DYNAMIC COMPONENT SUBSTITUTION IN WEB-BASED SIMULATION*

Dhananjai Madhava Rao and *Philip A. Wilsey*
Experimental Computing Laboratory
Dept. of ECECS, PO Box 210030, Cincinnati, OH 45221–0030.
(email: dmadhava@ececs.uc.edu, phil.wilsey@uc.edu)

ABSTRACT

Recent breakthroughs in communication and software engineering has resulted in significant growth of web-based computing. Web-based techniques have been employed for modeling, simulation, and analysis of systems. The models for simulation are usually developed using component based techniques. In a component based model, a system is represented as a set of interconnected components. A component is a well defined software module that is viewed as a "black box" *i.e.,* only its interface is of concern and not its implementation. However, the behavior of a component, which is necessary for simulation, could be implemented by different modelers including third party manufacturers. Web-based simulation environments enable effective sharing and reuse of components thereby minimizing model development overheads. In component based simulations, one or more components can be substituted during simulation with a functionally equivalent set of components. Such Dynamic Component Substitutions (DCS) provide an effective technique for selectively changing the level of abstraction of a model during simulation. It provides a tradeoff between simulation overheads and model details. It can be used to effectively study large systems and accelerate rare event simulations to desired scenarios of interest. DCS may also be used to achieve fault-tolerance in Web-based simulations. This paper presents the ongoing research to design and implement support for DCS in A Web-based Environment for Systems Engineering (WESE).

1 INTRODUCTION

The marked growth in communication technology and software engineering has resulted in significant growth in the use of the World Wide Web (WWW) [1]. The distributed resources of the WWW have been harnessed together using Web-based computing methodologies [1, 11, 10]. Web-based techniques enable active interaction between interconnected computing systems that can be individually or collectively used to provide a generic set of computational resources. These techniques have transformed the WWW into a giant computational infrastructure [11, 10]. The computational infrastructure of the WWW has been exploited to enable Web-based simulations [7, 11, 10]. Web-based simulation is an effective solution to address a number of issues exacerbating modeling, simulation, and analysis, such as: *(i)* effective sharing and reuse of simulation models developed by different modelers [11, 10]; *(ii)* availability and accessibility of the models [7] without loss of proprietary information [11, 10]; *(iii)* portability and inter-operability of the models [14]; *(iv)* capacity for large scale simulations [6, 10]. Due to its effectiveness, web-based simulations are steadily growing in importance.

In web-based simulation environments, the models for simulations are usually developed using component based modeling techniques [8, 11, 10]. In a component based model, a system is represented as a set of interconnected components [11, 10]. A component is a well defined entity which is viewed as a "black box", *i.e.,* only its interface is of interest and not its implementation. A component could in turn be specified using a set of sub-components. During simulation, each *atomic* component is associated with a specific, well defined software module that implements its behavior and functionality. The software modules could be those implemented by the modeler, available locally, or those obtained via the WWW from other third party model developers [11, 10]. Web-based simulation environments insulate the user from the intricacies involved utilizing third party models and the overheads of distributed simulation. Component based modeling techniques offer a number of advantages [8, 11, 10]. Components are not only useful modeling abstractions but are

*Support for this work was provided in part by the Advanced Research Projects Agency under contracts J–FBI–93–116 and DABT63–96–C–0055.

also convenient units for information exchange over the WWW. Sharing and reuse of components considerably reduces modeling and validation overheads. Component based modeling technique also eases exploration of design alternatives through "plug and play" of components [11, 10]. Hence, it is prevalently used for web-based modeling and simulation [1, 11, 10].

In component based simulation models, one or more components can be substituted by functionally equivalent set of components without altering the basic characteristics of the model. Substituting one or more components with a single component and vice versa is synonymous to varying the level of abstraction of the model. For example, in the case of logic simulations, a structural model of a component could be substituted by its behavioral model and vice versa to change the levels of abstraction. Substitution of components may be done statically, prior to simulation, or dynamically, during the course of simulation. Static component substitution has been employed to address capacity and performance of large scale simulations. Huang *et. al.* present techniques for selectively abstracting different components of network models to improve performance and capacity of network simulations [3]. Rao *et. al.* aggregate components that utilize a common implementation, increasing the capacity of simulators, to enable ultra-large scale simulations [12]. Levelized code compilation techniques, that selectively replace parts of combinatorial logic circuits with equivalent behavioral descriptions, are widely used to improve performance of circuit simulations [15]. The primary drawback of these techniques is that, functionality, observability, and model details cannot be altered during simulation. However, observability and model details are crucial for effectively studying large scale systems.

