description can be distributed using e-mail.

Conduct experiment (2d) Experimenting on the packages is done by modelling the case study provided by Accenture. The modelling steps are as follows:

- *Conceptualisation*: conceptual model design based on the package's modelling approach. This can be difficult, because the package may ask for a different modelling approach than what the selection team is used to.
- *Specification*: actual model construction. All experiments with the different simulation packages should use the same data sets and distribution functions as input, so the results in the output are truly comparable. It is important to keep a logbook for immediately writing down the experiences with the simulation packages.
- *Execution*: running the model and creating output to analyse further. Run-time and animation speed can, for instance, be analysed here if these are part of the quality criteria.
- *Output*: Analysing the output provided. If the quality criteria include specific aspects of output types or post-processing of the output, it can be done in this step.
- *Scenario management*: analyse different scenarios if needed.

For the experiments, vendors should be involved. Vendor participation is important because they can show the most efficient way of modelling the case using their packages. Furthermore, their level of cooperation in the project is one way of investigating the support level of the vendor. The vendors can be given three options for participation:

- Vendor's work on the case study is transparent to other vendors (vice versa) and they know with whom they are competing.
- Vendor's work is not transparent to other vendors and they only know with whom they are competing.

Table 2 Sensitivity analysis approach

Parameter	*Changes*	*Type of change*
Criteria weight	Value of weight	Set weights to 0% and 100% one by one
Criteria	Number of criteria	Only look at top-N of criteria
Package score	Value of score	Set score for whole category to minimum

- Vendors only work on the case study and vendors don't know with whom they are competing.

Consensus should be reached among the vendors on the way of working. The experiment is conducted based on their cooperation strategy.

Gather additional information (2e) In addition to conducting an experiment, additional information has to be collected to enrich the evaluation quality. The additional information collected is mainly considering the limits of the packages, and advantages and disadvantages of the packages. The additional information is collected from articles and research papers, and simulation experts who have used the packages. A very good source of input here can be a user group of the simulation package.

Ranking of software (2f) By using the output of the experiments and the additional information collected from various sources, a score for all the packages can be given. The following method is used:

- Define a scale for scoring: in this case, a four-level scale is defined. Scaling is done using scale 0–3.3 indicates 'Good', 2 indicates 'Sufficient', 1 indicates 'Insufficient', and 0 indicates 'Feature doesn't exist'.
- For all the packages, assign scores for each criterion based on the defined scale. The selection team, that has followed all efforts for working out the case study in detail, does the scoring.
- The detailed scores for all the criteria is averaged based on the criteria categories.
  For $\forall j \in P$, where P is the set of packages involved in the selection, then $\sum_{i=1}^{N} w_i s_i$ gives the score of each package according to the weights of the criteria, where $W = \{w_i\}$ is the set of weights involved in the evaluation.
- Again, by following the MCDM method SMART, criteria weights are multiplied by the scores of each package for all the criteria.
- The sum of the product of weight and score for each package is compared.

The result of the ranking is then given to the team members to get feedback. After this ranking, the packages receiving the highest score can be considered better than the ones scoring lower, as the comparison has been based on the (perceived) quality of the simulation package in a representative case study.

Sensitivity analysis (2g) After ranking is done and the best package is known, sensitivity analysis has to be done in order to assess the robustness of the result. A sensitivity analysis is performed because both scores and criteria weights are subjectively generated in MCDM. Table 2 shows the sensitivity analysis approach used.

Practical application of the methodology at Accenture

The methodology has been applied in practice for supporting decision making related to selecting a new discrete event simulation package for Accenture's worldwide simulation team. Following the methodology explained above, a detailed vision of the team was extracted in the first phase. The vision includes the team's strategy, work environment, type of clients, type of project, and team expertise. The criteria were generated with the weights, and categorised with the Hlupic *et al* (1999) and Nikoukaran *et al* (1999) frameworks (see Table 3). Based on the weights, the hard (essential) criteria were identified. The weights are not included in Table 3, as they are Accenture specific and therefore confidential.

