

A Component Model for Discrete Event Simulation

Boleslaw K. Szymanski¹ and Gilbert Chen¹

Department of Computer Science,
Rensselaer Polytechnic Institute,
Troy, NY 12180, U.S.A.
{szymansk, cheng3}@cs.rpi.edu

Abstract. The paper describes the ideas behind a technique for hierarchical linking of simulations as a way of building large-scale simulations from components that interact with each other via communication ports. The resulting component simulation view is compared with the presently used simulation world views. The classification of simulation components, which groups components into timeless, time-dependent, and time-independent categories, implies that components must be linked by type-specific simulation engines. The discovery of the properties of lookback, the ability of a component to change its past without affecting other components, enables another classification which categorizes communication ports into regular, virtual, and lookback ports. These two classifications enable a hierarchical simulation modeling methodology that addresses two important issues in large-scale simulation: composability and efficiency.

1 Introduction

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 these 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 must take the simulation time semantics into consideration. Simulation programs are hard to debug because an event handler procedure might be executed millions of times, each of which is invoked with different arguments and at a distinct point in the simulation time. More fundamentally, the two widely used simulation world views, event scheduling and process interaction, as discussed in this paper, do not scale well to large simulations. The modeling methodology has become a serious obstacle that limits the use of simulation technology outside the research community.

Traditionally, designers and implementers of simulations were more concerned with the performance than scalability. For instance, this is the case for Parallel Discrete Event Simulation (PDES), that is believed by many researchers to be the key to the ultimate success of the simulation. It has been an active research area for more than fifteen years. Various PDES algorithms have been devised and have proven to be effective. However, programmers still find PDES techniques unconvincing [13]. The main reason is that, as David Nicol pointed out [11], performance is a constraint, not an objective function. Richard Fujimoto also observed that PDES techniques are too difficult and take a considerable amount of time to program to be widely accepted [6].

Simulation technology has evolved to a stage in which the modeling methodology becomes more and more important for designing large and complex simulations. The shift towards methodology becomes evident as more and more researchers are now turning their interest to the integration approach to building simulations by interconnecting existing smaller ones. This is the driving force behind efforts such as the HLA (High Level Architecture) [3, 9]. Yet, the HLA has certain limitations that prevent it from being the generally acceptable component-based approach for simulation [12]. One of the most important among those limitations is the failure to address the performance issue; it has been designed for interoperability rather than execution speed.

A well-defined component model is at the heart of a component-based approach for simulation. Various conceptual models have been used to help researchers build simulators. The logical process paradigm [5], used in almost all parallel simulations, is one of such conceptual models. Conceptual models represent a primitive form of component models. The difference is apparent: while a conceptual model serves only as a guideline to facilitate the implementation, a component model contains a formal and strict specification describing the input/output behaviors of the components.

We propose a simulation component model. Essential to this model are the classifications of components and ports, which determine whether a set of components can be connected together and how they are connected. We start the discussion by first presenting the idea of component-based simulation.

2 Component-Based Simulation

The component-based approach is a natural and intuitive approach to the development of large-scale simulations. Traditional simulation world views, such as *event scheduling*, *activity scanning*, and *process interaction*, do not emphasize the composability of simulation models [2]. When adopting these world views, the designers tend to model the real system as a whole, which inevitably limits the reusability of the simulation model.

The logical process paradigm [5], arising from the necessity of simulated system partitioning in PDES (Parallel Discrete Event Simulation), achieves a certain degree of composability. However, this paradigm does not entirely sep-

arate the development of logical processes from the simulation into which they are integrated, thus preventing them from being reusable in other simulations.

2.1 Component-Oriented World View

We proposed a component-oriented world view in which a simulation is composed of a number of components. Two major differences distinguish a component from a logical process. First, components communicate with each other through input/output ports. To send out a message, the component simply place the message in the desired output port. Which input port will receive this message depends on the interconnection of the components. Second, a configuration phase must be performed for components before the execution or even compilation of the system. In the configuration phase, component parameters are assigned values and component ports are interconnected. The configuration phase, together with the indirect communication via ports, make the component development independent of the simulation composed of the components.

