Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
1–22
© 2014 The Author(s)
DOI: 10.1177/0037549714553151
sim.sagepub.com
SAGE SCS

Simulation integration: using multidatabase systems concepts

Leila Jalali, Sharad Mehrotra and Nalini Venkatasubramanian

Abstract
This paper considers the challenge of designing a framework that supports the integration of multiple existing simulation models into an integrated simulation environment (multisimulation). We aim to facilitate the process of fusing together the independently created simulators into an integrated simulation environment wherein we can model and execute complex scenarios involving multiple simulators. In this paper, we focus on solutions for the synchronization problem in multisimulation to orchestrate consistent information flow through multiple simulators. In particular, we provide: (1) a transaction-based approach to modeling the synchronization problem in multisimulations by mapping it to a problem similar to multidatabase concurrency; we express multisimulation synchronization as a scheduling problem where the goal is to generate "correct schedules" for time advancement and data exchange across simulators that meets the dependencies without loss of concurrency; (2) a hybrid scheduling strategy which adapts itself to the "right" level of pessimism/optimism based on the state of the execution and underlying dependencies. We also develop two key optimizations: (a) efficient checkpointing/rollback techniques; and (b) relaxation model for dependencies which guarantees bounded violation of consistency to support higher levels of concurrency. We evaluate our proposed techniques via a detailed case study from the emergency response domain by integrating three disparate simulators: a fire simulator (CFAST), an evacuation simulator (Drillsim) and a communication simulator (LTEsim).

Keywords
Simulation integration, synchronization, dependency, consistency

1. Introduction

In this paper, we consider the challenge of designing a middleware framework to integrate multiple pre-existing simulation models. Simulation models are typically developed by domain experts who have an in-depth understanding of the phenomena being modeled and are designed to be executed and evaluated independently. A grand challenge is to facilitate the process of fusing together the independently created simulators into an integrated simulation environment wherein we can model and execute complex scenarios involving multiple simulators. In this paper, we refer to such an integrated simulation environment as a *multisimulation*.

To motivate the need for multisimulations, let us consider the domain of emergency response. Modeling and simulation are very useful in emergency response domain where response plans and methods are validated by simulating disasters and their impact on people and infrastructure. A variety of simulators, e.g. loss estimation tools (HAZUS,[1] INLET,[2] CAPARS,[3] CFD[4]), fire-spread simulators (CFAST,[5] CCFM[6]), activity simulators (Drillsim,[7] SDSS[8]), transportation simulators (VISSIM,[9] PARAMICS[10]), etc., that model different aspects of disasters, their impacts, and response processes have been developed. While these simulators are individually important in understanding disasters, their integrated and concurrent execution can significantly enhance the understanding of the phenomena and dependencies between multiple aspects of the complex processes. Consider, for instance, a fire simulator, CFAST, that simulates the impact of fire and smoke in a specific region and calculates the evolving distribution of smoke. Since fire and smoke can affect health conditions of individuals in the region of fire, one may wish to further study its impact on the evacuation process as captured within an activity simulator, e.g. Drillsim. Similarly, the progress of fire (simulated by CFAST) may

Department of Computer Science, University of California, Irvine, USA

Corresponding author:
Leila Jalali, University of California, Irvine, 2060 DBH, Irvine, CA 92617, USA.
Email: jalalil@uci.edu

create infeasible paths/exits for evacuation (as simulated by Drillsim). Such what-if analyses can significantly improve the understanding of adverse impacts such as increase in evacuation times or increased exposure to undesired particulates enabling intelligent decision making to improve the response.

Integrated simulation environments are also useful in the healthcare domain where understanding complex health issues requires combining multiple simulation models including agriculture, transportation, distribution, climate, and social, economic, and physical systems in which people and food operate. It has been argued that combined simulation results in a more accurate representation of population health; this, in turn, can help devise meaningful policies for the future.[11]

The US Department of Defense (DoD) has promoted the development of standards to provide a common framework in which simulators can be integrated. These include standards such as SIMulator NETworking (SIMNET),[12] Distributed Interactive Simulation (DIS),[13] Aggregate Level Simulation Protocol (ALSP),[14] and High Level Architecture (HLA).[15] HLA, initially designed for military purposes, does not take the standards or technologies of other domains into account adequately.[16] While HLA is suited to developing new multidomain simulations, its broader applicability to combine pre-existing simulators is questionable.[17] It is a complex standard and is not transparent enough: too much low-level knowledge is needed from the practitioner. HLA forces developers to provide a particular functionality or to conform to specific standards in order to participate in the integration process; the rigid assumptions and limitations on participants makes it difficult to integrate pre-existing simulators without significant modification (especially in non-military domains). Other efforts exist that relate to simulation integration via HLA to provide run-time support, including HLA-based network simulations,[18] ModHel'X,[19] JAVAGPSS,[20] DEVSJAVA,[21] and PIOVRA.[22] Owing to the characteristics of programming-language-specific and platform-specific of RTI, the federate programmed by one language cannot work well with the federate programmed by another language.[23,24]

Web Services provide a loosely coupled mechanism for performing coarse-grained services with modest performance over both local-area networks (LAN) and wide-area networks (WAN). The web enabling HLA can combine the best of Web Services technologies and HLA mechanisms, bringing the above Web Services capabilities to HLA. There are two main development methods of the web-enabling HLA: web-enabled RTI and HLA evolved Web Services.[25-28] The goal of the web-enabled RTI is to enable simulators to communicate with an HLA RTI through web-based services. The long-term goal is to be able to have multiple simulators that are able to reside as Web Services on a WAN, enabling an end-user to compose a simulation from a browser. Previous research[29-31] introduced the Web Services-based HLA interface (API), and discussed three methods of integrating HLA into a web-based service: (a) using Web Services, (b) using Web Services to bridge an HLA system and an external system, and (c) using the HLA Evolved Web Service API. Researchers have proposed an approach that combines the idea of a service-oriented architecture (SOA) with HLA/RTI to solve problems associated with traditional RTI. Dragoicea et al.[32] proposed the integration of HLA and SOA into a Simulation Framework. The integration of HLA and SOA proposed by Wang and Zhang[33] to achieve better interoperability and reusability among heterogeneous simulation components in a distributed environment.

Many researchers have also realized the drawbacks of Web Services-based approaches. Turner et al.[34] revealed the defects of Web Service communication protocols in engineer applications. According to Byrne et al.[35] service-oriented HLA using Web Services are only appropriate for simulation applications that require coarse grain, low data update frequency, or non-real-time requests. Safety and stability are another drawbacks of simulation applications based on Web Services.[36]

In contrast to prior work on simulation integration,[13-15,37] in our architecture we do not need to integrate simulations tightly into a common framework, but we make it feasible to semi-automatically compose simulation models via a looser coupling approach. When we examine the nature of our challenges, we note properties of simulation integration which parallel certain concepts in multidatabase systems.[38,39] In multidatabase systems the individual database management systems need to be integrated in a single unified database environment while they desire to preserve the autonomy of the local database management systems. Using concepts from multidatabase systems, we model multisimulations as sequences of actions and capture dependencies across them. Our goal is to come up with sequences of actions from different simulators that can be executed with minimum synchronization overhead such that concurrent execution of the underlying simulators is consistent. We apply our techniques in a case study with fine-grained high data update frequency. In order to participate in the federation, wrappers are written for each simulator that enable us to intercept and control the execution of individual simulators in order to ensure effective and correct composite simulation models. Our goal is to preserve the autonomy of the individual simulators while accommodating the multiple time management and advancement mechanisms implemented internally by the participating simulators. The main contributions of this paper are as follows.

- A reflective middleware architecture for multisimulations based on structural reflection and metamodel concepts.

- A transaction-based approach to modeling the synchronization problem in multisimulations by mapping it to a problem similar to multidatabase concurrency.
- A novel hybrid scheduling strategy for multisimulation synchronization which adapts itself to the "right" level of pessimism/optimism based on the state of the execution and underlying dependencies.
- A relaxation model for dependencies across multiple simulators which guarantee bounded violation of consistency to support higher levels of concurrency.
- Applying proposed techniques in a detailed case study from the emergency response domain using multiple real-world simulators.

This paper is organized as follows. In Section 2, we discuss our multisimulation architecture and our methodology for simulation integration that supports the interoperability of multiple existing simulation models. In Section 3, we formalize the notion of multisimulation and present a transaction-based approach to modeling the synchronization problem. We propose a novel hybrid scheduling strategy and relaxation model for dependencies in Section 4. We describe system prototype implementation, a detailed case study, and evaluation in Section 5. Finally, Section 6 discusses related work and concluding remarks.

2. A reflective architecture for multisimulation

We address the challenges in multisimulations by developing a middleware to achieve integration of multiple simulators. A major advantage of developing a middleware for multisimulations is that it hides the details of the underlying independently created simulators and platform-specific issues. In particular, the reflective middleware architecture helps us in (i) observing specified changes in simulators and reflecting them back into other simulators, (ii) analyzing different aspects from simulators in different integration scenarios, and (iii) preserving the autonomy of individual simulators by separating concerns related to the simulation domain from those related to the integration mechanisms. Figure 1 illustrates our reflective middleware architecture for multisimulation. The base level consists of various pre-existing simulators that need to be integrated; each simulator is essentially a black box with its own internal data structures, models, and source code. The meta-level is built on base-level simulators; structural reification[40] of base-level entities yields data structures at the meta-level which capture relevant aspects of each simulator in a meta-model and provides support for data adaptations and exchange across simulators.[41,42] Simulators interface with the meta-level by using wrappers. In our architecture, simulators' data items are stored at base level simulators. The integration of various simulators results in

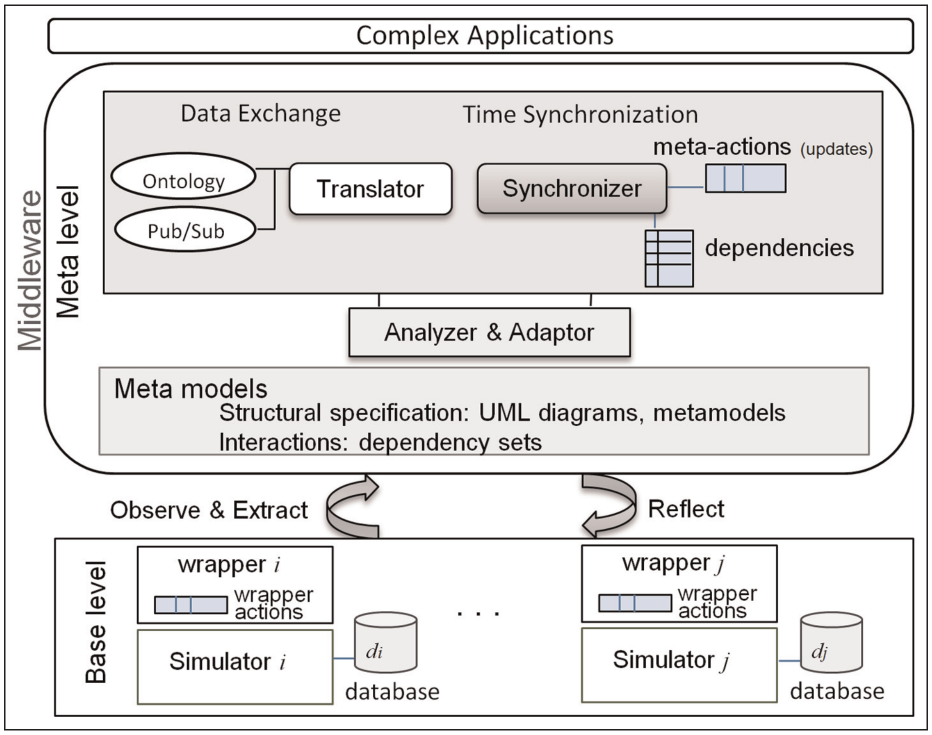


Figure 1. Multisimulation reflective architecture.

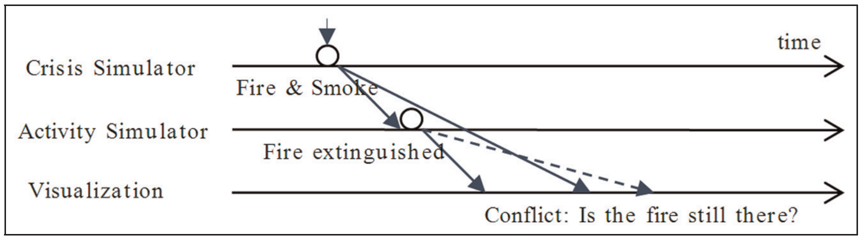


Figure 2. Causality violation due to lack of inter-simulator synchronization.

the dependencies between simulators. When there is a dependency between two data items, updates from one simulator need to be reflected into the other simulator. These updates are determined by the wrappers and sent to the meta-level. Upon receiving such updates from a simulator, the meta-level notifies dependent simulators.