The obtained model development criteria in Table 3 reflect the vision of the Accenture simulation team. The team indicated a number of areas that are key to the type of work they do, leading to high weights for those criteria. To give some examples, flexibility and package ease of use in modelling are very important as the team is involved in different application areas of simulation. Therefore, as an example, 'Graphical model building' scored quite high. In the Vendor Category, the maintenance support and documentation is very important because time is a very important issue. Efficient and fast support is highly required in a work environment with tight deadlines. The 'Multiple runs' criterion in the Execution Criteria Category scores high compared to others because of variance reduction. In the Animation Criteria Category, the '3D animation' criterion is for instance not important for the team because the group mainly carries out business process simulation as opposed to manufacturing where it could be more important. In the Testing and Efficiency Criteria Category, examples of high scores are 'Error checker' and 'Interactive debugger' because most of the projects managed by the group are complex and large. Without a good debugger, it is considered difficult to fine tune the model. Within the Output Criteria Category in Table 3, the 'Export to...' criteria are considered important because the output features are common tools for displaying results to clients. For the User Criteria Category, the 'Package application area'

Table 3 Criteria categories as obtained for Accenture

Model development and input category	*Vendor category*
Graphical model building	Documentation
Merging models	Maintenance support
Conditional routing	Pedigree
Statistical distribution	Pre-purchase facility
Queuing policies	Multiple runs
Reuse of user defined modules	Automatic batch runs
Built-in functions	Reset capability
Link to other languages	Start in non-empty state
Coding tools and utilities	Runtime Interaction with user
Input from text files	Warm up period
Input from database	Ability to calculate appropriate warm-up period and replications
Input from spreadsheets	Speed control
Automatic data collection	Self executable versions
Batch input mode	
Interactive input mode	
Random number generators	
Program generator	
Animation category	*Testing & efficiency category*
Integration of animation	Error checker
Library of icons	Interacting debugger
Screen layout	Multitasking
Concurrent animation mode	Display features
Animation on/off feature	Tracing
3D animation	Breakpoints
Animation development feature	Running backwards
	Limits
Output category	*User category*
Standard report generation	Cost
Report customization	Connectivity with internet
Integration with statistical packages	Package interoperability
Integration with other simulation packages	Package link to different animation packages
Exporting data to database	Package has open source code
Exporting data to spreadsheets	Package application area
Exporting data to text files or word processors	Flow oriented modelling approach
Optimization	High level architecture
Output analysis feature	Capability for continuous simulation
Business graphics	Simulation strategy

criterion scored high for the reason that the package has to be generic enough to be applied in different domains.

For the phase 1 evaluation, more than 50 packages were considered. A few software vendors were not able to participate in the project. By using the high weight (essential) criteria from Table 3, the packages were screened. Those packages that didn't satisfy all the essential criteria were eliminated. Packages that satisfied all the hard criteria were (in alphabetical order): Arena 5.0 (Rockwell Software), Kelton (2006), AutoMod 9.1 (Brooks Automation), Enterprise Dynamics 3.1 (Enterprise Dynamics), Extend 5.0 (Imagine That Inc.), ProModel 2001 (Promodel Corporation), Quest (DELMIA Corporation), Simul8 6.0 (Simul8 Corporation), and Witness 2000 (Lanner Group, Inc.).

Furthermore, these packages were ranked based on detailed feature they offer, that is, availability of features. The result indicates that the score for all eight packages is close and the distinction between them is not significant. Without even doing sensitivity analysis, it can be seen that by just using these results, it can't be concluded that one excels or one is the best. Additional analysis needed to be conducted on the quality of each feature to investigate how good the packages were. This was done in the next phase of the project.

In the second phase of the methodology the quality of the eight discrete-event simulation software selected in phase 1, was evaluated in detail. The quality of the packages regarding the following issue was tested:

- *Vendor*: includes vendor pedigree, maintenance support, and documentation.
- *Model development and input*: includes model building and coding aspect, batch processing, library of reusable modules, conditional routing, statistical distributions, queue policy, input modes, automatic documentation, batch input modes, random number generator, and standard commands.