A component has a clearly defined interface. This definition alone does not distinguish a component from an object. More important is that all interaction of a component with others must go through its interface. This is not the case in most object-oriented languages. In C++, for instance, it is taken for granted that an object can directly call a method of other objects. Such method calls are not reflected in the interface, causing problems for the object integrator who must look at the implementation of an object to derive an object's dependency on others. Even if this is possible, such method calls are not fully reconfigurable, for they are embedded in the implementation code. From this point of view, many existing self-claimed component-based approaches for general software engineering, such as CORBA, DCOM, and JavaBeans, are not real component models indeed. The key to a component is the definition of both input ports and output ports, abbreviated as inports and outports respectively. Inports, which are indeed the same as functions, define what functionalities the component can provide. Outports, on the other hand, define to what functionalities the component must have access. Such a distinction between functionality providers and consumers truly separates the development phase of components from the configuration phase, allowing for more flexibility and more composability.

2.2 Component Classification

Components are classified into three types with respect to the way they handle the time semantics: timeless, time-dependent and time-independent, called Type I, Type II and Type III respectively.

A Type I component does not have the notion of simulation time. It is passive in the sense that it never generates messages without first receiving a message. A component, when processing a message received from other components, may generate new messages that have the same timestamp as the incoming message that triggered it. Yet, the component itself is unaware of the time semantics. Neither does it know whether it is running as a part of a simulation program or

a part of an ordinary program. For this reason, the timeless component is said to be time-unaware.

The interface of timeless component is composed of parameters, inports and outports. Inports receive messages while outports send out messages. An example of timeless components is the FCFS (First-Come-First-Serve) queue without a server, whose interface is given below. The FCFS queue has an inport *in* that accepts arriving items, which are then stored in an internal queue in the order they are received. Whenever a null message arrives at the inport *next* requesting the next item, the first item in the queue will be sent to the outport *out*.

```
component FIFO
{
    param int capacity;
    inport in(ITEM item);
    inport next();
    outport out(ITEM item);
}
```

Type II components are time-aware. They cannot advance the simulation time themselves but they can request a time advance via a special entity called a *timer*. Timers provide a mechanism for components to schedule and receive events. To schedule a future 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 value preset by the timer, the corresponding component will be activated and then process the timer event.

A server is a simple time-dependent component, as shown below. The function of a server is to process a message for a certain amount of time. When there is a message arriving at the inport *in*, the server stores the message in an internal buffer and sets the delay duration by writing the duration value to the timer *wait*. The delay duration is randomly chosen from an exponential distribution with a mean value of *average_delay*. The timer *wait* becomes activated by the simulation engine when the global simulation clock assumes the value of the current simulation time plus the delay. The message is then released from the internal buffer and sent out through the outport *out*. At the same time, an empty message is sent out at the outport *next* to notify other components (possibly an FCFS queue).

```
component Server
{
    param double average_delay;
    inport in(ITEM item);
    outport out(ITEM item);
    outport next();
    timer wait;
}
```

Type III components maintain their own simulation clock themselves. They do not have any timers. Instead, they contain a *clock*, which indicates the simulation time throughout the execution. In conservative simulation, these components can receive a message only if such a message will not cause causality errors. The easiest way to guarantee it is to accept only those received messages that have a timestamp larger than the value of the simulation clock.

All stand-alone simulations can be viewed as time-independent components without ports. Some simulations may produce data available to other components or require additional information (often dynamic) during execution. For example, a Lyme disease simulation [4] needs to access the tick density at the location where an event related to mice activities has occurred. The following shows the interface of a Lyme component. When a mouse is bitten, the component will send a message to the outport *tickbite*, indicating the location where the bite occurs. The number of ticks biting the mouse will be returned in the argument *tick* since it is passed by reference. Similarly, when a mouse drops ticks that it carries, the component will send a message to the outport *tickdrop*, indicating the location of the mouse and the number of dropped ticks.

```
component Lyme
{
    param int width, height;
    outport tickbite(int x, int y, TICK& tick);
    outport tickdrop(int x, int y, TICK tick);
    clock c;
}
```

Simulation engines play an important role in the component-based simulation. components are always coupled together by a simulation engine. The simulation engine is responsible for parameterization and interconnection of components. During the interconnection, it sets up channels that directly connect matched ports. The simulation engine must also keep track of all activities on timers for Type II components in the runtime. 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.