In this paper, we focus on the synchronization problem in multisimulations. When integrating simulators, synchronization mechanisms are needed to provide causal correctness and ensure that the multiple simulators executing concurrently are proceeding in a coordinated manner as time progresses. In other words, it is essential to orchestrate consistent flow of information to multiple simulators in the concurrent execution. Figure 2 illustrates how consistency (causality) violations may occur in concurrently executing simulators without an effective time management system.[43] Consider a fire, modeled in CFAST, that communicates to other simulators, say, agents in activity simulators that are capable of extinguishing the fire. Without correct ordering of actions across simulators, the visualization interface might show that an agent extinguished the fire before the fire event started. Alternately, incorrect ordering might make it difficult for decision makers to determine whether the fire is ongoing or not. Such synchronization is complicated by the fact that the different simulators being integrated may represent time differently and implement varying time advancement mechanisms.

Time synchronization in distributed systems is a research area with a very long history. In general, the synchronization mechanisms can fall into three different categories: (1) conservative,[44] (2) optimistic,[45] and (3) scaled real time.[46] There are numerous variations of Time Warp that have been proposed that attempt to reduce the amount of rolled back computation that may occur.[47] Mainly, there are two classes of optimistic approaches: non-adaptive[48] and adaptive.[49] Non-adaptive schemes typically leave the specification of parameters to the user, or rely on heuristics to statically set their parameters at the beginning of the simulation. Adaptive schemes use a better potential of achieving speedups for varying workloads by changing the parameters ‘‘on-the-fly’’. Each of the standards provided by DoD has their own method for dealing with time synchronization.[50,51] In DIS there is no time coordination between individual simulators and all of the simulators run in real-time. Simulators that interoperate using ALSP must be discrete event simulators; here conservative time management services are employed to maintain event causality. The runtime infrastructure of HLA provides explicit *timeAdvanceRequest* and *timeAdvanceGrant* services to enable the management of a unified time across federates. The HLA time management services are strongly related to other services such as message exchange and attribute updates.[17] Using these services can be very complex for the integration of pre-existing simulators. Figure 3 demonstrates the basic steps of our methodology for simulation integration. The first step is to extract metadata from basic simulators and to describe it using metamodels. The next step analyzes metamodels to discover inter-dependencies between simulators. The first two steps are pre-processing steps that are human-in-loop process in which we would ask a user (for example, a data designer or administrator familiar with the simulators) to draw lines between elements that should contain related data in different simulators. Our methodology is described by Jalali et al.[42,52] The minimum requirement for the simulator is the ability to stop and to proceed the simulation. When multisimulation runs, meta-level modules ensure the correctness until the end of simulation.

3. Modeling simulators and multisimulations

In this section, we provide fundamental concepts used throughout the rest of the paper and formalize the multisimulation synchronization problem. When we examine the nature of our problem and challenges, we note properties of simulation integration which parallel certain concepts in multidatabase systems and transaction processing.[38,39] In multidatabase systems the individual database management systems need to be integrated in a single unified database environment while they desire to preserve the autonomy of the local database management systems. Interesting parallels also exist between the concepts of simulation execution synchronization and transaction serializability. Using concepts from multidatabase systems,

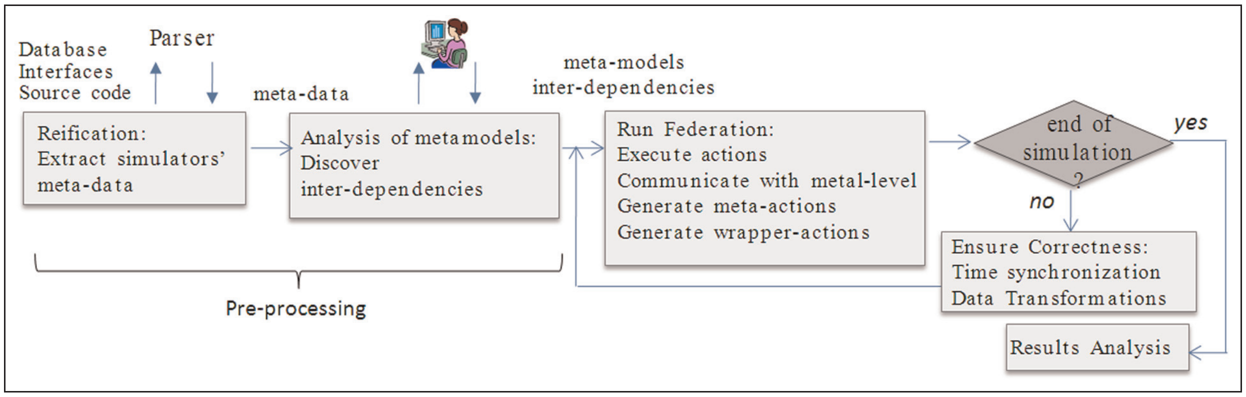


Figure 3. The basic steps of methodology.

we model multisimulations as sequences of actions and capture dependencies across them. Our goal is to come up with sequences of actions from different simulators that can be executed with minimum synchronization overhead such that concurrent execution of the underlying simulators in the multisimulation is consistent. We define relaxations (motivated by divergence control mechanisms[53]) that guarantee bounded violation of consistency to support high levels of concurrency in the multisimulation execution.

In a multisimulation, multiple simulators run concurrently. Each simulator's execution results in a sequence of actions which we refer to as a simulation. Simulation is the execution of a simulator's actions where there is a total order $<_i$ on the set of actions $A_i = \{a_k | k = 1, 2, \ldots\}$.

A *multisimulation* is the concurrent execution of independently created simulators, S_1, S_2,..., S_n that execute concurrently in an integrated environment. Before we develop the meta-simulator model and formally define the synchronization problem, we first need to define the concept of a simulator.

Simulator: Each simulator is modeled as a three tuple $S_i = < T_i, D_i, A_i >$, the components of which are defined as follows.

- T_i: the type of the simulator. We categorize simulators based on the time management mechanism that they employ as being time-stepped or event-based.[43] In time-stepped simulators, for each execution of the main control loop the simulation time is incremented by one quantum of time Δt. In the case of event-based simulators, execution is driven by an event list, $E = \{e_m | m = 1, 2, \ldots\}$, each event has a time stamp and the simulation time jumps from one event time stamp to the next without representing all of the values in between. When event e_a occurs before event e_b, we have the following property: $timestamp(e_a) \leqslant timestamp(e_b)$. In our notation, $T_i \in \{TS, EB\}$, where TS and EB correspond to time-stepped and event-based simulators respectively. We allow different simulators to have different levels of granularity in their event timestamps in event based simulators or time intervals in time stepped simulators.
- D_i : the set of entities associated with S_i that capture relevant aspects of each simulator as well as support for data adaptations and exchange across simulators. We refer to $d \in D_i$ as the data items associated with S_i. For each data item $d \in D_i$, $Dom(d)$ denotes the domain of d, which is a set of values that can be assigned to d
- A_i: the set of actions that is$A_i = \{a_k | k = 1, 2, \ldots\}$, where each action, a_k, of a simulator is an atomic unit of processing that reads and modifies data. For each action a_k, there is a set of data items, $a_k.Data \subseteq D_i$, associated that are either read or modified by the action. An action, a_k, captures changes that occur in a tick (or a step) in a time-stepped simulator or the execution of an event in an event-based simulator. For black box simulators, if we do not know which portion of data is associated with an action, we consider all data items maintained by the simulator as being associated with the action.

State of a simulator: The state, $\{\Phi_i\}_{ts}$ of the simulator S_i is the snapshot of its data items D_i and their values. A state maps every data item $d_i \in D_i$ to a value v_i, where $v_i \in Dom(d_i)$. Thus a state can be expressed as a set of ordered pairs of data items in D_i and their values, $\{\Phi_i\}_{ts} = \{(d_i, v_i) | d_i \in D_i, v_i \in Dom(d_i)\}$. Associated with each state is a timestamp, ts. Actions trigger state change; we use the notation $\{\Phi_i\}_{ts}$ a_k $\{\Phi_i\}_{ts'}$ to indicate that when an action a_k in simulator S_i executes from a state $\{\Phi_i\}_{ts}$, it results in a state$\{\Phi_i\}_{ts'}$. The timestamp of the state is changed after the execution of action. In a time-stepped simulator the timestamp advances by Δt, $\{\Phi_i\}_{ts}$ a_k $\{\Phi_i\}_{ts+\Delta t}$. In an event-based simulator it advances to the time stamp of the event e_m that is executed, $\{\Phi_i\}_{ts}$ a_k

$\{\Phi_i\}_{timestamp(e_m)}$, where $timestamp(e_m)$ is the timestamp associated with e_m.

3.1. Inter-simulator dependencies

The integration of various simulators results in the introduction of certain inter-simulator dependencies, which were not present prior to integration. Such dependencies enable us to observe only the portion of data items that are required for synchronization. An example of such a relationship is that smoke from a fire simulator can affect a person's health in an activity simulator. We use the notation $S_i \rightarrow S_j$ and $(d_i \in D_i) \rightarrow (d_j \in D_j)$ to express the dependency between data d_i in simulator S_i and d_j in simulator S_j, when updates on d_i need to be reflected into d_j. Note that the notion of dependency is directional.

3.1.1. Local data versus interdependent data. Dependencies enable us to partition the set of data items at a simulator, D_i, into local data items, LD_i, and interdependent data items, ID_i, such that $LD_i\, I\, D_i = \emptyset$, and $D_i = LD_i \cup ID_i$. Furthermore, if there is a dependency $(d_i \in D_i) \rightarrow (d_j \in D_j)$, $i \neq$j, then $d_i \in ID_i$ and $d_j \in ID_j$.

3.1.2. Internal actions versus external actions. Partitioning of data items at each simulator into local and interdependent data allows us to define the notion of internal and external actions. Internal actions, IA_i, are those actions that read or modify only the portion of data that is local to the simulator, $d_i \in LD_i$. External actions, EA_i, access at least one data item which is an interdependent data item, $d_i \in ID_j$. An action a_k is an external action if, $\exists d \in ID_i$ such that $d \in a_k.Data$, and a_k is an internal action otherwise. Synchronization is only needed during external action processing and we can eliminate it altogether during internal action processing periods.

3.1.3. Dependency set. For each simulator, S_j, we derive a dependency set $Dep(S_j)$ which is a set of simulators S_i where there exists a dependency $(d_i \in D_i) \rightarrow (d_j \in D_j)$ from S_i to S_j. That is: $Dep(S_j) = \{S_i | \exists (d_i \in D_i) \rightarrow (d_j \in D_j) s.t. d_i \in ID_i, d_j \in ID_j\}$.

We illustrate the above concepts through the examples taken from the activity simulator, Drillsim in our case study, when we integrate three pre-existing real-world simulators. Overall in Drillsim, we have 419 local data items that are internal to the simulator versus 45 interdependent data items for which there exists a dependency to/from other simulators. Out of 4220 actions of Drillsim executed, 1275 were external actions requiring synchronization with other simulators, and 2945 actions are internal actions that read or modify the portion of data that is local to Drillsim and do not require to be synchronized with other simulators' actions.

3.2. Simulators actions, meta-actions, and wrapper actions

Inter-simulator dependencies may cause causality violations where actions of different simulators interleaved arbitrarily. This results in the need for synchronization between multiple simulators to ensure causal correctness. We consider each simulator's execution as a sequence of actions (steps in time-stepped simulators or events in event-based simulators). To enable such synchronization in multisimulations, we extend each simulator with synchronization points where the simulator stops processing its actions and communicate with the meta-level in order to be synchronized with other simulators. We introduce a meta-level entity, *meta-synchronizer*, that coordinates the execution of actions from multiple simulators. In our synchronization protocol, we consider an action as *safe* to execute if the time in other dependent simulators is beyond the start time of the action. In other words, an action a_k from simulator S_i is safe if for all simulators in S_i's dependency set, $S_j \in Dep(S_i)$, we have $t_i < t_j$. When the meta-synchronizer encounters unsafe actions, it needs to delay the action to make sure the dependency will be preserved. Simulators interface with meta-synchronizer by using wrappers. The wrapper determines the external actions for which the simulator needs to communicate with meta-level and sends them to meta-synchronizer (reification). Upon receiving such actions from a simulator, the meta-level generates *meta-actions* and dependent simulators are notified through their wrappers. Wrappers translate the meta-actions into corresponding *wrapper actions* that are applied to the underlying simulators (reflection).