Table 4 Final ranking of simulation packages

Criteria	*Weight*	*Package A*	*Package B*	*Package C*	*Package D*	*Package E*
Vendor	5.6	3.00	2.00	2.67	3.00	2.33
Model development & input	9.5	2.71	2.57	2.71	2.00	2.43
Execution	7.6	2.00	2.33	2.33	2.00	2.00
Animation	6.3	2.67	2.33	1.33	2.67	1.00
Testing & efficiency	7.6	2.38	2.38	2.50	2.00	1.75
Output	6.6	2.33	1.67	2.33	2.00	2.67
Experimental design	5.9	3.00	2.00	2.00	3.00	2.00
User	5.6	2.00	1.50	2.50	2.00	3.00
Total		136.9	117.3	127.1	125.1	117.1
Rank		1	4	2	3	4

Table 5 Results of the sensitivity to number of criteria—ranking of the packages

Criteria considered	*Package A*	*Package B*	*Package C*	*Package D*	*Package E*
The 1st important criterion	1	2	1	4	3
The first two important criteria	3	2	1	5	4
The first three important criteria	3	2	1	5	4
The first four important criteria	2	3	1	5	4
The first five important criteria	1	2	2	3	4
The first six important criteria	1	2	2	3	4
The first seven important criteria	1	4	2	3	5
The first eight important criteria	1	4	3	2	5
All	1	4	2	3	4

- *Execution*: includes multiple run, automatic batch run, reset capability, start in non-empty state, interaction with user, and unit conversions.
- *Animation*: includes icon and animation development, screen layout, and animation running.
- *Testing and efficiency*: includes validation and verification tools, display features, tracing, step functions, breakpoints, model size, and model speed.
- *Output*: includes report, integration with external packages, and business graphics.
- *User*: required experience and cost of package,
- Experimental design.

Based on these criteria, a case study that considered most of the criteria was created. Out of the eight package vendors, five of them participated in the evaluation. Three others were not able to participate for several reasons. All vendors chose the same participation level that is, their work on the case study is transparent to other vendors in the list and they know with whom they are competing. After the evaluation, a document was prepared for each of the packages and sent for additional feedback to the respective vendors so that a clear image of the packages could be produced. Vendors were not given evaluation details of the packages of other vendors. Based on all this material, a detailed analysis of each package was produced. After the analysis, a total ranking of the packages was made. The overall ranking is shown in Table 4.

The scores indicate that Package A is the first. This is so because Package A scores high in most of the criteria. Package C comes second. Package B and E scored almost the same.

Sensitivity analysis on the results was conducted as getting the result is not the last step in methodology. Sensitivity analysis based on criteria weights showed the result obtained, that is package A, is stable. The sensitivity was checked by setting each criteria category to maximum (value of 10) and minimum (value of 0) one at a time. In addition, equal weights were considered as well. In all the results, it was seen that the score of Package A was still the highest. In other words, the result is stable to changes in weights of the criteria one at a time.

Sensitivity analysis was also done by limiting the number of criteria. The first few important criteria were considered. The results obtained are given in Table 5.

The result shows that the result is sensitive to the number of criteria considered. If very few important criteria category are considered, then Package A is not the first. However, as the number of criteria considered increases, Package A gets the first position. Sensitivity analysis based on Package A score was also done. By varying the score of Package A, the sensitivity of the output was calculated. Table 6 shows the results obtained.

