A Reflective Middleware Architecture for Simulation Integration

Leila Jalali
Department of Computer Science
University of California, Irvine
CA 92697 USA
jalalil@uci.edu

Nalini Venkatasubramanian
Department of Computer Science
University of California, Irvine
CA 92697 USA
nalini@ics.uci.edu

Sharad Mehrotra
Department of Computer Science
University of California, Irvine
CA 92697 USA
sharad@ics.uci.edu

ABSTRACT

This paper presents a reflective middleware architecture for simulation integration based on structural reflection and metamodel concepts. The proposed architecture extracts the simulator information as metamodels from the base-level simulators, determines the required features and modules using semantic constraints, and reflects the modified features to the base- level. It is shown that the reflective middleware architecture addresses various challenges in simulation integration. It also enables a design that is more adaptable, flexible and easier to extend. We present a detailed case study from the emergency response domain, where simulations are critical, to illustrate the potential benefits of applying the proposed architecture.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures Patterns (Reflection)

General Terms

Design

Keywords

Reflective Middleware, Simulation Integration, Structural Reflection, Metamodel

1. INTRODUCTION

Modeling and simulation (M&S) are accepted problem solving methodologies for the solution of many real-world problems. There is a wide variety of methods for studying models of real-world systems using software designed to imitate the system's operations or characteristics, often over time. There are several advantages of using simulations instead of experimenting with the real system itself: simulation is cheaper, quicker, and enables what-if analyses for better system design [1, 12]. This is particularly true in domains such as emergency response where the need to simulate disasters, their impact and the efficacy of response methods are often validated via simulations. These simulations (e.g. earthquake simulators, fire simulators) are often developed by domain experts who have a clear understanding of the field and are often difficult to recreate.

Building complex simulations to understand the joint effect of multiple phenomena (spread of hazardous material as a result of an earthquake, impact of earthquake on the cellular network in a region) is useful. Consider an existing simulator in the crisis management domain, HAZUS-MH, which estimates losses from potential hazards such as earthquake, wind, flood or release of hazardous material [18]. HAZUS-MH can be used in a scenario in which a hazardous material has been released as a consequence of an earthquake. In this scenario, authorities may wish to block traffic, the blocked roads may be captured within a transportation simulator, e.g. the INLET (Internet-based Loss Estimation Tool) transportation simulator [19]. Such integration is useful to 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 time. In the example above, one can study the actual effect of blocked roads on the number of people affected by the hazmat release. Additionally, traffic loads in the blocked area are reduced enabling emergency response teams to reach the crisis sooner.

One approach to studying joint impact is to build monolithic simulations; this is a cumbersome process; economic and organizational constraints make it infeasible to build these complex distributed simulations entirely from scratch. The second approach is to leverage existing simulators (developed by experts); since each simulator has its own models and entities, the integration is a big challenge. In this paper we focus on using reflective middleware solutions for integration and enhancement of simulations built by domain experts that are likely available. We design a reflective architecture for simulation integration in which interoperability of different simulators can be ultimately achieved via shared metadata. In Section 2, we discuss the related work in simulation integration and its limitations. In Section 3, we describe the reflective architecture, challenges, and our meta-model. We illustrate the power of the reflective model using real world case studies in Section 4 and compare the reflective approach with a popular simulation integration framework (the High level architecture). Finally we conclude in section 5 with future research directions.