In the real world there is only one kind of components, real-time driven components that have outports that produce real time data and inports that consume real time data. For a real-time component to cooperate with any simulation components, adaptors must be used to translate the real time into simulation time and vice versa. An example of such components is a repository of real time data gathered by sensors.

2.3 Implications of Component Classification

The classification of simulation components into three types clarifies the role of the simulation developers. A simulation program can be divided into two parts:

one part models the behavior of the real system and the other simply makes the model executable. In this sense, the Type III components have nothing to do with modeling of the real system, since they are only concerned with the underlying implementation issues. Hence, these components should be hidden from the model builders who usually have no specialized knowledge on simulation. On the contrary, the Type I and Type II components include both the code that represents the real system and the code that allows components to exchange data with the simulation engine. The latter is the sole responsibility of the simulator builders but should be transparent to the model builders.

The classification fits into parallel computation well. In practice, a simulation model of a real system may contain an excessive amount of parallelism. However, direct mapping into a parallel program is unnecessary and often inefficient, because the optimal granularity of runtime parallelism is determined only by the number of processors available in a simulation run. It is well known that multiple parallel programs running on one processor always require context switching, which incurs significant overhead.

The classification allows us to avoid this problem by mandating that all Type I and Type II components contained within a processor execute sequentially. Only one component can be active at any one time. A Type I or Type II component is immediately suspended after writing to an outport. The inport connected to this outport is then activated and the component where the inport resides starts to process the arriving message in this inport. If there are multiple inports connected to the outport, all connected inports should be activated in an implementation dependent order. The component that initiated the communication is blocked until all components connected by the outport have finished execution. Some of these components may write to one or more outports and transfer execution further to the other components.

A close examination reveals that components impose various requirements on the timestamp of the simulation time. Type I components require *copyable* timestamps, because they must copy the timestamp of a message arriving in an inport to outgoing messages which can only occur at the simulation time of the received message. Type II components require *addable* timestamps, because when they write a delay duration to a timer, they implicitly schedule an event whose timestamp is equal to the current simulation time plus the specified delay. Type III components naturally ask for *comparable* timestamp for the purpose of deciding causality. In practice, the simulation time is usually implemented through floating-point numbers, which possess all three properties discussed above. These properties are the most common requirement of each type, but not necessarily all the required properties. For example, a component modeling a time-variant system may require the property of being *readable*, which is different from any of the properties mentioned above.

3 Lookback

It is expected that the size of integrated simulations by combining components will become so large that multiple processors are necessary to carry out the simulations. The techniques developed for PDES (Parallel Discrete Event Simulation) are therefore necessary to make large-scale simulation feasible. Historically, there are two classes of PDES protocols [5]. In conservative simulations, a component processes an event only when this event cannot cause any causality errors. In most cases, for the simplicity of implementation, an event can be processed when it is guaranteed to contain the smallest timestamp among all future events, some of which may still be in transit from other components. In optimistic simulations, based on the notion of Virtual Time [8], a component is allowed to process events without considering the causality constraint. However, it must save any changes made to the state, for events might arrive with a smaller timestamp, in which case the previously processed event must be rolled back.

3.1 Lookback-Based Protocols

Our recent discovery of the property of lookback enables a third class of PDES protocols [1]. A component with a certain amount of lookback is able to process out-of-timestamp order events falling into its lookback windows. Hence, lookback allows a component to advance its simulation time more aggressively. We have shown that lookback is able to exploit the intra-component parallelism. It is also more commonly observed than lookahead, the ability to predict the future, upon which most conservative protocols depend. For optimistic simulations, lookback can be used to reduce the frequency of rollbacks. Of more theoretical importance is that, as we proved that in [1], lookback enables the conservative simulation to circumvent the speed limit imposed by the critical times of events, which was previously thought impossible by many researchers [7].

