Formal Specification of Multisimulations using Maude

Leila Jalali[1] jalali@uci.edu
Carolyn Talcott[2] clt@csl.sri.com
Nalini Venkatasubramanian[1] nalini@ics.uci.edu
Sharad Mehrotra[1] sharad@ics.uci.edu
[1] University of California, Irvine, Department of Computer Science
[2] SRI International, Computing Science Lab

Keywords: Formal Specifications, Dependency, Maude, Integration, Reflection.

Abstract

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 pulling all of independently created models together into an integrated simulation environment wherein we can model and execute complex scenarios involving multiple simulators. In this paper, we describe the use of the rewriting logic based Maude tool to specify and analyze such an integrated simulation environments (multisimulations). We discuss the representation of the underlying multisimulation concepts and describe the use of Maude capabilities to analyze multisimulations. We also discuss the use of Maude's reflective capability. The idea of multisimulations specifications using Maude opens up an exciting new world of challenging applications for formal methods in general and for rewriting logic based formalisms in particular.

1. INTRODUCTION

Building complex simulations to understand the joint effect of multiple phenomena (spread of hazardous material as a result of an earthquake, impact of obesity on health issues in a society) is very useful. A variety of simulators; e.g., loss estimation tools (HAZUS [13], INLET [14], CAPARS [15], CFD [12]), fire spread simulators (CFAST [10], CCFM [19]), activity simulators (Drillsim [9], SDSS [11]), transportation simulators (VISIM [17], PARAMICS [18]) etc., that model different aspects of disasters, their impacts, and response processes have been developed. For example, in domains such as emergency response where response plans and methods are validated by simulating disasters and their impact on people and infrastructure. Since these simulators are typically developed by domain experts who have an in-depth understanding of the phenomena being modeled and their evolution, they are designed to be executed and evaluated independently. Consider 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 study its impact on the evacuation process as captured within an evacuation simulator, e.g. Drillsim. Similarly, the progress of fire (captured by CFAST) may create infeasible paths/exits for evacuation (as captured by Drillsim). An integrated and concurrent execution of the two simulators is essential to understand the adverse impacts caused such as increase in evacuation times or increased exposure to undesired particulates. Such what-if analyses can enable intelligent decision making to improve the outcome of the response.

A grand challenge is to facilitate the process of pulling all of independently created simulators into an integrated simulation environment wherein we can model and execute complex scenarios involving multiple simulators. We refer to such an integrated simulation environment as a *multisimulation* [21], [22], [23]. To enable a system designer to reason about the overall correctness of a multisumilation, it becomes more important to have clear formulation and semantic modelling and to be able to carry out a variety of analyses based on these modelling in order to increase assurance of correct and expected behaviour. In this paper, we propose specifications to formally specify multisimulations and to enable a system designer to reason about the overall correctness of multisumilations. We describe the use of the rewriting logic based Maude tool [16] to specify and analyse multisimulations. Formulation and modeling as presented here can serve as a valuable form of documentation of systems designs and implementation decisions. It also provides a systematic mechanism to relate multisimulation requirements and the design/implementation descriptions.

In section 2, we discuss some background efforts in the area of simulation integration. In Section 3, we explain the need for formal specifications of multisimulations. In Section 4, we explain how the rewriting logic based Maude language can formalize multisimulations and we illistrate the multisimulations specifications in Maude. We use the relationship between smoke level and health conditions as a running example. In sections 5, we discuss a case study that integrates multiple real world simulators using our proposed approach and techniques. Finally we draw conclusions.

2. RELATED WORKS

To best of our knowledge, simulation integration has been studied in two domains – (a) military command-and-

