DEXSim: an experimental environment for distributed execution of replicated simulators using a concept of single-simulation multiple scenarios

Changbeom Choi, Kyung-Min Seo and Tag Gon Kim

SIMULATION published online 7 February 2014
DOI: 10.1177/0037549713520251

Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
1–22
© 2014 The Society for Modeling and Simulation International
DOI: 10.1177/0037549713520251
sim.sagepub.com

DEXSim: an experimental environment for distributed execution of replicated simulators using a concept of single simulation multiple scenarios

Changbeom Choi, Kyung-Min Seo and Tag Gon Kim

Abstract
This paper presents an efficient and scalable experimental environment for distributed execution of replicated simulators. By taking a performance-centered approach, the proposed technique makes the best use of distributed hardware resources for faster data collection. Accordingly, the primary contribution of this work is to describe how the environment improves scalability and utilizes distributed hardware resources efficiently. To do this, we suggest a new concept of single simulation multiple scenarios and propose a distributed execution simulation framework regarding the following three aspects: (1) layered architecture model design; (2) protocol definitions interacting with them; and (3) framework implementation. The proposed model architecture and protocol definitions guarantee a straightforward structural scalability and an efficient load-balanced utilization between hardware resources. Moreover, the framework operates simulation execution automatically without users' extra work. In order to prove the efficiency of the proposed framework, we performed three extensive experiments with different models, that is, different systems. The experimental results show that simulation performance increases proportionally with the number of hardware resources, minimizing the overhead of the proposed framework's utilization.

Keywords
Faster data collection, experimental frame, distributed simulation, multi-core computing

1. Introduction

Simulation is often used to resolve problems that are too hard to solve either explicitly or numerically.[1] Simulation-based experiments provide insight into all aspects of system development and acquisition, and in several ways that were not previously possible.[2] For example, in defense communities, a battle experiment through simulation provides new insight into the future operation capacity (FOC),[3,4] and gives a potential evaluation of military system acquisitions.[5,6] Also, in industrial fields, simulation-based experiments are utilized to solve scheduling problems such as estimating cycle time, which is required for a job or lot to traverse a given routing in a production system.[7] Finding a solution using these simulation-based experiments requires answering "what if" questions involving thousands or millions of different scenarios.[8] Because such problems lead to simulation being an extremely a time-consuming job, many studies have attempted faster data collection methods to mitigate the time-consuming phenomenon.[9]

Recently, multi-core processing techniques have become more widely used in various industries, which has resulted in parallel and distributed simulation techniques.[10,11] With the current industrial and research trend, we have focused on fully utilizing given hardware resources, such as multiple central processing units (CPU cores) or multiple machines, for faster data collection. In other words, we have made the best use of given distributed hardware resources by allocating a specific simulation application to each CPU core. A simple approach for utilization of the given hardware resources is to execute multiple simulations gradationally and manually. For example, a

Department of Electrical Engineering, Korea Advanced Institute of Science and Technology, Korea

Corresponding author:
Changbeom Choi, Department of Electrical Engineering, Korea Advanced Institute of Science and Technology, 373-1 Guseong-dong, Yuseong-gu, Daejeon, Korea 305-701.
Email: cbchoi@smslab.kaist.ac.kr

simulationist classifies overall experimental scenarios into small groups to map them into all available resources. Thereafter, the simulationist executes multiple simulations with every resource, and collects the experimental results. If additional hardware resources are available during simulation execution, the simulationist generates new scenario groups from the remaining scenarios and assigns them to supplementary resources. In addition, the simulationist should check simulation progress frequently to see whether some unexpected or abnormal behaviors occur or not. Due to manual operations for the overall process, it can easily be error-prone; for example, he not only repeats simulations several times for the same scenario, but also misses some important scenarios. Consequently, this causes different kinds of time-consuming phenomena. Explanation of the preceding situation demonstrates simple but vital reasons that many researchers have developed their own automatic execution framework for distributed simulation. Therefore, these frameworks should support four requirements: (1) the framework should fully utilize the initial assigned hardware resources (*effectiveness*); (2) it should manage overall scenarios and their results conveniently (*efficiency*); (3) it should control unexpected behaviors during simulation execution (*maintenance handling*); and (4) it should guarantee an expandable simulation execution, such as scenarios or hardware expansions (*scalability*).

In simulation fields, some researchers have attempted to find and explore the first two requirements. These researchers have proposed chip-multiprocessing[12] or parallel/distributed simulation techniques[13–16] to fulfill them. To be specific, some researchers have suggested a middleware framework to assist a simulationist with accessing computing resources easily and without modifying existing simulation applications;[17] others have implemented the cloud middleware that provides a cloud-enabled, component-based programming interface.[18] Notwithstanding their contributions, they missed some important considerations: the third and fourth requirements for *maintenance handling* and *scalability*. In detail, they may assume that simulation applications have been tested completely before simulation-based experiments. In practical fields, however, it is difficult to verify the application for all the scenario cases; unexpected or abnormal behaviors can occur during simulation execution.[19] Next, they have proposed and developed their own software tools, not based on a general theorem or standard, such as IEEE 1278 Distributed Interactive Simulation (DIS)[20] or IEEE 1516 High Level Architecture (HLA).[21] This limits facilitating more scalable simulation execution whenever scenarios or hardware resources are extended.

Our point of this study is, therefore, to propose an execution framework for meeting all the aforementioned requirements. For the theoretical basis, we propose a new concept, single simulation multiple scenarios (SSMSs), which is based on the taxonomy of the computer architecture. The SSMS concept guarantees that the framework can be scaled up the experiment environment depending on joined machines, while only a single simulation gives rise to simulation results about multiple experimental scenarios through distributed execution. With the theoretical basis of the SSMS concept, we developed a distributed execution simulation (DEXSim) framework that is capable of accelerating simulations by replicating simulation applications across multiple CPU cores in a single machine, or in multiple machines. The proposed DEXSim provides several techniques that support the fulfillment of the foregoing requirements of *effectiveness, efficiency*, and *maintenance handling*: (1) multiple machines/cores distribution; (2) input scenarios and output results management; and (3) execution and shutdown of the simulation application. Furthermore, all the techniques are designed and implemented based on IEEE 1516 HLA as a network infrastructure. As a result, the simulationists can easily extend the framework by including simulation modules as a federate to the framework (*scalability*).

In summary, the objectives of this paper are to suggest a theoretical basis for the DEXSim framework and to suggest an example of the framework realization. To do this, we first identify a structural design of the DEXSim framework. For functional classification, we propose a two-tiered management scheme, in which the upper layer manages entire simulations, whereas the lower layer controls the simulations on a single machine. Based on the structural modeling, we describe detailed interaction messages between the proposed models by defining four kinds of protocols: node (*node* refers to the communication terminal equipment, such as a PC, and in this paper, we regard a node as an individual machine to be joined to the DEXSim framework) management protocol; synchronization management protocol; scenario management protocol; and process management protocol. For a simple description of protocol definition, we use a formal semantics of the sequence diagram, which is part of the unified modeling language (UML).[22] In addition, we describe how the DEXSim framework controls the simulation jobs adaptively and guarantees to control various abnormal situations. Finally, we explain an implementation of the DEXSim framework in the Windows operating system environment. The outcomes of three experiments illustrate the usefulness of the proposed work.

The remainder of this paper is structured as follows: Section 2 explains a theoretical basis for the DEXSim framework, SSMS; Section 3 compares several previous works; Sections 4 and 5 explain the proposed DEXSim framework regarding structural design and protocols

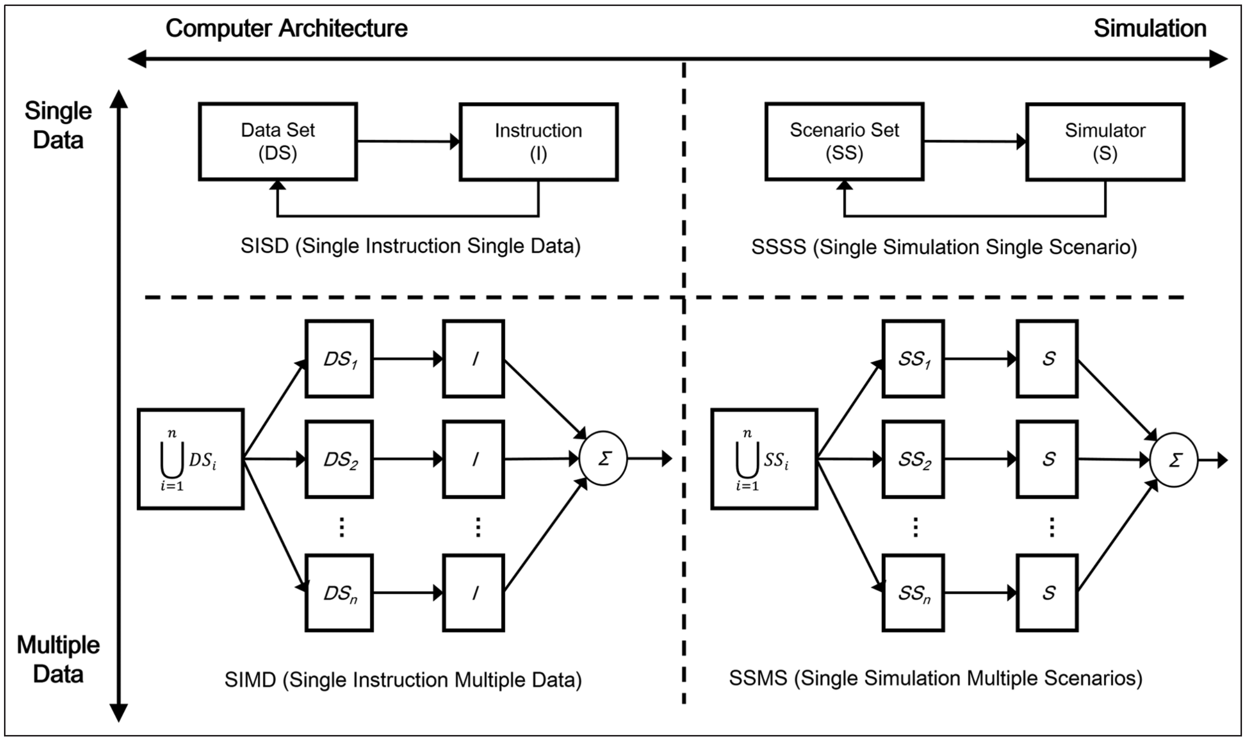


Figure 1. Comparison of computer architecture and simulation field.

behavior; and Section 6 introduces the implementation of the DEXSim framework in the Windows operating environment. Section 7 illustrates three experimental results to prove the efficiency of this work, and finally, Section 8 concludes this study and proposes future extensions for a more complete solution.

2. Problem description

For the theoretical starting point, we describe a new concept for simulation-based experiments with multiple hardware resources. The most well-known taxonomy for parallel computers was proposed by MJ Flynn in 1966.[23] It is based on the multiplicity of data and operations, which identifies four classes of computers: (1) single instruction stream single data stream (SISD); (2) single instruction multiple data (SIMD); (3) multiple instruction single data (MISD); and (4) multiple instruction multiple data (MIMD) computers. Among them, the SISD computer corresponds to non-parallel computers that can execute one instruction at a time on one data at a time, whereas the SIMD computer are based on a central program controller that drives the program flow and a set of processing elements that all execute the instructions from the central controller on their individual data items. Briefly, in the SIMD computers, all processors are given the same instruction and each processor operates on different data. In this paper, we pay attention to the concept of the SIMD computer, and expand the SIMD concept to the simulation field for faster data collection.

Figure 1 shows concept expansion from the computer architecture field to the simulation field. The left-hand side of Figure 1 indicates the computer architecture field, whereas the right-hand side corresponds to the simulation field. The vertical partition represents the amount of data to be processed at once. In the simulation field, the traditional simulation-based data collection can be explained with a concept of single simulation single scenario (SSSS). Therefore, simulationists run simulations thousands or millions of times to cover all scenarios in the scenario set. It is similar to the SISD architecture that is the traditional concept of the computer architecture. Due to the increasing necessity of computer architectural parallelism, the SIMD architecture has been widely used rather than the SISD architecture. Accordingly, based on the SIMD architecture, we propose a new concept, called SSMSs, for efficient utilization of the given hardware resources. The SSMS concept means that overall experimental scenarios can be conducted through only a single simulation if the computing resources are available. To realize the SSMS concept, it demands the advanced techniques such as simulator replication, scenario distribution, and concurrent execution of replicated simulators, as illustrated in Figure 1. However, our work relies on theoretical architecture—SSMS architecture—and we implemented an experimental environment based on that architecture. In other words, we defined an architecture for simulation execution and experiment management. Then, we implemented an extendable experiment environment utilizing the IEEE 1516 standard, building on the theoretical foundation.