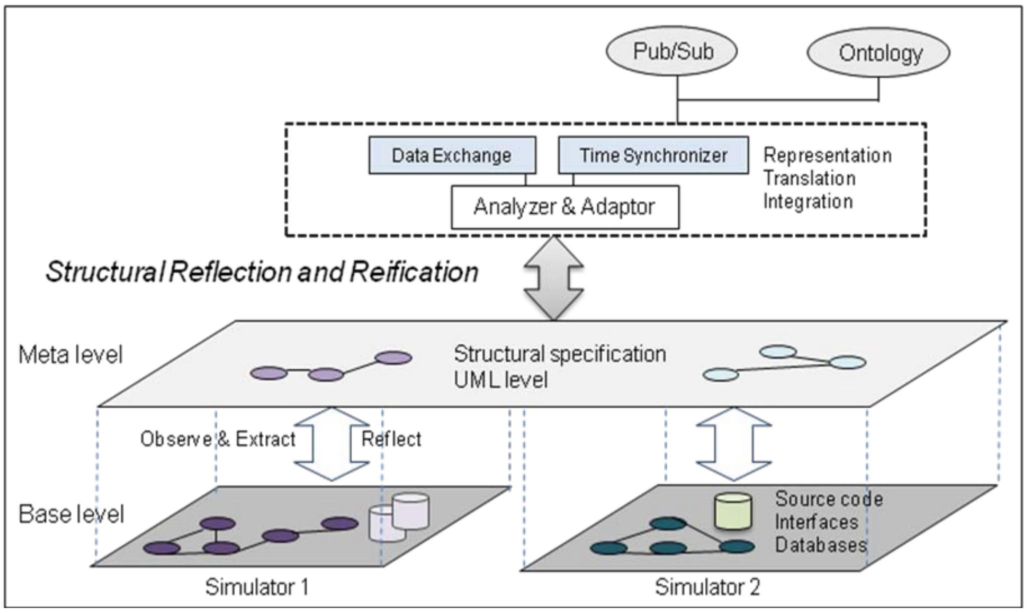

Figure 1. Reflective Architecture for Simulation Integration

2. RELATED WORK

To the best of our knowledge, simulation integration has been applied in two domains – (a) military command-and-control and (b) games. The U.S. Department of Defense (DoD) has promoted the development of distributed simulation standards to provide a common framework in which simulators can be integrated. These include standards such as SIMulator NETworking (SIMNET) [4], Distributed Interactive Simulation (DIS) [15], Aggregate Level Simulation Protocol (ALSP) [5], High Level Architecture (HLA) [6, 7]. These standards provide specific services for interoperability in niche applications, for example DIS for human-in-the-loop simulators or ALSP for war games. The recent HLA effort has become the defacto standard technical architecture for military simulations in the United States – it aims to promote interoperability, scalability, and reusability between simulators. One of the central components of HLA is the Runtime Infrastructure (RTI); ORBs and CORBA services are candidate tools for implementing HLA RTIs. While HLA led to some new insights in simulation integration, its broader applicability for general simulation integration is questionable. 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. Additionally it requires the participants to agree on a common interpretation of data that is produced and exchanged between them. Recently simulation integration methods have been used in the game community [8, 9, 10, 11] primarily to support interoperability. As in the case with the HLA architecture, solutions here are prescriptive - they force the developers to provide a particular functionality or to conform to specific standards to participate in the integration process.

In contrast to the above approaches where the participating simulators must conform to predefined standards, our goal is to leverage existing simulators, as is, while enabling data interchange between them. Existing middleware frameworks such as CORBA, DCOM or RMI [2, 3], XML tools support data exchange between software applications; supporting semantic interoperability requires capabilities beyond what these frameworks can provide. Time synchronization across multiple simulations is one such challenge. Time synchronization services offered by traditional middleware frameworks typically need the participants to agree on a common interpretation of time and on common time advancement methods. Although existing simulators model time in disparate ways, it is difficult to achieve a joint integration without a clear understanding of how each simulator represents and advances time – this is essential for accurate simulation.

3. A REFLECTIVE ARCHITECTURE FOR SIMULATION INTEGRATION

There are several challenges that must be addressed to fully realize simulation integration. The first challenge in modeling complex scenarios using multiple simulators is the analysis of cause-effect relationships between those simulators. The second challenge arises from the fact that each simulator uses its own models and entities; these must now be integrated in the context of a single simulation. The simulators need to exchange the data and have a correct interpretation of the data they send and receive. It is necessary to analyze the data types used internally by the simulators. Therefore, there are challenges in data transformation and data integration. Time synchronization is yet another challenge. When integrating simulators, there is a need for synchronization of time between the different models. The simulation clock that controls simulation time during execution of a simulation resides within each simulator itself. Time synchronization mechanisms are needed to ensure causal correctness for models that use different time advancement mechanisms.