Informally, lookback is defined as the ability of a component to execute out-of-timestamp order events, which are often referred to as *stragglers*, without impacting states of other components. We first give the formal definition of lookback and other related terms.

Definition 1 *If a component at simulation time T can execute correctly, without sending out anti-messages, any received event with timestamp between $T - L$ and T , then the time window $[T - L, T]$ is said to be the lookback window of the component at time T . The lower end of the lookback window, $T - L$, is referred to as the virtual lookback time. The procedure used to process events falling into the lookback window is called the lookback procedure. Finally, for any future event E at the given simulation time T , $LB(E, T)$ denotes the function that defines the value of the virtual rollback time after the event E is executed at time T .*

To avoid any causality error, components must always maintain a virtual lookback time that is less than or equal to the timestamp of any future event. Under such circumstances, any received event can be successfully processed by

either the regular event handler or the lookback procedure. Therefore, we define the lookback constraint as follows.

Definition 2 *A component obeys the lookback constraint if and only if after processing each event, the virtual lookback time is smaller or equal to the minimum timestamp value of all events (currently received and those that will arrive later). The lower bound on the timestamp of all the future events at the simulation time T is referred to as the Minimum Future Receive Time , $MFRT(T)$.*

The lookback constraint is basically a relaxation of the local causality constraint given by Fujimoto [5]. Adherence to the lookback constraint enables a new kind of synchronization protocols for the parallel discrete event simulation. We give the description of the most general protocol below.

```

while (the termination condition is not met)
  create E(T) according to some criteria
  success_flag := false
  while ( E(T) is nonempty )
    remove an event e from E(T)
    precompute LB(e,t)
    if ( LB(e,T) <= MFRT(T) )
      process e
      success_flag := true
    end if
  end while
  if ( success_flag = false )  recompute MFRT(T)
end while

```

This general lookback-based protocol first constructs a set of potentially eligible events $E(T)$. It then repeatedly removes an event from $E(T)$ and decides if this event can be processed according to the lookback constraint by precomputing $LB(e,T)$. Even if it cannot be processed, the while loop will continue to check the eligibility of other events in $E(T)$. The MFRT is recomputed only when $E(T)$ becomes empty and no event has been successfully processed.

The efficiency of creating the set $E(T)$ is important for overall efficiency of simulation because this operation is repeated for each iteration. Simply, we can select the event with the smallest timestamp as the only member of $E(T)$. Such a solution will minimize the amount of processing by minimizing the number of invocations of the lookback procedure which is usually more expensive than the regular event procedure. On the other hand, by ordering the future event list by the virtual lookback time of events and selecting the event with the smallest virtual lookback time as the only member of $E(T)$ may work better for simulations whose virtual lookback time is independent of the simulation time. This solution increases the parallelism of the simulation and minimizes the number of times MFRT is recomputed when the earliest event cannot be processed, but it may increase processing time because of the costly lookback procedure.

We have implemented two variants of the general lookback-based protocol, which always select the earliest event as the only member of $E(T)$ and reported their efficiency elsewhere [1]. These two variants differ in the estimations of the MFRT: one is using GVT (Global Virtual Time) and the other is based on EIT (Earliest Input Time). We abbreviate the two variants of the protocol as LB-GVT and LB-EIT, respectively. The LB-GVT protocol keeps a global estimation of the MFRT as equal to GVT. Hence, it does not take into account the interconnections between components. As a result the GVT is a crude estimation of the MFRT. In contrast, the LB-EIT protocol requires that each component maintain its own estimation of the MFRT based on the topology of the interconnections, thus, helping to improve the accuracy of the estimation. Unlike the LB-GVT which is deadlock free, the LB-EIT has been shown to be prone to deadlock, and deadlock avoidance mechanism must be incorporated into the protocol.

