Proc. Modeling and Simulation Conference, ESM2001, E.J.H. Kerckhoffs and M. Snorek, edts, Prague, Czech Republic, June 2001, SCS Press, Delft, Netherlands, 2001, pp. 68-75, Nominated for the Best Paper Award

Component-Based Simulation

Gilbert Chen and Boleslaw K. Szymanski
Department of Computer Science
Rensselaer Polytechnic Institute, Troy, NY 12180, USA

Abstract

Moore's law states that transistor density doubles approximately every eighteen months. Abetted by an increase in clock speed, it contributes to an improvement at an even faster rate in microprocessor overall performance. Based on this hardware improvements, parallel simulations nowadays are theoretically capable of executing far larger and more complex models than ever before.

However, there is a gap between the size of the simulation model that the state-of-the art computers can execute and the size and complexity of the simulation model that the programmers can reliably design and build. Simulations are hard to program because programmers need to take the time semantics into consideration. Simulation programs are hard to debug because a procedure might be executed millions of times, each of which is associated with different arguments and a distinct point in simulation time. More fundamentally, the two widely used simulation world views, event scheduling and process interaction, as discussed in this paper, don't scale well to large simulations. The modeling methodology has become an obstacle that limits the use of simulation technology outside research community.

We believe that component-based simulation is a natural and intuitive approach to the development of large-scale simulations. In this paper, we first briefly review classical simulation world views and the logical process paradigm. After discussing their limitations, we present a component-oriented view that facilitates the use of the component-based approach in simulation development. To support this component-oriented view, we classify simulation components into timeless components, time-dependent components, and time-independent components. This classification promotes a hierarchical simulation modeling methodology that addresses both composability and efficiency.

1 Introduction

Simulation technology has evolved to a stage in which the modeling methodology becomes more and more important for designing large and complex simulations. Traditionally, designers and implementors of simulations were more concerned with the performance issue. For instance, Parallel Discrete Event Simulation (PDES), which is believed by many researchers to be the key to the ultimate success of the simulation, has been an active research topic for more than fifteen years. Various PDES algorithms have been devised and have proven to be effective. However, engineers still find PDES techniques unconvincing. The reason is that, as David Nicol pointed out [15], performance is a constraint, not an objective function. Factors such as model reusability and maintenance, complexity of model development, easiness of analysis of results, and interoperability are often more important than performance. Richard Fujimoto also observed that PDES techniques were too difficult to apply and too inscrutable to the non-parallel programmer [10, 17].

The shift towards methodology becomes evident as more and more researchers are now interested in the integration approach to building simulations by interconnecting existing smaller ones. This is the driving force behind efforts such as HLA [1, 8] and multi-paradigm simulations [5, 6]. Yet, the HLA has certain limitations that prevent it from being the generally acceptable component-based approach for simulation [16]. One of the most important among those limitations is lack of addressing the performance issue because the HLA has been designed for interoperability rather than the execution speed.

A well-defined component model is at the heart of a component-based approach for simulation. Various conceptual models has long been used to help researchers build simulators. The logical process paradigm [9], used in almost all parallel simulations, is one of such conceptual models. These conceptual models represent primitive forms of component model. The difference is apparent: while a conceptual model only serves as a guideline to facilitate the implementation, a component model contains a formal and strict specification describing the input/output behaviors of the components.

In this paper, we will propose a component model, called Component-ORiented Simulation Architecture (CORSA). Before we discuss the component model, we first turn to the classical world views of simulation, since a component-based approach views simulations as compositions of components and therefore, as we shall see, deviates from all classical world views.

2 Simulation World Views

2.1 Classical World Views

A world view is used by the simulation modeler to describe the dynamic structure, which defines how the system state changes over time. Basically, there are three classical world views: *event scheduling*, *activity scanning*, and *process interaction* [19, 7].

Event Scheduling

In the event scheduling world view, a modeler defines events that can occur during the course of the simulation. An event may change the state of the system. In addition, an event can schedule other events to occur at later times, or cancel other events that have been scheduled before. All these actions are performed by a predefined procedure associated with that event, which is often referred to as event handler.