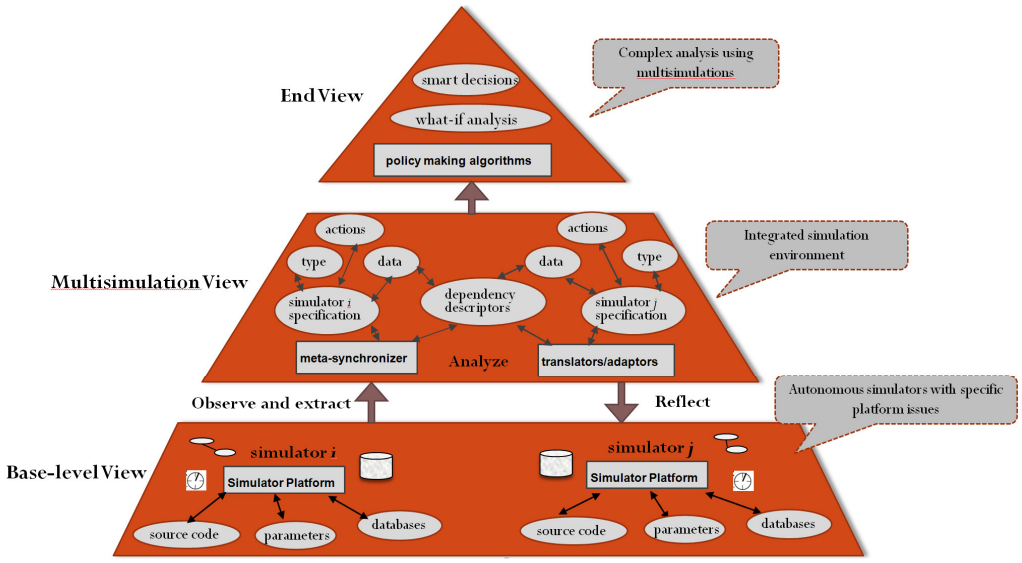

Fig. 1. Pyramid showing summary of specifications. The three levels indicates (a) a detailed base-level view that specifies simulators, (b) a multisimulation wide view with integration modules and abstract properties extracted from the underlying base-level simulators, and (c) the end view point from the perspective of the complex analysis using underlying multisimulations.

control [4] and (b) games and virtual environments [6]. At present, the universal framework for simulation integration is mainly referred to the high level architecture (HLA) standard [4]. The informal description of the HLA standard brings a lot of problems and limitations for not only the comprehension of system documentation, but also for developing new applications [5]. Recently, there are efforts intend to bridge the gap between HLA and current modeling formalisms [2], [3], [8]. The formal description of the HLA permits to highlight its key characteristics and limitations.

While HLA is suited to developing new simulators that can be easily integrated, its broader applicability to combine pre-existing simulators is questionable [20]. It is a complex standard designed specifically for the military domain 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).

In contrast to prior work on simulation integration, 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 which can hinder leveraging prior work. Unlike the significant code rewrite required in other approaches, our framework permits individual simulators to maintain their autonomy (i.e. retain their internal representations of time/state etc.), thereby avoiding the need for rigid common interfaces across simulators.

In particular, in this paper, we explore the use of Maude as a formal specification framework for writing multisimulatins specifications. Maude is an executable rewriting logic language which is well suited for specification of object oriented and distributed systems. We show how Maude can be used as a simple and accurate way of modeling multisimulations concepts which allows the specifications produced and offers a support for reasoning about multisimulations. The level of specification is detailed enough to capture essential operational aspects of multisimulations and to allow rapid prototyping and debugging by directly executing the specifications.

3. MULTISIMULATIONS FORMAL SPECIFICATION

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 autonomous simulators and platform specific issues. A reflective architecture [1] is well suited for this task since it provides mechanisms to: (i) observe changes in desired attributes of independently executing simulations at runtime (reification), (ii) analyze and evaluate whether changes in the observed attributes impact parameters in other simulators and, (iii) adapt the execution of the multisimulation by enforcing the changes back into the impacted simulators (reflection). Such an observe-analyze-adapt approach is at the crux of a reflective architecture for integrated simulation environments.

Figure 1 shows pyramid of specifications. The first level is a detailed base-level view that specifies simulators

including each simulator's specific platform, source code, databases and parameter models. The second level (meta-level) describes a multisimulation wide view with abstract properties extracted from the underlying components (e.g. inter-simulator's dependencies) and modules need for the integration. Meta-level is built on top of base-level entities to describe the base-level entities and the relationships between them (e.g. a simulator tuple to describe simulators properties, and a dependency descriptor to specify a dependency between simulators). Meta-level entities are captured in a pre-processing step using the base-level simulators. A variety of coordination and interaction mechanisms can be modeled using meta-level entities including synchronization and data exchange. We call these mechanisms meta-level modules. Meta-level modules are part of the run time system which ensure the correctness of multisimulation and controls the run-time behavior of base-level simulators. The third level specifies the end view point from the perspective of the services and tools provided on top of multisimulations for what-if analysis and smart decision making using multiple simulators and data sources.

Why use formal specification in Maude? To assure correct observe-analyze-adapt cycle in dynamically changing simulation models, it is important to have a rigorous semantic model of multisimulations: the base-level simulators, the data sources, the middleware that provide the coordination among multiple simulators (e.g. meta-level entities and modules), and the sharing and interactions among these elements. Using such a formal model, designs can be analyzed to clarify assumptions that must be made for the correct integration, and to establish consistent multisimulations based on assumptions.

In this paper, we specifically deal with the formulation and modeling that describe different components of multisimulation. In our approach, we describe a semantic model for specifying and reasoning about multisimulations and we show how different meta-level modules (e.g. synchronizer) can be used to coordinate the consistent flow of information between multiple simulators. From a higher level point of view, we specify the end to end relationship to provide a service in response to a request from a simulator (e.g. a request to get input parameters). From a low-level point of view we specify the properties of the dependencies between simulators (e.g. to reflect the updates from one simulator into another simulator).