Given the potential black-box nature of simulators developed by experts in diverse domains (earthquake engineering and transportation systems in the introductory example), 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. Our solution does not require simulator developers to adhere to a strict programming interface or conform to particular design styles - the ability to flexibly interoperate with multiple simulators is our goal.

We will use reflective middleware solutions to provide a principled, yet flexible approach to support the development of simulation integration platforms. There are two main forms of reflection in middleware platforms: structural and behavioral reflection [16]. In solutions for simulation integration, we are primarily interested in abstracting out the structural aspects of the underlying simulators. Structural reflection is concerned with the reification of the underlying structure of objects or components, e.g., in terms of the needs of the high level integration task. The expected mode of operation is that the actual interactions between simulations are specified and altered by a specialist who understands the purpose of the different simulators (and the rationale for their integration) but not necessarily their details.

3.1 Reflection for Simulation Integration

In this section we present a two-level reflective architecture for simulation integration, as illustrated in Figure 1. In the proposed architecture, integration of different simulators can be ultimately achieved by using the meta-level for specifying/modeling the properties of the different simulators and reasoning about the interactions among the different simulators – i.e. what we intend to design and develop is a meta-simulator. The meta-level is built on base-level simulators; reification of base-level entities yield data structures at the meta-level, modified features of these structures that implement the integration are then reflected to the base-level. A closer look at the base-level simulators themselves reveals that the structural aspects of the simulation application are not merely in the simulator code - backend databases and models stored in domain-specific formats contain aspects of the simulators that may need to be explored as well -- in general, there can be many kinds of meta-level entities to cover various integration aspects. By using the metamodeling capability the model elements that need to be integrated can be extracted.

A major challenge toward the realization of reflective architectures for simulation integration is related to the complexity associated with reification. Many reflective systems [2, 3] provide access to their internal operations in terms of a composition graph, describing the dependencies between their components. Such an approach requires the specification of all interfaces and objects involved. However, in using existing available simulators we may not have access to all details of the simulators including the specification of their interfaces and objects. In our approach, we formulate the metamodel that captures concepts of interest using a publish- subscribe mechanism for data exchange – here, subscribers (the simulation integration tasks) express interest in aspects that they want to observe (implemented by base-level simulators) – when changes in these monitored aspects occur at the base simulators, the meta-level entities receive information or updates of interest via publishers. A pre-existing set of ontology models assist in the matching process for the pub-sub implementation of the simulation integration task – these include domain ontologies that are representations of knowledge in a well-circumscribed domain. Interoperability of different simulators can be achieved by sharing and understanding the metamodels. Implementing the semantic

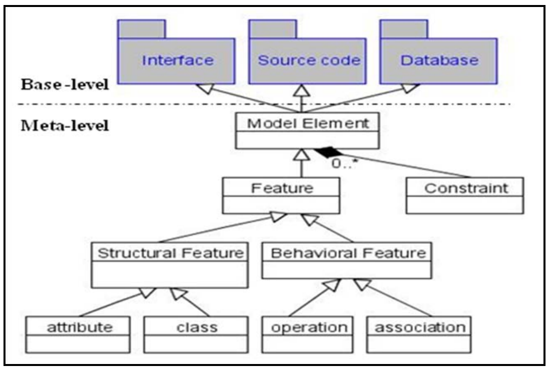

Figure 3. The meta-model

constraints for simulation integration is a human in the loop process which results in the annotations that are invisible to base-level computation and are provided to the meta-level. In the following section we propose our metamodel.

3.2 Meta-model

In this section we attempt to construct a metamodel that helps to reify the main features of the simulation platforms. Metamodels make the underlying simulators more understandable by abstracting out lower-level details of integration and interoperability. The main challenge in deriving a metamodel is what features need to be present in the metamodel. Since our metamodel needs to take several domain expert simulators into account, the metamodel should be comprehensive, yet extensible. The careful examination of the features in various simulators of the different domains has allowed us to identify and categorize common features using key classes. We describe the process and tools used in deriving key structural aspects of simulators to assist the integration process.