Activity Scanning

In the activity scanning world view, a modeler also defines events with their causal relationships and their impact on the system state. There is an additional type of events, called *contingent events*, which occur when some condition is met. They cannot be directly scheduled since they depend on the current system state. The events that are bound to happen are therefore called *determined events*. Contingent events often simulate the beginning of some activity and schedule another determined event that simulates the end of that activity.

Process Interaction

In the process interaction world view, a modeler defines a set of *processes* that are chronologically sequenced set of events and activities. Every process is associated with a procedure, which usually includes instructions for suspending execution of the procedure for an interval of time, and instructions for checking whether a condition became true. As a result, an essential distinction between process interaction and event scheduling is that execution of an event handler is instantaneous in terms of simulation time, while the execution of a process may span an interval of simulation time, or even a whole simulation run.

2.2 World Views and Composability

None of the above world views supports the notion of constructing the simulation by composition of well-defined component. They all tend to model the system as a whole, irrespective of the natural boundaries between different parts. For instance, when using the event scheduling world view to simulate an M/M/1 system, the modeler must define an event that happens when a job quits from the server. The procedure that handles this event should retrieve the first job in the queue (if there is any), put it in the server, and schedule another event of the same type that will happen after an interval of time equal to the service time needed by the new job. This modeling methodology gives no consideration to the fact that the queue and the server are two distinctive parts, and that changing the state of both in a single event procedure violates the criterion of encapsulation.

Bruce Cota and Robert Sargent have proposed a modification of the process interaction world view that supports modularity [7]. They argued that the shortcoming of the original process interaction view is that it allows one process to cancel another process's next event in order to deal with preemption of an activity. Structuring a simulation model in this way interferes with modularity because the knowledge of the behavior of a process has to be spread throughout other processes. They proposed using a special control structure which suspends the process for a specified amount of time if none of the predefined conditions is met during the suspension. If there is one, the statement after the clause that the true condition belongs to will be executed immediately.

However, this approach doesn't achieve pure encapsulation, because the conditions may depend on some shared variables or event variables possessed by other processes. Cota and Sargent realized this problem and suggested that when processes can only interact with each other by exchanging messages, the condition will be much simpler. It may only depend on the content of the received message. This variation of the modified process interaction view that is based on message passing was implemented in Maisie [2], a parallel simulation language, and its descendant, Parsec [14].

The relationship between classical world views and component-based approach still remains unclear. The scenario in which existing simulations are integrated to form a large simulation doesn't fit into any classical world views. These world views are often ignored when the component-based approach is being discussed. To help further discussion on their relationship, it is worthwhile to first examine the *logical process* paradigm, which bears some resemblance to component-based approach.

2.3 Logical Process Paradigm

The concept of logical process was first introduced by Fujimoto in his award-winning overview paper [9]. Using the logical process, the simulation is viewed as being composed of a set of *logical processes* that interact with each other by exchanging timestamped messages. Each logical process contains a portion of the

state, as well as a local clock that denotes how far the process has progressed. Shared variables are forbidden in the logical process paradigm, but as Fujimoto pointed out, the exclusion of shared variables may or may not be burdensome, depending on the applications and the necessity of parallelization.

Fujimoto didn't specify the world view used to describe logical processes. The reason becomes clear if we turn back to the definition of world views. The simulation world views is used to describe the *dynamic structure* of the simulation model, which defines how the system state changes over time. On the contrary is the *static structure*, which defines the possible states of the physical system [7, 13]. According to its definition, the logical process paradigm partitions the set of the system state variables into a collection of disjoint subsets. Therefore, this paradigm deals only with the static structure.

The orthogonality between world views and logical process paradigm leads to two kinds of logical process implementations, based on either event scheduling or process interaction. Parallel simulation frameworks using event scheduling include Georgia Time Warp [11], ROSS [3], while Maisie [2] and TeD [18] are based on process interaction. There is no logical process implementation that uses the activity scanning world view. Due to the absence of shared variables, the conditions used by the activity scanning and process interaction world view now depend only on the content of received messages. As a result, the activity scanning regresses to event scheduling, and the only significant difference between the process interaction and the event scheduling is that the process interaction allows the use of *wait* statement, which suspends the logical process for a certain amount of time.