Therefore, the central part of this paper is directed at how the SSMS concept is applied to the proposed DEXSim framework. Before describing the SSMS-applied DEXSim framework, we explain previous research concerning faster data collection in the following section.

3. Related works

Before moving to the central part of our work, we review several previous studies of faster data collection. An earlier study of faster data collection was explored by Heidelberger in the 1980s.[24] In his research, he compared the obtained performance of replicated simulation and parallel simulation based on statistical analysis. Moreover, several simulation software tools were proposed for replicated simulation.[25–30] These tools were based upon various technologies, such as parallel and distributed computing technologies. Among them, five recent representative studies were selected and compared. Table 1 shows the comparison among the representative simulation software and our DEXSim in the perspective of the experimental environment, simulation execution, and the design of experiment. The guideline for classification of the studies is provided by Taylor et al.[17]

A framework for Discrete Event Simulation and Modeling in Java (DESMO-J) was developed at the University of Hamburg in the late 1990s.[31] Its goal is to provide a framework that supports an efficient implementation of simulation optimization projects. In order to implement the DESMO-J, the authors utilized the Java Remote Method Invocation (RMI) technology. Therefore, the framework can instantiate a simulation model, execute the model, and collect information from remote machines. In addition, the framework's integrated design of experiment scheme allows simulationists to customize parameters of simulation scenarios during experiments. Since the DESMO-J is implemented on the Java RMI, the simulationists should implement their simulation models in Java. Moreover, it may be difficult to implement an additional functionality to the framework, since the architecture of the framework does not consider the extension of the framework.

The MITRE Elastic Goal-directed simulation framework (MEG) is a middleware framework to allow simulationists to access computing resources easily without modifying existing simulation applications.[32] In particular, the objective of MEG is to embrace various grid schedulers; thus, the authors have adopted the Gridway metascheduler,[33] which works with various distributed resource management systems, such as HT Condor,[26] Globus,[27] and Sun Grid Engine (the name has since changed to Oracle Grid Engine).[28] Moreover, the MEG utilized third-party data processing and visualization tools to help the simulationist to gather simulation results from distributed resources and to analyze the simulation results easily. Therefore, the structure of the MEG is flexible compared to the DESMO-J. Nevertheless, it may be difficult to develop MEG-compatible experimental components by other developers, since the architecture of the MEG does not consider the third-party components to participate in the experiments. The distributed resource management systems that MEG utilizes are designed as general-purpose systems, so that a special distributed resource management system is necessary for the simulation domain.

The Concurrent Replications of Parallel and Distributed Simulations (CR-PADS) framework is a framework to maximize the speedup of the simulation processes and the utilization of computing and communication resources.[34] To maximize speedup and utilization, the CR-PADS framework adopts the replication concept that duplicates the logical processes, and executes them independently through distributed computing resources. As a result, many independent simulation runs are executed concurrently by replicating them. In order to implement the framework, the authors developed new parallel and distributed simulation middleware based on the IEEE 1516 standard.[35] Therefore, when the simulationist wants to utilize the framework, he or she should compile their simulator with the middleware. It may also be hard to extend the functionality of the simulation software, since the replication architecture is tightly embedded within the middleware

In 2011, the ICT Innovation Group at Brunel University and Saker Solutions developed simulation software, called the SakerGrid,[36] to support the deployment of simulations across a desktop grid. The SakerGrid has three components, which are the client, manager, and worker, and each of these components is implemented by utilizing the same grid middleware. The architecture of the SakerGrid can be divided into two elements, the frontend and the backend. The backend of the SakerGrid comprises the manager and worker components. The manager component controls the simulation jobs and dispatches them to the workers on the grid. When the worker finishes the simulation jobs, the manager component collects them and combines the results from the other workers. The simulationist uses the frontend of the SakerGrid, the client component, to submit simulation jobs to the manager. In addition, the simulationists can monitor the progress of their execution and download the results when they are complete. The SakerGrid may be the powerful industrial simulation software that utilizes the computing resource of a desktop grid; however, the SakerGrid executes simulation models so that the simulationist should convert their simulation models. Moreover, it may be difficult to extend functionality of the SakerGrid to the third parties.

Table 1. Comparisons of previous research and the DEXSim framework.

		DESMO-J	MEG	CR-PADS	SakerGrid	mJADEs	DEXSim
Experimental Environment Perspective	Type of Experimental Environment	Single run task approach	Installed multiple run approach	Multiple run task approach	Installed multiple run approach	Installed multiple run approach	Multiple run task approach
	Load balancing	None	None	None	None	Dispatch simulation tasks from largest to the smallest	Simulation process allocationin federate DEXSim
Simulation Perspective	Type of Simulator/ Simulation Model	Simulation Model	Simulation Model	Standalone Simulator	Simulation Model	Simulation Model	Standalone Simulator
	Requirements of Simulator/ Simulation Model	Simulation model must be implemented using Java RMI	None	User's simulator code should be complied with middleware	Simulator must be integrated with the middleware	Simulation model must be implemented using Java	None
	Simulator Fault Management	None	None	None	None	None	Abnormal termination management andabnormal execution management
Design of Experiment Perspective	Experiment Management	Simulation Parameters Control	Simulation Parameters Control	None	None	None	Scenario generation infederation experimental frame

The mJADES simulation system[18] is implemented based on two technologies, the Java-based simulation library[37] and the cloud middleware that provides a cloud-enabled, component-based programming interface.[38] The JADES simulation engine supports the development and evaluation of discrete event simulation models. In the JADES simulation engine, the simulationists develop the simulation model in a process-oriented view, so that the modeler describes his model in terms of processes, which interact with other processes. After the simulationists develop the simulation models using JADES, the simulationists may utilize the mOSAIC to execute the simulation on the cloud service providers. In mJADES, the user submits the simulation requests to the mJADES manager through the HTTP gateway of the mOSAIC. Then, the mJADES manager transforms the simulation requests to simulation jobs and dispatches them to the cloud infrastructure. Because the simulation models are represented as processes in mJADES, the time required for simulation jobs may vary. To help the load balancing, the mJADES dispatches simulation jobs from the largest to the smallest set of simulation process. Similar to other simulation software, the architecture of mJADES is fixed, and it is hard to extend the functionality of this software. In addition, the simulationist should describe the simulation models in a process-oriented view, so that the simulationist should rebuild the legacy simulation models to utilize the mJADES framework.

These studies, as abovementioned, were focused on the simulation software, which manages the experiments on the parallel and distributed computing resources. From another perspective, several studies have focused on chip-multiprocessing techniques or parallel and distributed simulation techniques for the full utilization of the given hardware resources.[13–16] Our approach in this paper is identical to the above studies, but from the viewpoint of the efficient utilization of hardware resources.

Nevertheless, there are three major differences from previous studies in the present work. Firstly, our DEXSim framework executes and controls the simulators, not the simulation models. Several studies need simulation algorithms or data structures to simulate target simulation models on a concrete platform.[14,15] The DEXSim framework is a simulator-independent execution environment, and it executes at the simulator level, not the simulation model level. Therefore, the DEXSim framework can utilize any simulator regardless of its system type, such as discrete event simulators, continuous simulators, and discrete time simulators.

Secondly, we aimed for the target simulator to be complete and to operate in a standalone environment, which means that the simulator does not divide into component simulators for interoperation with each other. Often the simulator, which measures and analyzes effectiveness by collecting enormous amount of data, is composed of an integrated simulator and does not contain several distributed components.[16] This may be particularly true in simulators for finding an optimal solution.[39–42] Thus, we constrained the simulator, in this paper, to being complete itself and to being able to operate in a standalone environment. This allows the DEXSim framework to assign a single CPU core to a certain simulator.

Thirdly, we aimed to implement a scalable and extendable experiment environment. In order to speed up the simulation experiment, the simulationist may add additional computing resources during the experiments. In this situation, the simulation software should identify the resources and dispatch simulation jobs immediately. Moreover, the architecture of the simulation software should allow the third party to extend the experiment environment without changing core components of the software. For instance, a third-party simulationist may want to apply an advanced design of experiments scheme to the experiment environment. Moreover, various visualization techniques may assist the analysis of the simulation results, if the simulationist applies their visualization results to the simulation software. In order to achieve these requirements, we adopted the aforementioned simulation architecture to build a protocol. Then, we defined a protocol based on the IEEE 1516 standard. By defining the protocol based on the HLA/RTI, a third-party simulationist not only extends the functionality under the given protocol, but also utilizes another simulator as a scenario generator by interoperation. With these three differences in mind, we will explain our DEXSim framework in detail in the following sections.

4. DEXSim framework: structural design

In describing our DEXSim framework, it may be useful to begin with the suggestion of high-level structural design preferentially. Accordingly, we will first describe a structural approach from a high-level viewpoint of the DEXSim framework; then, we will explain the detailed model construction and its roles of the components.

4.1. Proposed two-tiered DEXSim framework

Because the purpose of the DEXSim framework is to make the best use of the given hardware resources, we utilized multiple machines, which are deployed in a distributed environment. In addition, we utilized the multiple CPU cores of the machine to collect simulation data quickly. In order to manage the experiments on distributed machines, we designed the two-tiered DEXSim structure. This two-tiered structure provides not only utilization of multiple cores of a single machine condition, but also utilization of multiple machines in a distributed environment.

Figure 2 illustrates the high-level structure of the DEXSim framework that reflects these characteristics. The

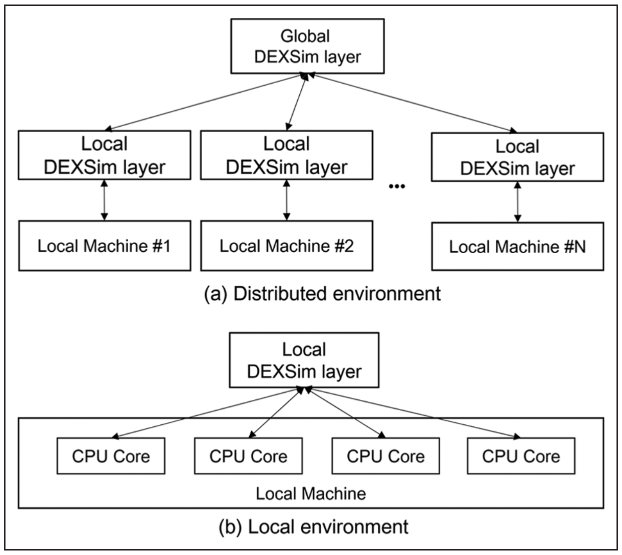


Figure 2. High-level structure of the DEXSim framework.

structure of the DEXSim framework consists of two layers: the global DEXSim layer and the local DEXSim layer. The global DEXSim layer in Figure 2(a) corresponds to a central simulation manager and the local DEXSim layer in Figure 2(b) corresponds to a single machine. In other words, the global DEXSim layer is responsible for the overall simulation execution environment, whereas the local DEXSim layer is in charge of a single machine. The global DEXSim layer manages three types of files: executable simulators, executable scenarios, and an overall scenario list. The scenario list contains a set of particular scenario sets, which combine an executable simulator and its scenario name. The global DEXSim layer properly assigns partial simulation jobs, that is, fragmentary scenario sets, to local DEXSim layers and arrays simulation results by utilizing the scenario list.

When the local DEXSim layer receives the assigned simulation jobs, it repartitions them to the CPU cores of the machine it controls and executes the simulation. For example, in terms of the simulation jobs' distribution, we assume that we have one specific simulator, sim_A, with 160 separate scenario cases to be carried out, and the given simulation execution environment contains four machines with quad-core processors. In this case, the global DEXSim layer can assign sim_A and 40 separate cases to every local DEXSim layer, and each local DEXSim layer distributes 10 separate cases to every CPU core in the machine. The share-out—the sets of 40 and 10 cases, in this example—comes from a simple arithmetical calculation. This may not be a problem because we provide a load-balancing technique at the local DEXSim layer, which we explain in detail in the following section. In addition to this characteristic, it is possible for the DEXSim framework to simulate various distinct simulators with numerous scenarios, as well as one simulator. Consequently, this hierarchical DEXSim structure provides efficiency for managing multiple executions of simulators by separating roles and assigning between the two-tiered DEXSim layers.

4.2. Model construction of the DEXSim framework

Based on the structural approach, we will describe the functional model construction of the DEXSim framework. Figure 3 illustrates an overall model construction. Here, we designate a global DEXSim layer and a local DEXSim layer as a federation DEXSim and a federate DEXSim, respectively. As we can see in Figure 3, the proposed DEXSim framework comprises one essential federation DEXSim and at least one federate DEXSim for transmitting messages to the federation DEXSim.