Figure 4 illustrates how synchronization works in our architecture. First, we initialize the meta-actions-queue, MAQ, the delayed actions queue, DAQ, and the list of simulators, SimList. MAQ is a queue of meta actions $< ma(d_i, d_j), ts >$ in which $ma(d_i, d_j)$ represents an update on d_i that needs to be reflected on d_j because of a dependency $(d_i \in D_i) \rightarrow (d_j \in D_j)$. Associated with each meta-action is the update time ts. DAQ is the queue of delayed actions, $< a_k, S_i, t_i, S_j >$, in which a_k is an action from S_i at time t_i that is delayed waiting for simulator S_j to proceed. The protocol is a conservative strategy that ensures the correctness of schedules by delaying the actions such that the dependencies are preserved in the concurrent execution of actions of simulators. For each external action, a_k, the simulator's wrapper sends a request, Request-Execute $< S_i, a_k >$ to the meta-synchronizer. The meta-synchronizer, on receiving of the request, may do any of the following actions:

```
Meta-synchronizer:
1: initialize the meta-actions-queue MAQ as a queue of $< ma(d_i, d_j), ts >$;
2: initialize the delayed-actions-queue DAQ as a queue of $< a_k, S_i, t_i, S_j >$;
3: initialize a list of simulators SimList as a list of $< S_i, t_i, Dep(S_i) >$;
4: Upon a request from a simulator $S_j$ to get the updates: Request-Updates$< S_j, t_j >$
5:    find and remove all meta-actions $< ma(d_i, d_j), ts >$ from MAQ  s.t. $d_j \in D_j$, $ts < t_j$
6:       send $< ma(d_i, d_j), ts >$ to $S_j$;
7:    update time of simulator $S_j$ to $t_j$ in SimList;
8:    find delayed actions $< a_k, S_i, t_i, S_j >$ from DAQ s.t.  $t_i < t_j$ waiting for $S_j$
9:                                  where $t_j$ is the time of $S_j$ extracted from SimList;
10:         allow($a_k$) and remove $< a_k, S_i, t_i, S_j >$ from DAQ;
11: Upon receiving a request to execute an external action: Request-Execute$< S_i, t_i, a_k >$
12:    if $a_k$ is safe  then  /* $< S_i, a_k >$ is safe for all $S_j \in Dep(S_i)$ if $t_i < t_j$ */
13:       allow($a_k$);
14:    else    /* $a_k$ is potentially unsafe $\exists S_j \in Dep(S_i)$ s.t.  $t_i > t_j$ */
15:       delay($a_k$);
16:       add $< a_k, S_i, t_i, S_j >$  to DAQ; /*$a_k$ is a pending action */
17: Upon receiving updates $< S_i, d_i, t_i' >$ from $S_i$: Update$< S_j, d_i, t_i' >$
18:       for all dependencies $(d_i \in D_i) \rightarrow (d_j \in D_j)$ do
19:             generate meta-actions $< ma(d_i, d_j), t_i' >$ and post it to MAQ;
Simulator's wrapper:
1:  while (simulation still in progress){
2:    for each (time step or event) do{
3:       $t_i$=current-time;
4:       send a Request-Updates $<S_i, t_i >$ to get meta-actions and apply updates using wrapper actions;
5:       action $a_k$= nextAction; /* in a time-stepped simulator $a_k$ is the next step,
                                    in an event-based it is the next event $e_m$*/
6:       if ($a_k$ is an internal action)
7:             proceed-simulator ($a_k$) and update simulation time to $t_i'$; /* allow the simulator to execute the internal action*/
8:       else /*$a_k$ is external action */
9:             send a Request-Execute $<a_k, t_i >$ to meta-synchronizer;  /*$t_i$=current-time*/
10:            wait until allow($a_k$) received from meta-synchronizer then
11:            proceed-simulator ($a_k$) and update simulation time to $t_i'$; /*execute the external action*/
12:             send Update$< S_i, d_i, t_i' >$ to meta-level for all interdependent data items $d_i \in ID_i$; }}
```

Figure 4. The protocol for synchronization in multisimulation.

- allow(a_k): a_k is granted to be safely executed.
- delay(a_k): a_k is delayed because its execution is not safe.

The delay will cause the simulator to freeze until the synchronizer allows it to proceed. Before the execution of an external action a_k, the simulator wrapper will ask meta-synchronizer for confirmation. If the action is allowed by meta-synchronizer, the action will be executed and the simulation time will proceed to the next action. For example, consider a simulator S_i in state $\{\Phi_i\}_{t_i}$ that performs an external action a_k which changes its state and updates an interdependent data item, d_i, to its new value, v_i', such that $(d_i, v_i') \in \{\Phi_i\}_{t_i'}$. Then S_i posts this update and the current state's timestamp, $< a_k, t_i' >$, to meta-level. Let $(d_i \in D_i) \rightarrow (d_j \in D_j)$ be a dependency. Upon receiving such an update, the meta-synchronizer generates a meta-action $< ma(d_i, d_j), t_i' >$ and posts it to MAQ. When S_j sends a request $< S_j, t_j >$ to receive the corresponding meta-actions, the meta-synchronizer determines the meta-actions $< ma(d_i, d_j), t_i' >$ that need to be reflected to S_j; that is, $d_j \in D_j$ and $t_i' \leqslant t_j$. Note that actions that correspond to the future state, i.e., $t_j < t_i'$, will not be reflected into S_j immediately and will be deferred until the simulator S_j advances to the time when the action is applicable. A meta-action $< ma(d_i, d_j), t_i >$ results in a wrapper action, $< wa(d_i, d_j), t_r >$, at the wrapper corresponding to the simulator S_j, to reflect the update of the meta-action into S_j. Associated with a wrapper action is a time, t_r, that is the timestamp of the state of S_j when the actual update is applied.

3.3. Schedules and correctness

In this section, we will define the notion of schedule and discuss the correctness of schedules in the presence of dependencies.

3.3.1. Schedule. Given a set of simulators $S = \{S_1, S_2, \ldots, S_n\}$, each simulator $S_i = < T_i, A_i, D_i\} >$, a set of wrapper actions of each simulator, $wa \in WA_i$, and a set of meta-actions, $ma \in MA$, a schedule, $\Psi_{Sch} = (Sch, <_{Sch})$ is an interleaved sequence of the external actions of each simulator (EA_i), meta-actions (MA), and wrapper actions (WA_i), $Sch = (\cup_{i=1}^{n} EA_i) \cup MA \cup (\cup_{i=1}^{n} WA_i)$ and $<_{Sch}$ is the ordering over Sch.

For example, consider the following two simulators $S_i = <TS, A_i, D_i>$ and $S_j = <EB, A_j, D_j>$: $S_i : S_i.a_1\ S_i.a_2$, and $S_j : S_j.a'_1 S_j.a'_2$. where $a_1, a_2 \in A_i$ and $a'_1, a'_2 \in A_j$ represent the actions of simulator S_i and S_j respectively. Consider a dependency $(d_i \in D_i) \rightarrow (d_j \in D_j)$, between d_i in simulator S_i and d_j in simulator S_j. Both simulators start at time 0. Assume $S_i.a_1$ and $S_i.a_2$ update data item, d_i, resulting in new states with time stamps 5 and 10. Here ma_1 and ma_2 are meta-actions corresponding to those updates, wa_1 and wa_2 are wrapper actions that reflect those updates to S_j, and $S_j.a'_1$ and $S_j.a'_2$ update d_j and result in states with timestamps 18 and 20. One schedule resulting from the concurrent execution of S_i and S_j can be $\Psi_{Sch1} : S_i.a_1 S_j.a'_1 S_j.a'_2 S_i.a_2$.

3.3.2. Well-formed schedule. A schedule, $\Psi_{Sch} = (Sch, <_{Sch})$, is a well-formed schedule (WFS) if for all external action a_k and dependencies $(d_i \in D_i) \rightarrow (d_j \in D_j)$ if $d_i \in a_k.Data$, then Sch contains:

- a meta-action $ma(d_i, d_j)$ corresponding to the action a_k and the dependency $(d_i \in D_i) \rightarrow (d_j \in D_j)$;
- a wrapper action $wa(d_i, d_j)$ corresponding to the meta-action $ma(d_i, d_j)$.

Furthermore, in the schedule Ψ_{Sch} we have: $a_k <_{Sch} ma(d_i, d_j) <_{Sch} wa(d_i, d_j)$.
An example of a WFS will be $\Psi_{Sch2} : S_i.a_1 ma_1\ wa_1\ S_j.a'_1\ S_j.a'_2\ S_i.a_2\ ma_2\ wa_2$.

We next introduce the concept of Dependency Preserved Well-Formed Schedule (DPWFS). Recall that we use the notation $\{\Phi_i\}_{ts}\ a_k\ \{\Phi_i\}_{ts'}$ to indicate that when an action a_k in simulator S_i executes from a state $\{\Phi_i\}_{ts}$, it results in a state $\{\Phi_i\}_{ts'}$.

3.3.3. Dependency preservation. A well-formed schedule preserves a dependency $(d_i \in D_i) \rightarrow (d_j \in D_j)$ if the effects of an action a_k are correctly reflected on all the other actions corresponding to other simulators that depend upon a_k based on $(d_i \in D_i) \rightarrow (d_j \in D_j)$. Formally, let $\Psi_{Sch} = (Sch, <_{Sch})$ be a schedule and let $(d_i \in D_i) \rightarrow (d_j \in D_j)$ be a dependency between two simulators S_i and S_j. Ψ_{Sch} preserves the dependency, if for all external actions a_k and $a_k{}'$ in simulators S_i and S_j respectively such that $d_i \in a_k.Data$ and $d_j \in a_k{}'.Data$, we have $ts_1 < ts'_1$ if and only if $a_k <_{Sch} a_k{}'$ in the schedule, where $\{\Phi_i\}_{ts_0}\ a_k\ \{\Phi_i\}_{ts_1}$and $\{\Phi_j\}_{ts'_1}\ a_k{}'\ \{\Phi_j\}_{ts'_2}$. In other words Ψ_{Sch} ensures that the ending timestamp of a_k occurs before the starting timestamp of $a_k{}'$.

Note that the notion of WFS ensures that the internal mechanisms to transfer updates across simulators through meta-actions and wrapper-actions. The notion of DPWFS on the other hand ensures that the ordering of action in the interacting simulation executes correctly (i.e., preserves causality). For example consider the following schedule:

$$\Psi_{Sch3} : S_i.a_1\ ma_1\ wa_1 S_j.a'_1\ S_i.a_2\ ma_2\ wa_2\ S_j.a'_2.$$

Here Ψ_{Sch3} preserves the dependency because $S_j.a'_2$ is executed after the update on d_i by $S_i.a_2$. In the previous example, Ψ_{Sch2} does not preserve the dependency $(d_i \in D_i) \rightarrow (d_j \in D_j)$, between d_i in simulator S_i and d_j in simulator S_j, because $S_j.a'_1$ starts at time 0 (there is no updates at this point) and results in a new state with time stamp 18. Then, $S_j.a'_2$ is executed at time 18, but it did not reflect the updates on d_i that is performed by $S_i.a_2$ (the update time, 10, is less than the start time of the action, 18), therefore the dependency is not preserved.

It is not only enough to make sure that the effects of the external actions are reflected from S_i into S_j to preserve the dependency, but we also need to make sure that the actions correspondent to the future state of S_i will not be reflected into S_j and will be deferred until the simulator S_j advances to the time when the action is applicable. For example, assume $S_i.a_1$ and $S_i.a_2$ update data item, d_i, resulting in new states with time stamps 10 and 20. $S_j.a'_1$ and $S_j.a'_2$ update d_j and result in states with timestamps 18 and 20. Consider the following schedule:

$$\Psi_{Sch4} : S_i.a_1\ ma_1\ wa_1\ S_j.a'_1\ S_i.a_2\ ma_2\ wa_2\ S_j.a'_2.$$

Here Ψ_{Sch4} does not preserve the dependency since ma_2 and wa_2 correspondent to $S_i.a_2$ are executed before $S_j.a'_2$ while S_j does not advance to the time, 20, to reflect the update performed by $S_i.a_2$.

3.3.4. Correctness of schedules. Consider a set of simulators $S = \{S_1, S_2, \ldots, S_n\}$, and dependency sets for each simulator, a set of meta-actions MA, and a set of wrapper actions WA_i, a well-formed schedule $\Psi_{Sch} = (Sch, <_{Sch})$is said to be a correct schedule if the schedule preserves all dependencies.

Our goal is to address the synchronization problem in multisimulations by ensuring that any concurrent execution of actions of different simulators is a ''correct schedule'' that preserves all of the dependencies. Consider a lock-step schedule in which all simulators advance step by step, at each step, one action from a simulator is executed. In our previous example, Ψ_{Sch3} is a lock-step schedule. A lock-step schedule is a correct schedule that guarantees dependency preservation; however with very heavy overhead, e.g. if there is no dependency between two simulators actions, then we can schedule the actions independently. In this paper, we address the synchronization problem to ensure that concurrent execution of actions from multiple simulators is a correct schedule, and at the same time we also want to minimize the expected execution time for each simulator in the integrated execution. In order to generate correct schedules, first, the actions that correspond to the future state, will not be reflected into S_j and will be deferred until the simulator S_j advances to the time when the update is applicable. This can be achieved by the protocol in Figure 4 as we discussed in Section 3. Second, the effects of any external action are correctly reflected on all of the other actions corresponding to other dependent simulators, for which we need to schedule simulators actions as we will discuss in the next section.

4. Metascheduling for synchronization

In this section, we discuss our approach to address the synchronization problem in multisimulations. To address the synchronization problem, our goal has become that of scheduling simulators actions such that the resulting schedule from concurrent execution is a correct schedule. In our approach, synchronization is only needed when there is data dependency among simulators. This eliminates synchronization altogether during internal action processing periods.

4.1. Baseline strategies

We first illustrate how the correctness of schedules can be guaranteed using two baseline strategies: (a) conservative, and (b) optimistic as described in the following.