The logical process paradigm became so popular that many papers on PDES used it to help the discussion and almost every parallel simulator was built in accordance with this paradigm. Its success may be attributed to the need to partition the simulation program in order to carry out parallel execution, but we are more willing to believe that the real reason is its similarity to the component-based approach; the complexity of developing parallel simulation model is drastically reduced with logical process paradigm.

However, the sender/receiver relationship in the logical process paradigm is not well-defined. This incompleteness leads to various implementations. For instance, in some simulators, such as Maisie and Parsec, a logical process is required to explicitly specify the destination of each sent message. The shortcoming of this approach is that the interconnection topology of the logical processes is embedded into the code of the logical process, which prevents the process to be reused in other simulations. In others simulators like Georgia Time Warp and ROSS, logical processes are not allowed to directly communicate with each other. Exchange of messages is achieved by scheduling events for other processes. Such approach has an additional disadvantage of introducing an inefficiency because extra communication overhead is incurred.

We identify the sender/receiver relationship as a key issue in the component-based approach for simulation. It is required that the development of a component should be completely isolated from the context in which the component will be used. This independence requirement implies that, prior to the simulation, the component knows neither the source of the messages that it will receive nor the destination of messages that it will send. In the next section we will present a component-oriented world view that adheres to this requirement.

3 Component-Oriented World View

In the component-oriented world view, a simulation is viewed as a collections of components that interact with each other by exchanging messages.

Each component contains a set of parameters, which must be initialized during the parameterization phase. After the parameterization, the components of the same type may exhibit different behavior if their parameters are assigned different values. The simulation is actually made up of component instances.

The most distinct feature of component-oriented world view is that it utilizes *ports* to send and received messages. Ports act as a communication interface. There are two types of ports, *inports* and *outports*. Components use outports to send messages and use inports to receive messages.

Ports are typed. Each inport or outport can pass on only a certain type of messages, which is the same as the type of the inport or outport. We say an inport and an outport are matched if they are of the same type. An inport and an outport can be connected together if they are matched. An inport can be connected with multiple outports, and similarly multiple inports can be connected with one outport.

Event scheduling world view is used to describe the behavior of components. Every component contains an event handler, which will be invoked when the component is activated. The component is activated upon arrival of two types of events. Incoming messages at an inport are called external events, because such messages are always caused by an event in another component. Besides, a component may become activated upon an internal event scheduled by the component itself. When such an event occurs, the simulation clock is adjusted to the timestamp of that event.

The sender/receiver relationship in the component-oriented world view is different from that in the logical process paradigm. As discussed earlier, when designing a logical process, the modeler must explicitly specify the destination of an outgoing messages, and there is no way to know which other processes may send messages to this process. However, when ports are used as a communication interface in the component-oriented world view, defining the destination of a message placed on an outport can be postponed until the configuration phase. It also becomes possible to determine the set of components that will be the senders to a particular component by connecting their outports to the inport of that component. This idea is similar to *delayed binding* proposed by Zeigler in his DEVS framework [21].

Achieving a degree of sender/receiver independence might be viable in logical process paradigm by specifying the destination of an outgoing message as a parameter, as in Maisie/Parsec. This approach runs into difficulties, however, when a message must be sent to multiple processes. A solution is to use a sizable parameter array instead of a single parameter. Specifying a sizable parameter array as the destination of a message means that the message is sent to every process that belongs to the set denoted by the parameter array. Indeed, this solution is equivalent to the delayed binding, because connecting an inport with an outport can be viewed as adding the component to an imaginary destination parameter array associated with the inport. But still, it is impossible to determine the set of sender processes.

Interestingly, the sender/receiver relationship is more dynamic and more implicit in the HLA. A simulation may publish its objects to the RTI, and when another simulation subscribes to one of these objects, the two simulations implicitly become a sender-receiver pair. However, some communication overhead is incurred when a message is dynamically routed from one simulation to another. The sender-receiver relationship must be established before the execution, yet the HLA does not require it to be coded into the simulation program. Hence, these two requirements together necessitates a configuration phase, which many existing simulators lack.