As illustrated in Figure 3, a federation DEXSim has three components: (1) a simulation manager; (2) a node manager; and (3) a synchronization manager. The simulation manager is in charge of generating simulation jobs, managing distributed experiments, and controlling simulation runs. Firstly, the simulation manager generates the set of simulation jobs, which are the pairs of the simulator and its scenarios. The set of simulation jobs is the simulation tasks, which are executed in the federate DEXSim. Secondly, the simulation manager properly assigns partial simulation jobs to one of the joined federate DEXSims. Finally, the simulation manager controls the execution of the simulator by forwarding a command for simulation execution, for example, simulation start or simulation stop, to every federate DEXSim and receives the response of the control messages. The node manager controls the conditions of all node machines (each node machine is controlled by a federate DEXSim, which we explain later). It receives computing situation information periodically, such as the number of node machines, the share occupancy ratios of their CPU cores and memory, and the status of the processes during execution. This is due to users manually calling to halt a certain simulator and restarting it, if necessary. Whenever the simulation execution is completed without a problem, the node manager receives the simulation result and arranges it. The synchronization manager performs the proper initialization and synchronization process of every federate DEXSim. In particular, the synchronization manager synchronizes simulation jobs, which are generated by the simulation manager, to every federate DEXSim. Hence, the synchronization manager broadcasts all execution files, such as the executable simulators, scenarios, and a partial scenario list, as mentioned in the previous section, to every machine. These files are stored in the particular local storage space of the machine and, consequently, any CPU core can access them.

Like the federation DEXSim, the federate DEXSim also has three functional components: a scenario manager; a

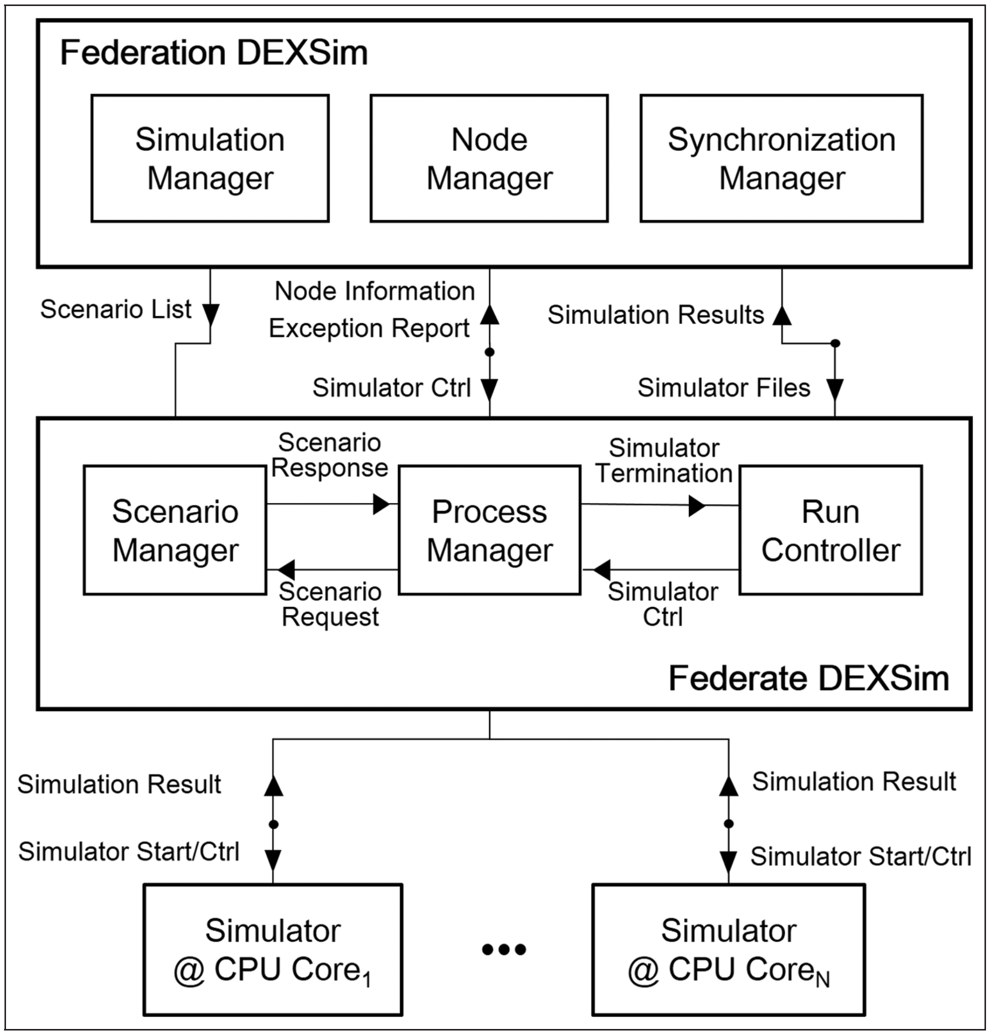


Figure 3. Descriptions and relationships between the DEXSim framework and simulator.

process manager; and a run-controller. The scenario manager handles a partial scenario list to be assigned by the synchronization manager. When the scenario manager receives a request for an executable scenario from the process manager, the scenario manager finds a relevant scenario name and transfers it to the process manager. For systematic management, the scenario manager classifies scenarios into a completion group or a preparation group. For the final stage, the process manager is closely involved in the simulation execution. It performs like a bridge between the machine's CPU cores and the federation DEXSim. When the process manager receives a control message for simulation start from the simulation manager, it chooses the simulator and requests a scenario name to be sent to the scenario manager. Because the execution files are simultaneously accessible to all CPU cores, the process manager actually assigns them to a distinct CPU core and executes simulation. We achieved the load-balancing technique, referred to in the previous section, with this physical resource access authority and appropriate message definitions. For example, we assumed that CPU core$_1$ in the machine completes the simulation execution, whereas CPU core$_2$ still performs the simulation execution. In this case, the process manager receives a next execution set from the scenario manager and assigns a simulation job to idle CPU core$_1$ instead of busy CPU core$_2$. After the simulation is completed, it sends the result to the run-controller to notify the simulation results to the federation DEXSim. The process manager repeats this process until all scenarios are finished. Since several simulation instances may occupy the resources of the local experimental environment infinitely, the run-controller continuously monitors the local experimental environment. The run-controller monitors the occupancy ratios of CPU cores and memory usages, and the status of the processes during execution, and sends them to the federation DEXSim periodically. The run-controller also monitors the execution result of a simulator, and reports the

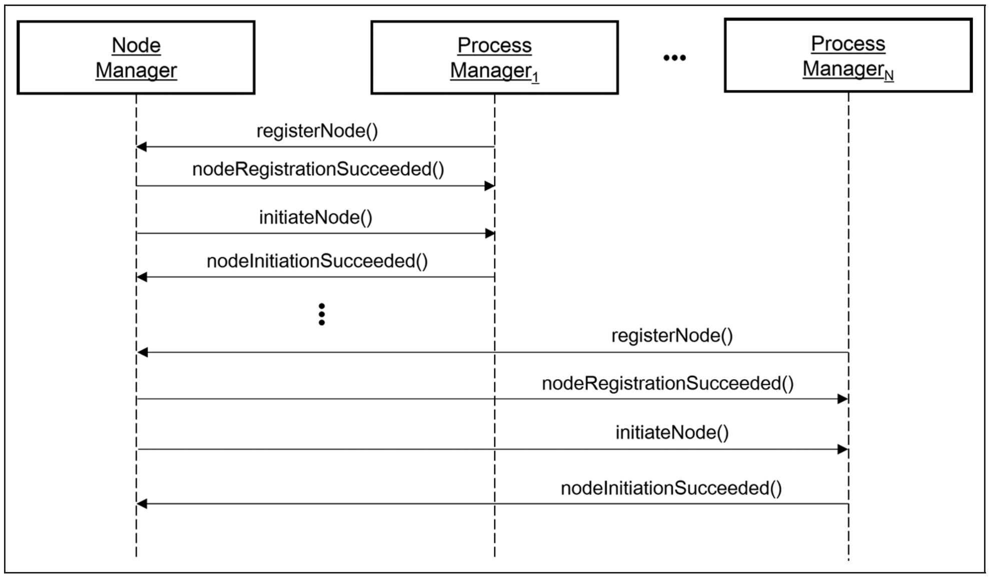


Figure 4. Messages for configuring the experimental environment.

simulation results to the federation DEXSim. In addition, when the federation DEXSim sends the execution control messages as responses of the monitoring reports, the run-controller takes an action, such as terminating the designated simulator or changing the process affinity for multiple CPU cores.

5. DEXSim framework: protocol design

The previous section focuses on the structural model design of the DEXSim framework. We have not yet described how the DEXSim framework connects to target simulators, executes them, and manages the experimental results according to time sequence. Thus, in this section, we move to detailed behaviors to interact with the DEXSim component models. In other words, the proposed DEXSim framework interacts with some specified messages among the component models. Hence, we define four kinds of protocols according to functional requirements, which associate exchanged information and thereby coordinate interacted activities.[43] At the end of this section, we additionally explain that the DEXSim protocols can handle abnormal situations during the experiments.

5.1. DEXSim protocol definition

The proposed DEXSim protocols perform both data transfer and control functions[43] among the DEXSim component models and the users. For data transfer, performance goals include delivering execution files of simulators and scenarios and transmitting simulation results. For control functions, reliability goals involve the proper initialization and synchronization on both sides of a connection, simulation execution, and simulation termination. This section describes our proposed protocol from these two standpoints. In this paper, we classify the proposed protocol into four separate protocols: (1) node management protocol; (2) synchronization management protocol; (3) process management protocol; and (4) scenario management protocol. In the following sections, we describe these four protocols with UML sequence diagrams to suggest a specification technique for the proposed protocols with both formal and intuitive semantics and a user-friendly graphical notation.[44]

5.1.1. Node management protocol. The node management protocol, which is responsible for the federation DEXSim, supervises all the information within the nodes. Thus, node management is one of the preconditions of simulation execution.

Figure 4 and 5 depict the protocols expressed as UML sequence diagrams for node management. As shown in Figure 4, the node management protocol takes on the roles of the registration and initiation processes for all joined federate DEXSim. In advance, a process manager sends a message, *registerNode()*, which contains the hardware resource information of the node, such as how many CPU cores the machine employs.

When the node manager receives the message properly, it sends a message, *nodeRegistrationSucceed()*, for registration completion, while it issues two kinds of identification (ID), for example, the machine ID and its CPU core ID, on the basis of the received message. With the distinct IDs, the node manager sends the message *initiateNode()* to the relevant process manager to notify its machine ID and core ID. When the process manager safely receives the

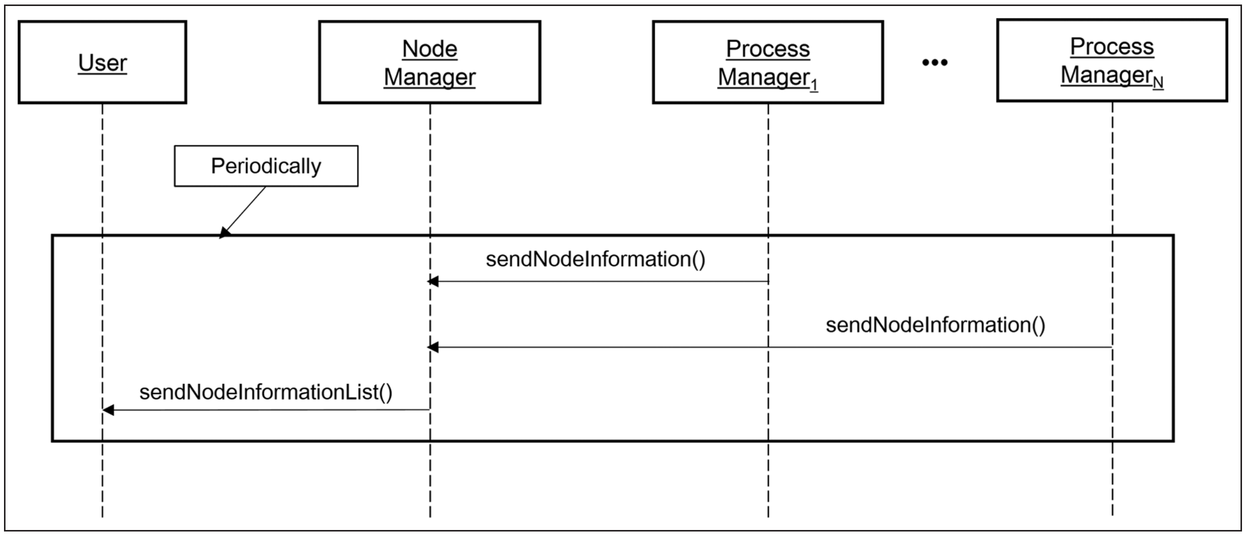


Figure 5. Node monitoring messages in the node management protocol.

message *initiateNode()*, it records the IDs and sends the message *nodeInitiateSucceeded()* to the node manager. This process is repeated for every federate DEXSim. In addition, the process manager periodically reports the status of the machine, such as the share ratios of its CPU cores and memory and current process status. The node manager, likewise, periodically informs users of synthesized information for all the joined machines, such as the number of joined federate DEXSims and their hardware resources status. In Figure 5, *sendNodeInformation()* and *sendNodeInformationList()* correspond to the aforementioned processes. These periodic reporting processes are for two primary reasons. Firstly, users should perceive whether the situation is ready for simulation execution. In our proposed method, a user can extend the experimental environment by adding additional resources. Therefore, the user can easily receive information about a machine status periodically, in order to check that the additional resources are ready to execute a simulator. Secondly, users can check the simulation status to see whether the simulator has a problem. These actions are taken in order to recover from abnormal situations during the experiments and will be discussed in the following section.