On the other hand, substituting components during simulation provides a dynamic tradeoff between model details and performance of the simulation. Dynamic Component Substitution (DCS) not only encompasses the utility of its static counterpart but also provides a number of other useful features. DCS enables effective "What-if" analyses and exploration of design alternatives to be carried out during the life time of a simulation. DCS is a novel approach for simulation of "multiple futures" [4]. It is an alternative approach for fast simulations and provides an attractive solution to accelerate rare event simulations [13]. It is an effective technique for debugging and validating large simulations. The technique can also be used to dynamically alter the tradeoffs between resource consumption and model details during simulation. DCS can be used to selectively abstract parts of a model thereby enabling simulation and analysis of large systems in reasonable time frames. The technique can also be used to achieve better fault-tolerance in web-based simulations. However, implementing support for DCS in optimistically synchronized simulations, Time Warp simulations in particular, is a complicated task. This paper presents the issues involved in implementing support for DCS in a Web-based Environment for Systems Engineering (WESE). A brief background on the distributed synchronization mechanism and the simulation kernel used in WESE is presented in Section 2. An overview of WESE is presented in Section 3. The issues involved in implemented DCS in WESE are presented in Section 4. Some of the experiments conducted using the DCS feature of WESE are presented in Section 5. Section 6 presents some concluding remarks along with pointers to future work.

2 BACKGROUND

The distributed simulation capabilities of WESE have been enabled using WARPED, a parallel optimistic simulator. WARPED uses the Time Warp [5] paradigm to achieve distributed synchronization. A Time Warp synchronized simulation is organized as a set of communicating asynchronous logical processes (LPs). The LPs communicate between each other by exchanging discrete *virtual time* stamped events [5]. Virtual Time is used to model the passage of time and define a total order on the events in the system. Each LP processes its events incrementing its local virtual time (LVT), changing its state, and generating new events. The LPs must be synchronized in order to maintain the causality of the simulation; although each LP processes local events in their correct time-stamp order, events are not globally ordered. Causal violations may occur due to the optimistic nature of Time Warp. Causality violations are detected by a LP when it receives an event with time-stamps lower than its LVT (a *straggler* event). On receiving a straggler, a *rollback* mechanism [5] is invoked to recover from the causality error. The rollback process recovers the LP's state prior to the causal violation, canceling the erroneous output events generated by sending out anti-messages, and re-processing the events in their correct causal order [5]. Each LP maintains a list of state transitions along with lists of input and output events corresponding to each state to enable the recovery process. A periodic garbage collection technique based on Global Virtual Time (GVT) [5] is used to prune the queues by discarding history items that are no longer needed. The distributed simulation is deemed to have terminated when all the events in the system have been processed in their correct causal order.

The WARPED kernel presents an interface to build

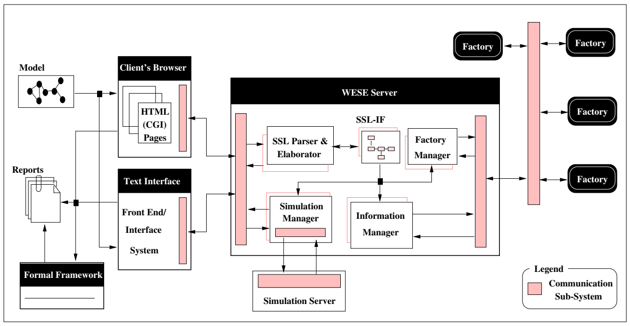

Figure 1: Overview of WESE

logical processes based on Jefferson's original definition [5] of Time Warp [9]. The kernel provides an application program interface (API) to build different LPs with unique definitions of state [9]. The basic functionality for sending and receiving events between LPs using a message passing system is supported by the kernel. In WARPED, LPs are placed into groups called "clusters". LPs on the same cluster communicate with each other without the intervention of the message passing system, which is faster than communication through the message system [9]. Although LPs are grouped together into clusters they are not coerced into synchronizing with each other. Control is exchanged between the application and the simulation kernel through cooperative use of function calls. Further details on the API and working of WARPED is available in the literature [9].