Conservative scheduling. A conservative strategy ensures the correctness of schedules by delaying the actions such that the dependencies are preserved in the concurrent execution of actions of simulators. The delay will cause the simulator to freeze until the synchronizer allows it to proceed. We discussed the synchronization protocol based on conservative scheduling in Section 3.2 (Figure 4).

Optimistic scheduling. In some applications it is quite common to be in a situation where although simulators are working simultaneously on interdependent data, violations are infrequent and dependencies continue to be preserved. When this is the case, an optimistic strategy becomes efficient. In the optimistic approach, we accept the fact that violations occur, but instead of trying to prevent them by delaying the actions, we choose to detect them after the action has executed and resolve the violations when they occur. The simulator executes each external action, a_k, without considering the dependencies. If anytime later the meta-synchronizer determined a dependency violation, it will notify the simulator to abort the actions that are executed.

Conservative and optimistic strategies may become more (or less) effective as a multisimulation progresses. The efficacy of a specific strategy at a point in time is a factor of the underlying dependencies and actions taken by the concurrently executing simulators. Initially, the cost of abort is small, so the optimistic strategy performs better than conservative strategy. However, as the simulator proceeds, abort costs become increasingly high. Therefore, the conservative strategy becomes more effective. In the next section, we propose a hybrid approach that combines the benefits of both the optimistic and conservative strategies by considering the underlying dependencies to make an informed decision to abort or delay an action.

4.2. Hybrid scheduling

The main goal of the hybrid approach is to minimize the expected execution time for each simulator in the integrated execution. To minimize the execution time, we examine the expected cost of both conservative and optimistic scheduling of each action, and choose the method with the lower expected cost. In the hybrid policy, we consider an action as safe to execute if it can be scheduled without the potential danger of abort. When the hybrid scheduling strategy encounters unsafe actions, the meta-synchronizer needs to make a decision on whether to allow the action or to delay it. Figure 5 illustrates an example of two concurrently executing simulators, S_i and S_j. Here $a_i \in A_i$ and $a_j \in A_j$ represent the actions of simulator S_i and S_j respectively. Consider a dependency $(d_i \in D_i) \rightarrow \left(d_j \in D_j\right)$, between d_i in S_i and d_j in S_j. Let t_i and t_j refer to the current time in S_i and S_j, respectively. The next external action of S_j, a_k, is unsafe if the current time in S_i is less than the current time in S_j, $t_i < t_j$. The reason is that a dependency might be violated by an update received from S_i, that leads to the abort of S_j.

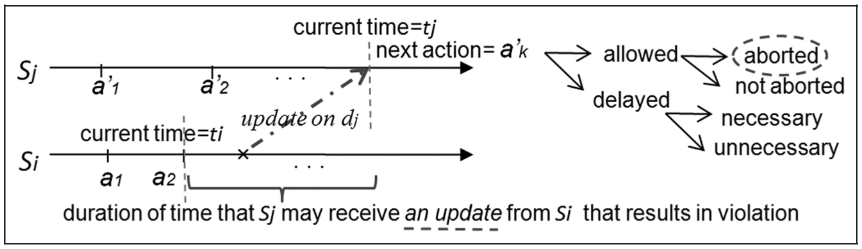


Figure 5. The example of two concurrently executing simulators.

```
Meta-Synchronizer
1:  initialize the meta-actions-queue MAQ as a queue of $< ma(d_i, d_j), ts >$;
2:  initialize the delayed-actions-queue DAQ as a queue of $< a_k, S_i, t_i, S_j >$;
3:  initialize the list of $S_i$.track as a list of $< S_j, t_{start_i} >$;
4:  initialize the list of $S_i$.cascade as a list of $< S_j, ts_j >$;
5:  initialize a list of simulators SimList as a list of $< S_i, t_i, Dep(S_i), S_i.abort$, $S_i$.track, $S_i$.cascade, $S_i$.log >;
6:  Upon a request from a simulator $S_j$ to get the updates: Request-Updates$< S_j, t_j >$
7:      find and remove all meta-actions $< ma(d_i, d_j), ts >$ from MAQ s.t. $d_j \in D_j$, $ts < t_j$
8:      send $< ma(d_i, d_j), ts >$ them to $S_j$;
9:      update time of simulator $S_j$ to $t_j$ in SimList;
10:     update $S_i$.log ; /* adding $< ma(d_i, d_j), ts >$ to the log to replay the updates in the case of abort */
11:     add $S_j$ to $S_i$.cascade ;
12:     find delayed actions $< a_k, S_i, t_i, S_j >$ from DAQ s.t. $t_i < t_j$ waiting for $S_j$
                                    where $t_j$ is the time of $S_j$ extracted from SimList;
13: allow($a_k$) and remove $< a_k, S_i, t_i, S_j >$ from DAQ;
14: Upon receiving a request to execute an external action: Request-Execute$< S_i, a_k >$
15:       if $a_k$ is safe then /* $< S_i, a_k >$ is safe if $t_i < t_j$ for all $S_j \in Dep(S_i)$*/
16:           allow($a_k$);
17:       else      /* $a_k$ is potentially unsafe $\exists S_j \in Dep(S_i)$ s.t. $t_i > t_j$ */
18:           $C_D = \frac{\sum_{a_j \in A_{du}} dt}{du_j}$; /*the cost of unnecessary delay*/
19:           $C_A = t_j - t_{dl}$;   /*the cost of abort- restart from beginning*/
20:           $p^A = \prod^{i}_{S_i \in Dep(S_j)} exp(\frac{-n_{ij}^{updates}.max(0, t_j - t_i)}{t_i})$ ; /*the probability of abort*/
21:           if $C_D < \frac{p^A}{(1-p^A)} . C_A$ then /*making a decision to delay, abort or allow*/
22:               delay($a_k$);  /*delay the action until notified by meta-synchronizer*/
23:     else
24:           allow ($a_k$);
25:           $t_{start}$=$t_i$;  /* the start time of action */
26:           add $< S_j, t_{start} >$ to $S_i$.track;
27: Upon receiving updates $< d_i, t'_i >$ from $S_i$: Update$< S_j, d_i, t_i' >$
28:     for all dependencies $(d_i \in D_i) \rightarrow (d_j \in D_j)$ do
29:           generate meta-actions $< ma(d_i, d_j), t_i' >$ and post it to MAQ;
30:     for all simulator $S_j$ s.t. $< S_i, t_{start_i} > \in S_j$.track
31:         find all updates $< ma(d_i, d_j), t_i' >$ from MQA s.t. $d_i \in S_i$ and $d_j \in S_j$
32:     if $t_i' < t_{start_i}$ then
33:       set $S_j.abort$=true in SimList; /*set abort flag for the simulator*/
34:       send abort($t_a$) to $S_j$ notify to abort where $t_a = t_i'$ extracted from $< ma(d_i, d_j), t_i' >$;
35:       set $S_k.abort$=true for all $< S_k, ts_k > \in S_j$.cascade s.t. $ts_k > t_a$;
36:       send abort($t_a$) to $S_k$ ; /*abort those simulators receiving an update from $S_i$ */
37:     Upon receiving abort-done$< S_i >$ from a simulator $S_i$
38:       set $S_i.abort$=false in SimList;
```

Figure 6. Basic structure for hybrid scheduling (*HS*): meta-synchronizer.

The meta-synchronizer can make an informed decision about an action based on EC_D, the expected cost of conservatively scheduling (delay), and EC_A, the expected cost of optimistically scheduling (abort). For each simulator, the meta-synchronizer maintains the statistics that will be used by the hybrid strategy for decision making. Let p^A refer to the probability of the abort if the action is unsafe, C_A be the cost of abort, and C_D be the cost of unnecessary delay. The following are the expected costs. For a given action a_k, if the expected cost of the delay is less than the expected cost of an abort, $EC_D(a_k) < EC_A(a_k)$, the action will be delayed: $C_D < \frac{p^A}{(1-p^A)}.C_A$,

$$EC_A(a_k) = p(abort|optimistically\ scheduled).C_A \cong p^A.C_A \quad (1)$$

$$EC_D(a_k) = p(unnecessary\ delay|conservatively\ scheduled).C_D \cong (1-p^A).C_D \quad (2)$$

4.2.1. Computing the probability of abort. Assume that an action from simulator S_j executes from a state $\{\Phi_j\}_{t_j}$ and results in a new state $\{\Phi_i\}_{t_j'}$. This action will result in an abort, if S_j receives an update from a simulator $S_i \in Dep(S_j)$ via a meta-action $< ma(d_i, d_j), ts' >$ such that: $ts' < t_j$

```
Simulator's wrapper
1: while (simulation still in progress){
2:   for each (time step or event) do{
3:     $t_i$=current-time;
4:     send a Request-Update <$S_i, t_i$ > to get meta-actions and apply updates using wrapper actions;
5:     action $a_k$= nextAction;
6:     if ($a_k$ is an internal action)
7:         proceed-simulator ($a_k$) and update simulation time to $t_i'$; /* allow the simulator to execute the internal action*/
8:     else /* $a_k$ is external action */
9:         send a Request-Execute <$a_k, t_i$ > to meta-synchronizer and wait until allow;
10:        proceed-simulator ($a_k$) and update simulation time to $t_i'$; /*execute the external action*/
11:        send Updates< $S_i, d_i, t_i'$ > to meta-synchronizer;
12:    if (abort($t_a$) received from meta-synchronizer) then
13:        restart the simulation from the beginning until $t_a$ and applying updates from $S_i$.log ;
14:        send abort-done< $S_i$ > ;}}
```

Figure 7. Basic structure for hybrid scheduling (*HS*): simulator's wrapper.

(where t_j is the start time of the action). This indicates that the update by S_i should have been reflected into S_j before the action's execution. Let $p^U_{i,j}$ refer to the probability of receiving an update from S_i that needs to be reflected intoS_j. We assume that updates in each simulator occur independently of the time since the last update. We use Poisson distribution to express the probability of a number of updates occurring in a period of time. If the expected number of occurrence of an update in an interval is λ, then the probability that there are exactly n updates will be equal to: $p^\lambda_T(n) = \frac{(\lambda T)^n e^{-\lambda T}}{\lambda!}$. We will have $p^U_{i,j} = 1 - p^\lambda_{t_j - t_i}(0) = 1 - e^{-r_{ij}.max(0,t_j - t_i)}$ where $\lambda = r_{ij} = \frac{n^{updates}_{ij}}{t_i}$. In which, r_{ij} is the rate of sending updates from S_i to S_j which is, $n^{updates}_{ij}$, the number of updates sent so far, divided by, t_i, the current time in S_i. We consider $max(0, t_j - t_i)$ because we are only interested in those simulators in S_j's dependency set lagging behind the time in S_j. Then, p^A is derived by the following equation:

$$p^A = 1 - \prod\nolimits^{i}_{S_i Dep(S_j)} (1 - p^U_{i,j}) = \prod\nolimits^{i}_{S_i Dep(S_j)} exp\left[\frac{-n^{updates}_{ij}.max(0, t_j - t_i)}{t_i}\right] \tag{3}$$

4.2.2. Computing the costs. The cost of abort, C_A, is the estimate of the amount of time the simulator needs to redo its work, t_j, from which we subtract the total time the simulator spent in delay, t_{dl}. We consider C_D, the cost of unnecessary delays, as the average time for delay which is calculated by monitoring and adding all the delay times, dt, of actions that are delayed unnecessarily ($a_j \in A_{du}$) and dividing it by the number of actions that were delayed unnecessarily,du_j:

$$C_A = t_j - t_{dl},\ C_D = \frac{\sum_{a_j \in A_{du}} dt}{du_j} \tag{4}$$

Unnecessary delayed actions are those actions that are delayed but if optimistically scheduled they will not result in an abort. They are determined from the updates sent by suppliers. If an update is received with the timestamp smaller than the start time of the action then the delay is necessary, otherwise it is un-necessary.

The algorithm for hybrid strategy is shown in Figures 6 and 7. The gray area represents the difference from the protocol in Figure 4. First, we determine whether an action is safe or not. We consider an action as safe to execute if it can be scheduled without the potential danger of abort. Therefore an action $a_k \in S_j$ is safe if the current time in S_j is greater than the current time in S_i for all $S_i \in Dep(S_j)$. Then the strategy estimates costs (lines 17–22) to make an informed decision about an action. Note that the hybrid strategy is self-adaptive. Initially the cost of abort is small, so the strategy will prefer to be optimistic. However, as the simulator proceeds, it becomes increasingly conservative.

There is one transition from conservative strategy to optimistic strategy in the whole simulation life. The convergence from conservative to optimistic depends on the computing costs which might be different from one simulation to another. In the following theorems, we show that the schedules resulting from hybrid approach are correct schedules without deadlocks.

Theorem 1. For a set of simulators $S = \{S_1, S_2, \ldots, S_n\}$, and their corresponding dependency sets, every schedule $\Psi_{Sch} = (Sch, <_s)$ that is generated as a result of hybrid scheduling, is a correct schedule.