5.1.2. Synchronization management protocol. In the DEXSim framework, the federation DEXSim has all the execution files and input scenarios of a simulator. To execute multiple simulations concurrently by distributing them to the joined federate DEXSim, the federation DEXSim needs to replicate actual execution files and their scenarios as well as assign scenario lists to each federate DEXSim. A synchronization protocol carries out the task by transferring appropriate data to every federate DEXSim. Figure 6 illustrates interacting messages for synchronization management expressed as a UML sequence diagram.

When the node manager completes the registration and initialization processes, the synchronization manager synchronizes all information that is requested for simulation execution about every joined machine. Thus, the synchronization management is responsible for the following three tasks. Firstly, it generates a whole scenario list that contains all sets of an executable simulator and a scenario. Secondly, it divides the whole scenario list into several parts and assigns them to every joined federate DEXSim respectively. Finally, it delivers the execution files and their scenarios of the simulator, including a separate scenario list, to every federate DEXSim. Because the scenario list contains combinations of an executable simulator name and a corresponding scenario filename, the federate DEXSim can discriminate and pick a scenario that is suitable for the current experiment environment. Then, the federate DEXSim performs a simulation for the scenario for each scenario within the scenario pool. Accordingly, the content of the partial scenario list enables the execution of a single simulator with multiple scenarios. Thus, the proposed protocol definitions provide a basis for the SSMS concept. Returning to the protocol explanation, the synchronization manager conducts the first and second roles in advance, and the third role is conducted by sending a message, *synchronizationScenarioNode()*, to every federate DEXSim, namely the scenario manager. Upon receiving the message, each scenario manager saves three types of files and sends a message, *scenarioSynchronizationAchieved()*, to the synchronization manager. When the synchronization manager receives the message from all federate DEXSims, it sends a synchronization completion message, *allNodesSynchronized()*, to users. Clearly, the

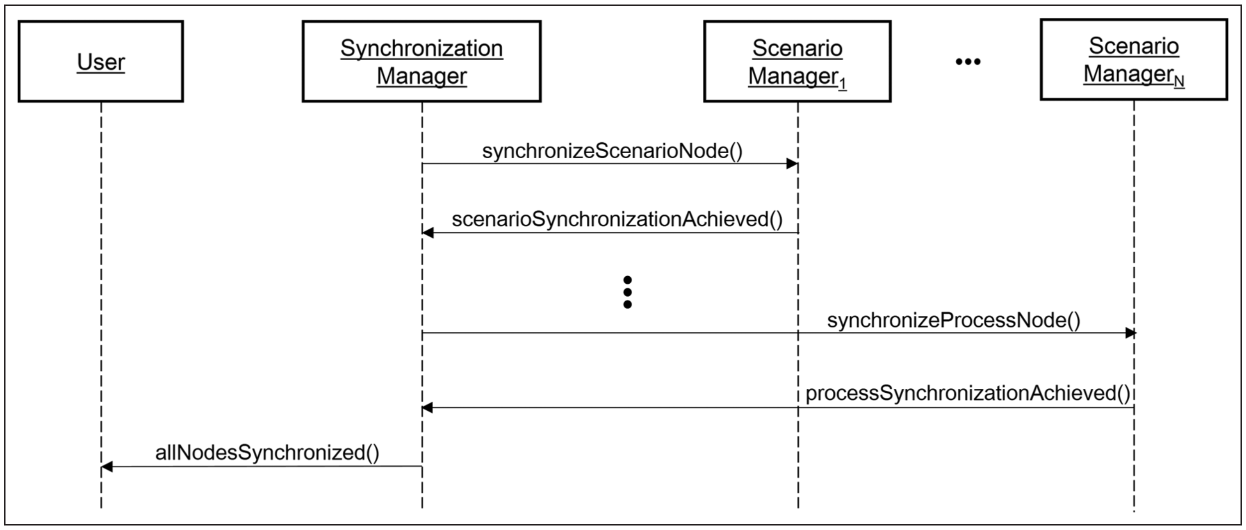


Figure 6. Interacting messages for synchronization management in the federation DEXSim.

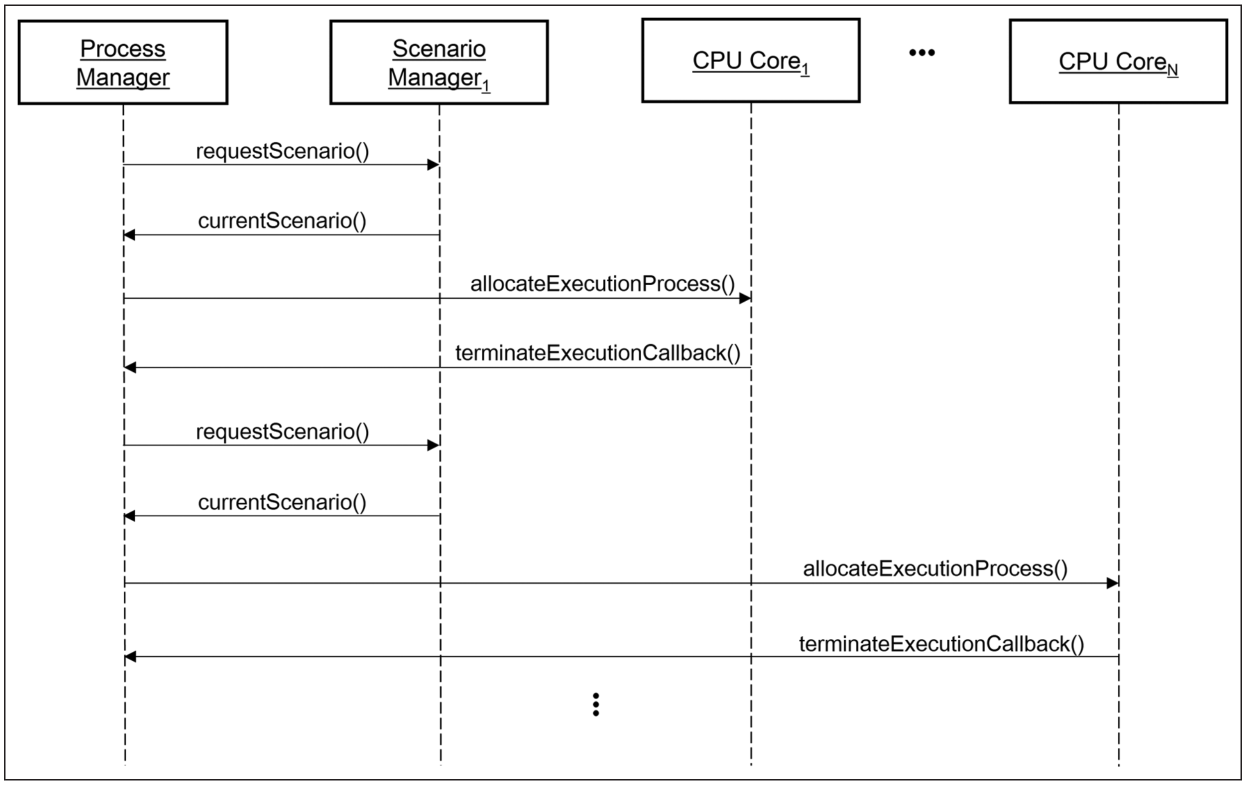


Figure 7. Interacting messages for scenario management in the federate DEXSim.

success of the node and synchronization management means that we are ready to execute multiple simulations concurrently with the proposed DEXSim framework.

5.1.3. Scenario management protocol. Node and synchronization management, as explained earlier, correspond to the establishment process before actual experiments. We will now describe two protocols, scenario and process management, regarding actual simulation execution. Figure 7 depicts message interaction for the scenario management protocol expressed as a UML sequence diagram.

Through the synchronization protocol, the scenario manager receives a partial scenario list, which should be executed in its own machine. The role of scenario management is similar to that of synchronization management. The difference between the two is that, while

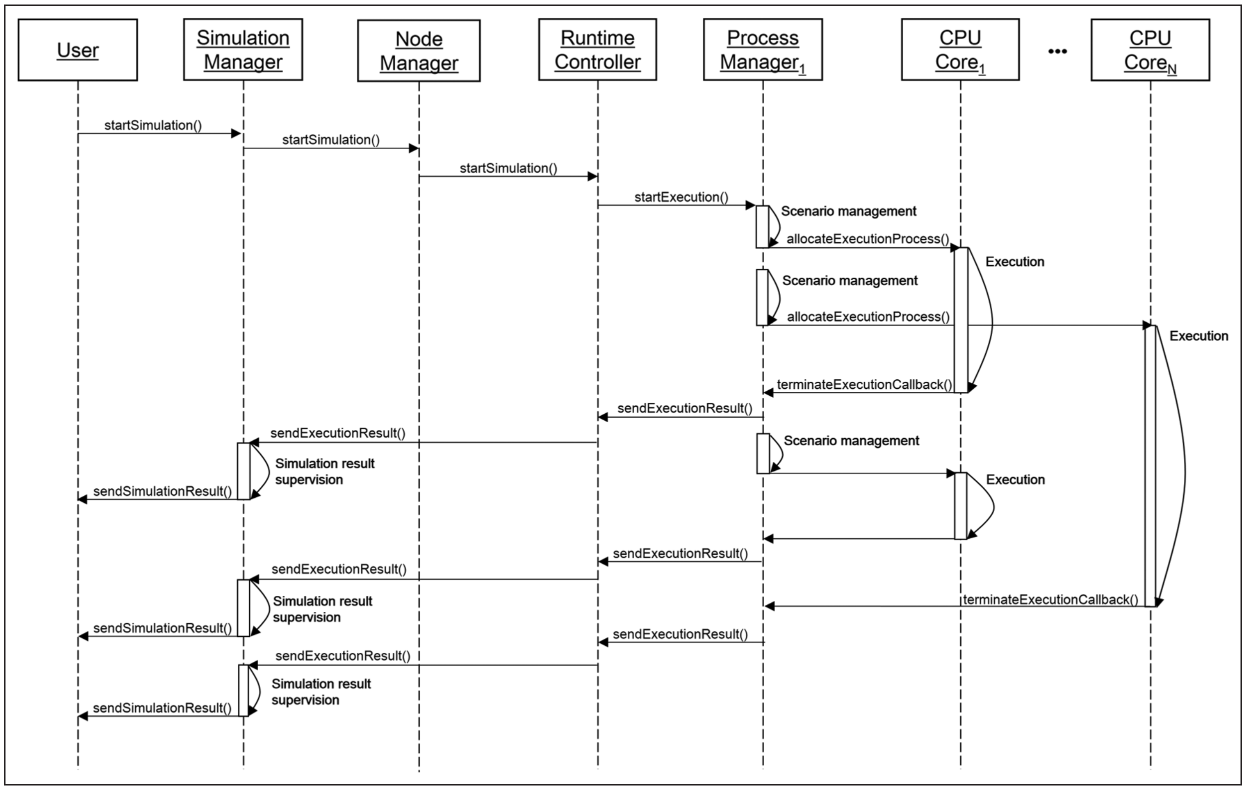


Figure 8. Interacting messages for process management in the federate DEXSim.

synchronization management distributes partial scenario lists out of the whole list, scenario management does not distribute them but rather processes one scenario at a time according to the order of requests from the process manager. Let us explain this situation with the sequence diagram in Figure 7. Firstly, the process manager sends a message, *requestScenario()*, to request a scenario for simulation execution. The *requestScenario()* message includes a set about the previous executed simulator and its scenario if it exists. If it is the first time for the process manager to request a scenario, the message *requestScenario()* contains a null message. After receiving *requestScenario()*, the scenario manager checks the scenario pool, chooses a new scenario, which additionally includes the number of iterations, and sends a message including the scenario, *currentScenario()*, to the process manager. If the simulator completes iterative simulation executions for the relevant scenario (we will explain this process when we address the process management protocol), the process manager requests a new scenario to be sent to the scenario manager.