3 WESE

The Web-based Environment for Systems Engineering (WESE) was developed to ease modeling and simulation of systems over the WWW [10]. In WESE the model of a system is represented using a set of interconnected components. A component is treated as a "black box" with a set of inputs and outputs; *i.e.,* only the interface specification of the component is of concern and not its implementation. The actual implementation of a component could be developed by the modeler or by other third party designers. Accordingly, WESE provides a component based modeling language, a framework for developing a web-based repository of components, and the infrastructure for distributed simulation. An overview of WESE is shown in Figure 1. As shown in Figure 1, the environment provides a Hyper Text Markup Language (HTML) interface and a text based frontend that can be used to interact with the WESE Server. The server forms the core of WESE and orchestrates the various parallel and distributed activities of the system. The input to WESE is the model of the system described using the System Specification Language (SSL). The Backus Normal Form for SSL grammar is shown in Figure 2. As shown in Figure 2, the specification of a model or a SSL design file consists of a set of interconnected *modules*. Each module consists of three main sections, namely; *(i)* the *component definition section* that contains the details of the components to be used to specify a module (such as the Universal Resource Locator (URL) of a factory and name of the source object along with initial parameters); *(ii)* the *component instantiation section* that defines the various components constituting the module; and *(iii)* the *netlist section* that defines the interconnectivity between the various instantiated components. SSL permits a *label* to be associated with each module. The *label* may be used as a component definition in subsequent module specifications to nest a module within another. In other words, the *labels*, when used to instantiate a component, result in the complete module associated with the label to be embedded within the instantiating module. This technique can be employed to reuse module descriptions and develop hierarchical specifications. As shown in Figure 1, the input SSL source is parsed into an object oriented (OO) in-memory *intermediate form* (SSL-IF) using the SSL parser. Hierarchical SSL models are elaborated or "flattened" at the end of parsing by the elaborator [11]. Elaboration is a recursive process that flattens a hierarchical model by substituting each module reference (made through the use of *labels*) with an unique instance of the module. As shown in Figure 1, the elaborated model, which is also represented using SSL-IF, forms the primary input to all the other modules of WESE.

The WESE Server also performs the task of collaborating with the distributed factories and coordinating the simulations. As shown in Figure 1, the *simulation manager* performs the activities associated with coordinating with the object factories (via the *factory manager*) to setup a distributed simulation. The *factory manager* performs the tasks of interacting with the distributed factories using a predefined protocol. It not only provides a uniform interface to communicate with different object factories but also insulates the other modules of the server from the intricacies of the underlying protocols. The *information manager* is responsible for interacting with the factories (via the *factory manager*) and constructing the formal specifications used by WESE's formal framework. The current implementation of WESE is geared to generate formal specifications in PVS, a higher order logic specification language. The PVS specification can be used to formally verify different attributes of the system by using a mechanized theorem prover.

To ease design, development, and use of compo-

```
design_file ::=
    include_list ssl_design_module | ssl_design_module
include_list ::= include_clause | include_clause include_list
include_clause ::= include " file_name ";
file_name ::= identifier | identifier . identifier
ssl_design_module ::=
    label ssl_module | ssl_module ssl_design_module |
    ssl_module | label ssl_module ssl_design_module
ssl_module ::=
    { component_definition_section }
    { component_instantiation_section }
    { net_list_section }
label ::=
    identifier (number,number) |
    identifier (number,number label_string )
label_string ::= , identifier | , identifier label_string
component_definition_section ::=
    component_definition |
    component_definition component_definition_section
component_definition ::=
    component_name (number , number):url optional_parameter
optional_parameter ::= parameter ; | ;
component_name ::= identifier (number, number)
parameter ::= " string " | ""
url ::= host_name : port_number . factory
host_name ::= identifier | identifier . host_name
factory ::= identifier | identifier . factory
port_number ::= number
component_instantiation_section ::=
    component_instantiation |
    component_instantiation component_instantiation_section
component_instantiation ::=
    identifier . identifier optional_parameter |    identifier . identifier
number optional_parameter
net_list_section ::= net_list | net_list net_list_section
net_list ::= identifier ( mode , number ) :  instance_list ;
instance_list := instance | instance , instance_list
instance ::= identifier ( mode , number )
mode ::= in | out
identifier ::= start_char any_char
start_char ::= [a - z, A - Z]
any_char ::= [a - z, A - Z, 0 - 9, _ ]
string ::= string_char | string_char string
string_char ::=  []
number ::= [0 - 9]
```

Figure 2: BNF for SSL grammar

nents WESE provides a framework for constructing web-based *object factories*. An object factory can be viewed as a web-based repository of components with an added capability for simulating them. The object factories play a pivotal role in providing a framework for management of components and the infrastructure for distributed simulation. Figure 3 illustrates the layout of a WESE factory. The initial handle to a factory is provided by the *gateway* module. The module hooks on to a specified IP (Internet Protocol) address via the communication backbone and processes the initial requests from different simulation managers. This IP address that should be specified in the configuration file to locate and communicate with a factory. The task of interacting with a simulation manager to create components and to set up a simulation is handled by the *session manager module*. The session manager also handles some of the specific semantics of the simulation engine. The *configuration manager* tailors the components generated by the factory to meet the user's specifications. The simulation sub-system constitutes the actual simulation engine of the factory. A WESE factory is built from sub-factories and *object stubs*. The object stubs are the atomic components of a factory. Object stubs contain attributes of the physical component (such as cost, size, and speed) along with the formal specifications for the component. The object