Given a set of sample simulators, we used Creole as an Eclipse plug in to examine source code dependencies and to extract the simulator's features. Since Creole did not help us with complex and large simulators, we implemented a parser using a tool for large scale code repositories search [17] to extract the entities and attributes from a Java simulator 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. The key classes in our metamodel will be: model elements, features which could be structural features or behavioral features, and constraints. Model elements are the main elements of a simulation and can be captured from the interfaces, the source code, or databases. Since we are interested in structural reflection, currently we only use structural features which include classes and attributes. We may also take behavioral features into account to represent operations and associations in future. Constraints are the number of limits for the simulation parameters in the simulation model.

Figure 2 shows the key classes that comprise our metamodel. We construct our metamodel using UML (Unified Modeling Language). The reflective UML metamodel has several metamodel propositions including class diagrams for describing the main elements and the static relations among them. We used the Eclipse Modeling Framework (EMF) to define and customize our metamodel which also allows the automatic generation of

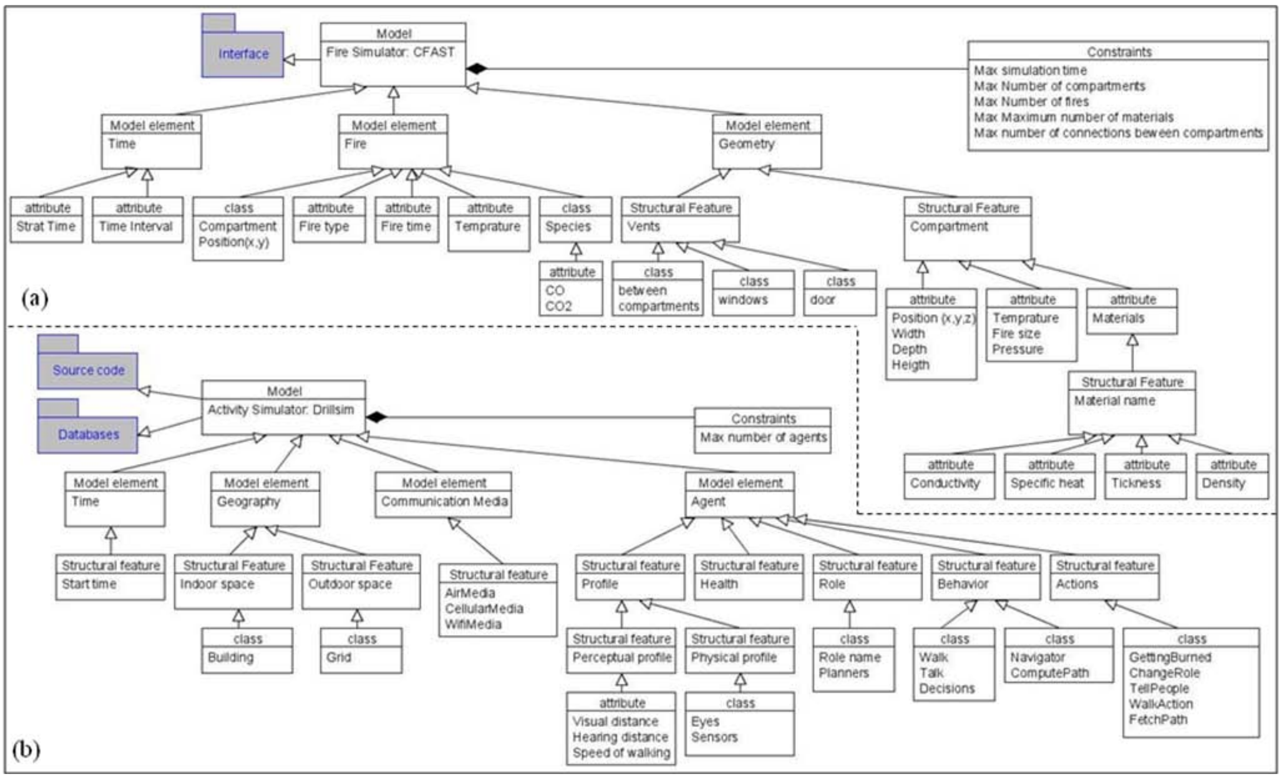

Figure 3. (a) CFAST Metamodel, (b) Drillsim Metamodel

tools (such as a repository). We will explain some examples of the key classes in our case study in next section.