5.1.4. Process management protocol. The process management protocol plays a direct role in executing and terminating simulations. While simulation commonly terminates automatically without any problem, manual termination is necessary when problems occur. Therefore, we describe process management by differentiating between the two cases, but we shall first explain the normal situation illustrated in Figure 8.

As we have seen in the previous sections, users are ready to execute simulation if they safely receive *sendNodeInformationList()* and *allNodeSynchronized()*. When users send a start message, *startSimulation()*, to a simulation manager, the simulation manager also broadcasts a message for simulation execution, *startExecution()*, to every process manager. After the process managers receive the message, each process manager brings up information for an executable simulator and its scenario using the scenario management protocol. Then the process manager utilizes the following load-balancing scheme to achieve load-balanced resource utilization for the experiment environment.

Algorithm 1. Load-balancing Scheme for federate DEXSim

```
Input: Set of usable CPU cores CPU_IDs; set of simulation tasks SimulationTasks
Output: Set of simulation tasks and CPU core pairs AllocationMap
1: procedure AllocateSimulationTask(CPU_IDs, SimulationTasks)
2: if(SimulationTasks = Ø)
3:     return null;
4: else
5:     AllocationMap ← Ø;
5:     for Each SimulationTask, st do
6:         for Each CPU_ID, cpu_id do
7:             Insert (cpu_id, st) into AllocationMap;
8:         end for
9:     end for
10:    Remove allocated simulation task from SimulationTasks
11:    return AllocationMap;
12: end if
```

As shown in Algorithm 1, the algorithm maps the given simulation tasks to the available CPU cores. When the process manager receives the allocation information from the *AllocateSimulationTask* method, it sends an execution message, *allocateExecutionProcess()*, that contains the information. Figure 8 illustrates the simulator allocation process mentioned above. In Figure 8, after process manager$_1$ receives the *startExecution()* message, it brings up information for an executable simulator and its scenario using the scenario management protocol. Afterwards, process manager$_1$ finds available CPU cores and sends execution messages, *allocateExecutionProcess()*, that contain the information. When the ordered simulation execution is complete, the process manager receives the *terminateExecutionCallback()* message as a completion message.

The received simulation result through *terminateExecutionCallback()* is transferred to users gradually using the messages *sendExecutionResult()* and *sendSimulationResult()*, respectively. By controlling multiple CPU cores simultaneously, process manager$_1$ can perform multiple execution processes at the same time from core$_1$ to core$_N$. Thus, process manager$_1$ allocates the next simulation execution to any idle core based on Algorithm 1. Consequently, the proposed protocols and physical file accessibility, as suggested above, are powerful advantages for guaranteeing load-balanced utilization.

From the users' perspective, when they send an execution message, *startSimulation()*, to the simulation manager, the simulation execution proceeds automatically in the DEXSim framework, and users just need to wait while all the simulations finish and the results are checked without their involvement. Thus, only a single simulation command enables the execution of multiple scenarios, that is, the SSMS process. Although this automatic process is another advantage of our DEXSim framework, we should consider how the proposed DEXSim framework handles exceptional conditions, such as abnormal behavior of the simulator.

As explained earlier, the DEXSim framework guarantees to recover abnormal execution situations, such as abnormal termination of a simulator or abnormal execution of a simulator. Since the DEXSim framework does not give any limitations to the simulators, such simulators may terminate abnormally, or they may cause the framework to stall. Generally, abnormal termination of a simulator may happen when the simulators have defects, such as divide by zero errors. Due to theses defects, the simulation result may be unreliable so that not all experiment results can be trusted. The upper part of Figure 9 shows how the

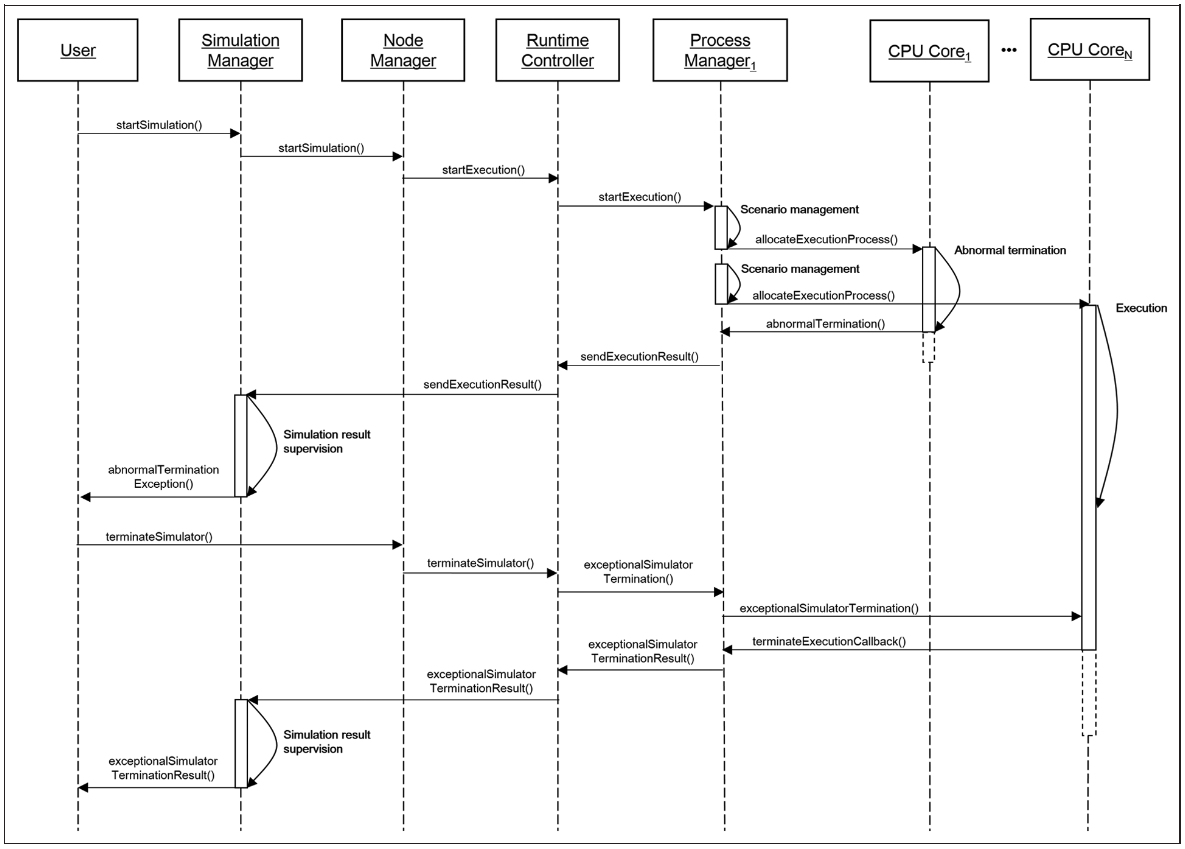

User
Simulation Manager
Node Manager
Runtime Controller
Process Manager1
CPU Core1
CPU CoreN
startSimulation()
startSimulation()
startExecution()
startExecution()
Scenario management
allocateExecutionProcess()
Abnormal termination
Scenario management
allocateExecutionProcess()
Execution
abnormalTermination()
sendExecutionResult()
sendExecutionResult()
Simulation result supervision
abnormalTermination Exception()
terminateSimulator()
terminateSimulator()
exceptionalSimulator Termination()
exceptionalSimulatorTermination()
terminateExecutionCallback()
exceptionalSimulator TerminationResult()
exceptionalSimulator TerminationResult()
Simulation result supervision
exceptionalSimulator TerminationResult()

Figure 9. Process management protocol for recovery from abnormal simulator status.

DEXSim framework handles abnormal termination of the simulator. When the simulator terminates abnormally, the process manager detects the termination and sends its simulation results to the run-controller. Then, the run-controller reports the execution result to the simulation manager. The simulation manager supervises the execution status and handles the exceptional status, for example, reports to the user or suspends entire simulation runs.

Stalling of the DEXSim framework is another abnormal execution situation. Such situations may come from abnormal behaviors of the simulator, for example, the actual execution time exceeds the abnormal execution criteria given by the simulationist and still occupies the CPU core without any simulation results. In the worst-case scenario, an unreliable simulator occupies all CPU cores in the DEXSim framework so that the simulator eventually leads to an abnormal core occupancy status in the framework. This issue deserves consideration because it is not possible to verify all input scenarios and simulators; therefore, unexpected behaviors can occur at any time, especially in the experimental stage.

The lower part of Figure 9 shows how our DEXSim framework processes this problem. Figure 9 represents simulation situations analogous to Figure 8, except for the activities of $core_N$. In Figure 9, job_1 is allocated to $core_N$, and it carries out the simulation execution, job_1. While $core_N$ conducts the simulation execution, users monitor how well the simulation is proceeding using the node management protocol. Unfortunately, abnormal execution may be very likely when job_1 shows no signs of completion. Various factors for abnormal execution can be found in $core_N$, such as problems of hardware resources or the assigned simulator. To overcome abnormal execution, the process management protocol provides an additional recovery message set, *exceptionalSimulatorTermination()* and *exceptionalSimulatorTerminationResult()*. In Figure 9, the run-controller bypasses the recovery messages to the process manager, and the process manager sends a halt event to job_1 in $core_N$ and reallocates the reassigned job_2 to $core_N$. Therefore, this message set provides intellectual and efficient utilization of the DEXSim framework to recover from unnatural simulation executions. In the following section, we will describe the abnormal occupancy recovery mechanism and verify it using the finite state machine (FSM).

5.2. Abnormal occupancy recovery mechanism

In order to overcome the stalling situations in the DEXSim framework, the framework provides two mechanisms: (1) monitoring all CPU cores' conditions and (2) users' intervention in the unexpected termination of a simulator. As we have seen, the former is attained using the node management protocol, while the latter is accomplished by the process manager protocol. In this section, we show how the DEXSim protocols recover from stalling situations using the FSM. The FSM-based protocol verification is used to verify various protocols[45,46] to show that the protocols do not contain a deadlock behavior. However, in this paper, we focus on the recovery technique in the case of an abnormal occupancy of the CPU cores, not the verification method, to maintain deadlock-free situations at all times. Consequently, our aim for recovering from abnormal occupancy status using the FSM is to show that there exists a transition, which breaks the abnormal occupancy status in every running CPU core. To this end, first, we conduct a simple CPU core model using the FSM, as illustrated in Figure 10. In Figure 10, we show three CPU core models that behave depending on different situations, namely normal, abnormal, and recovering abnormal occupancy situations. Each CPU core model has two states, *Idle* and *Occupy*, represented by an ellipse and solid lines, respectively, in the model that denote the behavior of the simulator, and a dotted line that denotes the exceptional message from the federation DEXSim.

As Figure 10(a) illustrates, when a process manager allocates a CPU core for a reliable simulator, the CPU

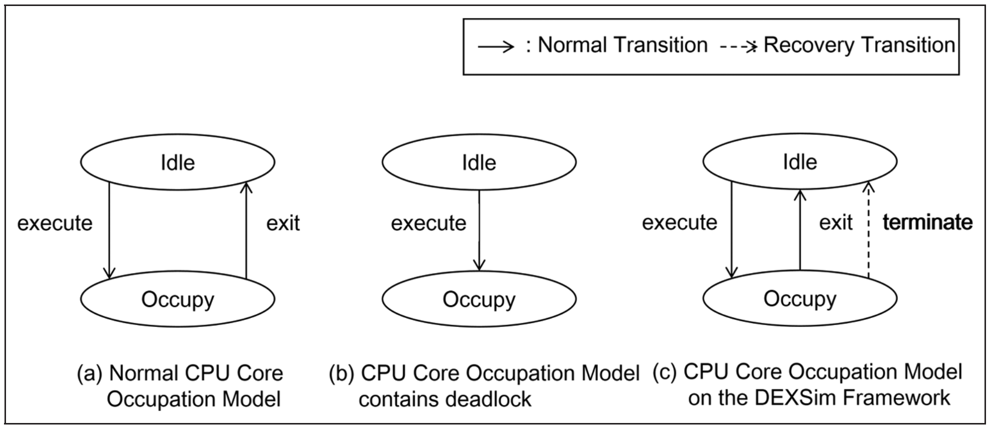

→ : Normal Transition --→ Recovery Transition
Idle
execute
exit
Occupy
(a) Normal CPU Core Occupation Model
Idle
execute
Occupy
(b) CPU Core Occupation Model contains deadlock
Idle
execute
exit
terminate
Occupy
(c) CPU Core Occupation Model on the DEXSim Framework

Figure 10. Representation of central processing unit (CPU) core model in three situations.

core model accomplishes an *execute* transition by changing the state from *Idle* to *Occupy*. Because a normal simulation is carried out easily and finished after a while, the model can be turned into the *Idle* state through an *exit* transition. In contrast, an abnormal simulator cannot terminate itself, which means that it is impossible for the model to conduct any *exit* transitions. Therefore, the model is unable to change its state indefinitely, and then the DEXSim framework can be said to be in an abnormal occupancy status (see Figure 10(b)). After an abnormal occupancy has occurred, however, it can be corrected by using several techniques.[46] In this paper, we employ a process termination method in order to abort all processes involved in the abnormal occupancy manually, and we offer a *termination* transition, as illustrated in Figure 10(c). Accordingly, the DEXSim framework allows the CPU core model to escape the abnormal occupancy state, *Occupy*, using the *exit* transition in normal situations or the added *termination* transition in abnormal situations. The proposed process termination is feasible because the federate DEXSim and its simulator are separate processes and are executed independently.