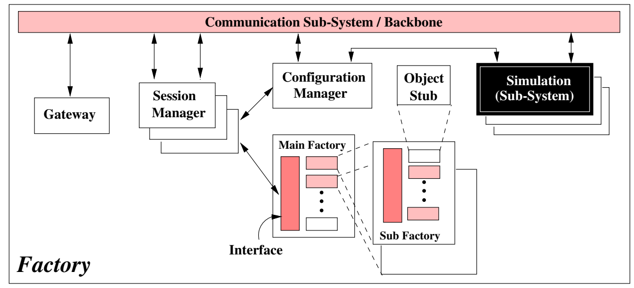

Figure 3: A WESE Factory

factories collaborate with the WESE Server to enable web-based simulations. WESE provides a simple, yet robust *application program interface* (API) for developing simulation models. Further details on the API and WESE are available in the literature [10].

4 IMPLEMENTING DCS IN WESE

DCS may be achieved by replacing a given LP, or a set of LPs, in a simulation with a functionally equivalent LP, or a set of LPs. Some of the scenarios that could arise in DCS are illustrated in Figure 4. The *"1 to 1"* case, shown in Figure 4, in which one LP is replaced by another, is the simplest instance of DCS. As shown in the figure, the *"N to 1"* scenario, where in a set of LPs are replaced with a equivalent LP, arises when a compound component, consisting of a set of sub-components, is replaced with an atomic component. This scenario is equivalent to abstracting a part of the model. The *"N to M"* instance is one where in a set of LPs (N LPs) are replaced with a equivalent set of LPs (M LPs). This scenario arise when a compound component, is replaced with another compound component. However, this instance can be viewed as sequence of atomic component substitutions. An atomic component may be replaced with a compound component, reducing the level of abstraction, causing a single LP

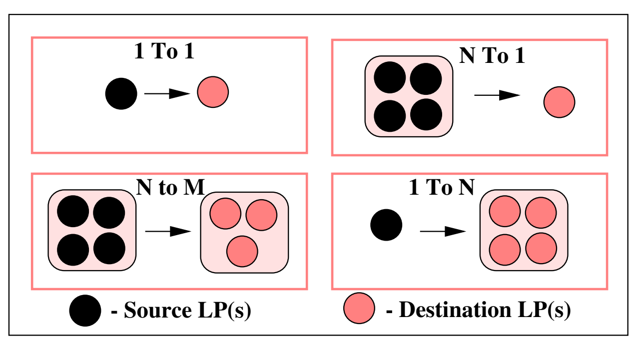

Figure 4: Scenarios in DCS

```
ssl_module ::= auxiliary_component ssl_module_body | ssl_module_body
ssl_module_body ::= { component_definition_section }
    { component_instantiation_section }
    { net_list_section }
auxiliary_component ::= label | : url optional_parameter
```

Figure 5: Modified BNF for SSL Module

to be replaced with a set of LPs. The *"1 to N"* scenario, shown in Figure 4, illustrates this case. To enable modeling of the different scenarios and effectively utilize support for DCS, modifications to the modeling language and simulation infrastructure are required. Consequently, to enable DCS in WESE, modifications to SSL, the SSL parser, SSL-IF, elaborator, and the simulation infrastructure were carried out. The issues involved in the implementation of these modifications along with the tradeoffs in their design are discussed in the following subsections.

4.1 Modifications for modeling DCS

The initial phase of implementing support for DCS in WESE involved extending SSL to include additional constructs for modeling the different scenarios illustrated in Figure 4. Care was taken to ensure that the extensions were minimal so that the language continues to be simple, flexible, and easy to process. The primary extension was to permit an auxiliary *module* or *component definition* to be associated with a module. The BNF of the modified grammar rule for a module is shown in Figure 5. When DCS for a module is requested, the set of components contained by the module are substituted using the auxiliary *module* or *component definition* and vice versa, as the case may be. Modeling the *"N to 1"*, the *"N to M"*, and the *"1 to N"* scenarios (illustrated in Figure 4) using this extension is straightforward. In WESE, DCS can be performed only at a module level. However, a module can contain a single component and it can be replaced it with an auxiliary component. This feature can be exploited for modeling the *"1 to 1"* DCS scenario. The semantics of the *netlist* was also extended to include references to the auxiliary module and component definitions.

SSL-IF was also extended to correspondingly reflect the changes to the grammar. The elaborator was also modified to account for the auxiliary components. The elaborator also flattens auxiliary modules and component definitions. It results in the creation of unique instances of the auxiliary components. The auxiliary components are an integral part of the elaborated SSL-IF and are identified using special flags in the various data structures. The elaborator was extended to identify primary input and output components, *i.e.,* components that are directly connected to the input and output ports of the enclosing module. This information is utilized during simulation to update netlist entries when components are substituted with other components. The elaborator also collates information on the set of components contained by each module. This information is utilized to identify a set of components that need to be replaced when DCS is initiated. The data collated by the elaborator is embedded into the corresponding SSL-IF nodes generated during elaboration. The data is passed on to the simulation modules of WESE that utilize them for enabling efficient DCS. Modifications to the simulation infrastructure of WESE to enable DCS are presented in the following subsection.