The results show that Package A still holds its number one position even if its score for 'Testing & efficiency', 'Output', or 'User' is set to minimum. However, other changes such as

Table 6 Results of the sensitivity to Package A scores—ranks of the packages

Case considered for Package A package	*Package A rank*	*Package B rank*	*Package C rank*	*Package D rank*	*Package E rank*
Poor score (=1) for Vendor criteria category	2	4	1	3	4
Poor score (=1) for Model dev. & input criteria category	3	4	1	2	4
Poor score (=1) for Execution criteria category	3	2	1	5	4
Poor score (=1) for Testing & Efficiency criteria category	1	3	1	2	3
Poor score (=1) for Animation criteria category	2	4	1	3	4
Poor score (=1) for Output criteria category	1	4	2	3	4
Poor score (=1) for User criteria category	1	4	2	3	4
Poor score (=1) for Experimental design criteria category	2	3	1	2	3

low value on 'Model development and input' lowers it to 3rd position and Package C becomes the first.

Out of the list of criteria category, the most subjective criteria are Vendor and User. Vendor criteria category is very subjective because it includes criteria such as quality of maintenance support, which might differ slightly from person to person. User criteria category is also sensitive because cost is not usually fixed (cost is negotiable) and required experience depends on background of people. As these two criteria category are very subjective, sensitivity analysis on both criteria was done. The result shows that the Package A is sensitive to 'Vendor' criteria category and not to 'User' category. Changes made to the score of Package A for 'Vendor' criteria changes the ranking. Setting score of Package A for 'Vendor' category from value of 3 (good) to value of 1 (not good) changes the ranking. In that case, Package C becomes the first. However, changes made to other packages score (for Vendor category) don't make any difference. Package A is stable to changes in the score of other packages for 'Vendor' as well as 'User' criteria category. This allows the organisation to re-evaluate the obtained scores for those categories that would change the overall result from Package A to Package C.

Discussion and implications of this research

This research produced a structured two-phase selection methodology that has been successfully used to support decision making in practice. During the process of simulation selection, objectivity was considered at all times. Some of the issues considered in this respect are:

- The way of modelling: which are different in different simulation packages, were not fixed so as to be objective as possible.
- Involvement of vendors: vendors were involved in the model development of the case study so that efficient modelling approach for each of the packages to be illustrated.
- Involvement of all simulation team members through the process of action research.
- Facilitation of interactions among simulation team members.

The methodology used is generic and can be applied to support decision making related to other types of evaluation and selection cases. From the practical experiment, it was seen that it is simple and effective to implement in a large organisation with a distributed setting. In addition, the methodology allowed for checking the robustness of the final selection. The methodology developed satisfies all major criteria identified at the beginning of this research project: it supports decision-making process, it is objective and not biased to a particular software package, it can be repeatable in different environments, it is structured, easy to implement, simple to follow, it allows all stakeholders to participate in the selection process, it enables parallel execution of some activities, at least some aspects of the methodology are automated and the scoring algorithm used in the methodology does not yield results that are sensitive to small changes in scores and weights.

The results obtained through applying this methodology, indicate that the package chosen is a package that best fits Accenture simulation team's needs. The package chosen was not the best simulation tool in general, but the one that best fitted the need of Accenture simulation team at that time. Sensitivity analysis indicates that the result is fairly stable to changes made in the weights of the criteria, scores for the alternatives, and limiting the criteria considered. The criteria considered were just mirrors to the vision and need of the team. Some of the most important needs were:

- *Complexity of problems*: Most projects the team deals with are complex. This point implies the use of tools that can help to model correctly.
- *Time*: Time is very important for the team. The work environment of the team encompasses strong deadlines.
- *Simplicity of package*: Simplicity is important because it decreases the time it takes to model.
- *Different precision required for different applications*: Quality of random number generators is important because of the different precision used in different simulation application areas.
- *Client interaction*: Ease of use and good ergonomics are important features for end users.

The criteria list generated was similar to the criteria list mentioned by Hlupic *et al* (1999); Nikoukaran *et al* (1999) and, most recently by Bosilj-Vuksic *et al* (2007). However, the weighting of the criteria depends on the application area. Hence it is not possible to compare the weights of the criteria with weights specified for other organisations. Even though the package that scored the highest was found out to be the best for Accenture simulation team, some other packages considered are also highly competitive. The five packages considered in the second phase were able to construct the case study under investigation with different level of quality.