To summarize, in the component-oriented world view a simulation is built by composing a collection of components. Each component contains parameters, inports and outports. Components place outgoing messages on outports and receive incoming messages from inport. Before the execution, each component must undergo a configuration phase, in which two operations are performed. First, all parameters are initialized. Second, the corresponding pairs of inport and outport are connected.

The idea of configuration phase is not new. Equational Programming Language (EPL) [20], for instance, uses *configurations* which define interconnections between ports of different processes to allow the programmer to reuse the same EPL programs in different computations.

4 CORSA Component Model

The component-oriented world view advocates an approach to building simulations which is based on integrating components. The next step is to formalize this approach, i.e., to develop an interface description language that can be used to describe the input/output behavior of the components. However, we found that there are differences among the components. For example, some components that can be used to form a simulation are existing simulations, while others are ordinary programs. Still others could be simulation models obtained from the model repositories which are neither a simulation nor an ordinary program. So the fundamental question is, how we can distinguish them, and how to reflect such differences in the interface description language?

Here, we propose CORSA (Component-ORiented Simulation Architecture) component model to characterize different components. It prescribes the functionality and the interface that each type of component should implement in order to enable integration with other components under the CORSA framework.

The core issue in using CORSA model to build large simulations is to support an efficient synchronization of the linked components. Traditionally there are two mechanisms to synchronize simulation components. The first one is *conservative* algorithm, which strictly avoids causality error by processing events that are certain not to cause any causality error. The second one is *optimistic* algorithm, which processes all ready events, optimistically assuming that there are no causality error, but it is able to recover from such errors if they happened. We observe that usually a simulation is either entirely conservative or entirely optimistic. Although a conservative component could be used in an optimistic simulation, the interaction between them is very limited and in most cases not mutual. For example, a conservative component can only process messages sent by the optimistic component with a timestamp equal to the Global Virtual Time (GVT) of the optimistic component.

From this point of view, every simulation belongs to one of the two worlds: *conventional world* where causality errors are strictly avoided and *virtual world* where they are detected and recovered by rollback mechanisms. In different worlds, components exhibit

different properties that dictate the way in which they can be linked together.

4.1 Conventional World

In the conventional world, only messages that cannot create causality errors are allowed to be generated. A widely used and simple approach to achieving this it to delay sending a message with a timestamp larger than current simulation time until the simulation time reaches the value of its timestamp.

4.1.1 Component Types

A simulation program or a simulation model differs from an ordinary program in the way they process the simulation time. Components are classified according to the way they handle the time semantics. There are three types of components: timeless, time-dependent and time-independent.

Type I: Timeless Component

A timeless component doesn't have the notion of simulation time. It is passive in the sense that it never generates messages without first receiving a message. When processing received messages sent from other components it may produce new messages that have the same timestamp as that of the incoming message. But the component itself is not aware of the existence of the time semantics. Neither does it know whether it is running as part of a simulation program or part of an ordinary program. For this reason, the timeless component is said to be time-unaware.

Type II: Time-Dependent Component

Time-dependent components are time-aware. They cannot advance the simulation time themselves but they can request to advance the time through a special entity called *timer*. Timers provide a mechanism for the component to schedule and receive events. To schedule an event, a timer is set with a specified value representing a future simulation time at which the event will occur. As soon as the simulation time reaches the preset value of that timer, it will be activated and we say that the component received the event it scheduled before.

Type III: Time-Independent Component

Time-independent components always maintain their simulation clock themselves. They don't have any timers. Instead, they contain a clock, which indicates the simulation time at any point throughout the simulation. These components can receive messages only if it is guaranteed that no causality errors will occur. In many cases, they can only receive messages with a timestamp larger that the value of the clock. However, it is also possible for a component of such type to process messages in out-of-timestamp order, provided that a lately arriving message doesn't invalidate any messages that have been processed. More strictly, if m_1 and m_2 are two messages with timestamps $t_1 < t_2$, and it is known that processing them in reverse order or in timestamp order would produce exactly the same state, the component can process m_2 first without violating the causality constraint.

All stand-alone simulations can be viewed as time-independent components without outports. Some simulations may produce data available to other component or require additional information (often dynamic) during the execution.