Now, let us consider an extended case of the DEXSim framework that contains two CPU cores. As illustrated in Figure 11, we can assume that the CPU core model has four states, ($Idle_1$, $Idel_2$), ($Occupy_1$, $Idle_2$), ($Idle_2$, $Occupy_2$), and ($Occupy_1$, $Occupy_2$), which represent combinations of the two cores' status. In Figure 11, we define termination transitions to recover from abnormal occupancy. Because the DEXSim framework is extendable simply by joining a federate DEXSim, the CPU core model can be generalized including *N* units of cores. Analogously, we can say that there exist exceptional termination transitions, which are triggered by the federation DEXSim, in every state of the CPU core model.

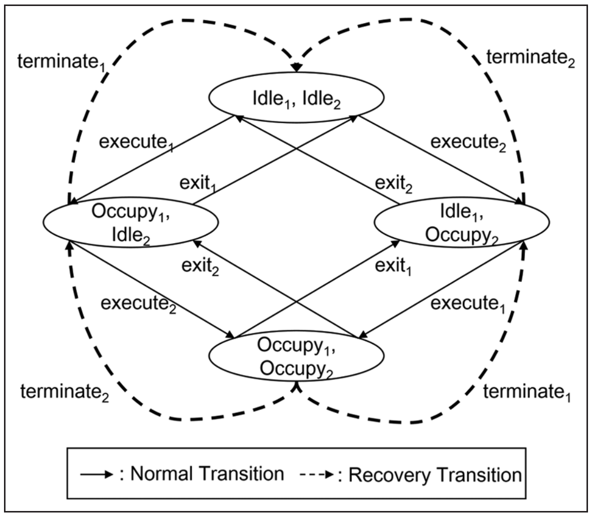


Figure 11. Central processing unit core model including two cores.

6. DEXSim framework: implementation

Now that we have theoretical descriptions of the proposed DEXSim framework, we may better understand how to implement the proposed DEXSim framework. In this section, we will present an implementation of an instance of our proposed framework for the Windows operating system environment. We implemented two types of messages and components of the DEXSim framework, using runtime infrastructure (RTI). RTI, the fundamental component of HLA, provides a set of software services to coordinate operations and data exchange during a runtime execution. In HLA/RTI, the federate denotes as an HLA-compliant entity, and federation denotes multiple entities connected via the RTI using a shared data model. In this research, we have adopted the HLA/RTI as the infrastructure of the DEXSim framework. Firstly, we have utilized the federation management (FM) service for federate DEXSim management. In the HLA/RTI, FM service defines how federates create, join, and resign the federation. In the DEXSim framework, we consider the federation and federate DEXSim as federate in the HLA/RTI, which interoperate with each other using the aforementioned messages. Therefore, a user can easily append an additional computing resource by executing federate DEXSim on the computer. Then, the federate DEXSim may join the experiment environment and receive the necessity data for the experiment.

Secondly, we have utilized declaration management (DM) and object management (OM) services to control and distribute necessary data to each federate. Since the federation DEXSim controls the experiment and distributed necessity data for the experiment centrally, the federation DEXSim publishes the process control message and necessity data, and the federate DEXSim subscribes the messages using DM services. The federation DEXSim utilizes the *object* of the OM services to distribute the simulator, scenario, and scenario list files. Therefore, when an additional computing resource joins the DEXSim framework, necessity data can be synchronized by OM services. Moreover, since the federate DEXSim subscribes the process control messages, the federation DEXSim can control the local execution environment through federate DEXSim by sending process control message as *interaction*.

Finally, we use the time management (TM) services for abnormal situation-handling schemes for the DEXSim framework. In the HLA/RTI, each federate manages the local time and coordinates data exchange with other members of a federation based on the local time. In order to implement the abnormal situation-handling scheme for the DEXSim framework, the federation DEXSim does not proceed its local time until it receives each federate DEXSim experiment status. When the federation DEXSim receives the experiment status of the federate DEXSim, the federation DEXSim decides on the abnormal terminations or

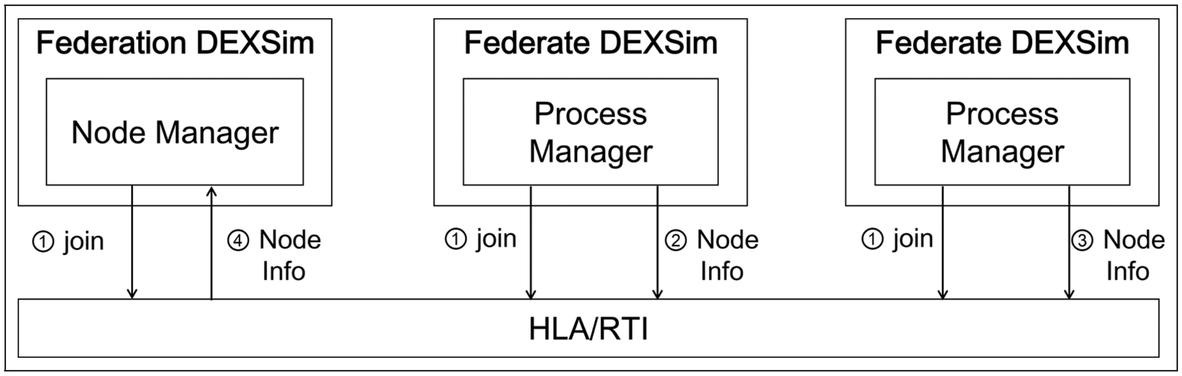


Figure 12. Implementation of node management services in the DEXSim framework.

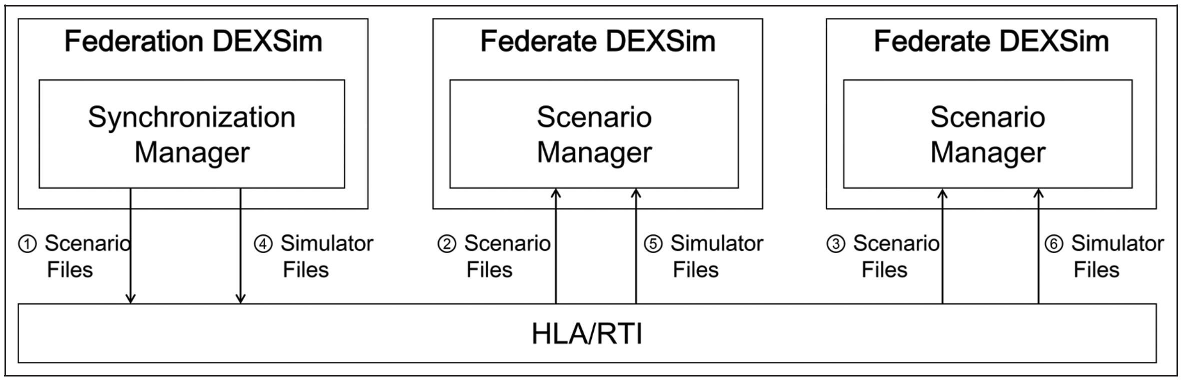


Figure 13. Implementation of synchronization management services in the DEXSim framework.

abnormal occupancy of replicated simulators under certain conditions, which are given by the user.

6.1. Federation DEXSim implementation: node and synchronization management

The requirements of the proposed framework are to manage the distributed and parallel experimental environment simultaneously. In the distributed environment, federation and multiple federate DEXSims have their own local memory, and they exchange messages with other components of the DEXSim if necessary. For federation DEXSim implementation, two types of messages, process control and data transfer, are utilized between the federation and multiple federate DEXSim. As we suggested in previous sections, messages for process control include registration of multiple federate DEXSims or notification of node information, etc., whereas data transfer messages contain execution files such as simulators, scenarios, and scenario lists.

Figures 12 and 13 represent an implementation of node and synchronization management. Every federation and federate DEXSim is implemented as an independent process, and they all join the DEXSim framework using the HLA/RTI. Therefore, any simulationists can implement their own components to interoperate with other federation and federate DEXSim based on the aforementioned protocols.

6.2. Federate DEXSim implementation: scenario and process management

As was the case of the federation DEXSim implementation, the federate DEXSim implementation also is based on the HLA/RTI. The messages about simulation controls and scenario results in Figure 14 are transferred though the HLA/RTI. There is one other point that claims our attention, which is how the process manager allocates a particular simulator to the machine's CPU core. To resolve this situation, we use a system call method, which is the process to request action by the operating system. In Figure 14, the process manager allocates the execution process, which is the simulator, to every CPU core. This is possible because each federate DEXSim and simulator is a separate program and is executed independently. Accordingly, the federate DEXSim utilizes the Windows-specific system call function to start the simulator on the operating system.

Figure 14 shows the normal simulation situation for the DEXSim framework without any problems. We also

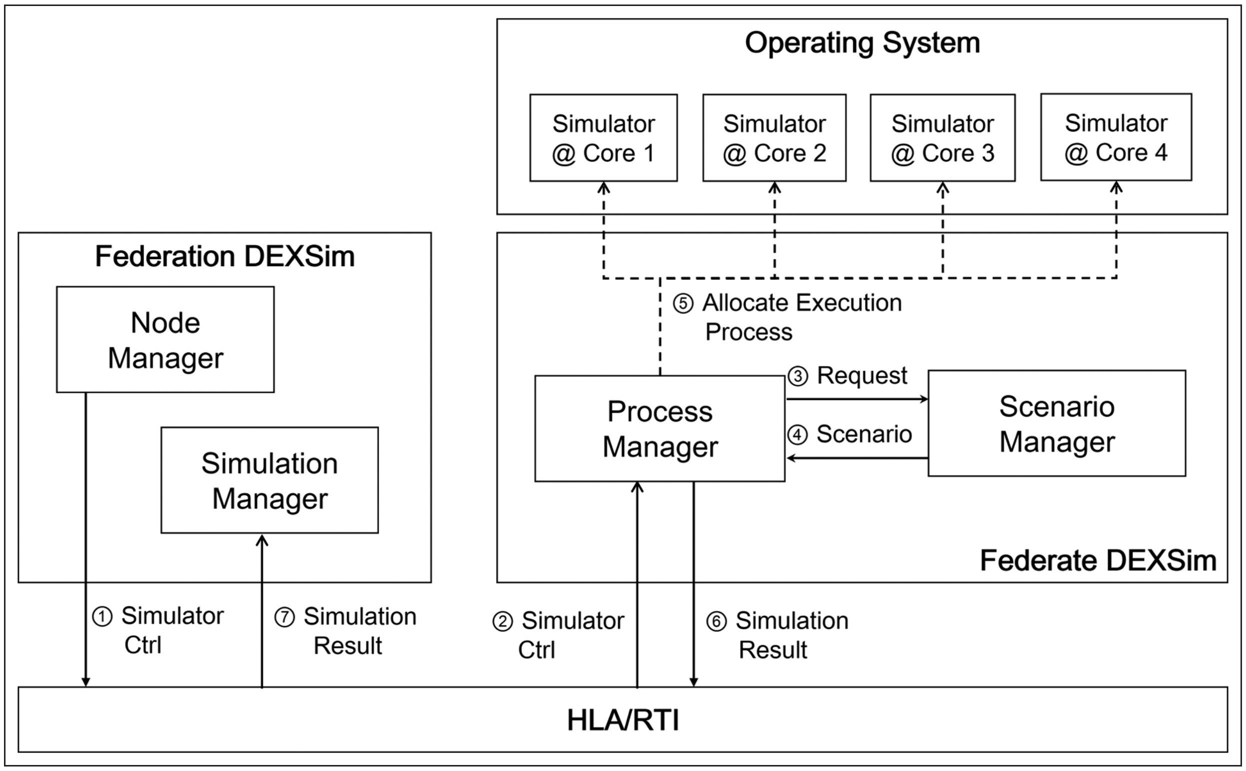


Figure 14. Instance of federate DEXSim implementation: scenario and process management.

propose a method to recover from abnormal occupancy, which is shown in Figure 15. A specific exceptional transition, a *terminate transition*, which we described in the previous section and is easily implemented using a Windows command, is called a kill process.