Although the developed methodology is relatively simple to implement, it is robust enough to provide comprehensive support for simulation software selection, and no other comparable methodology is available in the literature.

Conclusions and future work

The literature research shows that other software selection methods available are largely theoretical. However, the methodology proposed in this paper has successfully been used for decision making in one of the world's largest management and technology consulting firm. The result was found to be stable for reasonable changes in the scores and weights.

It is believed that the software selection methodology provided in this paper represents a research contribution towards a standardisation of simulation software selection in business environments. On the whole, the evaluation and selection methodology used was found out to be efficient and objective, providing a robust decision support tool. As the main concept behind the methodology is to be objective and effective, the result found is reliable. In addition, the methodology used is so generic that it can be applied in different application domains. It is believed that this methodology is more comprehensive than those few found in the literature. As it has been successfully tested in practice with Accenture, it is reasonable to expect that this methodology can be used by any other company or educational institution planning to purchase simulation software.

There are several avenues for future research in this area. Further application of the proposed methodology in other companies would provide a useful feedback on its effectiveness. As the important element of this methodology is the use of teams during selection process, it would be useful to investigate the usability of Group Support Systems (GSS) for this process (Hlupic and de Vreede, 2005), (Vogel and Vreede, 1999). GSS can be used to speed up the process of selection and improve the interaction among team members by providing:

- Computer-based voting mechanisms,
- Computer-based ranking of alternatives,
- Automation of routine tasks,
- Teleconferencing for team located in different places.

For future research, the methodology can be expanded by considering two criteria that were not analysed in detail: Optimisation and Random number generators. Conducting tests on optimisation and random number generator is a specific task that would require many theoretical tests, algorithm checks, and statistical tests. Conducting such kind of tests could take months. As conducting such an extensive test was beyond the scope of the project, recommendations from simulation experts and research papers were used. For random number generators, for instance, recommendations from experts such as Banks and Carson (2004); Law and Kelton (2000) were used. Finally, for a user to successfully use and implement this methodology, the necessary skills and knowledge of the simulation domain is required.

Acknowledgements—The authors would like to thank software vendors who participated in the evaluation processes especially Simul8 Corporation, Imagine That Inc., Enterprise Dynamics Corp., ProModel Corporation, and Rockwell Software. The authors would also like to thank Mr Gilles Bardonnet of Accenture, Paris, for his contribution and feedback provided for the project described in this manuscript. Dr Tewoldeberhan has conducted research described in this paper when he was at Delft University of Technology.

References

Balci O (2001). A methodology for certification of modeling and simulation applications. *ACM T Model and Comput Simul (TOMACS)* **11**(4): 352–377.

Banks J (1991). Selecting simulation software. In: Nelson BL and Kelton WD (eds). *Proceeding of the 1991 Winter Simulation Conference*. IEEE: Piscataway: NJ, pp 15–20.

Banks J and Carson JS (2004). *Discrete-Event System Simulation*, 4th edn Prentice Hall: United States.

Banks J and Gibson RR (1997). Selecting simulation software. *IIE Solutions* May, 30–32.

Bosilj-Vuksic V, Ceric V and Hlupic V (2007). Criteria for the evaluation of business process simulation tools. *Interdisciplinary J Inform Knowl Manage* **2**: 73–88.

Bots PWG and Hulshof JAM (2000). Designing multi-criteria decision analysis processes for priority setting in health policy. *J Multi-Criteria Decis Anal* **9**(1–3): 56–75.

Bovone M, de Ferarri V and Manuelli R (1989). How to choose a useful simulation software. In: Lazeolla G, Lehman A and van den Herik HJ (eds). *Proceedings of the 1989 European Simulation Multiconference*. SCS: San Diego, pp 39–43.

Burns D (2007). *Systemic Action Research: A Strategy for Whole System Change*. Policy Press: Bristol.