Although our framework has been developed in Java, we used the rewriting logic language Maude for formal specification of multisimulations, because a logic-based language is better suited to deal with constraints, reflection, and the planning process. The reflective capability of Maude makes it well suited to programming simulators' interactions in multisimulations that has been crucial in our studies to date. Maude specifications can be executed using the Maude rewrite engine which allows their use for system prototyping and debugging of specifications. In the following sections, we discuss the representation of the underlying multisimulation concepts and describe the use of model checking and reasoning capabilities of Maude to analyze multisimulations. The idea of multisimulations specifications using Maude opens up an exciting new world of challenging applications for formal methods in general and for rewriting logic based formalisms in particular.

4. FORMAL SPECIFICATION USING MAUDE

In this section, we provide some fundamental concepts to specify simulators and components of multisimulation and describe how the concepts can be presented in Maude. We want to provide a formalism that gives us the structure to have simulation knowledge without committing to a particular simulator or its internal properties (e.g. time advancement). Our formalism provides a semantic model for specifying and reasoning about multisimulations.

4.1. Rewriting Logic and Maude

Maude is a language and a system based on rewriting logic that supports membership equational logic and rewriting logic specification and programming of systems [16]. Rewriting logic is a flexible and expressive logical framework that can naturally deal with state and with highly nondeterministic concurrent computations.

In Maude, object-oriented systems are specified by object-oriented modules in which classes and subclasses are declared. A class definition $C|a_1: S_1, \dots, a_n: S_n$ defines a class of name C with attribute name a_i and their corresponding sorts S_i. Objects of a class C are record-like structures of the form $< O: C|a_1: v_1, \dots, a_n: v_n >$, where O is the name of the object, and v_i are the values of its attributes. Objects can interact in a number of different ways, including messages. Messages are declared in Maude in *Msg* clauses, in which the syntax and arguments of the messages are defined.The concurrent state of a system is called a *configuration* that has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting using rules that describe the effects of the communication events of objects and messages. The general form of such rewrite rules is:

$crl[r]$:

$$M_1 \dots M_m < O_1: C_1|atts_1 > \cdots < O_n: C_n|atts_n > \\ => < O_{i_1}: C'_{i_1}|atts'_{i_1} > \cdots < O_{i_k}: C'_{i_k}|atts'_{i_k} > \\ < Q_1: C''_1|atts''_1 > \cdots < Q_p: C''_p|atts''_p > \\ M'_1 \dots M'_q$$

$if\ Cond$.

where r is the rule name, $M_1 \dots M_m$ and $M'_1 \dots M'_q$ are messages, $O_1 \dots O_n$ and $Q_1 \dots Q_p$ are object identifiers, $C_1 \dots C_n$, $C'_{i_1} \dots C'_{i_k}$ and $C''_1 \dots C''_p$ are classes, $i_1 \dots i_k$ is a subset of $1 \dots n$, and $Cond$ is a Boolean condition. The result of applying such a rule is that the messages are consumed

and the state and possibly the classes of objects $O_{i_1} \dots O_{i_k}$ may change and new messages $M'_1 \dots M'_q$ and new objects $Q_1 \dots Q_p$ are created.

4.2. Modeling Multisimulations in Maude

In this section we describe how multisimulations can be presented in Maude. A multisimulation, MS, consists of a set of autonomous pre-existing simulators, S_1, S_2,…, S_n that execute concurrently in an integrated environment. Before we develop the multisimulation model, we first need to define the concept of a simulator and develop it using Maude. We consider each simulator as a *simulator template*. Simulator templates will be presented by Maude classes. In Maude, each class is defined by a name and a set of attributes (of certain sort) that describe the simulators. Each simulator will be then represented by Maude objects. Each object belongs to a class and it may change during its lifetime. All simulator templates will inherit from class `Sim-Oid`, which described the common features that any simulator exhibits.

```
Class Sim-Oid | conf : configuration .
```

Predefined sort configuration allows us to store configuration of Maude objects and messages. Attribute `conf` store a set of Maude objects representing the properties of multisimulation. Each simulator is modeled as a three tuple $S_i =< T_i, A_i, D_i >$ where T_i is the type of the simulator, D_i is the data items that the simulator reads or updates, and A_i is the set of actions executed by the simulator. We consider the type of the simulator based on the time management mechanism that they employ as time stepped or event based [7]. 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, $E = \{e_m | m = 1,2, \dots\}$, each event has a time stamp (usually causality preserving) and the simulation time jumps from one event time stamp to the next. For every two events e_a and e_b we have the following property: $timestamp(e_a) \leq timestamp(e_b)$ when $a \leq b$. In our notation, $T_i \in \{TS, EB\}$, where *TS* and *EB* correspond to time stepped and event based simulators respectively. We represent T_i by an operation `clock` which incremented using Maude rules. D_i is the data items that the simulator reads or updates. 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. In the following example, `smokeHigh` represents the smoke level in the activity simulator, `ACS` with a specific assigned value using Maude variables.

```
sort Sim-Oid .
op ACS : -> Sim-Oid . *** Activity Simulator
op clock:_ : Int -> Attr [ctor] .
op smokeHigh:_ : Bool -> Attr [ctor] .
op f1 : Int Int -> Bool .
```