4. CASE STUDY

In this section we develop a case study for simulation integration using two available existing simulators – the primary goal is to validate the proposed reflective architecture and understand issues in its realization. The two simulators are (a) a fire simulator that simulates the effects of fire and smoke inside a building and (b) an activity simulator that model a response activity – evacuation.

4.1 Fire Simulator: CFAST

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 [14]. CFAST have several interfaces to input the parameters that contain information about the building geometry (compartment sizes, materials of construction, and material properties), connections between compartments (horizontal flow openings such as doors, windows), fire properties (fire size and species production rates as a function of time), and specifications for detectors. The simulator produces outputs that contain information about temperatures, ignition times, gas concentrations such as CO and CO2, and etc.

Figure 3-a shows the representation of CFAST using our metamodel. Since in CFAST we only have access to several interfaces and we do not have access to the source code and databases, the model elements are captured from the interfaces. In the figure the model that we try to capture is CFAST-fire. The model elements include time, fire, and geometry. Each element has its own structural features. We try to make our model general and easy to extend to be used for any other fire simulator. A major property associated with any simulation is time. CAFST is a time stepped simulator which has a simulation environment interface in which the simulation time duration and time intervals can be defined. Structural features of time are simulation start time, time interval, and current time. The second model element is the fire which has several structural features, such as location, fire type, time, temperature, etc. Finally, geometry is another model element in CFAST which includes information on the building geometry such as compartments and connections between compartments. CFAST has also some limitation on its parameters that can be specified in the metamodel using constraints. There are two significant properties of the metamodel that we want to briefly mention: flexibility and scalability. First our metamodel is flexible because it is easy to add other key classes to the metamodel using UML specifications. Second, we made our metamodel scalable by developing the meta-adaptors. When using another fire simulator we can adapt its metamodel to the CFAST metamodel using the adaptor. Detailed information about the meta-adaptors is beyond scope of this paper.

4.2 Activity Simulator: Drillsim

Drillsim is a simulation environment that plays out the activities of a crisis response (e.g., evacuation), which is a multi-agent system that simulates human behavior in a crisis [13]. Agents represent an evacuee, a building captain, etc. Every agent has a set of properties associated with it, such as physical and perceptual profile (e.g., range of sight, speed of walking) and the current health status of the agent (e.g. injured, unconscious). At any given time, agents are associated with a given location in the geographical space. Indoor space consists of floors, rooms, corridors, stairways, etc. Outdoor space is represented by a grid in Drillsim. Figure 3-b shows Drillsim using our metamodel key classes extracted from the source code and databases.

4.3 Drillsim-CFAST integration

In order to integrate Drillsim and CFAST using our reflective architecture we first need to extract semantic constraints by using the metadata to capture where we need to integrate the two simulators. The careful examination of the features in the metadata and the cause-effect analysis allow us to extract the following constraints. The main constraint is that fire from CFAST can affect an agent's health in Drillsim. Therefore we need to extract the harmful condition caused by fire and smoke from CFAST and update agent health condition in Drillsim. The following are the examples on how the integration enables information interchanged between two simulators:

- Harmful condition from CFAST can affect someone's health in Drillsim.
- Agents in Drillsim can talk about the fire and its location – this will prevent agents from entering dangerous areas.
- Smoke from CFAST can decrease someone's visual distance in Drillsim.
- Harmful conditions from CFAST can affect the evacuation process in Drillsim e.g. increase walking speed.

Since the harmful condition in CFAST is associated with the specific time and location, we need a time synchronizer and geometry transformer. Each simulator has its own internal time management mechanism. This implies that we need a time synchronization method to guarantee the logical correctness and causality during simulation. The two simulators also use different geometry representations – translators are required to translate specific locations in CFAST to corresponding locations in Drillsim. In the following sections we will describe each integration module in more detail.