Proof. By contradiction. Assume $\Psi_{Sch} = (Sch, <_{Sch})$ is not a correct schedule. Without loss of generality we assume, a'_k, is executed by S_j, $\{\Phi_j\}_{t'_1}\ a'_k\ \{\Phi_j\}_{t'_2}$ and a dependency, $(d_i \in D_i) \rightarrow \left(d_j \in D_j\right)$ between two simulators S_i and S_j, such that by executing the action a'_k from S_j, the dependency is not preserved. If the dependency $(d_i \in D_i) \rightarrow \left(d_j \in D_j\right)$ is not preserved, we have the following cases.

(1) There exists an external action a_k, $\{\Phi_i\}_{t_1}\ a_k\ \{\Phi_i\}_{t_2}$, in another simulator S_i such that $d_i \in a_k.Data$ and $d_j \in a'_k.Data$, where $t_1 < t'_1$ and a_k executed after a'_k in the schedule $a'_k <_{Sch} a_k$. We have the following cases. (a) Assume that a_k is conservatively scheduled. The action a'_k is allowed by the meta-synchronizer therefore a'_k is safe which implies that the time in all dependent supplier simulators (such as S_i) is greater than the timestamp of the current state of S_j, t'_1, thus $t'_1 < t_1$, this is a contradiction. (b) Assume that a_k is optimistically scheduled. In the hybrid scheduling, if there exist another action $\{\Phi_i\}_{t_1}\ a_k\ \{\Phi_i\}_{t_2}$, from another simulator S_i s.t. $d_i \in a_k.Data$ and $d_j \in a_k{}'.Data$, when $t_1 < t'_1$ and $a'_k <_{Sch} a_k$ then a'_k will be aborted, therefore a'_k will be removed from the schedule. This is a contradiction.

(2) Or there exists an external action a_k, $\{\Phi_i\}_{t_1}\ a_k\ \{\Phi_i\}_{t_2}$, in another simulator S_i s.t. $d_i \in a_k.Data$ and $d_j \in a'_k.Data$, where $t'_1 < t_1$ and $a_k <_{Sch} a'_k$ which leads to contradiction same as the above cases.

Theorem 2. Given a set of simulators $S = \{S_1, S_2, \ldots, S_n\}$, every schedule $\Psi_{Sch} = (Sch, <_{Sch})$ that is generated as a result of hybrid scheduling cannot lead to a deadlock.

Proof. By contradiction. Assume $\Psi_{Sch} = (Sch, <_{Sch})$ is not a deadlock-free schedule. Then there is a cyclic dependency between simulators that leads to the deadlock. Consider a dependency between two data items in S_i and S_j, $(d_i \in D_i) \rightarrow \left(d_j \in D_j\right)$, and another dependency, $\left(d_n \in D_j\right) \rightarrow (d_m \in D_i)$, between S_j and S_i. We need to show that such dependencies cannot lead to deadlocks. To see this, note that in hybrid scheduling, a simulator S_i only waits for simulator S_j if its state timestamp is greater than that of S_j. Assume the current state of S_i and S_j are $\{\Phi_i\}_{ts_i}$ and $\{\Phi_j\}_{ts_j}$ respectively. Then S_i waits for S_j if and only if $ts_i < ts_j$. However, if S_j waits for S_i then $ts_j < ts_i$, this is a contradiction. Thus, a cycle in the wait-for graph of simulators is not possible. Therefore, the deadlock is not possible.

To generalize the discussion to the cycle of dependencies involving more than two simulators, consider the cyclic dependency: $S_i \rightarrow S_{i+1} \rightarrow S_{i+2} \rightarrow \ldots \rightarrow S_j \rightarrow S_i$, which implies $ts_i < ts_{i+1} < ts_{i+2} < \ldots < ts_j$ and $ts_j < ts_i$ which is a contradiction. Furthermore, for an external action a_k when we delay the meta-action $ma\left(d_i, d_j\right)$ and the wrapper action $wa\left(d_i, d_j\right)$ corresponding to a_k and a dependency $(d_i \in D_i) \rightarrow \left(d_j \in D_j\right)$, the delay will not cause the deadlock since it does not cause the action a_k to be delayed.

In the presence of dependencies between simulators, the multisimulation as a whole might not have forward progress (i.e. endless cycles of aborts). When simulators received updates from meta-synchronizer and aborted and then immediately restarted, the possibility of ‘‘cyclic restarts’’ exists. Some simulators may repeatedly get a new update from a supplier simulator which results in a dependency violation, be aborted, and restarted, only to reach the same state again. The simulators actions involved in a cycle do not terminate by themselves. Consider a cyclic dependency between two data items in S_i and S_j, $(d_i \in D_i) \rightarrow \left(d_j \in D_j\right)$ and $\left(d_n \in D_j\right) \rightarrow (d_m \in D_i)$. Assume that S_j needs to abort because it received an update from S_i for which the timestamp of the update is smaller than the start time of the action in S_j. In the meantime, S_j also sent an update to S_i. Since S_j is aborted this update that is received by S_i needs to be undone; therefore S_i is also aborted. The same scenario might occur again and again. To ensure the forward progress of multisimulation, in the case of cyclic dependencies we add an operation at the meta-level to remove the updates from the meta-action queue. In the case that the updates which need to be undone are not sent to the consumer simulator, the remove operation is enough to delete the correspondent meta-action. If the updates are sent to the consumer, then the meta-synchronizer send a message to the consumer to undo the updates. The undo is possible because S_i in a cyclic dependency is behind the S_j. (Note that its update in the first place that results in roll-back had a timestamp smaller than the time in S_j.) Therefore S_i does not reach the point to apply the update received from S_j.

4.3. Optimization 1: relaxed dependencies

In a multisimulation, ideally, dependencies need to be reflected from one simulator into another as soon as update in one simulator becomes valid in another. However, in most applications, ideal behavior results in unnecessary synchronization overhead and loss of concurrency among simulators in the integrated simulation. In this section, we describe our approach in which we relax the dependencies and capture the extent to which simulators can deviate from ideal behavior. By using relaxations, the resulting schedules will tolerate some amount of inconsistency but they still will be allowed by meta-synchronizer. Note that the structure of proposed scheduling algorithms will not change.

Consider a dependency between two data items in two simulators S_i and S_j: $(d_i \in D_i) \rightarrow \left(d_j \in D_j\right)$. Each data item has a value at any given state, $(d_i, v_i) \in \{\Phi_i\}_{t_i}$ and $\left(d_j, v_j\right) \in \left\{\Phi_j\right\}_{t_j}$. Now we consider three types of deviations.

- **Time***(t-bound): t-bound* works as the delay condition which states how much time the consumer can use a value behind the new update of the supplier. At any time, if $\left\langle ma\left(d_j, d_i\right), t_i{}'\right\rangle$ is the meta-action posted as a result of an update by S_i, $\langle wa(d_j), t_r\rangle$ is the corresponding wrapper action to reflect the update into S_j, and t_j is the current time in S_i then $t_r - t_j \leqslant t\text{-}bound$.

- **Value** (v-distance): let v_i be the value of d_i updated by S_i and v_j be the value of d_j updated by S_j, we consider the difference between the values of two data items using a user defined distance function $distance(v_i, v_j)$.[54] v-distance is the maximum amount of distance permitted between the values at any time: $distance(v_i, v_j) \leqslant$ v-distance.
- **Number of changes** (n-update): captures the maximum number of updates on d_i before they become reflected on d_j. In other words, at any time, the consumer, S_j, can skip at most n-update of d_i that have been performed by the supplier, S_i.

We use a logical expression to represent the relaxations as conjunction and/or disjunction of three deviation types: *t-bound, v-distance, n-updates*.

4.3.1. Relaxation. Given two simulators S_i and S_j, $i \neq j$, and two data items $d_i \in D_i$ and $d_j \in D_j$, and a dependency $(d_i \in D_i) \rightarrow (d_j \in D_j)$, a relaxation is a 3-tuple $c_r(d_i, d_j, exp)$ where $c_r.exp = (t\text{-}bound{=}t\ [\wedge|\vee]\ n\text{-}update{=}n\ [\wedge|\vee] v\text{-}distance{=}v)$, $t \in T_p$, $n \in \mathbb{N}$, and $v \in Dom(d_i)$. We illustrate this through an example.

4.3.2. Example. Consider the case of the dependency between d_i, the smoke level, in a fire simulator S_i and, d_j, the person's health condition, in an activity simulator S_j. We express this dependency using a relaxation: $c_r(d_i, d_j, t\text{-}bound{=}5 \wedge n\text{-}update{=}2)$. Here c_r indicates that the smoke level will not affect the health condition immediately, but instead in $5s$ (expressed as *t-bound=5);* this implies that health information from the fire simulator must be reflected to health conditions in an activity simulator with at most a $5s$ delay to be consistent. Since the activity simulator can process its actions without considering updates from the fire simulators in $5s$, the relaxation helps the overall speed up of multisimulation. In addition, *n-update=2* indicates that there should be at most two updates on the smoke level in S_i before it is reflected into S_j. This reduces the synchronization overhead since the activity simulator can skip one update that has been performed by the fire simulator.

4.4. Optimization 2: checkpointing and rollback mechanism

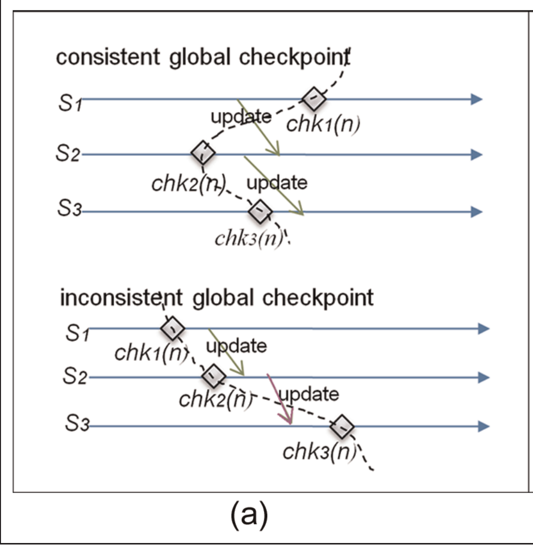
The implementation of allow, delay, and abort operations in the wrapper is straightforward; it will proceed, freeze, or restart the simulation from its start time, respectively. The abort operation (restart) in multisimulations is not always cost effective because (i) long-running scientific simulation programs designed to run for days need to restart from the beginning, and (ii) as the result of aborting one simulator, multiple dependent simulators that used its updates need to restart as well, which results in very significant overhead on multisimulation execution. Therefore, we explore a checkpointing and rollback mechanism[55] that enables a simulator to restart effectively from a checkpoint.

4.4.1. Local checkpoint. A local checkpoint, $chk_i(n)$, of a simulator S_i is the saving and restoration of the simulator's state that provides the information needed to restart the simulation in the case of rollback. Checkpoints are assigned ordinal numbers by the meta-synchronizer from a monotonically increasing counter. This checkpoint number is piggybacked to each updates sent from simulators to the meta-synchronizer.

4.4.2. Global multisimulation checkpoint. A global multisimulation checkpoint, $GMC(n) = \underset{i}{\cup} chk_i(n)$, is the set of local checkpoints, $chk_i(n)$, one for each simulator. Each local checkpoint belonging to the same global checkpoint identifies by the same ordinal number. This number is incremented every time a global multisimulation checkpoint is created. A global checkpoint is consistent if:

- for every update sent from S_i to the meta-synchronizer and reflected into S_j before $chk_j(n)$, then the sending of that update must have happened before $chk_i(n)$ (see Figure 8(a));
- for every update sent from S_i to the meta-synchronizer before $chk_i(n)$ and reflected into S_j after $chk_j(n)$, the corresponding meta-action of that update must be saved to meta-actions queue to be replayed if the simulators has to roll back.

4.4.3. Checkpointing protocol. We now describe the protocol for coordinated application-level checkpointing. The protocol (Figure 8(b)) is independent of the technique used by individual simulators to save their state. Each local checkpoint belonging to the same global checkpoint (global rollback line) has the same ordinal number. The checkpointing operation is triggered periodically by a timer mechanism. When this timer expires, the meta-synchronizer forces all simulators to take a checkpoint, by sending a message - *chkp-req(n)*. For consistency, we need to prevent an update by S_i, within $chk_i(n)$, from being received by another simulator S_j, with $chk_j(n-1)$. We do not have to block S_j, until S_i has taken its checkpoint. In fact, the n counter attached to each update will solve this problem, and simulators do not have to be blocked by the algorithm. If a simulator receives an update from another simulator, that is already running in the next checkpoint interval, the corresponding simulator takes a checkpoint of its execution state before



```
Meta-synchronizer checkpointing:
  if $timer$ expires then
    send $chkp$-$req(n)$ to all $S_i$ s.t $S_i.chkp$ = on
    if receive $chkp$-$ack(n)$ from all $S_i$ then
      build recovery-line $GMC(n) = \cup_i chk_i(n)$
      restart($timer$);
      send $done.GMC(n)$ to all $S_i$ s.t $S_i.chkp$ = on
      $n$++;

Simulator checkpointing($S_i$):
  $chk_i(n)$.state= state of $S_i$;
  $chk_i(n).\ t_{chk_n} = t_i$ ;
  send $chkp$-$ack(n)$ to meta-synchronizer;
  if $done.GMC(n)$ received from meta-level then
        discard $chk_i(n-1)$;
```
(b)

Figure 8. (a) Multisimulation global checkpoint. (b) The checkpointing protocol.

reflecting that update. Each simulator after taking a checkpoint of its state, has to send an acknowledgment message to the meta-synchronizer, designated by *chkp-ack(n)*, so that later a new checkpoint can be initiated.