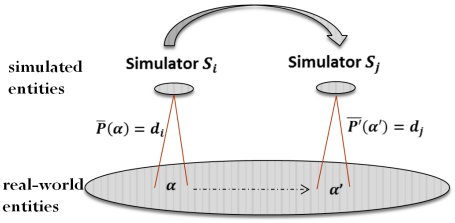

Fig. 2. Inter-simulators dependencies.

```
vars n t c : Int .
var b : Bool .
rl[TS] :
[acs : ACS | smokeHigh: b, clock: c]
=> [acs : ACS | smokeHigh: f1(n,t), clock:
(c + Δt)] .
rl[EB] :
[acs : ACS | smokeHigh: b, clock: c]
=> [acs : ACS | smokeHigh: f1(n,t), clock:
(c + timestamp(e)] .
```

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 $d \in D_i$ to a value v, where $v \in Dom(d)$. Thus a state can be expressed as a set of ordered pairs of data items in D_i and their values, $\{\Phi_i\}_{ts} = \{(d, v) | \, d \in D_i, \quad v \in Dom(d)\}$. Associated with each state is a timestamp, *ts*. Actions trigger state change; we use the notation $\{\Phi_i\}_{ts} \xrightarrow{a_k} \{\Phi'_i\}_{ts'}$ to indicate that when an action a_k in simulator S_i executes from a state $\{\Phi_i\}_{ts}$, it results in a state$\{\Phi'_i\}_{ts'}$. Therefore, the state is changed after the execution of action. A_i is the set of actions that is $A_i = \{a_k | \, k = 1, \dots, n\}$. There is a total order $<_i$ on the set of actions of a simulator. Each action, a_k, of a simulator is an atomic unit of processing that reads and modifies the data and changes simulator's state. Each 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. In a time stepped simulator it advances by Δt, $\{\Phi_i\}_{ts} \xrightarrow{a_k} \{\Phi'_i\}_{ts+\Delta t}$. In an event based simulator it advances to the time stamp of the event e_m that is executed, $\{\Phi_i\}_{ts} \xrightarrow{a_k} \{\Phi_i\}_{ts+e_m(ts)}$, where $e_m(ts)$ is the timestamp advanced by e_m. Therefore, an action a_k captures changes that occur in a clock tick (or a step) in a time stepped simulator or the execution of an event in an event based simulator.

4.2.1. Inter-simulator Dependencies

There are several challenges in building multisimulations by integrating multiple autonomous simulation models. Given the potential black-box nature of simulators developed by experts in diverse domains we believe that achieving a completely automated plug-and-play integration of simulators is a very difficult, if not infeasible challenge. Our goals are more modest – we intend to develop enabling tools that will simplify the task of simulation integration

with a wide range of simulators that vary in the degree to which they expose their interfaces and implementations. One challenge arises from the fact that each simulator in a multsimulation uses its own models and entities; these must now be integrated in the context of a single simulation. Data items in simulators are abstractions of real world entities. In order to integrate various simulators, first we need to discover and analyze the relationship between multiple simulators' data items. Consider two different simulators S_i and S_j. As depicted in Figure 2, each simulator abstracts some real world entities into its own simulated world data items (using a function, $\bar{P}$). Assume $d_i \in D_i$ represents the real-world entity α ($\bar{P}(\alpha) = d_i$), and $d_j \in D_j$ represents the real-world entity α' ($\bar{P}'(\alpha') = d_j$). There are three possible relationships between α and α':

(1) Dependent entities: α and α' are dependent entities in real world where changes in one entity can affect the other one (e.g. α represents the amount of smoke in the fire simulator and α' represents the health conditions in the activity simulator).

(2) Exact match entities: α and α' represent the exact same entity in different simulators (both α and α' represent current temperature).

(3) No match: α and α' represent two independent entities without any semantic relationship (e.g. α represents the individual's speed of walking in the activity simulator and α' represents the thickness of walls of the building in the fire simulator).

In this paper, we consider the first type of relationship, dependent entities, and capture it by the concept of inter-simulator dependency.

Dependency: Let α and α' be two real world entities with respect to the data items $d_i \in D_i$ and $d_j \in D_j$ in two different simulators, d_i is dependent on d_j denoted by $\Theta(d_i, d_j)$ if a change to α implies a change to α'.