4.2 Simulation infrastructure for DCS

The process of dynamically substituting components during simulation (as shown in Figure 4) involves the following steps: triggering DCS in the simulation, creation of new LPs that model the components, updation of states and events of the LPs, and updation of kernel information. In Time Warp synchronized simulations, additional care must be exercised to implement these phases in the presence of rollbacks that could occur in a Time Warp synchronized simulation. A number of modifications were carried out to the simulation modules of WESE to enable DCS. The most significant change was a modification to the structure and API of a LP. The API was modified to utilize object oriented (OO) techniques to completely disassociate a user-defined LP from the simulator core, as shown in Figure 6. In the earlier API, the `UserDefined Object` class would be directly inherited from the `Kernel Object` class. As illustrated in the figure, the `Kernel Object` and `User Object` are linked using pointer references. The `User Object` translates the API function calls to corresponding `Kernel Object` methods while

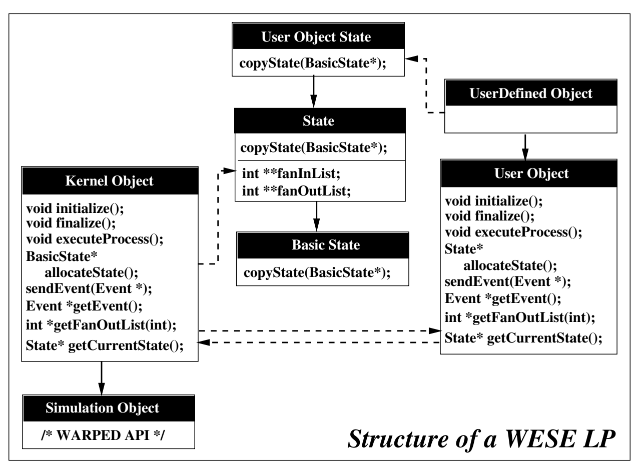

Figure 6: Modified structure of a WESE LP

the Kernel Object translates WARPED API function calls to corresponding User Object methods. The API presented by the User Object class is similar to the earlier API of WESE. Hence, the changes required to the existing components of WESE were minimal.

This design is motivated primarily by two factors. The primary issue being that the WARPED kernel does not support creation and deletion of LPs during simulation. In other words, the WARPED kernel does not permit the structure and composition to change once simulation commences. However, DCS involves changes in structure and composition during simulation. This issue is resolved by using the class hierarchy, shown in Figure 6, wherein the Kernel Objects are static (*i.e.,* they do not change during simulation) while the User Object class hierarchy is dynamic (*i.e.,* it can change during simulation). The Kernel Objects provide the static interface to the WARPED kernel, while different UserDefined Objects can be plugged into the Kernel Object during simulation. This technique enables dynamic substitution of components while adhering to the specifications and semantics of the WARPED kernel. However, this design does not provide an effective technique for creating new components that may be necessary during DCS. Hence, in WESE, the auxiliary components that could potentially be used during simulation are also created. However, these components merely as place holders and do not perform any activity until they are activated through DCS.

The second motivation for the design is that the Kernel Object class provides a convenient spot for implementing support for DCS by utilizing the simulation infrastructure of WARPED. The WARPED kernel insulates the Kernel Objects from rollbacks which considerably reduces the complexity and overheads involved in implementing DCS. Also, with this design, the overheads and process of DCS is transparent to the components. This solution is independent of the underlying synchronization mechanism. Accordingly, in WESE, an event driven approach has been adopted for carrying out the sequence of steps involved in dynamically substituting components. The set of *kernel events* used by WESE was extended to include events for sequencing the different phases of DCS. The primary drawback of this design is that it introduces additional state saving overheads in Time Warp simulations. However, a number of Time Warp optimizations can be employed to minimize state saving overheads [2]. This design also introduces additional overheads during simulation since each API function call involves one extra level of indirection. Also, maintaining the auxiliary components could prove to be a bottleneck for large simulations [12]. However, component aggregation techniques can be employed to minimize the overheads [12].