3.2 Implications of Lookback

We observe that the three classes of PDES protocols arise from different ways of manipulating the simulation time. At first, the simulation time was treated in the same way as the physical time. An analogy between the simulation time of distributed systems and the physical time was given by Lamport [10]. Later, Jefferson proposed the notion of Virtual Time, in which the simulation time could be reversed. Now, lookback allows us to ignore, to some extent, the timestamp order imposed by the simulation time.

The discovery of lookback forces us to rethink several fundamental concepts in PDES, the first being event processing. Traditionally, any event is processed by the component using the state information at the point in the simulation time when the event is bound to take effect. In a lookback-enabled component, however, not only the snapshot of the state, but also the history of the event activities must be taken into consideration. For example, if an event arrives in the timestamp order, it can be processed without considering other events. If it is a straggler, then the damage caused by already processed events with a larger timestamp has to be repaired, and repairing becomes as an indispensable part of the event handling procedure for the straggler. The component must keep track of all those events that have been prematurely processed; otherwise it would be impossible to repair the damage. Therefore, the term ‘event processing’ has a broader meaning. It includes the operations to recover from the erroneous computation resulting from wrong order of events execution. According to this meaning, an event is said to be processed correctly even if other processed events have to be rolled back and re-executed.

Another important concept that has to be considered carefully is the meaning of causality. A general rule has been widely accepted in simulation systems that an event must be executed later than the preceding event in the same component. This is actually a more strict causality than the physical cause-and-effect relationship. Two independent events that do not affect one another may be executed concurrently, regardless of the times at which they occur. However, with lookback, we are able to process two events in out-of-timestamp order even if

one affects the other. On the other hand, we must adhere to the timestamp order if one causes the other, like in the case where the execution of one event generates the other event. From this point of view, we define a simulation causality determined only by the possible execution order.

Definition 3 *There is a causal antecedence between two events if one generates the other.*

This new definition of simulation causality introduces less strict dependence among events than those imposed by the physical cause-and-effect relationships. It should be noticed that the simulation world is different from the real world; we have more power to manipulate what happens in the simulation world, because everything is represented by variables stored in the computer memory, including the simulation time. Relaxed event dependences allow events to be processed in out-of-timestamp order, and, as a result, there may be more events eligible for execution in each component. Out-of-timestamp order execution based on independent events was thought possible, and has been implemented successfully by the PDES community. However, it is only the lookback-based synchronization protocols that provide a unified yet efficient way to enable out-of-timestamp order execution for events dependent in the traditional view.

The simulation causality defined in this way is flexible. Whether two events can be executed in the reverse order is determined by how the simulation is implemented. For example, in the lookback-based protocols an event A is dependent on an earlier event B if the execution of the event A produces an outgoing message and the content of this message could be affected by the event B. The event A must then be delayed until the event B is processed first. However, in optimistic protocol anti-messages are allowed, so the event A can be process first regardless of the event B. If the execution of B changes the outgoing message produced by A, an anti-message is sufficient to cancel out the message, which means the event A and B become independent to each other.

Lookback-based protocols differ from the other two classes of protocols in the part of the computation that is certain to be correct. The conservative protocols allow an event to be executed only when no other events can affect it, therefore every processed event is a *committed* event. In contrast, every event in an optimistic simulation is uncommitted after execution. It becomes committed only when the GVT passes beyond the timestamp of the event. In the lookback-based protocols, however, it is the local state of the component that is subject to changes caused by late events. Figure 1 illustrates the differences among these three classes of synchronization protocols.

4 Port Classification

The three classes of synchronization protocols actually correspond to three types of communication ports in simulation components:

- regular ports which accept only positive messages with the timestamps larger than the current simulation time (normal messages),