If a change in one entity triggers also changes in the dependent one, we observe transformational dependency. The integration of various simulators results in the introduction of certain inter-simulator dependencies, which were not present prior to integration. An example of such a relationship is that smoke from fire simulator can affect someone's health in an activity simulator. We use a **dependency descriptor** to specify the transformational dependency $\Theta(d_i, d_j)$between a data item d_i in simulator S_i and a data item d_j in simulator S_j when updates on d_i need to be reflected into d_j: $S_i \xrightarrow{d} S_j =< d_i \in D_i, d_j \in D_j, f >$. Note that dependency notion is directional. S_i is the supplier simulator, S_j is the consumer simulator. In general, there can be more than one dependency between two simulators describing multiple aspects of their relationships. A ***dependency function***, *f*, defines the relationship between two data items values. Each data item has a value at any given state, $(d_i, v_i) \in \{\Phi_i\}_{ts}$ and $(d_j, v_j) \in \{\Phi_j\}_{ts'}$. At each iteration, the new value of d_j is determined by the dependency function $f: Dom(d_i) \rightarrow Dom(d_j)$, that is $v_j = f(v_i)$. For each dependency between simulators such a dependency function is defined at meta-level.

Local data vs. Interdependent data. Dependencies described by dependency descriptors enable us to partition the set of data items at a simulator, D_i, into local data items, LD_i, and interdependent data items, SD_i, such that $LD_i \cap SD_i$=∅, and $D_i = LD_i \cup SD_i$. Furthermore, if there is a $(d' \in D_j) \rightarrow (d \in D_i)$, $i \neq$j, then $d \epsilon\, SD_i$ and $d' \epsilon\, SD_j$. For example, smoke level in Fire simulator and health condition in an activity simulator are an interdependent data items.

Internal actions vs. 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. Synchronization among simulators in a multisimulation is only needed during external action processing, and we can eliminate it altogether during internal action processing periods. Internal actions, IA_i, are those actions that read or modify only the portion of data that is local to the simulator. External actions, EA_i, access at least one data item which is an interdependent data item.

Dependency sets. For each simulator, S_j, we create a dependency set that includes those simulators updating data items which are interdependent with the data items of S_i. The dependency set includes all the simulators S_i for which S_j depends upon a data item updated by S_i. That is: $Dep(S_j) = \{S_i | \exists\, S_i \xrightarrow{d} S_j =< d_i \in D_i, d_j \in D_j, f, R > s.t.\ d_i \in SD_i, d_j \in SD_j\}$. Finally, we represent all the dependency sets in the multisimulation by $DEP = \cup_{i=1}^{n} \left(Dep(S_i)\right)$.

4.2.2. Simulators interactions using Maude messages

Inter-simulator dependencies result in the need for sending updates (i.e. Muade messages) from one simulator to another simulator in order to preserve dependencies. We consider each simulator's execution as a sequence of actions (steps in time stepped simulators or events in event based simulators). We introduce *synchronizer* object that coordinates the execution of actions from multiple simulators. Thus multisimulation execution state consists of set of simulator objects `Sim-Oid` (actors), synchronizer object, and any messages pending delivery. A simulator can execute an internal action whenever enabled. However, it requests permission to execute an external action and executes when a granted message arrives, and sends updates. The strategy used in the synchronizer to address the synchronization problem in multisimulations is out of the scope of this article [23]. Simulators communicate with synchronizer by using Maude messages. For the external actions (i.e. read or modify interdependent data items) the simulator needs to communicate with meta-level and sends them synchronizer (reification). Upon receiving such

external actions from a simulator, the synchronizer generates messages to notify dependent simulators (reflection). Figure 3 shows the sequence diagram of the interactions between simulator templates and synchronizer. Consider a simulator S_i in state $\{\Phi_i\}_{t_i}$ 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 $(d_i, v_i') \in \{\Phi_i\}_{t_i'}$. S_i posts this update and the current state's timestamp to synchronizer. Let $S_i \xrightarrow{d} S_j =< d_i \in D_i, d_j \in D_j, f >$ be a dependency. A meta-action $<ma(d_i, d_j), t_i' >$ is an action generated by the synchronizer as the result of an external action, $\{\Phi_i\}_{t_i}$ a_k $\{\Phi_i\}_{t_i'}$, executed in a supplier simulator, S_i, performing an update on an interdependent data item d_i. Associated with each meta-action is a timestamp which is the timestamp of the state of S_i that contains the new value of d_i after the update performed by a_k. In our reflective architecture, the updates are reified from supplier simulators and represented at meta-level using meta-actions. A wrapper action, $<wa(d_j), t_r >$, is the action taken by consumer simulator's wrapper (S_j's wrapper), to reflect the update of the corresponding meta-action into d_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.