4.3.1 Data Issues

We discussed one example of the semantic constraints between the two simulators. In our example harmful conditions extracted from CFAST can affect the health of agents in Drillsim. In the data management module we need to update an agent's health level in Drillsim based on the harmful condition caused by fire and smoke in CFAST. In general, the data management module provides data transfer that preserves the meaning and relationships of the data exchanged between two simulators. Since we are working with existing simulators, we cannot use the methods based on the common representation of data. Each simulator may have its own data representation which can not be easily modified. We used data translators that work based on the constraints. Clearly one of the most difficult portions is to extract these constraints. Finally if the data translators are implemented correctly, they can provide immediate conduits to publish or subscribe to information.

4.3.2 Time Synchronizer

Time synchronization can be implemented differently in simulators: clock synchronization and timescale transformation are two common techniques. In clock synchronization the simulators' clocks have the same time at any given moment which is a costly approach and sometime impossible because the internal time advance manner of a simulator might not be accessible. In timescale transformation we can transform the internal time of one simulation into the internal time of another simulation. When a message or update is sent to another simulator, it has a time stamp which is transformed to the timescale of receiver. For messages sent over multiple simulators the transformation is repeated. Time stamp transformation can be achieved by computing the difference from message creation and its arrival. Since CFAST and Drillsim are both time stepped simulators the timescale transformation can be performed by means of time calibration.

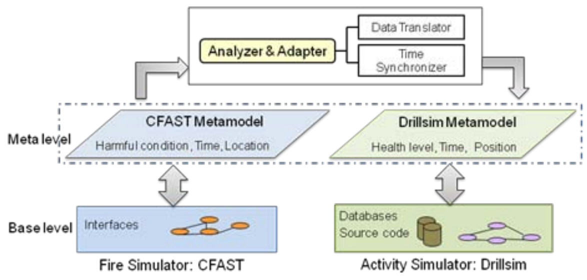

Figure 4. Using the proposed architecture

4.3.3 Geometry Transformer

Geometry translators have the responsibility of performing coordinate conversions between geographies that use different coordinate systems. To create this translator module we create a set of guide points in both geographies and determine a coordinate transform matrix. Figure 4 illustrates how to use the proposed architecture in our case study 1) it allows to extract the simulator design information as metamodels from the base-level simulators using their interfaces, source code, and databases. In CFAST we do not access to the simulator's source code and databases but we can access several interfaces that contain simulation parameters. On the other hand, Drillsim does not provide us with powerful interfaces but we can access the source code and databases; 2) by using the metamodels capability the model elements that need to be integrated can be extracted. This step is a human-in-the-loop process and we use the semantic constraints for integration. In our case study, the fire and smoke from the fire simulator can affect someone's health in the activity simulator; 3) using the main model elements involve in the integration we need: (i) the data translator to transform the data on fire and smoke and to update agent's health, (ii) the time synchronizer to synchronize the time of fire and smoke to the time in Drillsim, (iii) the geometry transformer to perform coordinate conversion between the two simulators. Finally; 4) by using reflection we reflect the modified features to the base-level that means we modify health conditions using the source code and databases in Drillsim.

Comparing with HLA: Table 1 presents a brief comparison of the reflective architecture to HLA. Using HLA outside the defense domain such as our case study is very complex, if not impossible. In HLA low level knowledge needed from participants. Each simulator must use the common data format that leads to simulations that are very closely coupled to an underlying database. Since the HLA environment is a fully distributed simulation environment, the simulators must fully conform to the designated features of the HLA standard. Note that transforming existing simulators to conform to the standard may not always be feasible. In our reflective architecture each simulator can have its own data representation, internal time management, and data management. Therefore, we do not force the simulators to change their internal properties. Another advantage of our reflective architecture is separation of concerns, that is, separate the concerns related to the simulation domain from those related to the integration mechanisms. Additionally it provides a design that is more adaptable, flexible and easier to extend.