In order to identify the abnormal occupancy situation, the DEXSim framework utilized TM services in the HLA standard. During the simulation experiment, the federation DEXSim maintains time-out clocks for each federate DEXSim and monitors the simulation results. The time-out clocks are used to check the simulation progress of simulators in each federate DEXSim. Before staring the simulation, the simulationists set up the deadline in the time-out clock based on the types of simulators. If the simulation process does not generate simulation results until the given time-out period, the DEXSim framework terminates the process and notifies the users. For abnormal termination situations, the framework monitors the simulation results to check for the abnormal termination of the simulation process.

7. Case study

In the preceding sections, we described our DEXSim framework design and implementation. This section presents the experimental results of three target simulators using DEXSim implementation. The experimental results provided in this section were all obtained on a homogeneous cluster of 16 machines. Each machine within the cluster had dual quad-core Intel i7-2600 CPUs running at 3.40 GHz and 8 GB of DRAM, and running on Windows 7. The machines within the cluster were connected to a gigabit Ethernet switch with two trunked gigabit ports per machine. We will first briefly describe the three target simulators, and then display the DEXSim framework during multiple simulations, and compare and finally analyze the experimental results.

7.1. Target simulator description

Target simulators were applied to military analysis and acquisition missions of the Korean military.[47] Among three experiments, two used simulators designed by simulation models, whereas the other used a simulator represented by a complicated and stochastic analytic model.

For the first experiment, we used the naval air defense simulator, which was developed to validate an anti-air defense doctrine on the sea. We applied the discrete event system formalism to make a simulator that can reflect the anti-air defense doctrine as well. For the second experiment, we utilized the naval underwater warfare simulator, which was developed to acquire the performance indices of underwater weapon systems. Detailed information on these two simulators is provided by Kim et al.[3]

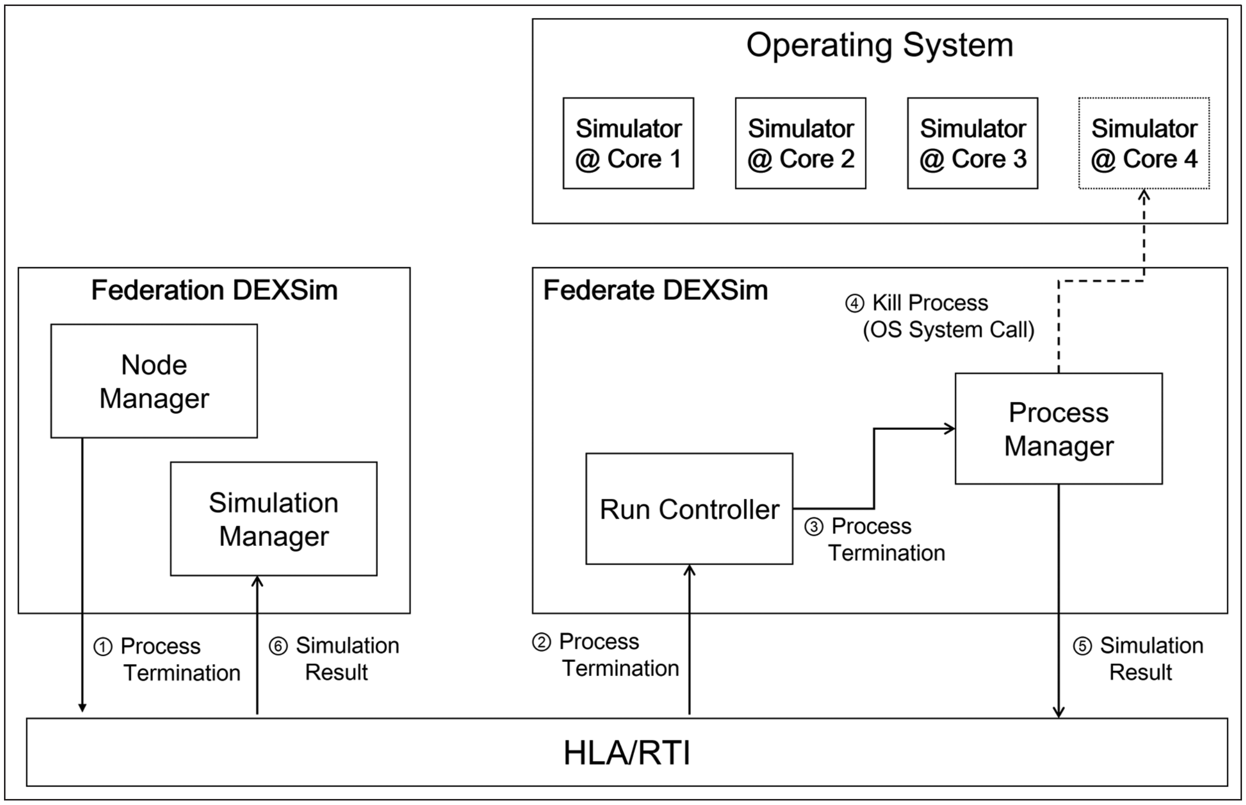


Figure 15. Instance of abnormal occupancy recovery method in DEXSim implementation.

Finally, we used the analytical model in use in the Korea Combat Training Center (KCTC).[48] The KCTC has been constructed to conduct exercises, combat training, and field experiments in a virtual training environment. Every soldier wears multiple integrated laser engagement system (MILES) gear, which uses lasers, sensors, and blank cartridges to simulate an actual battle. During training exercises at the KCTC, troops are engaged in virtual battles equipped with laser shooters and detectors, and the central control system receives detailed portrayals of battle scenes and evaluates them.[49] The KCTC simulator was developed on the theoretical basis of Driels.[50] The three experiments require a large number of experiment sets and significant computational time, as illustrated in Table 2. In the following section, we explain the experimental results in detail.

7.2. Experiment results

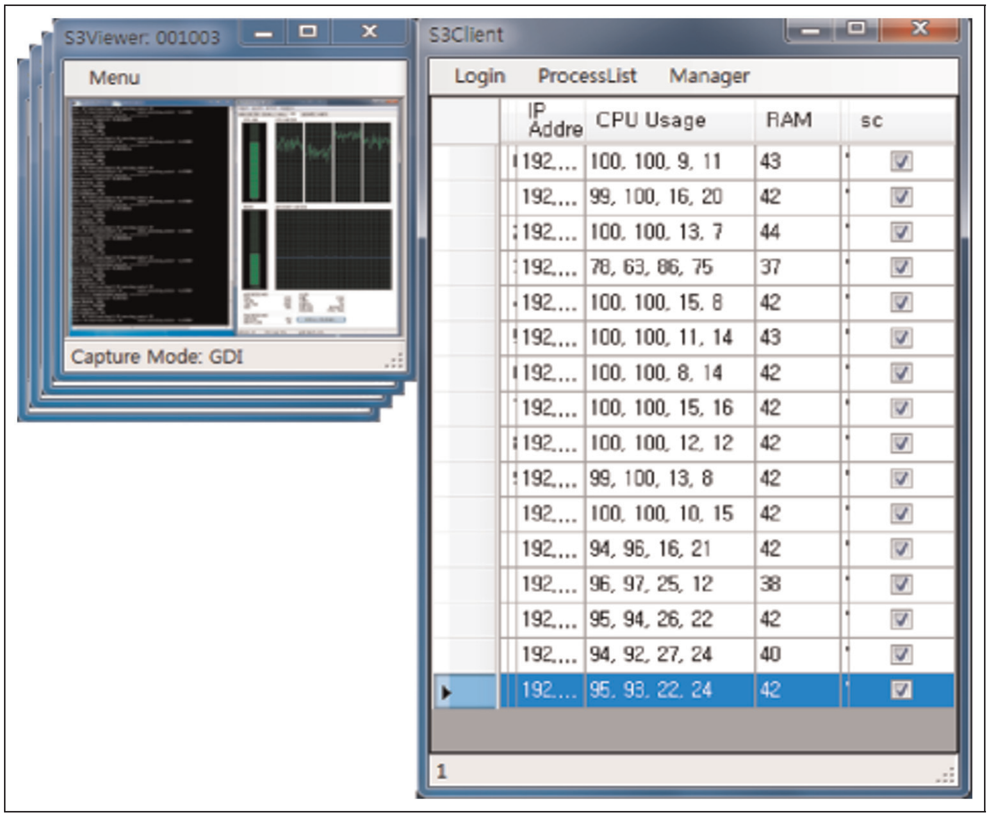
We describe our experimental results in two ways. The first step is to explain the execution situation of multiple simulations using the DEXSim framework, which shows how the DEXSim framework controls the simulations. Secondly, we arrange the scenario cases and their results into a table that shows how much total execution time for overall simulation is accelerated due to our DEXSim framework.

7.2.1. Simulation using the DEXSim framework. Figure 16 shows the simulation monitoring of all joined multiple federate DEXSims. The left-hand side of Figure 16 shows the overlapping screens of all the federate DEXSims and the right-hand side shows their current CPU and RAM occupancy ratios. The node management protocol transfers this information to the node manager, and the node manager displays it. During multiple simulations, the users check the situation of the current simulation, that is, how much the simulator occupies the CPU core or for how long the simulator runs, and determine whether the simulations are proceeding normally or abnormally. The check boxes on the right-hand side of Figure 16 show that the multiple federate DEXSims are well conducted in the relevant machine. Therefore, the users can check the simulation situation at run-time simulation, and can take an action if necessary (e.g., when the behavior of the particular simulator is unreliable).

7.2.2. Execution time measurement. This section analyzes how much the proposed DEXSim framework can accelerate simulations. To do this, we construct a table that represents scenario cases and their elapsed time for simulation. As described in Table 2, various simulation models were used in the first and second simulations. Therefore, the elapsed time for a single simulation is a bit longer than the third simulator. Nevertheless, the third experiment, the

Table 2. Design and results of three experiments.

Experimental index	Simulator 1	Simulator 2	Simulator 3	Implications
Varied parameter cases	$3^4 + 3^3 = 108$	$5 \times 4^2 \times 2 = 160$	$5^4 \times 4^4 = 40{,}000$	Total variation of cases for full-scale experiment
Total cases	$(3^4 + 3^3) \times 30 = 3{,}240$	$160 \times 100 = 16{,}000$	$40{,}000 \times 30 = 120{,}000$	No. of replication per case (Sim_1, Sim_2, Sim_3): (30, 100, 100)
Elapsed time per an experiment (sec)	60	90	40	The average time for one simulation
Elapsed time with existing environment (hour)	$3{,}240 \times 60 \div 3{,}600 = 54$	$16{,}000 \times 90 \div 3{,}600 = 400$	$120{,}000 \times 40 \div 3{,}600 = 1{,}333$	The result of the classical simulation environment: one standalone PC using one CPU core
Elapsed time with proposed DEXSim framework (hour)	0.982	7.143	24.24	Theoretical improvement: 64 times faster Practical improvement: 55 times faster due to overhead, such as network latency or task switching cost



IP Addre	CPU Usage	RAM	sc
192....	100, 100, 9, 11	43	✓
192....	99, 100, 16, 20	42	✓
192....	100, 100, 13, 7	44	✓
192....	78, 63, 86, 75	37	✓
192....	100, 100, 15, 8	42	✓
192....	100, 100, 11, 14	43	✓
192....	100, 100, 8, 14	42	✓
192....	100, 100, 15, 16	42	✓
192....	100, 100, 12, 12	42	✓
192....	99, 100, 13, 8	42	✓
192....	100, 100, 10, 15	42	✓
192....	94, 96, 16, 21	42	✓
192....	96, 97, 25, 12	38	✓
192....	95, 94, 26, 22	42	✓
192....	94, 92, 27, 24	40	✓
192....	95, 93, 22, 24	42	✓

1

Figure 16. Simulation monitoring of all joined multiple federate DEXSims.

KCTC simulator, had the most numerous scenario cases among the three experiments. Consequently, it took time to execute simulations.