In the case of rollback, associated with the rollback notification from metasynchronizer is a checkpoint, chk_n, which indicates the checkpoint to which simulation needs to be rolled back. The cost of rollback, C_R, is the estimate of the amount of simulation time the simulator needs to redo its work from t_{chk_n}, from which we subtract the time in between the simulator spent in delay (the blocking time), t_{dl_j}. It also involves the overhead for storing/checkpointing the simulator's state (C_{CHK_j}):

$$C_R = t_j - t_{dl_j} - t_{chk_n} + C_{CHK_j} \tag{5}$$

Consider the metasynchronizer decides to rollback a simulator S_i. Assume the time of the update that leads to the rollback occurs at time ts'. First, the metasynchronizer must determine the rollback line to which simulators should rollback. This will be the latest rollback line before time ts'. Then it forces S_i and all simulators that communicate with S_i (directly or through other simulators) to roll back to the recovery line. Simulators need to keep only the last checkpoint and associated meta-actions of those updates received from other simulators after the last checkpoint. If the state of a simulator cannot be saved as a checkpoint (because of the black box nature of the simulation), the simulation restarts from the beginning in the proposed algorithms in Section 4.

5. Prototype system implementation

In this section we discuss our methodology, its relevant issues, and the implementation of a prototype multisimulation system. Figure 9 shows different modules in the prototype system. Consistent with our architecture design philosophy, the design aims to separate the base-level aspects of each simulator (this includes the simulator code, the back-end databases and models stored in domain-specific formats) from the meta-level synchronization and adaptation mechanisms.

Each simulator at base level is a black box with its own internal data structures, models and source code. Associated with each simulator is a set of entities at the meta-level that is the result of the reification process to capture relevant aspects of each simulator stored in meta-models. We maintain the set of interdependent data items for which there exists at least one dependency to data items in other simulators. Dependencies are stored in a dependency table at meta-level and indexed by its corresponding interdependent data items. Once a simulator updates a data item, the wrapper determines whether the update is on an interdependent data item and sends it to meta-synchronizer. There are three key modules at the meta-level: (a) a *Meta-Synchronizer* which uses the proposed approaches to monitor and control concurrent execution in the multisimulation; this module also makes use of a lock manager to coordinate concurrent access to simulators data items; (b) an *Analyzer* which analyzes the interactions between simulators using meta-models to capture the dependencies; (c) an *Adaptor* which manages the data exchange and adapts information that is passed between simulators through the design of wrapper modules for each simulator.

Figure 9 shows the basic metamodel and the sequence of interactions between base level simulators, simulator wrappers, and the meta-synchronizer. In the initialization steps the wrapper sends a request for connection to the meta-synchronizer. The meta-synchronizer confirms the

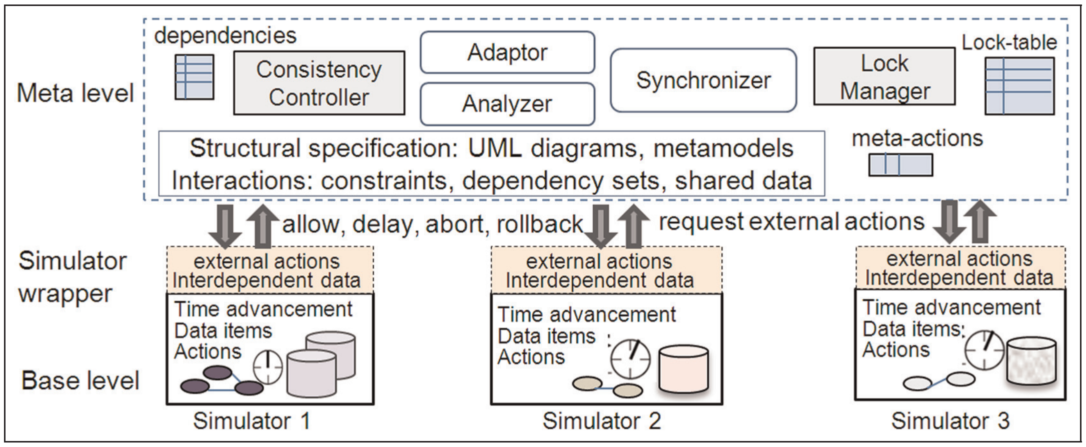


Figure 9. Multisimulation framework.

connection and sends information about interdependent data items and dependencies to the wrapper. Once the simulation starts, the wrapper determines which actions are external actions and sends corresponding requests to the meta-synchronizer. Upon receiving a request, the meta-level modules evaluate the dependencies/relaxations and respond to the wrapper with a decision allow, delay or restart (abort or rollback). It also sends to the wrapper, those meta-actions that contain the updates by other simulators. The wrapper reflects the received meta-actions on the execution of the underlying simulator using wrapper actions. This loop is continued until the end of simulation.

With growing complexity, the scalability of a multisimulation environment becomes an important aspect. There are several salability issues that can be considered in a multisimulation framework (i.e. scalability number of simulation models, number of dependencies between simulators, number of entities in simulators, etc.). Different simulators in our framework can run on different machines which make a system that is executed on a distributed architecture. Relaxations allow us to handle more updates on simulators' entities with less synchronization overhead. For example, in our framework by using *n-update* in relaxations we can capture the maximum number of updates on an entity in a simulator before sending it to another simulator. In our experiments, we test the scalability performance with different number of dependencies to study the behavior of our approaches.

5.1. Integrating real-world simulators

To ground our work in reality, we develop a case study for simulation integration using three pre-existing real world simulators from the emergency response domain. Table 1 summarizes the three simulators and their properties. In our case study, we focus primarily on integrating simulation and models aimed at informing emergency response policy decision making, but we expect our framework and methods will be applicable to other complex problem domains. The specific simulators are as follows.

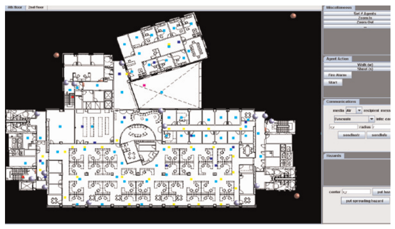
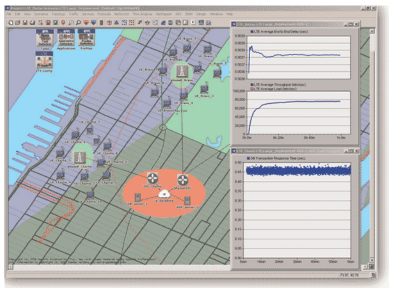
***(1) Fire simulator*:** CFAST, the Consolidated Model of Fire and Smoke Transport, is a simulator that simulates the impact of fires and smoke in a specific building environment and calculates the evolving distribution of smoke, fire gases, and temperature.[5] CFAST is a black box simulator (we do not have access to the source code of the simulator) which has several interfaces to input the parameters that contain information about the building geometry, fire properties, etc. The simulator produces outputs that contain information about temperatures, ignition times, gas concentrations, etc.

***(2) Activity simulator*:** Drillsim is a multiagent that plays out the activities of a crisis response process, e.g. building evacuation in response to an evolving fire hazard. Drillsim simulates human behavior in a crisis at fine granularities[7] agents represent an evacuee, a building captain, etc. Every agent has a set of properties associated with it, such as physical perceptual profile (e.g. range of sight, speed of walking) and the current health status of the agent (e.g. injured, unconscious).

***(3) Communication simulator*:** LTEsim, the communication simulator in our case study, is a LTE System Level simulator[56] which abstracts the physical layer and performs network-level simulations of 3GPP Long-Term Evolution with lower complexity. We chose LTEsim because the LTE standard has several improvements in capacity, speed, and latency and will be the technology of choice for most existing 3GPP mobile operators.[57] LTEsim considers several parameters to model the communication infrastructure (such as network layout, bandwidth, pathloss, etc.).

In our integration scenario, the fire simulator, CFAST, is used to simulate the impact of fire in a specific region and calculates the evolving distribution of smoke; fire and smoke can affect evacuation process, e.g. people's health condition, in the evacuation simulator, Drillsim, which has impacts on communication patterns in communication

Table 1. Three real-world simulators.

Activity simulator	Communication simulator	Fire simulator
 • DrillSim,[7] Time-stepped • Open source (in Java) • Parameters: health profile, visual distance, speeds of walking, number of ongoing call, etc. Output: number of evacuees, injuries, etc.	 • LTESim[56] • Event-based • Open source (in Matlab) • Parameters: number of transmit and receive antennas, uplink delay, network layout, channel model, bandwidth, frequency, etc. Output: pathloss, throughput, etc.	• CFAST,[5] Time-stepped • Black box (no access to source) • Parameters: building geometry, materials of construction, fire properties, etc. Output: temperatures, pressure, gas concentrations, etc.

simulator, LTEsim. Such integration is useful to conduct better what-if analyses and understand various factors that can adversely delay evacuation times or increase exposure and consequently used to make decisions that can improve safety and emergency response times.

The first step is to specify meta-models for the three base-level simulators and dependencies across them. Metamodels are UML diagrams that help us to understanding/reasoning about interactions between different simulators. We developed a set of tools that help in creating the metamodels. There are several key classes in the metamodel: model type, actions, model elements (data items) which could be local data or interdependent data, input or output parameters, actions, and constraints. Model type includes information about the type of simulation model. Model elements are the main elements of a simulation and can be captured from the interfaces, the source code, or databases. We have implemented a parser using tools for large scale code repositories search[58] to extract the entities and attributes using the simulator's source code, interfaces, and databases. Then we group extracted information into features to capture the structure of the simulator. The features are put into the same class if they are considered equivalent. Since our metamodel needs to take several domain expert simulators into account, the metamodel should be comprehensive, yet extensible. In our metamodel, we also consider input and output parameters. Finally, constraints are the number of limits for the simulation parameters in the simulation model. Figure 10 illustrates the meta-models for the three simulators. The following are the examples of information interchanged among simulators:

- a harmful condition in CFAST can affect an individual's health in Drillsim;
- smoke in CFAST can decrease an agent's visual distance in Drillsim;
- pathloss in LTEsim can be used to determine connectivity/coverage in Drillsim.

In our current implementation, several such dependencies specified (the actual number of dependencies required was in the range of 10-50 for most situations). Table 2 shows some examples of relaxations using our notation.

5.2. Experimental evaluations

Our experiments are based on the case study described above where we integrate three real-world simulators: Drillsim, CFAST, and LTESim. Drillsim has been modified to develop checkpointing mechanism using libckpt[59] user-level incremental checkpointing. The experiments are conducted on three machines (Intel(R) Core 2 Duo P8700 2.53 GHz, 4 GB; Pentium 4 3 GHz, 1 GB; Intel(R) Core 2 Duo E8400 3.0 GHz, 4 GB) connected to each other with a Gigabit Ethernet controller.