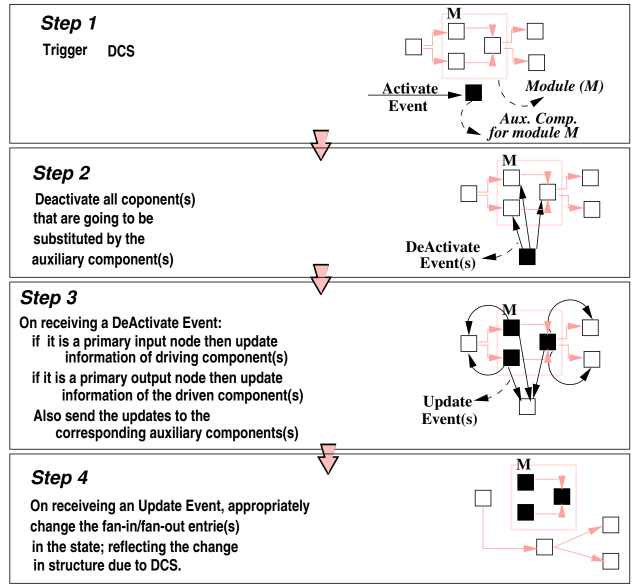

Figure 7: Sequence of operations during DCS

A typical sequence of steps performed by the Kernel Objects to achieve DCS are shown in Figure 7. The figure also illustrates the corresponding sequence of transformations that occur to the model during the different phases. The kernel events that participate in DCS are also shown. The initial phase involves triggering DCS in the simulation by scheduling an Activate or a DeActivate event, as the case may be, to the corresponding auxiliary component(s). DCS could be triggered externally, by using interactive simulation features, or internally, by the simulation model based on certain application-specific conditions. On receiving a Activate or a DeActivate event, the Kernel Objects initiates the process of DCS. During the second phase of DCS, the activated set of auxiliary components schedule DeActivate events to the set of components that they are going to substitute. The information on the set of components to be replaced is collated during elaboration and is passed onto the corresponding Kernel Objects by the WESE server during initialization. The server also passes the primary input and output component flags collated by the elaborator (as explained in subsection 4.1) along with the netlist data to the respective Kernel Objects. In the next phase, the Kernel Objects that receive the DeActivate Event utilize this information to schedule Update Events to all the related components. The related components are those components with which a given component communicates. This list of related components is obtained from the from the netlist data of the component. The primary input and output components also schedule Update Events to the auxiliary component to

Model Name	Description	Number of Components		
		Regular	Aux.	Replaced by Aux.
M1	4-bit adder	56	4	9
M2	5-bit Mux.	33	6	4
M3	Cascaded half-adders	16	6	2
M4	Chain of not gates	70	5	6
M5	Chain of not gates	330	1	30

Table 1: Models used in experiments

provide the list of related components. This information is required to build the netlist of the new component. On receiving the various `Update Events`, the various `Kernel Objects` update their netlists reflecting the change in structure. As shown in Figure 7, during subsequent simulation cycles, the events generated would be passed on to the new components while the old components get deactivated. To handle the different scenarios that could arise during DCS (as shown in Figure 4, additional sub-tasks, such as instantiating new components in different object factories, are performed in the corresponding DCS phases.

The kernel events used during the different phases of DCS are scheduled using WARPED's simulation infrastructure. The usage is similar to that of any other WARPED application. Hence, to ensure that the events are scheduled in the correct sequence, a *delta* delay is introduced between each event using a two tuple definition for simulation time. The use of a two tuple definition for simulation time is hidden from the user by the API. Since the process of DCS proceeds in *delta* cycles, it appears to occur at a particular instant in simulation time. The *delta* delays also ensure consistent recovery from rollbacks. The data pertaining to the component is stored in the state of the `Kernel Object`. When a rollback occurs, WARPED appropriately restores the state of the `Kernel Object` ensuring coherence of the different phases in DCS. The disadvantages of the event driven design for DCS is that a large number of events could be scheduled during DCS. Hence, if DCS occurs frequently, the performance of the simulation could deteriorate. One of the limitations of the current implementation is that it can be used to substitute only "memory less" components (*i.e.,* components that do not have an explicit notion of state). In other words, the current implementation does not provide support to map the state of the old module to that of the new module. Research is underway to provide a support for mapping the state space of one module to another during DCS. Also, it must be noted that the transient events that were already scheduled for the old set of modules do not get reassigned to the new set of components. They continue to get processed by the substituted set of components. The experiments conducted using DCS in WESE are presented in the following section.