4.1.2 Properties of Components

Property 1 *Type I and II components can be coupled together.*

Type I and II components are time-implicit since messages they exchange don't have an explicit timestamp. We say these messages are implicitly timestamped. These components have no way to modify the value of timestamps. But they do know the value because the timestamp of any message is always equal to current simulation time which can be easily obtained by components. They can immediately start processing whenever they received a message.

Property 2 *A Type III component can be linked with Type I components unconditionally, or with Type III components and Type II components under the condition that causality errors can be guaranteed not to arise.*

A Type III component can possibly be linked with another Type III or Type II component. Each messages from these components contains timestamp that indicates the specific point in the simulation time at which the events carried by such a message are bound to take place. As stated earlier, only if it is guaranteed that causality error can be strictly avoided can such a message be processed.

A Type III component can be arbitrarily linked with a Type I component without violating any causality conditions. Unlike Type II components, a Type I component never outputs a messages without receiving a message first. When a message sent from a Type III component arrives, the Type I component simply ignores the timestamp of that message. In the middle of processing this message, the component of Type I may produce new messages, but their timestamps will

be equal to the timestamp of the received message. The Type III component that sent the original message with this timestamp will have no problem with receiving these messages because their timestamps will be exactly the same as the current simulation time.

4.1.3 Simulation Engine

Components of Type I and Type II can be coupled together by a simulation engine (Figure 1), which is responsible for parameterization and interconnection of components. During the interconnection, it sets up channels that directly connect matched ports. This mechanism enables components to communicate directly.

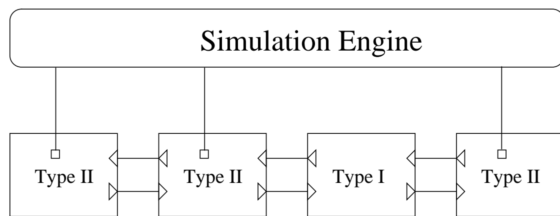

Figure 1: Simulation Engine for Type I and Type II Components in Conventional World

The simulation engine must keep track of all activities of timers. When a timer is scheduled to be active in a future simulation time, the simulation engine inserts a corresponding event into a priority queue. It repeatedly removes the event with the smallest timestamp from the priority queue and then activates the corresponding timer by invoking the event handler of the component to which the timer belongs.

Notice that a Type I component doesn't have any direct interaction with the simulation engine.

A simulation engine can be of Type II or Type III. A Type II simulation engine contains a timer and an event handler. The timer is always set with the smallest timestamp of the event in the priority queue. The event handler is called when the time is activated. It then activates the smallest timer (which belongs to the component, not the simulation engine) and invokes the event handler of the corresponding component. Upon completion of the event handler, the simulation engine will set the timer again with the new smallest timestamp in the priority queue. Finally, the control is passed back to the higher level simulation engine (if it exists). Constructing the simulation engine as a Type II component allows hierarchical simulation modeling.

4.2 Virtual World

In the virtual world, causality errors are allowed to happen. Arrival of a message that generates a causality error will force the component to rollback by undoing all processed messages that are affected by the arriving message. Hence, in the virtual world there is a distinction between regular messages that forward the simulation and therefore are called *positive messages*, and the messages produced in the process of a rollback and referred to as *anti-messages*.

4.2.1 Component Types

In the virtual world, the classification of three types still holds. However, components exhibit slightly different properties.

Type I: Timeless Component

A timeless component accepts anti-messages, as well as positive messages. Upon arrival of an anti-message, it will undo the effect caused by the corresponding positive message. It needs to generate new anti-messages, if the corresponding positive message had produced new positive messages to other component. A timeless component never initiates new anti-messages without arrival of an anti-message.

keyword

Type II: Time-Dependent Component

Besides anti-messages sent from component, a time-dependent component also receives anti-messages from the simulation engine. These anti-messages occur when timers have been prematurely activated. The component must then respond to these anti-messages by counteracting the impact of positive messages and producing new anti-messages to other component if necessary.

All anti-messages are implicitly stamped with the timestamp of the corresponding positive messages.

Type III: Time-Independent Component

Time-independent components communicate by exchanging timestamped positive messages and anti-messages with one another. If a positive message arrives with a timestamp smaller than current simulation time, the component will roll back to the time specified by the timestamp of the received message and then process the message. If the timestamp of the arriving message is equal to or larger than current simulation time, it will be inserted into the priority queue.