In our first set of experiments, we implemented four different meta-scheduling techniques for synchronization across the three simulators: conservative scheduling (*CS*), optimistic scheduling (*OS*), hybrid scheduling (*HS*) and lock-step scheduling (*LS*) which is a variant of *CS*. *LS* implements a lock step schedule in which all simulators advance step by step; it is the most conservative approach since at any step the simulators synchronize by locking at data item level until the next step. Locking data at the

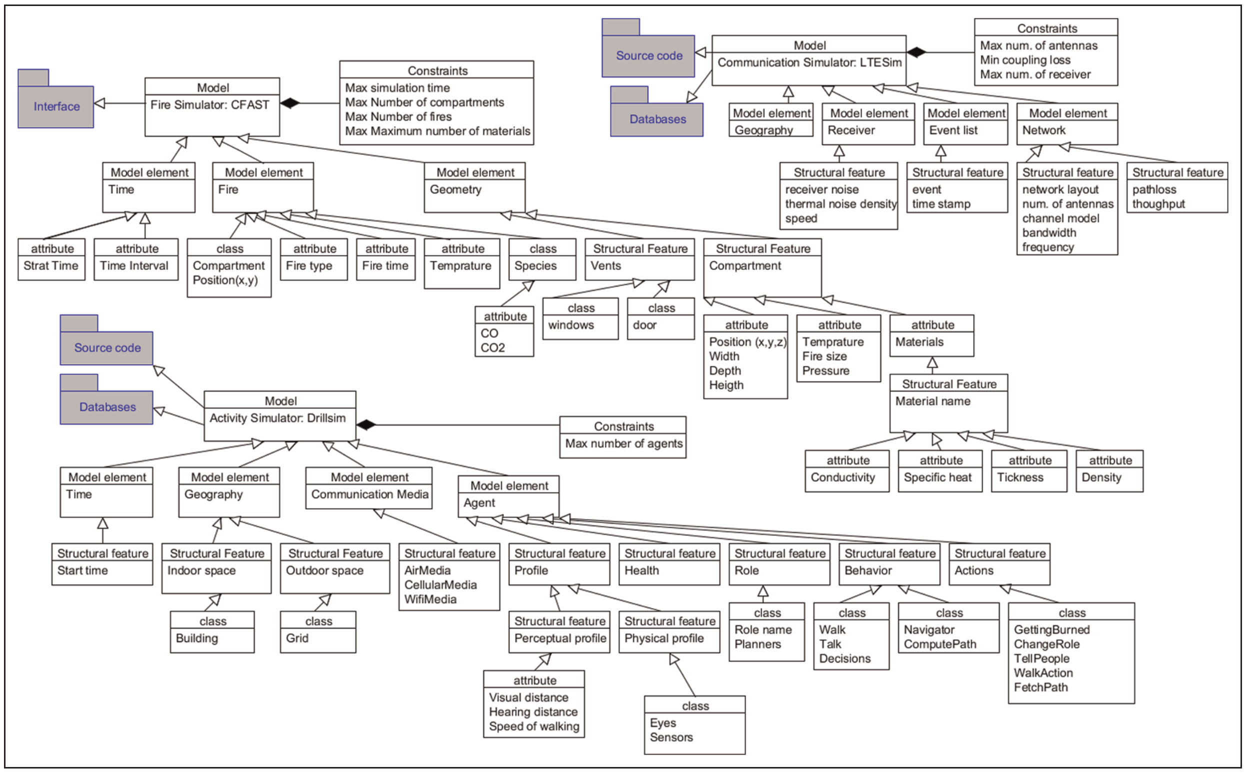


Figure 10. Metamodels of (a) LTEsim, (b) CFAST, and (c) Drillsim.

Table 2. Relaxations for dependencies using our notation.

Relaxations: $c_r(d_i, d_j, exp)$	**Interdependent data**
c_1(health, smoke, *tbound* = 5ms)	health ∊ Drillsim, smoke ∊ CFAST
c_2(visualDistance, smoke, *tbound* = 7ms)	visualDistance ∊ Drillsim, smoke ∊ CFAST
c_3(cellCoverage, throughput, *tbound* = 1min)	cellCoverage ∊ Drillsim, throughput ∊ CFAST
c_4(numofUsers, ongoingCalls, *tbound* = 1min $\wedge$ *nupdates* = 5)	numofUsers ∊ LTEsim, ongoingCalls ∊ Drillsim
c_5(bldgGeo, numAntenna, *nupdates* = 1)	bldgGeo ∊ CFAST, numAntenna ∊ LTEsim
c_6(walkingSpeed, userSpeed, *vdistance* = 3m/s)	walkingSpeed ∊ Drillsim, userSpeed ∊ LTEsim

beginning of action and releasing the locks at the end enables us to prevent deadlocks.

In the second set of experiments we applied relaxations to the above techniques: conservative scheduling using relaxations (*CSR*), optimistic scheduling using relaxations (*OSR*), and hybrid scheduling using relaxations (*HSR*). We consider wait time before each abort/rollback to be 0.5*s* (to make sure that the rollback scenario will not occur immediately). In *HS* and *HSR* we considered the statistics of the most recent 50 actions in each simulator in order to calculate the expected costs. Every measurement in our results is an average of 5 runs.

As stated in Section 3, using concepts from multidatabases, we model multisimulations as sequences of actions, where each action of a simulator is an atomic unit of processing that reads and modifies data. In our framework for experimental evaluations, an action captures changes that occur in a tick (or a step) in a time-stepped simulator or the execution of an event in an event-based simulator. We studied the performance results in three different phases of simulation life time: the early phase (where the total number of actions from all three simulators is 1000 actions), the intermediate phase (where the total number of actions is 2500), and the late phase (in which the total actions is around 3500). This helps us to study the effect of our synchronization mechanisms in different phases during simulation. Prior to integration, each of three simulator s involved in the experiments runs autonomously with the average time of 1 second per action. We consider three performance measures in our evaluations.

***(1) Timing performance*:** We studied the synchronization overhead and the total execution time using different

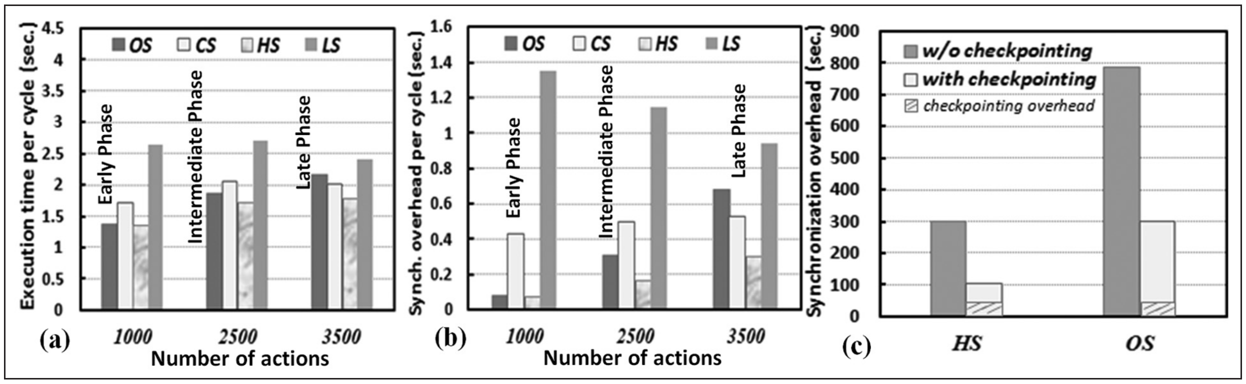


Figure 11. (a) Total execution time per action, (b) average synchronization overhead per action, in different phases of execution, and (c) synchronization overhead with and w/o checkpointing in *Drillsim* using *OS* and *HS*. (Number of dependencies=100.)

techniques. We measured synchronization overhead by adding the synchronization overhead in all simulators. In *LS* and *CS*, we consider the total delay time, i.e. the duration a simulator is blocked and the locking overhead, i.e. the time needed for acquiring or releasing locks, to calculate the synchronization overhead. In *OS*, we consider the total abort/rollback time and the checkpointing time. In *HS*, the synchronization overhead is calculated by adding the delay times and the abort/rollback times and checkpointing times. Figure 11(a) and (b) illustrate the average execution time per action and average synchronization overhead per action during different phases of execution (i.e. for different numbers of actions). The synchronization overhead in *CS* and *HS* is lower as compared with *OS* and *LS* during later phases of execution. This is due to the high blocking time in *LS* and abort time in *OS*. In *HS*, initially, the cost of abort is small, so the strategy will lean towards being optimistic and is hence closer to *OS* in performance. However, as the number of actions increases, *HS* becomes increasingly conservative and approaches the behavior of *CS*. We eliminate *LS* from further consideration due to its poor overall performance. Figure 11(c) shows the synchronization overhead of *OS* and *HS* in Drillsim with and without checkpointing. This experiment considered the modified version of the simulator using incremental checkpointing mechanism of libckpt.[30] We observe that synchronization overhead is significantly lower when checkpointing is used. This is because in the case of abort, the simulator restarts from a checkpoint instead of restarting from the beginning.

***(2) Scalability performance*:** We studied the behavior of our approaches for different number of dependencies. We introduce synthetic dependencies to investigate the scalability of our approaches as the number of interdependent data items in different simulators increases. Figure 12(a) represents the synchronization overhead in *OS, CS*, and *HS* for different number of dependencies in the late phase of simulation. We see from the results that overall performance of *HS* is better than *CS* and *OS* over a broad range of dependencies. Another observation is that increasing the number of dependencies has an adverse effect on *OS* as compared the *CS* and *HS*. The reason is that increasing the number of dependencies causes an increase in the number of conflicts/aborts, and the cost of aborts in the case of violations is high. We also studied the abort behavior (for *OS* and *HS*) and blocking behavior (for *CS* and *HS*); Table 3 illustrates our results. We observe that the number of aborts in *HS* is significantly lower compared to *OS* because it moves toward a more conservative approach when the number of aborts increases. Table 3 also illustrates lower blocking times for *HS* (as compared to *CS*) since *HS* makes better decisions on when to delay execution. Figure 12(b) and (c) capture the behavior of the three different simulators when executed using the *OS, CS*, and *HS* policies. A key aspect to note is that the synchronization overhead in LTESim is very high when *OS* is used. It is because LTESim is an event-based simulator with a large number of external events that access the interdependent data items (e.g. throughput); this results in a high number of aborts when *OS* is used. This shows that the synchronization approach chosen must take into account the type of simulator and number of dependencies.

(3) Applying relaxations to different meta-scheduling techniques: Table 4 summarizes the results for different approaches with and without using relaxations. Appling relaxations results in better performance for all three simulators for all of the approaches. In particular, note the significant difference between OS and OSR (see the highlighted cells in Table 4). The dramatic reduction is due to the fact that applying relaxations decreases the number of aborts when optimistic scheduling is used. The following are some key observations from our experiments:

- *HS* exhibits superior overall performance to other approaches;
- the choice of the approach is also dependent on the simulator, e.g. for event-based simulators when the

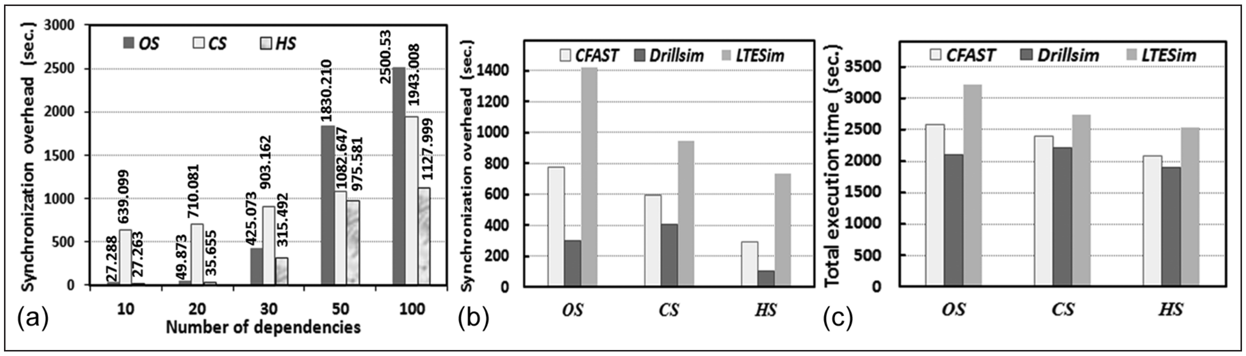


Figure 12. (a) Synchronization overhead versus the number of dependencies, (b) synchronization overhead, and (c) total execution time of *CFAST, DrillSim*, and *LTEsim*. (Number of dependencies =100.)

Table 3. Number of aborts and blocking time in *OS, HS*, and *CS*.

Number of dependencies	*OS* *Number of aborts*	*HS* *Number of aborts*	*CS* *Blocking time*	*HS* *Blocking time*
50	64	19	1435.674	492.273
100	223	71	1812.008	603.018

Table 4. The results for synchronization overhead (*Synch. overhead*) and total execution time (*Total time*).

Strategy	*Metric*	CFAST	Drillsim	LTESim	Total
OS	*Synch. overhead*	778.364	301.182	1420.091	2499.637
	Total time	2521.501	2112.796	3218.013	7852.310
OSR	*Synch. overhead*	353.856	296.341	825.120	1475.317
	Total time	2149.009	2101.772	2629.836	6880.617
CS	*Synch. overhead*	592.228	404.293	946.432	1943.013
	Total time	2387.793	2208.015	2751.959	7347.767
CSR	*Synch. overhead*	510.376	393.634	883.812	1787.882
	Total time	2308.651	2195.409	2677.111	7181.171
HS	*Synch. overhead*	292.283	104.293	731.423	1127.999
	Total time	2088.802	1906.548	2529.580	6524.930
HSR	*Synch. overhead*	201.598	98.781	423.923	724.302
	Total time	2003.656	1897.523	2224.612	6125.791

number of events is large we need to avoid using *OS*;

- checkpointing/rollback mechanisms and using relaxations always results in better performance regarding synchronization overhead and total execution time.

6. Conclusions and future work

When we examine the nature of our problem and challenges to facilitate the process of fusing together the independently created simulators into an integrated simulation environment, we note properties of simulation integration which parallel certain concepts in multidatabase systems and transaction processing. In multidatabase systems the individual database management systems need to be integrated in a single unified database environment while they desire to preserve the autonomy of the local database management systems. Using concepts from multidatabase systems, we make it feasible to semi-automatically compose simulation models via a looser coupling approach. In order to participate in the federation, wrappers are written for each simulator that enable us to intercept and control the execution of individual simulators in order to ensure effective and correct composite simulation models. We discussed a hybrid scheduling strategy for multisimulation synchronization which works based on the state of the execution and underlying dependencies. We also presented a

Table 5. Comparison between HLA and multisimulation architecture.