Chikofsky EJ, Martin DE and Chang H (1992). Assessing the state of tools—assessment. *IEEE Software* May, 18–21.

Christy DP and Watson HJ (1983). The application of simulation: A survey of industry practice. *Interfaces* **13**(5): 47–52.

Davis L and Williams G (1994). Evaluating and selecting simulation software using the analytic hierarchy process. *Integr Manuf Syst* **5**(1): 23–32.

Deaver RA (1987). Selecting a manufacturing simulation system. *CIM Rev* **3**(3): 6–8.

Ekere NN and Hannam RG (1989). An evaluation of approaches to modelling and simulating manufacturing systems. *Int J Prod Res* **27**(4): 599–611.

Feinstein AH and Cannon HH (2002). Constructs of simulation evaluation. *Simulat Gaming* **33**(4): 425–440.

Hlupic V (1997). Selecting simulation software using SimSelect. *Simulation* **69**(4): 231–239.

Hlupic V and de Vreede GJ (2005). Process modelling using discrete-event simulation: Current opportunities and future challenges. *Int J Simul Process Model* **1**(1/2): 72–78.

Hlupic V and Paul RJ (1996). A methodological approach to manufacturing simulation software selection. *Comput Integr Manuf Syst* **9**(1): 49–55.

Hlupic V, Irani Z and Paul RJ (1999). Evaluation framework for simulation software. *Int J Adv Manuf Tech* **15**: 366–382.

Holder K (1990). Selecting simulation software. *OR Insight* **3**(4): 19–24.

Kelton DW (2006). *Simulation with Arena*. McGraw Hill Higher Education: Boston, USA.

Law AM and Kelton DW (2000). *Simulation Modeling and Analysis*, 3rd edn McGraw-Hill: Boston, USA.

Lootsma FA (1999). *Multi-criteria Decision Analysis via Ratio and Difference Judgment*. Kluwer Academic Publishers: Amsterdam, The Netherlands.

Lyons AC, Nemat M and Rowe WB (2000). A comparative study of alternative approaches to modelling the operations of a small enterprise. *Work Study* **49**(3): 107–114.

Mackulak GT, Cochran JK and Savory PA (1994). Ascertaining important features for industrial simulation environments. *Simulation* **63**(4): 211–221.

Nikoukaran J, Hlupic V and Paul RJ (1999). A hierarchical framework for evaluating simulation software. *Simulat Pract Theory* **7**: 219–231.

Pidd M (1989). Choosing discrete simulation software. *OR Insight* **2**(3): 22–23.

Pidd M (2004). *Computer Simulation In Management Science*, 5th edn John Wiley & Sons: Chichester, UK.

Robinson S (2003). *Simulation: The Practice of Model Development and Use*. John Wiley and Sons Ltd: Chichester, UK.

Sahay BS and Gupta AK (2003). Development of software selection criteria for supply chain solutions. *Ind Manage Data Syst* **103**(2): 97–110.

Shannon RE (1992). Introduction to simulation. In: Swain JJ, Goldsman D, Crain RC and Wilson JR (eds). *Proceedings of the 1992 Winter Simulation Conference*. SCS: San Diego, pp 65–73.

Shanklin T, Roper K, Yegneswaran PK and Marten MR (2001). Selection of bioprocess simulation software for industrial applications. *Biotec Bioeng* **72**(4): 483–489.

Swain JJ (1999). Imagine new worlds. *OR/MS Today*, February, pp 38–41.

Tobias R and Hofmann C (2004). Evaluation of free Java-libraries for social-scientific agent based simulation. *J Artif Soc Soc Simulat* **7**(1), http://jasss.soc.surrey.ac.uk/7/1/6.html.

Vogel DR and Vreede GJ (1999). The organizational application of GSS: A comparison across time and national boundaries. In: Vreede GJ, de Ackermann F (eds), *Proceedings of the 10th EuroGDSS Workshop*, Copenhagen.

Received January 2008;
accepted August 2009 after one revision