A time-independent component in the virtual world also needs the information about Global Virtual Time (GVT). GVT is the lower bound on the timestamp of all messages that will be received in the future. Time-independent component obtains the GVT value from

the simulation engine. To be able to calculate the GVT, the simulation engine must know the local simulation time of every components and the timestamp of every messages sent by the components.

4.2.2 Component Properties

Property 3 *Type I and II components can be coupled together.*

This is the same as in the conventional world.

Property 4 *A Type III component can only be linked with other Type III components.*

Time III components can be coupled together because their ability to receive stragglers (messages that cause causality errors). On the contrary, Type I and Type II components only receive positive messages and anti-messages. They are time-implicit, which means that they lack the ability to detect whether a message is a normal one or a straggler. They must rely on the simulation engine which translates a straggler into a set of anti-messages. Therefore, a Time III component cannot be directly linked with Type I or Type II components.

4.2.3 Simulation Engine

A simulation engine for Type I and Type II component in the virtual world acts the same way as its counterpart in the conventional world, if there are no stragglers. Otherwise, a straggler will be translated into a set of anti-messages that will be sent to the components contained within the simulation engine.

Fossils (messages with a timestamp older than GVT) can be collected in two places. It can be done by the simulation engine, which will periodically check all the saved messages and released those older than GVT. It can also be done by ports and timers themselves. In order to do so, ports and timers must stamp every received message with current simulation time and then save them in an internal memory. To collect fossils, they need to compare the GVT value with the timestamp of the message. In either case, there is dedicated communication channel between the simulation engine and every port or timer to enable the port or timer to obtain the GVT value or to request and release memory for saved messages, as shown in Figure 2. Messages are still exchanged directly between components without going through the simulation engine.

Similarly as in the conventional world, a simulation engine for Type I and II components can be of Type II or Type III. A Type II simulation engine enables

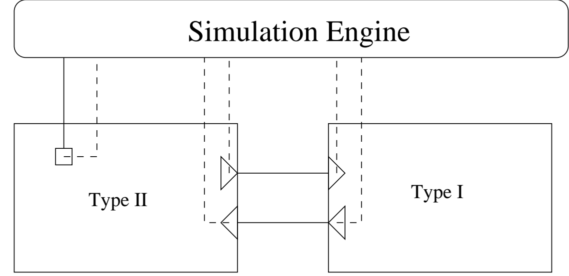

Figure 2: Simulation Engine for Type I and Type II Components in virtual world

further integration. A simulation engine for Type III components acts as a message passing provider. Its main task is to calculate the GVT value based on the information provided by each component.

4.3 Between Conventional World and Virtual World

It is worthwhile to discuss the relations between the conventional world and the virtual world. For instance, a virtual component can be used in the conventional world without modification. On the other hand, when augmented with the functionality of state-saving, a conventional component becomes a virtual component.

The fact that a conventional component can be converted into a virtual component by adding state-saving implies that the introduction of Virtual Time [12] is solely for the sake of efficiency. If there was infinite memory, the fossil collection based on the value of the GVT would be unnecessary. Likewise, various state-saving techniques, such as incremental state-saving, infrequent state-saving, and reverse computation [4], are merely different kinds of optimizations whose purpose is to reduce the amount of memory required by the simulation. More interestingly, for perfectly-reversible components [4] that require no extra memory to perform the reverse computation, it is possible that a conventional component and a virtual component derive from the same implementation. For example, a conventional component that acts as a linked list can possibly be upgraded into a virtual component by only modifying the interface: the reverse of action that appends an item to the end of the linked list is the action which removes the last item of the linked list and vice versa.

5 Conclusion

In this paper we pointed out the orthogonality between the component-based approach and the classical simulation world view. From this observation, we derived the component-oriented world view. To support this world view, components are classified into three

types, each of which plays a different role in a simulation.

It becomes clear that an ordinary program is a timeless component, a simulation model is a time-dependent component, and an existing simulation is a time-independent component. Both timeless component and time-dependent components can be part of a simulation.