5 EXPERIMENTS

The experiments conducted to evaluate the support for DCS in WESE consisted of two phases. During the first phase an object factory consisting of a collection of logic gates was developed. The factory contained logic gates such as two input `and` gate, two input `or` gate, two input `exclusive-or` gate, and `not` gate. More complex components, such as a `half adder` and a `full adder`, were included in the hardware factory. The factory also contained a bit pattern generation component and a bit display component. The pattern generator can generate all possible bit patterns of a given length and can be used to exercise the inputs of a model developed using components from the factory. The display component can be used to generate a set of bits as outputs from the simulations. The factory also contained a *controller* component that provides a convenient interface to trigger DCS. The second phase of the experiment consisted of developing logic models in SSL using the various components from the hardware factory. The characteristics of some of the models using the experiments is shown in Table 1. The models included auxiliary component specifications for the modules that had equivalent higher level abstractions. The number of components replaced by each auxiliary component in the models is also shown in the table (column `Replaced by Aux.`). For example, model M1 was implemented using structural models of full adders. The structural models of also included an auxiliary specification to use the `full adder` component available in the factory. The `full adder` component substitutes nine components constituting the structural model. The SSL descriptions also used the `controller` components activate the auxiliary modules (trigger DCS) at different time points during simulation.

The simulation experiments were conducted on a network of shared memory multi-processor (SMP) workstations. Each workstation consisted of two Pentium pro Processors (166 Mhz.) with 128 mega bytes (MB) or main memory (RAM). The workstations were interconnected using fast Ethernet. The graph in Figure 8 presents the change in the total number of events processed with respect to the duration of simulation time in which the auxiliary components were active. These statistics were collated from the experiments conducted using a single factory where in no rollbacks occur. The data points shown with zero durations did not involve any DCS and represent the basic number of events executed by each model. As illustrated by the graphs, for short durations during which the auxiliary component

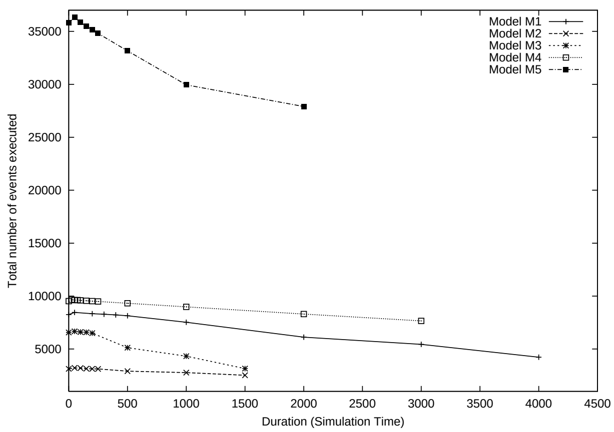

Figure 8: Events versus Duration of DCS

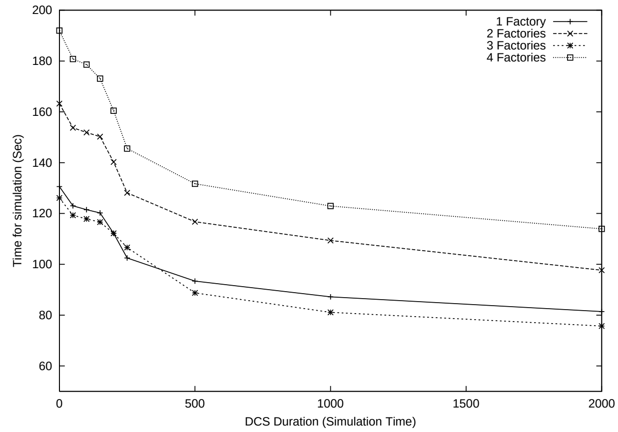

Figure 9: Time for Parallel Simulation

is active, the total number of events processed is higher. The increase in number of events is due to the additional kernel events used to activate and deactivate the components during DCS. However, as the duration increases the number of events processed decreases. The number of events decrease since a set of components are replaced by a single component which results in the elimination of a number intermediate events used and the total number of events in the simulation decreases. As shown in Figure 8, the duration of simulation time for which DCS reduces the number of events varies with respect to the model characteristics. This value plays a crucial role in the effectiveness of DCS to improve performance of the simulations. If the duration is smaller than this threshold value, then as the number of substitutions increases, the total number of events in the simulation increases and the performance of the simulation decreases, and vice versa.

Figure 9 presents the time for simulating model `M5` in parallel using a varying number of factories. These experiments were conducted by deploying the object factory on different workstations and modifying the SSL descriptions to choose components from the different factories. The components were chosen from the different factories at random. The timing information shown in the graph is the average of 10 simulation runs. As illustrated by the graph, the performance of the simulations increases as the duration during which the auxiliary components are active increase. As shown in Figure 8, the improvement in performance is due to the decrease in the total number of events that need to be processed. As illustrated by Figure 9, the parallel simulations performed using 3 factories performs better than those performed using a single factory. The performance improves since the simulation overheads get distributed across the three processors. In the 2 factories case the computational overheads dominate the simulation, while in the 4 factories case communication overheads dominate. Hence, in these cases the overheads dominate the gains accrued by employing parallel simulation and the performance of the simulations do not improve. As illustrated by Figure 8 and Figure 9 the performance of parallel simulations can be improved through DCS.