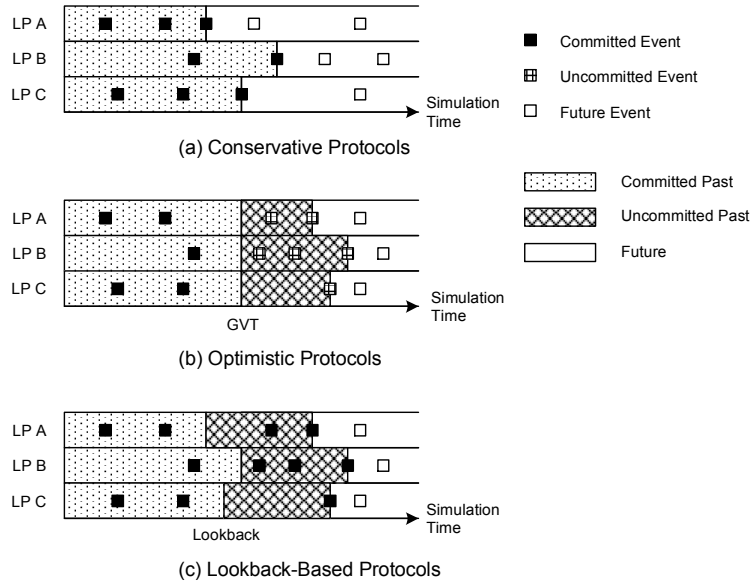


Fig. 1. Comparing Lookback-Based Protocols with Conservative and Optimistic Protocols

- virtual ports which may receive positive messages and anti-messages, and
- lookback ports which may receive positive messages with the timestamps larger than the lookback virtual time, including stragglers.

A virtual port could also have the element of lookback type, as in the case when lookback is exploited in the optimistic simulation to reduce the rollback frequency, but not to completely avoid rollbacks. The first two port types are widely used since the optimistic simulators became available. The third type has been recently discovered, yet lookback ports existed in optimistic simulation engines even before the concept of lookback was introduced. Optimistic simulation engines are able to accept and handle stragglers by translating them into a set of anti-messages. As a result, the components that reside in such a simulation engine never notices the existence of the stragglers. The explicit identification of lookback makes lookback ports possible and directly exposes components to the stragglers.

Different types of ports may coexist in the same component. For example, imagine an FCFS server with a lookback inport. It deals with stragglers without difficulty, because such stragglers can be correctly processed by inserting them into the internal list sorted in the timestamp order. The output of the FCFS server, however, can be of the regular type. The nature of the FCFS server can guarantee that it never outputs events in out-of-timestamp order. A regular port can therefore be connected to this output, simplifying the simulation modeling of other components that receive events from this FCFS server.

There are three rules regulating how ports can be connected together.

Rule 1 *An outpost must be connected to an inport of the same type.*

This is almost self-explanatory, because otherwise the inport would have difficulties in processing the messages sent by the outpost. The opposite is not true, however. It is possible, for instance, to link a virtual inport with a regular outpost. In such a case, the virtual inport will never receive an anti-message.

Rule 2 *Simulation engines must deliver the messages consistent with the type of the ports.*

The simulation engine must determine the consistency of the message in the component level, not within individual port. For example, sending messages in timestamp order to a regular inport does not necessarily imply the correctness of message delivery. It is possible that the timestamp of a message, though greater than the last one sent to the same inport, will be less than the current simulation time of the component where the inport resides, making the message a straggler. Hence, the simulation engine must also take into consideration the current simulation time of the component receiving the messages.

Rule 3 *Components residing in different simulation engines cannot be linked together.*

They cannot be connected for two reasons. First, components in different simulation engines may have different understanding of the time semantics. For example, in a Type II component the timestamp of any message has the property of being addable, while in a Type III component the timestamp is comparable. Such a disparity in time semantics must be resolved by a simulation engine. Second, even if two components are of the same type, they cannot be linked together because there is no way to guarantee a message delivery mechanism that is consistent with the type of the ports. As we already know, the information of the port type alone is not sufficient to determine the consistency of the message.

5 Comparison with HLA

The High Level Architecture (HLA) [3, 9] provides a software architecture for integration of a wide variety of simulations. The main design goal of the HLA was to provide a modeling mechanism for reusing existing simulations so that the cost and time required to create new ones can be dramatically decreased.