Criterion	Multisimulation architecture	HLA	Web services technologies
Objective	• Semantic Interoperability, • Reusability • Flexibility	• Interoperability • Reusability[16]	• Interoperability Reusability among heterogeneous simulation components in a distributed environment[33]
Domain	• Flexible via use of domain ontologies	• Defense[17]	• Different applications[32]
Formal specifications	• Maude[52]	• DEVS[21]	• Multiple APIs but no formal specifications[29,31]
Complexity	• No need to conform the internal properties • Semantic constraints implemented at the metalevel[42]	• Low-level knowledge needed • Lack of semantic interoperability[16]	• Lack of semantic interoperability • Lack of safety and stability[36]
Time management	• Optimistic, Conservative, and Hybrid methods • Relaxed dependencies	• Optimistic and conservative methods[50]	• Appropriate for coarse grain, low data update frequency, or non-real-time requests[35]
Separation of concerns	• Separate concerns related to simulation domain to those related to integration mechanisms	• Merges domain-specific and integrated simulation aspects[17]	• Merges domain-specific and integrated simulation aspects

relaxation model for dependencies across multiple simulators which guarantee bounded violation of consistency to support higher levels of concurrency. To ground our work in reality, we have developed a detailed case study from the emergency response domain using multiple real-world simulators. Table 5 presents a representation of multisimulation architecture properties in comparison with HLA-based architecture and Web Services technologies. Ongoing work includes development of adaptive plans for chackpointing and rollback mechanism. Future research will focus on addressing challenges in the complexity associated with generalizing the meta-models for simulators, integrating simulators in other domains including earthquake and transportation simulators, and addressing the challenges of data transformation.

Funding

Support of this research by an NSF grant, CYPRESS, and Sensorium awards (numbers 1063596, 106356, and 1059436) is gratefully acknowledged.

References

1. HAZUS-MH. Multi-hazard Loss Estimation Methodology. *User Manual*, 2003.
2. Cho S, Huyck CK, Ghosh S and Eguchi RT. Development of a web-based transportation platform for emergency response. In: *8th Conference on Earthquake Engineering*, 2006.
3. CAPARS. The Computer-assisted Protective Action Recommendation System, 1995. http://www.alphatrac.com/PlumeModelingSystem
4. Abanades A, Sordo F, Lafuente A, Martinez-Val JM and Munoz J. Application of CFD codes as design tools. In: *5th Conference on ISFA*, 2007.
5. Peacock R, Jones W, Reneke P and Forney G. CFAST–Consolidated Model of Fire Growth and Smoke Transport (Version 6) User's Guide, NIST Special Publication, 2005.
6. Cooper LY and Forney GP. The consolidated compartment fire model (CCFM) computer code application CCFM. VENTS - Part I: Physical basis. NISTIR 4342, 1990.
7. Balasubramanian V, Massaguer D, Mehrotra S and Venkatasubramanian N. DrillSim: A Simulation Framework for Emergency Response Drills. In: *ISI*, 2006, pp. 237-248.
8. De Silva FN and Eglese RW. Integrating simulation modeling and GIS: spatial decision support systems for evacuation planning. *JORS* 2000; 51(4): 423–430.
9. VISSIM, Planung Transport Verkehr AG, ''Vissim User Manual V3.61'', Germany, 2001.
10. Cameron G, Wylie B and McArthur D. PARAMICS - moving vehicles on the connection machine. In: *Conference on High Performance Networking and Computer*, 1994, pp. 291–300.
11. Maglio P, Cefkin M, Haas P and Seninger P. Social factors in creating an integrated capability for health system modeling and simulation. In *SBP 2010*. New York: Springer, 2010.
12. Pope A. *The SIMNET Network and Protocols*. Technical Report 7102, BBN, 1989.
13. Davis PK. Distributed interactive simulation (DIS). *Proc IEEE* 1995; 83(8): 1138–1155.
14. Weatherly R, Seidel D and Weissman J. Aggregate level simulation protocol. In *Summer Simulation Conference*, 1991.
15. Kuhl F, Weatherly R and Dahmann J. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1999.

16. Boer C, Brulin A and Vebraeck A. Distributed simulation in industry- a survey: part 2 - - experts on distributed simulation. In: *Winter Simulation Conference*, 2006, pp. 1061-1068.
17. Boer C, Bruin A and Vebraeck A. Distributed simulation in industry - a survey, part 3- the HLA standard in industry. In: *Proceedings of the 40th Winter Simulation Conference*, 2008, pp. 1094-1102.
18. Zhang F and Huang B. HLA-based network simulation for interactive communication system. In *Proceedings of the First Asia International Conference on Modelling & Simulation*, 2007, pp. 177–180.
19. Hardebolle C and Boulanger F. ModHel'X: a component-oriented approach to multi-formalism modeling. *MODELS* 2007; 5002(2008): 247-258.
20. Klein U, Straburger S and Beikrich J. Distributed Simulation with JavaGPSS based on the high level architecture. *Proceedings of the International Conference on Web-based Modeling & Simulation*, 1998.
21. Zeigler BP, Ball G, Cho HJ, Lee JS and Sar-joughian H. Implementation of the DEVS formalism over the HLA/ RTI: problems and solutions. In *SIW*, SISO, 1999.
22. Zacharewicz G, Frydman C and Giambiasi N. Mapping PIOVRA in GDEVS/HLA environment. In *Summer Simulation Multiconference*, 2007, pp. 1086–1093.
23. Tolk A and Diallo SY. Using a formal approach to simulation interoperability to specify languages for ambassador agents. In *Proceedings of the Winter Simulation Conference*, 2010, pp. 359-370.
24. Hemingway G, Neema H, Nine H, Sztipanovits J and Karsai G. Rapid synthesis of high level architecture-based heterogeneous simulation: a model-based integration. *Simulation* 2011; 88(2): 217–232.
25. Wenguang W, Wenguang Y, Qun L and Weiping W. Service-oriented high level architecture. In *European Simulation Interoperability Workshop*, 2008, 08E-SIW-022.
26. Jiang Z, Huang J and Huang K. HLA/RTI simulation system extension based on Web Services. *Adv Manuf Management* 2008; 27(1): 39-51.
27. Zhou X, Wei J, Li P and Su Q. Design and implementation of distributed RTI based on Web Services. *J Syst Sim* 2008; 20(8): 2064-2067.
28. Shucai T, Tianyuan X and Wenhui F. A collaborative platform for complex product design with an extended HLA integration architecture. *Sim Modell Practice Theory* 2010; 18: 1048-1068.
29. Moller B. SMixing service oriented and high level architectures in support of GIG. In *Proceedings of the 2005 Fall Simulation Interoperability Workshop*, 2005.
30. Moller B. SA management overview of the HLA Evolved Web Service API. In *Proceedings of the 2006 Fall Simulation Interoperability Workshop*, 2006.
31. Moller B and Dahlin C. A first look at the HLA Evolved Web Service API. In *Proceedings of the 2006 European Simulation Interoperability Workshop*, 2006.
32. Dragoicea M, Bucur L, Tsai W and Sarjoughian H. Integrating HLA and service-oriented architecture in a simulation framework. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012, pp. 861-866.
33. Wang H and Zhang H. An integrated and collaborative approach for complex product development in distributed heterogeneous environment. *Int J Production Res* 2008; 46(9): 2345-2361.
34. Turner M, Zhu F, Kotsiopoulos I, et al. Using Web Service technologies to create an information broker: an experience report. In *Proceedings of the 26th International Conference on Software Engineering*. Los Alamitos, CA: IEEE Computer Society Press, 2004, pp. 831-839.
35. Byrne J, Heaveya C and Byrneb P. A review of web based simulation and supporting tools. *Sim Modell Practice Theory* 2010; 18: 253-276.
36. Boyens C and Gnther O. Trust is not enough: privacy and security in ASP and Web service environments. In: Manolopoulos Y et al. (eds), *Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*. Bratislava: Springer-Verlag, 2002, pp. 8-22.
37. Jain S and McLean CR. Integrated simulation and gaming architecture for incident management training. In *Proceedings of the Winter Simulation Conference*, 2005, pp. 904-913.
38. Breitbart Y, Garcia-Molina H and Silberschatz A. Overview of multidatabase transaction management. *VLDB* 1992; 1: 181-240.
39. Garcia-Molina H, et al. *Database Systems: The Complete Book*. Englewood Cliffs, NJ: Prentice-Hall, 2002.
40. Kon F, Costa F, Blair G and Campbell RH. The case for reflective middleware. *Commun ACM* 2002; 45(6): 33–38.
41. Jalali L, Venkatasubramanian N and Mehrotra S. Reflective Middleware Architecture for Simulation Integration. In *Proceedings of ARM'09*, Urbana Champaign, IL, 2009.
42. Jalali L, Venkatasubramanian N and Mehrotra S. Middleware solutions for integrated simulation environments. In *Proceedings of the 7th Middleware Doctoral Symposium*, 2010.
43. Fujimoto MR. *Parallel and Distributed Simulation Systems*. New York: John Wiley & Sons Inc., 2000.
44. Nicol DM. Principles of conservative simulations. In *Proceedings of the Winter Simulation Conference*, 1996.
45. Jefferson D. Virtual time. *ACM Trans Program Lang Sys 1985*; 3: 404-425.
46. Smith R. Synchronizing virtual worlds: volume 3 in the Simulation 2000 series. 2000. http://www.modelbenders.com/papers/sim2000/SynchDistVW.PDF
47. Fujimoto R. Parallel discrete event simulation. *Commun ACAL* 1990; 33(10): 30-53.
48. Steinman JS. Breathing time warp. In *7th Workshop on Parallel and Distributed Simulation* (SCS Simulation Series, vol. 23), 1993, pp. 109-118.
49. Frescha A and Luthi J. Simulating rollback overhead for optimism control in time warp. In *Proceedings of the IEEE Annual Simulation Symposium*, 1990, pp. 2-12.
50. Huang J, Tung M, Hui L and Ming-Che L. An approach for the unified time management mechanism for HLA source. *Simulation* 2005; 81(1): 45-56.

51. Liu B, Yao Y, Tao J and Wang H. Implementation of time management in a runtime infrastructure. In *Proceedings of the Winter Simulation Conference*, 2006.
52. Jalali L, Talcott C, Venkatasubramanian N and Mehrotra S. Formal specification of multisimulations using Maude. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, 2012, article 22.
53. Wu K, Yu PS and Pu C. Divergence control for epsilon-serializability. In *ICDE*, 1992, 506-515.
54. Ramamritham K and Calton P. A formal characterization of epsilon serializability. *IEEE Trans Knowledge Data Eng* 7(6): 997-1007.
55. Elnozahy EN, Alvisi L, Wang Y and Johnson D. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput Surv* 2002; 34(3): 375-408.
56. LTE System Level Simulator, 2007. http://www.nt.tuwien.ac.at/about-us/staff/josep-colom-ikuno/lte-simulators.
57. McQueen D. 3GPP LTE: the momentum behind LTE adoption. *IEEE Commun Mag* 2009; 47(2): 44–45.
58. Lemos O, Bajracharya S and Ossher J. CodeGenie: a tool for test-driven source code search. In *Companion To the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications Companion*, Montreal, Quebec, Canada, 2007, pp. 917-918.
59. LIBCKPT. Libckpt: transparent checkpointing under Unix, 1995. http://web.eecs.utk.edu/~plank/plank/papers/USENIX-95W.pdf

Author biographies

Leila Jalali was born and raised in Iran and obtained her BSc in computer science (2004) from the Isfahan University of Technology. In 2007, she completed her master's degree at Amirkabir University of Technology, Tehran. She received her PhD in computer science from the University of California, Irvine (2012) in the field of data management. During her time at UC Irvine, she worked on a multidisciplinary research on the data management systems and model/simulators integration. She is the author of about 10 books, peer-reviewed scientific publications, and conference papers. Currently she is working on SAP HANA. She is specializing in the calculation engine models to provide the ability to analyze, aggregate, compare, and forecast extremely large data volumes. Prior to joining SAP, Leila worked at IBM DB2 (performance team), and she was also an intern at Stanford Research Institute (SRI) and IBM Almaden. When not working on data management problems, Leila loves spending time in nature, hiking, as well as reading books on positive psychology.

Sharad Mehrotra is a professor in the School of Information and Computer Science at the University of California, Irvine. Prior to joining UCI, he was a member of the faculty at University of Illinois, Urbana-Champaign in the Department of Computer Science, where he was the recipient of the C. W. Gear Outstanding Junior Faculty Award. He also served as a scientist at Matsushita Information Technology Laboratory immediately after graduating with a PhD from the University of Texas at Austin (1988–1993). His research expertise is in data management and distributed systems areas, in which he has made many pioneering contributions.

Nalini Venkatasubramanian is a professor of computer science at the University of California, Irvine. She is known for her work in effective management and utilization of resources in the evolving global information infrastructure. Her research interests are networked and distributed systems, Internet technologies and applications, ubiquitous computing, and urban crisis responses. Born and raised in Bangalore, she received her PhD in computer science from University of Illinois, Urbana-Champaign, in 1998.