We describe the set of all messages by Msg which includes messages between simulators and synchronizer (e.g. request to execute an action, allow). In the following example, fire simulator sends the smoke level to activity simulator using a Maude operation to create a message `smokeLevel`. This message includes the amount of smoke and the time in fire simulator, both defining as integers.

```
op smokeLevel : Sim-Oid Sim-Oid Int Int ->
Msg [ctor] .
rl[FS] :
[fs : FS | asim: acs, smoke: n, temp: t]
=> [fs : FS | asim: acs, smoke: n, temp: t]
smokeLevel(fs,acs,n,t) .
rl[AS] :
smokeLevel(fs,acs,n,t)
[acs : ACS | smokeHigh: b, clock: c]
=> [acs : ACS | smokeHigh: f1(n,t), clock:
(c + 1)] .
```

4.2.3. Multisimulations configuration

A Multisimulation *configuration*, C, represents a snapshot of the multisimulation state. It includes the states of all base-level simulators ($\{\Phi_i\}_{ts}$ represents the recent state of S_i), dependency sets of each simulator ($Dep(S_i)$), and a set of messages (Msg): $C^x = (\bigcup_{i=1}^{n}\{\Phi_i\}_{ts}) \cup (\bigcup_{i=1}^{n} DEP) \cup Msg$. Figure 4 shows an example of multisimulation configuration, C^x. The multisimulation includes two simulators S_i and S_j each has their own state,

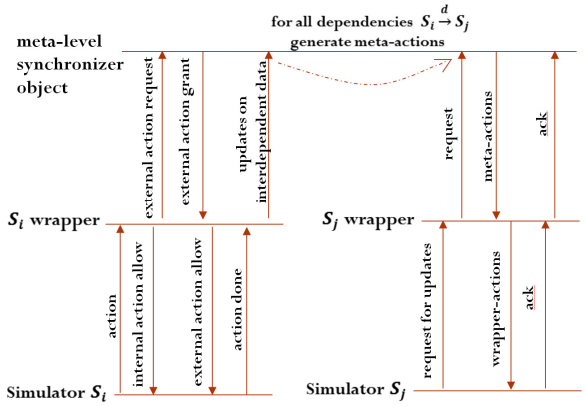

Fig. 3. Simulators interactions.

represented by $\{\Phi_i\}_{t_i}$ and $\{\Phi_j\}_{t_i}$ respectively. There exists a dependency between smoke level S_i in and health condition S_j in specified by $S_i \xrightarrow{d} S_j =< smoke \in D_i, health \in D_j, f >$. As a result of an update on smoke level by S_i a message is generated $<ma(smoke, health), t_i >$. The message has been sent to S_j to actually reflect the into update health condition. When the update performed an ack will be sent to the meta-level synchronizer. In this example, the actual update is not performed yet. Therefore the message is included in the configuration.

Configurations change by transitions from one configuration to the next configuration. In other word, each transition results in a new multisimulation configuration. Formally a transition τ has the form $\tau: C^x \xrightarrow{l} C^{x+1}$ where l is the label of the trigger that initiates the transition. We define a set of trigger labels, L, for transitions among different configurations:

$L = \{ \boldsymbol{REQ}(S_i, a_k), \boldsymbol{UPD}(a_k, ma(d_i, t_i')), \boldsymbol{GET}(S_j, t_j), \boldsymbol{ACK}(S_j, < wa(d_j, t_r) >, \boldsymbol{MDF}(Dep)\}$

These labels work as the rules in the multisimulation framework:

- $\boldsymbol{REQ}(S_i, a_k)$ indicates that S_i requests to execute an external action a_k. (i.e. on an interdependent data item).
- $\boldsymbol{UPD}(a_k, ma(d_i, t_i'))$ indicates that an external action executed by a simulator at time t_i' which results in the change of the state of the simulator and sending the updates on an interdependent data item (d_i) to meta-level.
- $\boldsymbol{GET}(S_j, t_j)$ indicates a request at time t_j from a simulator, S_j to receive the updates.
- $\boldsymbol{ACK}(S_j, wa(d_j, t_r))$ indicates that S_j executed corresponding wrapper-actions to reflect the updates.
- $\boldsymbol{MDF}(Dep)$ indicates the updates on the dependencies among multiple simulators (i.e. remove a dependency, add a new dependency).

We can have sequence of transitions, π, in multisimulations: $\pi = [\, C^x \xrightarrow{l} C^{x+1} | x \in \mathbb{N}]$ in which each

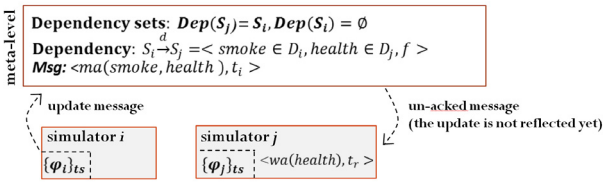

Fig. 4. Multisimulation configuration.

transition has a trigger label. An example of such a sequence is (Fig. 5):

$$\pi = C^0 \xrightarrow{REQ(S_i,a_k)} C^1 \xrightarrow{UPD(a_k,ma(d_i,t_i'))} C^2 \xrightarrow{GET(S_j,t_j)} C^3 \xrightarrow{ACK(S_j,wa(d_j,t_r))} C^4.$$

The sequence specifies an external action performed at a supplier simulator S_i which results in sending the updates to the consumer simulator S_j. When the consumer simulator S_j requests to receive the meta-actions, $GET < S_j, t_j >$, meta-actions will be sent to S_j which results in generating the correspondent message in C^2 to actually reflect the updates to the target data. When the updates applied the ack will be sent to the meta-level, $ACK(S_j, wa(d_j, t_r))$, and the message will be removed from the configuration in C^4.