Current research on component-based approach simulation often tends to design an omniscient simulation engine for all kinds of components. For example, HLA offers five different services for advancing time in order to support a variety of simulations. Such a design idea usually results in a cumbersome implementation containing too many functionalities, most of which are unnecessary for a single task.

The classification developed by us captures differences between components. The integrator can select the simulation engine of the most suitable type for the components that need to be linked. If the components are special and no existing simulation engine fits them, the integrator can develop a new simulation engine. This new simulation engine becomes reusable immediately because it can be used to link components that belong to the same subtype.

The classification also unifies the program-level development and the application-level integration. It has been realized that a good component-based approach should provide an easy and fast mechanism to develop customized components, as well as the mechanism to integrate existing components. Using the CORSA component model, the way to integrate existing simulations to form a larger simulation is the same as the way to build a single simulation by linking timeless and time-dependent components. The advantage of unified modeling methodology is that it addresses the problem of scalability. People are never able to predicate the scale of the simulation they need to handle. Simulations built today by integrating smaller ones will probably become a part of an even larger simulation in the future. Unified modeling methodology allows us to construct the simulation hierarchically, without considering the size of the simulation.

References

[1] High level architecture. http://hla.dmso.mil. Defense Modeling and Simulation Office.

[2] Rajive L. Bagrodia and Wen-Toh Liao. Maisie: A language for the design of efficient discrete-event simulations. *IEEE Transactions on software Engineering*, April 1994.

[3] Christopher D. Carothers, David Bauer, and Shawn Pearce. ROSS: A high-performance, low memory, modular Time Warp system. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, 2000.

[4] Christopher D. Carothers, Kaylan S. Perumall, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, 1999.

[5] Gilbert Cheng, Boleslaw K. Szymanski and Thomas Caraco. "Multiparadigm Simulations in Modeling Spread of Lyme Disease," *Proc. ESM2000: 14th European Simulation Multiconference*, Ghent, Belgium, May 23-26, 2000, Rik Van Landeghem (edt), SCS Press, Delft, Netherland, 2000, pp. 631-638.

[6] Gilbert Cheng and Boleslaw K. Szymanski, "Linking spatially explicit parallel continuous and discrete models," *Proc. Winter Simulation Conference*, Orlando, Florida, December 2000 IEEE Computer Press, Los Alamitos, CA, pp. 1705-1712.

[7] Bruce A. Cota and Robert G. Sargent. A modification of the process interaction world view. *ACM Transactions on Modeling and Computer Simulation*, April 1992.

[8] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. The DoD high level architecture: An update. In *Proceedings of the 1998 Winter Simulation Conference*, 1998.

[9] Richard M. Fujimoto. Parallel discrete event simulation. *Communication of the ACM*, October 1990.

[10] Richard M. Fujimoto. Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, 1993.

[11] Richard M. Fujimoto et al. Georgia Tech Time Warp programmer's manual for distributed network of workstations. Technical report, Georgia Institute of Technology, 1997.

[12] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, July 1985.

[13] P. J. Kiviat. *Computer Simulation Experiments with Models of Economic Systems*, chapter Simulation Languages. Wiley, 1971.

[14] UCLA Parallel Computing Laboratory. Parsec user manual. http://pcl.cs.ncla.edu.

[15] David M. Nicol. Parallel discrete event simulation: So who cares? In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 1997.

[16] Ernest H. Page. The rise of web-based simulation: Implications for the high level architecture. In *1998 Winter Simulation Conference Proceedings*, 1998.

[17] Ernest H. Page. Beyond speedup: PADS, the HLA and web-based simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, 1999.

[18] Kalyan S. Perumalla, Richard M. Fujimoto, and Andrew T. Ogielski. MetaTeD - a meta language for modeling telecommunication networks. Technical report, Georgia Institute of Technology, 1996.

[19] Michael Pidd and Ricardo A. Cassel. Taking cues from Java. *IEEE Potential*, February/March 2000.

[20] Boleslaw K. Szymanski. *Parallel Functional Language and Compilers*, chapter EPL–Parallel Programming with Recurrent Equations. ACM Press, 1991.

[21] Bernard P. Zeigler. *Object-oriented Simulation with Hierarchical, Modular Models.* Academic Press, 1990.