The HLA does achieve interoperability at the component level. It allows easy linking of existing simulations, if these simulations were built in accordance with the HLA framework. However, it accomplishes this goal at the cost of efficiency. For instance, the subscribe-and-publish scheme in the HLA does not allow peer simulations to communicate directly with each other. Instead, peer-to-peer communication is achieved by placing messages on the Run Time Infrastructure

(RTI), which acts as a communication bus. These messages are then received by other simulations listening to the RTI. This solution supports composed simulations; new components can be easily added without any modification to the existing system. However, the communication bus becomes the system bottleneck when the number of connected simulations is large.

The main problem with the HLA, from our point of view, is that it treats all components uniformly. As made evident by the classifications of components and ports, a simulation model is characterized by the types of its components and ports, based on the way how the simulation time is handled. It is difficult, if ever possible, to directly connect simulation models that are not compatible with each other in terms of component type and port type. The attempt of the HLA to link together components without considering their types results in a centralized bus that is too cumbersome to implement and too inefficient to execute.

Realizing the differences among components, we propose the component-based approach described above that adopts typed simulation engines to couple together components of compatible types. There are no predefined simulation engines. Instead, the simulation engine is treated as a component too, either of Type II or of Type III, and it can only link components of the type for which it was designed. A new type of components may require a new simulation engine, but once the simulation engine has been designed it could be used to link any components of the same type. More importantly, since a simulation engine is also a component, it could be linked with other components by another simulation engine. This hierarchical structure of linkage may avoid the bottleneck problem caused by the communication bus in the HLA.

6 Conclusion and Future Direction

In this paper, we presented a component model for discrete event simulation. The differences between timeless, time-dependent, and time-independent components were illustrated. With a brief introduction to lookback, three types of communication ports were suggested, corresponding to three classes of synchronization protocols for PDES.

The two classifications form the basis of a hierarchical component-based approach for simulation. We plan to develop a simulation infrastructure, referred to as Component-ORiented Simulation Architecture (CORSA). Two key issues in the design of such a simulation infrastructure is the interoperability and interchangeability of the simulation models. Interoperability enables the simulation integrator to reuse existing simulations, while interchangeability allows the same simulation model to be used in different simulations. They actually represent two levels of reusability, one at the simulation level and the other at the model level. Therefore, the ultimate goal of the CORSA is to achieve the maximum reusability and to preserve efficiency at the same time.

Acknowledgment

This work was partially supported by the NSF Grant KDI-9873139 and IBM Shared University Research Program. The content of this paper does not necessarily reflect the position of policy of the U.S. Government or IBM Corporation - no official endorsement should be inferred or implied.

References

1. Gilbert Chen and Boleslaw K. Szymanski. Lookback: A new way of exploiting parallelism in discrete event simulation. Technical Report 01-11, Department of Computer Science, Rensselaer Polytechnic Institute, October 2001.
2. Bruce A. Cota and Robert G. Sargent. A modification of the process interaction world view. *ACM Transactions on Modeling and Computer Simulation*, pages 109–129, April 1992.
3. 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*, pages 797–804, 1998.
4. Ewa Deelman, Boleslaw K. Szymanski and Thomas Caraco. Simulating Lyme Disease Using Parallel Discrete Event Simulation. In *Proceeding of the 1996 Winter Simulation Conference*, pages 1191-1198, 1996.
5. Richard M. Fujimoto. Parallel discrete event simulation. *Communication of the ACM*, pages 30–53, October 1990.
6. Richard M. Fujimoto. Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, pages 213–230, 1993.
7. David Jefferson and Peter Reiher. Supercritical speedup. In *Proceedings of the 24th Annual Simulation Symposium*, pages 159–168, 1991.
8. David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, pages 404–425, July 1985.
9. Frederick Kuhl et al. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall, 1999.
10. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
11. David M. Nicol. Parallel discrete event simulation: So who cares? In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 1997.
12. Ernest H. Page. The rise of web-based simulation: Implications for the high level architecture. In *1998 Winter Simulation Conference Proceedings*, pages 1663–1668, 1998.
13. Ernest H. Page. Beyond speedup: PADS, the HLA and web-based simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 2–9, 1999.