4.3. Using the Reflective Capability of Maude

There is a variety of metadata associated with the executable model of a multisimulation. This includes information justifying or qualifying a rule and rate information about a rule. Furthermore, as the multisimulation system grows, consistency checks become more important and it is useful to be able to answer simple questions such as "What are all the rule labels?" or "What constants of sort `Sim-Oid` have been declared?". Using the reflective capability of Maude make it easy to formalize such metadata and use it to further control experiments or to answer questions and carry out various meta-analyses. Rather than giving here a detailed explanation of the representations of terms and modules for which we refer the reader to [16] we instead illustrate them with simple examples. There are a number of important metalevel functions that has been efficiently implemented in the `META-LEVEL` module. For our purposes, we focus on the functions `meta-reduce` and `meta-apply`. The function `meta-reduce` returns the representation of fully reduced from a term using the equations as a parameter. The most important such function for our purposes is the `meta-apply` function, that simulates the application of a given rewrite rule to a term. We use `meta-reduce` and `meta-apply` as basic strategy expressions and then extend the module `META-LEVEL` by additional strategy expressions and corresponding semantic rules.

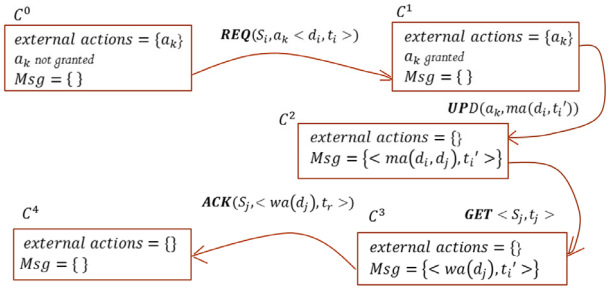

Fig. 5. Example of a sequence of configurations.

In particular, we can analyze the behavior of a module such as the specification of interactions by writing an adequate strategy that will explore all the behaviors from an initial state up to termination. One correctness criterion in multisimulations is checking that in all execution sequences all messages arrive to their destinations. One can easily write a predicate to check this requirement.

5. INTEGRATING REAL-WORLD SIMULATORS

To ground our work in reality, we develop a case study using three pre-existing real world simulators: Drillsim [9], CFAST [10], and LTESim [24]. Table 1 summarizes the three simulators and their properties. 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 activity simulator, Drillsim, which has impacts on communication patterns in communication 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. There are 3 key modules developed at meta-level: (a) a *Meta-Synchronizer* which uses the proposed approaches to monitor and control concurrent execution in the multisimulation, (b) an *Analyzer* which analyzes the interactions between simulators using meta-models to capture the dependencies, (c) which

Table 1. Three Real-World Simulators.

Activity Simulator	♦ DrillSim [9],time-stepped, open source (Java) ♦ Parameters: health profile, visual distance, speed of walking, num. of ongoing call, etc. Output: num. of evacuees, injuries, etc.
Communication Simulator	♦ LTESim [24], event-based, open source (Matlab) ♦ Parameters: num. of transmit and receive antennas, uplink delay, network layout, channel model, bandwidth, frequency, etc. Output: pathloss, throughput, etc.
Fire Simulator	♦ CFAST [10], time-stepped, black-box (no access to source) ♦ Parameters: building geometry, materials of construction, fire properties, etc. Output: temperatures, gas concentrations: CO2, etc.

Table 2. Comparison between HLA and Multisimulations.

Criterion	HLA	Multisimulations
Objective	– Interoperability – Reusability	– Semantic Interoperability, Reusability, Flexibility
Formal Specification	– DVES/HLA	– **Maude**
Domain	– Defense	– Flexible via use of domain ontologies [21]
Complexity	– Low level knowledge needed – Lack of semantic interoperability	– No need to conform the internal properties – Semantic constraints implemented at the metalevel [22]
Time Management	– Optimistic and conservative methods	– Optimistic, Conservative, and Hybrid methods – Relaxed dependencies [23]
Separation of Concerns	– Merges domain-specific and integrated simulation aspects	– Separate concerns related to simulation domain to those related to integration mechanisms [21]

manages the data exchange between simulators through the design of wrapper modules for each simulator. Table 2 presents a brief comparison of our approach to HLA. Our proposed architecture makes it feasible to semi-automatically compose simulation models via a looser coupling approach and provides a design that is more adaptable, flexible and easier to extend.