6 CONCLUSIONS

Component based modeling techniques provide a effective means to study systems through "plug and play" of components. In this paper the issues involved in substituting the components dynamically, during simulation were presented. The design and implementation of the support for Dynamic Component Substitution in WESE was illustrated. The experiments in which DCS was used to change the level of abstraction of the model during simulation were described. The results obtained from the experiments indicate that considerable gains in the performance of simulation can be accrued by employing DCS. The technique can be used to accelerate simulations, rare event simulation in particular, to scenarios of interest. DCS can be used to replace a single component with multiple components and simultaneously study the effects of different decisions. This provides a novel technique for simulating multiple futures. DCS can also be used to selectively study parts of a large simulation thereby increasing the performance and the capacity to simulate large scale models over the WWW.

References

[1] Fishwick, P. A. Web-based simulation: Some personal observations. In *Proc. of the 1996 Winter Simulation Conference* (Dec. 1996), pp. 772–779.

[2] Fujimoto, R. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (Oct. 1990), 30–53.

[3] Huang, P., Estrin, D., and Heidemann, J. Enabling large-scale simulations: Selective abstraction approach to the study of multicast protocols. In *In Proceedings of International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Networks* (Oct. 1998).

[4] Hybinette, M., and Fujimoto, R. Optimistic computations in virtual environments. In *Proceedings of the Virtual Worlds and Simulation Conference (VWSIM'99)* (Jan. 1999), pp. 39–44.

[5] Jefferson, D. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 405–425.

[6] Page, E. H., Griffin, S. P., and Rother, L. S. Providing conceptual framework support for distributed web-based simulation within the high level architecture. In *Proceedings of SPIE: Enabling Technologies for Simulation Scicence II* (Apr. 1998).

[7] Page, E. H., and Nance, R. E. Parallel discrete event simulation: A modeling methodological perspective. In *Proceedings of the ACM/IEEE/SCS 8th Workshop on Parallel and Distributed Simulation* (Dec. 1994), pp. 88–93.

[8] Pidd, M., Oses, N., and Cassel, R. A. Component-based simulation on the web? In *In Proceedings of the 1999 Winter Simulation Conference (WSC'99)* (Dec. 1999).

[9] Radhakrishnan, R., Martin, D. E., Chetlur, M., Rao, D. M., and Wilsey, P. A. An Object-Oriented Time Warp Simulation Kernel. In *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, D. Caromel, R. R. Oldehoeft, and M. Tholburn, Eds., vol. LNCS 1505. Springer-Verlag, Dec. 1998, pp. 13–23.

[10] Rao, D. M., Chernyakhovsky, V., and Wilsey, P. A. WESE: A Web-based Environment for Systems Engineering. In *2000 International Conference On Web-Based Modelling & Simulation (WebSim'2000)* (Jan. 2000), Society for Computer Simulation.

[11] Rao, D. M., Radhakrishnan, R., and Wilsey, P. A. FWNS: A Framework for Web-based Network Simulation. In *1999 International Conference On Web-Based Modelling & Simulation (WebSim'99)* (Jan. 1999), A. G. Bruzzone, A. Uhrmacher, and E. H. Page, Eds., vol. 31, Society for Computer Simulation, pp. 9–14.

[12] Rao, D. M., and Wilsey, P. A. Simulation of ultra-large communication networks. In *Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (Oct. 1999), pp. 112–119.

[13] Villén-Altamirano, M., and Villén-Altamirano, J. Restart: A straightforward method for fast simulation of rare events. In *In Proceedings of the 1994 Winter Simulation Conference (WSC'94)* (Sept. 1994), pp. 282–289.

[14] Vinoski, S. Corba: Integrating diverse applications within distributed heterogenous environments. *IEEE Communications Magazine, Vol 35, No. 2* (Feb. 1997).

[15] Wang, Z., and Maurer, P. M. Lecsim: A levelized event driven compiled logic simulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC'90)* (June 1990), pp. 491–496.

AUTHOR BIOGRAPHIES

Dhananjai M. Rao is a Ph.D. student in the Department of Electrical and Computer Engineering & Computer Science at the University of Cincinnati. He received his Bachelor's degree in Computer Science and Engineering from the University of Madras in 1996. His research interests include parallel discrete event driven simulation, distributed computing, network simulation, object oriented design patterns, and web-based simulation.

Philip A. Wilsey is an Assistant Professor in the Department of Electrical and Computer Engineering & Computer Science at the University of Cincinnati. He received PhD and MS degrees in Computer Science from the University of Southwestern Louisiana and a BS degree in Mathematics from Illinois State University. His current research interests are parallel and distributed processing, parallel discrete event driven simulation, computer aided design, formal methods and design verification, and computer architecture.