As described previously, we performed a full simulation using 16 target PCs with quad-core processors for the scaling study. From Table 2, we can identify an important finding. When we used the proposed DEXSim framework, expected performance improvement is about 64 times, ideally. However, we improved the simulation execution time by about 55 times compared to the classic simulation environment that used a single PC with a CPU core. The number accounts for overheads, such as network latency or task-switching costs. Particularly in order to solve the problem of the KCTC, the simulation had to be accomplished in one day instead of over the course of 55 days. In summary, the experimental results show that the total elapsed simulation time is reduced in proportion to the hardware resources, which means that the extra overhead of the DEXSim framework is much less than that of the simulation execution. Among the experiments, two used simulators designed by simulation models consisting of a set of rules and timed-state transitions, whereas the other used a simulator represented by a complicated and stochastic analytic model. Therefore, the experiments show that the DEXSim framework enables the application of any simulator regardless of the simulator type.

8. Conclusion

Due to the increasing necessity of simulation-based experiments and improvements in hardware performance, distributed simulation techniques fully utilizing the given hardware resources are becoming essential in the modeling and simulation field. To satisfy these demands, this paper has attempted to describe an advanced simulation technique, the DEXSim framework, to utilize multiple distributed hardware resources for faster data collection. For the theoretical starting point, we propose the SSMS concept for the DEXSim framework, which is based on taxonomy of the computer architecture. To sum up, the following kinds of methods for the DEXSim framework were introduced in this paper: (1) a structural model design with two hierarchies; (2) component identification of two hierarchical models according to functional classification; and (3) IEEE 1516 standard based protocols. The hierarchical DEXSim structure provides efficiency by separating roles between the two-tiered DEXSim layers, and providing scalability by containing multiple federate DEXSim. In addition, component models and their interacting protocols in the DEXSim framework bring highly efficient load balancing in the local experimental environment and additionally offer an intellectual utilization of the framework to recover from unnatural simulation executions. The DEXSim framework also hides the technical details of multi-core programming from general users, allowing them to benefit from increased computing capacity with minimal knowledge of the execution environment. Finally, the DEXSim framework is implemented on the HLA/RTI, so that the simulationist can easily extend and scale up the experiment environment. For the scaling study, we performed a full simulation using 16 target PCs, and the experimental results showed that the total elapsed simulation time was reduced in proportion to the hardware resources, which means that the extra overhead of the DEXSim framework is much less than that of the simulation execution.

For further research, there are several considerations to improve the current DEXSim framework. In particular, we assume that experiments are executed on homogenous computing resources. However, the execution time of simulators with heterogeneous computing resources may differ from each other. As a result, the load balancing between federate DEXSims may be considered at the federation DEXSim level. Moreover, research to support various simulation architectures in the DEXSim framework is a notable topic for the further study.

The successful execution of this study will offer an immediate application for most complicated, large-scale, and stochastic simulators, and the technical cooperation with existing software-based methods will result in a considerable synergy effect for faster data collection. Furthermore, we expect that this work will provide guidance for decisions in the modeling and simulation fields.

Funding

This work was supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract UD110006MD, Korea.

References

1. Rubinstein RY and Melaned B. *Modern simulation and modeling*. New York, NY, USA: Wiley, 1998.
2. Piplani LK, Mercer JG and Roop RO. *System acquisition manager's guide for the use of models and simulation*. Fort Belvoir, VA, USA: Defense Systems Management College Press, 1994.
3. Kim JH, Choi CB and Kim TG. Battle experiments of naval air defense with discrete event system-based mission-level modeling and simulations. *J Defens Model Simulat* 2011; 8: 173–187.
4. Kim JH, Moon IC and Kim TG. New insight into doctrine via simulation interoperation of heterogeneous levels of models in battle experimentation. *Simulation* 2012; 88: 649–667.
5. Seo KM, et al. Measurement of effectiveness for an anti-torpedo combat system using a discrete event systems specification-based underwater warfare simulator. *J Defens Model Simulat* 2011; 8: 157–171.
6. Higgins TM, Turriff AE and Patrone DM. Simulation-based undersea warfare assessment. *Johns Hopkins Tech Digest* 2002; 23: 396–402.

7. Fowler JW and Rose O. Grand challenges in modeling and simulation of complex manufacturing systems. *Simulation* 2004; 80: 469–476.
8. Yang F, Ankenman B and Lelson BL. Efficient generation of cycle time-throughput curves through simulation based and metamodeling. *Nav Res Logist* 2007; 54: 78–93.
9. McGeoch C. Analyzing algorithms by simulation: variance reduction techniques and simulation speedups. *ACM Comput Surv* 1992; 24: 195–212.
10. Gepner P and Kowalik MF. Multi-core processors: new way to achieve high system performance. In: *proceedings of the PAR ELEC 2006, international symposium*, 2006.
11. Rybacki S, Himmelspach J and Uhrmacher AM. Experiments with single core, multi-core, and GPU based computation of cellular automata. In: *proceedings of the advances in system simulation*, 2009.
12. Liu Q and Wainer G. Multicore acceleration of discrete event system specification systems. *Simulation* 2012; 88: 801–831.
13. Kutanoglu E and Sabuncuoglu I. Experimental investigation of iterative simulation-based scheduling in a dynamic and stochastic job shop. *J Manuf Syst* 2001; 20: 264–279.
14. Himmelspach J, Ewald R and Uhrmacher AM. A flexible and scalable experimentation layer. In: *proceedings of the 40th conference on winter simulation*, 2008.
15. Miller JE, et al. Graphite: a distributed parallel simulator for multicores. In: *proceedings of the 2010 IEEE 16th international symposium on high performance computer architecture (HPCA)*, 2010.
16. Zhou SP, et al. Flexible state update mechanism for large-scale distributed wargame simulations. *Simulation* 2007; 83: 707–719.
17. Taylor SJE, et al. Distributed computing and modeling and simulation: speeding up simulations and creating large models. In: *proceedings of the winter simulation conference*, 2011, pp.161–175.
18. Rak M, Cuomo A and Villano U. mJADES: Concurrent simulation in the cloud. In: *the proceedings of sixth international conference on complex, intelligent and software intensive systems (CISIS)*, 2012.
19. Van Vliet H. *Software engineering: principles and practice*. Chichester, UK: Wiley, 2006.
20. IEEE 1278.1 and 1278.1A:1995. IEEE standard for distributed interactive simulation-application protocols (set).
21. IEEE 1516:2000. IEEE standard for modeling and simulation (M&S) high level architecture (HLA) – framework and rules.
22. Miles R and Hamilton K. *Learning UML 2.0*. Sebastopol, CA, USA: O'Reilly, 2006.
23. Hord RM. *Parallel supercomputing in SIMD architectures*. Boca Raton, FL, USA: CRC Press, 1990.
24. Heidelberger P. Statistical analysis of parallel simulations. In: *proceedings of winter simulation conference*, 1983, pp.290–295.
25. Marr C, et al. A Java-based simulation manager for web-based simulation. In: *proceedings of the 32nd winter simulation conference*, 2000, pp.1815–1822.
26. Globus alliance. GRAM – Globus, http://dev.globus.org/wiki/GRAM (accessed August 2013).
27. HT Condor. HTCondor Home, http://research.cs.wisc.edu/htcondor/ (accessed August 2013).
28. Oracle. Oracle Grid Engine, http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html (accessed August 2013).
29. Berkeley. Boinc, http://boinc.berkeley.edu/ (accessed August 2013).
30. Digipede. Grid computing for Windows: distributed computing: cluster computing: grid computing for .NET, http://www.digipede.net (accessed August 2013).
31. Gehlsen B and Page B. A framework for distributed simulation optimization. In: *proceedings of the winter simulation conference*, 2001, pp.508–514.
32. Page E, et al. Goal-directed grid-enabled computing for legacy simulations. In: *proceedings of the cluster, cloud and grid computing, ACM/IEEE international symposium*, 2012, pp.873–879.
33. Globus Alliance. Gridway home, http://dev.globus.org/wiki/GridWay (accessed August 2013).
34. Bononi L, et al. Concurrent replication of parallel and distributed simulations. In: *proceedings of the workshop on principles of advanced and distributed simulation*, 2005, pp.234–243.
35. Bononi L, et al. ARTIS: a parallel and distributed simulation middleware for performance evaluation. In: *proceedings of the 19th international symposium on computer and information sciences (ISCIS 2004)*, 2004, pp.627–637.
36. Kite S and Taylor SJE. SakerGrid: simulation experimentation using grid enabled simulation software. In: *proceedings of the winter simulation conference*, 2011, pp.2278–2288.
37. Cuomo A, Rak M and Villano U. Process-oriented discrete-event simulation in Java with continuations: quantitative performance evaluation. In: *proceedings of international conference on simulation and modeling methodologies, technologies and applications (SIMULTECH)*, Rome, 2012, pp.87–96.
38. Petcu D, Craciun C and Rak M. Towards a cross platform cloud API - components for cloud federation. In: *proceedings of the international conference on cloud computing and services science*, 2011.
39. Wang D, et al. DRAMsim: a memory system simulator. *Comput Architect News* 2005; 33: 100–107.
40. MacGougan G, et al. Performance analysis of a stand-alone high-sensitivity receiver. *GPS Solutions* 2002; 6: 179–195.
41. Brase JM and Brown DL. Modeling, simulation and analysis of complex networked systems: a program plan. Lawrence Livermore National Laboratory, LLNL-TR-4112733, 2009. http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Complex_networked_systems_program_final.pdf.
42. Butler KL, Ehsani M and Kamath P. A Matlab-based modeling and simulation package for electric and hybrid electric vehicle design. *IEEE Trans Veh Tech* 1999; 48: 1770–1778.
43. Sunshine CA. Survey of protocol definition and verification techniques. *Comput Network (1976)* 1978; 2: 346–350.
44. Odell JH, Parunak VD and Bauer B. Representing agent interaction protocols in UML. In: *proceedings of agent-oriented software engineering*, 2001.
45. Yuang MC. Survey of protocol verification techniques based on finite state machine models. In: *proceedings of the computer networking symposium*, 1988.

46. Ling Y, Chen S and Chiang CYJ. On optimal deadlock detection scheduling. *IEEE Trans Comput* 2006; 55: 1178–1187.
47. Kim TG, et al. DEVSim++ toolset for defense modeling and simulation and interoperation. *J Defens Model Simulat* 2011; 8: 129–142.
48. Department of the Army, Korea Combat Training Center, http://www.kctc.mil.kr/main.html (accessed December 2012).
49. Best BJ, Lebiere C and Scarpinatto KC. Modeling synthetic opponents in MOUT training simulations using the ACT-R cognitive architecture. In: *proceedings of the 11th conference on computer generated forces and behavior representation*, 2002.
50. Driels MR. *Weaponeering conventional weapon system effectiveness*. Reston, VA, USA: AIAA Education Series, 2004.

Author biographies

Changbeom Choi received his BS in computer engineering from Kyung Hee University and his MS in computer science from the Korea Advanced Institute of Science and Technology (KAIST) in 2005 and 2007, respectively. He is currently a PhD candidate at the Department of Electrical Engineering at KAIST. His research interests include discrete event systems modeling/simulation, verification, validation, and accreditation (VV&A), and agent-based simulation.

Kyung-Min Seo received his BS degree in electrical engineering from Pusan National University and his MS degree in electrical engineering and computer science from KAIST in 2006 and 2008, respectively. Currently, he is a PhD candidate at the System Modeling Simulation Laboratory (SMSLAB) at KAIST. His research interests include underwater warfare modeling and simulation, methodologies for the modeling and simulation of hybrid systems, and simulation-based experiment.

Tag Gon Kim received his PhD in computer engineering with specialization in systems modeling and simulation from the University of Arizona, Tucson, Arizona, 1988. He was an Assistant Professor of Electrical and Computer Engineering, University of Kansas, Lawrence, Kansas, from 1989 to 1991. He joined the Electrical Engineering Department, KAIST, Daejeon, Korea in fall, 1991, and has been a Full Professor at the EECS Department since fall, 1998. He was the President of The Korea Society for Simulation (KSS). He was the Editor-In-Chief for *Simulation: Transactions for Society for Computer Modeling and Simulation International* (SCS). He is a co-author of the text book, *Theory of Modeling and Simulation*, Academic Press, 2000. He has published about 200 papers on modeling and simulation theory and practice in international journals and conference proceedings. He is very active in defense modeling and simulation in Korea. He was/is a consultant for defense modeling and simulation technology at various Korea government organizations, including the Ministry of Defense, Defense Agency for Technology and Quality (DTAQ), Korea Institute for Defense Analysis (KIDA), and Agency for Defense Development (ADD). He is a Fellow of the SCS and a Senior Member of the IEEE.