Table 1. Comparison between HLA and Reflective Architecture for simulation integration

Criterion	HLA	Reflective Architecture
Objective	– Interoperability – Reusability	– Semantic Interoperability – Reusability – Flexibility
Domain	– Defense	– Flexible via use of domain ontologies
Complexity	– Low level knowledge needed – Lack of semantic interoperability	– No need to conform the internal properties – Semantic constraints implemented at the meta-level
Time Management	– Optimistic and conservative methods	– Allows Timescale transformation
Separation of Concerns	– Merges domain-specific and integrated simulation aspects	– Separate concerns related to simulation domain to those related to integration mechanisms

.

5. CONCLUSIONS AND FUTURE WORK

In this paper we proposed a reflective middleware architecture for simulation integration that implements structural reflection to alleviate the flexibility issues in current simulation integration techniques. In this architecture, the meta-level is structured as a series of metamodels representing the various simulators. We have implemented a detailed case study using two available simulators and illustrated the utility of the reflective architecture. In the near term, we intend to extend our approach to integrate more than two simulators. Future research will focus on addressing challenges in the complexity associated with reification, generalizing the metamodels for other evacuation and fire simulators, integrating simulators in other domains including earthquake and transportation simulators as well as supporting non-Java simulators.

6. REFERENCES

[1] Kheir, N.A., Dekker, M. 1995. Systems modeling and computer simulation, 2nd ed., Springer, New York, USA.

[2] Buss, A., Jackson, L. 1998. Distributed Simulation Modeling: A Comparison of HLA, CORBA and RMI, Proceedings of the Winter Simulation Conference, 818-825.

[3] Verbraeck, A., Valentin, E., Saanen, Y.A. 2000. Simulation as a Real-time Logistic Control System: AGV Control with Simple++, The New Simulation in Production and Logistics Prospects, Views and Attitudes ed., Germany, 245-255.

[4] Pope, A. 1989. The SIMNET Network and Protocols, Technical Report 7102, MA: BBN Systems and Technologies, Cambridge, Massachusetts.

[5] Weatherly, R., Seidel, D., Weissman, J. 1991. Aggregate Level Simulation Protocol, Summer Computer Simulation Conference, Baltimore, Maryland, 953-958.

[6] Xiaoxia, S.. Quihai, Z. 2003. The introduction on High Level Architecture(HLA) and Run Time Infrastructure(RTI), SICE Annual Conference, Japan, 1136-1139.

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

[8] Ling, Y., Zhang, M., Lu, X., Wang, W., Lao, S. 2006. Model Searching Algorithm Based on Response Order and Access Order in War-Game Simulation Grid, Edutainment, Springer-Verlag Berlin Heidelberg, 627-637.

[9] Huang, H., Wu, W., Tang, X., Zhou, Z. 2006. Federate Migration in Grid-Based Virtual Wargame Collaborative Environment, 606-615 .

[10] Rhalibi, A.E., Merabti, M., Shen, Y. 2006. Improving Game Processing in Multithreading and Multiprocessor Architecture, Edutainment, Springer-Verlag,669 – 679.

[11] Jain, S., McLean, C.R., 2005, Integrated simulation and gaming architecture for incident management training, Simulation, Proceedings of the Winter Simulation Conference, 904-913.

[12] Fujimoto, M.R., 2000. Parallel and Distributed Simulation Systems, John Wiley and Sons, Inc., New York.

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

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

[15] Davis, P.K. 1995. Distributed Interactive Simulation (DIS) in the Evolution of DoD Warfare Modeling and Simulation, Proceedings of the IEEE 83(8), 1138-1155.

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

[17] Lemos, O., Bajracharya, S., Ossher, J. 2007. 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, 917-918.

[18] HAZUS-MH 2003 Multi-hazard Loss Estimation Methodology. HAZUS-MH User Manual, FEMA.

[19] Cho, S., Huyck, C.K., Ghosh, S. Eguchi, R.T., 2006. Development of a Web-based Transportation Modeling Platform for Emergency Response. 8th Conference on Earthquake Engineering, San Francisco.