6. CONCLUSION

We used the rewriting logic language Maude for formal specification of multisimulations. We discuss the use of Maude's reflective capability for meta modeling and analyzing multisimulations. To our knowledge this is the first achieved architecture for simulation integration through the use of reflection and Maude. Reflection provides the mechanisms needed to access and modify the environment of a given simulator and the flexibility provided by the reflective architecture is a perfect match for the challenges in simulation integration. The reflective capability of Maude makes it well suited to programming simulators' interactions in multisimulations that has been crucial in multisimulations. 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 in multisimulations.

References

[1] Kon, F., Costa, F., Blair, G., Campbell, R.H.: The Case for Reflective Middleware, Communications of the ACM, 45(6), 33–38 (2002)

[2] Bernard P. Zeigler, Steve B. Hall and Hessam S. Sarjoughian, Exploiting HLA and DEVS To Promote Interoperability and Reuse in Lockheed's Corporate Environment, SIMULATION, pp. 73-288 (1999)

[3] Chen, J., Wu, D., Zhang, J., Oquendo, F., Formal modelling and analysis of HLA architectural style, International Journal of Modelling, Identification and Control, Vol. 9, (2010)

[4] Kuhl, F., Weatherly, R., Dahmann, J.: Creating Computer Simulation Systems: An Introduction to the High Level Architecture, New Jersey, Prentice Hall (1999)

[5] Tolk, A., Diallo, S., Y.: Using a Formal Approach to Simulation Interoperability to Specify Languages for Ambassador Agents, Winter Simulation, pp. 359-370 (2010)

[6] Jain, S., McLean, C.R.: Integrated simulation and gaming architecture for incident management training, Simulation, Proc. of the Winter Simulation, 904-913 (2005)

[7] Fujimoto, M.R.: Parallel and Distributed Simulation Systems, John Wiley Inc. (2000)

[8] Fernando J. Barros. Describing the HLA Using the DFSS Formalism. In Proceedings of AIS'2004. pp.117-127 (2004)

[9] Balasubramanian, V., Massaguer, D., Mehrotra, S., Venkatasubramanian, N.: DrillSim: A Simulation Framework for Emergency Response Drills, ISI, 237-248 (2006)

[10] Peacock, R., Jones, W., Reneke, P., Forney, G: CFAST–Consolidated Model of Fire Growth and Smoke Transport (Version 6) User's Guide, NIST Special Publication (2005)

[11] De Silva, F.N., Eglese, R.W.: Integrating Simulation Modeling and GIS: Spatial Decision Support Systems for Evacuation Planning, JORS 51(4), 423–430 (2000)

[12] Abanades, A., et al.: Application of CFD codes as design tools, 5th Conf. on ISFA (2007)

[13] HAZUS-MH: Multi-hazard Loss Estimation Methodology. User Manual (2003)

[14] Cho, S., Huyck, C.K., Ghosh, S. Eguchi, R.T.: Development of a Web-based Transportation Platform for Emergency Response. 8th Conf. on Earthquake Eng. (2006)

[15] CAPARS: http://www.alphatrac.com/PlumeModelingSystem

[16] Clavel, M., Dur´an, F., Eker, S., Lincoln, P., Mart´ı-Oliet, N., Meseguer, J., Talcott, C.: All About Maude- A High-Performance Logical Framework. Number 4350 in Lecture Notes in Computer Science. Springer (2007)

[17] Verkehr, A.: VISIM V3.6 Innovative Transportation (2001)

[18] Cameron, G., Wylie, B., McArthur, D.: PARAMICS- Moving Vehicles on the Connection Machine, Conf. on High Performance Networking and Computer, 291– 300 (1994)

[19] Cooper, L.Y, Forney, G. P.: The consolidated compartment fire model (CCFM) computer code application CCFM.VENTS - Part I: Physical basis. NISTIR 4342. (1990)

[20] Boer, C., Bruin, A., Vebraeck, A.: Distributed Simulation in Industry - a survey, part 3- the HLA standard in industry, Proc. of the 40th Conf. on Winter Sim., 1094-1102 (2008)

[21] Jalali, L., Venkatasubramanian, N., Mehrotra, S.: Reflective Middleware Architecture for Simulation Integration, ARM'09, Urbana Champaign, Illinois (2009)

[22] Jalali, L., Venkatasubramanian, N., Mehrotra, S.: Middleware Solutions for Integrated Simulation Environments, The Proc. of 7th Middleware Doctoral Symposium (2010)

[23] Jalali, L., Venkatasubramanian, N., Mehrotra, S.: Interoperability of Multiple Autonomous Simulators in Integrated Simulation Environments, Spring SIW'11 (2011)

[24] LTE System Level Simulator: http://www.nt.tuwien.ac.at/about-us/staff/josep-colom-ikuno/lte-simulators/