Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler eds.

DESIGN AND IMPLEMENTATION OF AN XML-BASED, TECHNOLOGY-UNIFIED DATA PIPELINE FOR INTERACTIVE SIMULATION

François Rioux
Dept. of Electrical and Computer Engineering
Laval University
Québec, QC, G1V 0A6, Canada

François Bernier
Systems of Systems Section, DRDC-Valcartier
2459 boul. Pie-XI Nord
Québec, QC, G3J 1X5, Canada

Denis Laurendeau
Dept. of Electrical and Computer Engineering
Laval University
Québec, QC, G1V 0A6, Canada

ABSTRACT

Providing software that is efficient, flexible, reusable and easy to work with is a hard task for simulation developers. In this paper we propose the use of XML and its related tools (e.g. JAXB, XQuery, XSLT, and Native XML Database) for the implementation of a technology-unified data pipeline targeted to interactive simulation. We introduce a technology-independent conceptual data model as the basis for every simulation framework. We show that XML is a well-suited technology to be used in that context. We propose a data modeling methodology that takes its roots from Model-Driven Engineering (MDE). We also show a sample implementation that uses XML for transmitting data over the entire simulation loop. We thus present our experience in implementing that kind of architecture and discuss how the use of XML and associated technologies help in building a unified and generic data pipeline for interactive simulation.

1 INTRODUCTION

Nowadays, simulation is used extensively by scientists and engineers for designing complex systems or for understanding intricate phenomena. Typically, "batch run simulations" are exploited for extracting knowledge from virtual experiments. Many batch run simulations lead to wasted computation time because errors in simulations or poor choice of simulation parameters are only discovered after a run, something that could have been avoided should interactive simulation be exploited. For being considered as interactive, a simulation should take a reasonable amount of time to execute, typically a few seconds to a few minutes.

Interactive simulation can be seen as the process of steering a simulation while it is executing. Many approaches have been proposed for steering simulations (Parker et al. 1997, Brooke et al. 2003). Interactive simulation is a 3-step process (see Figure 1). Firstly, a model of the simulation needs to be built in the *Simulation Modeling Module*. This step consists of defining the actors participating in the simulation, the properties of each actor, and the interactions between actors leading to the desired behaviors. The scope of this step depends on the architecture of the simulator and on the level of detail (and fidelity) required for the models. In addition to the definition of the actors and the interactions, a scenario includes the initial values for the parameters of the actors/models/interactions and the scheduling of "outside" events that will occur during the simulation and whose effect may be, among other things, to modify behaviors of the actors.

At the second step, the *Simulation Execution Module* accepts the scenario and models that were designed at the modeling stage. It is clear that the *Execution Module* must be structured so as to understand both the models and the scenario in order to execute the simulation properly and to maintain a coherent internal state. Then, the *Execution Module* runs the simulation and updates its internal state accordingly while taking into account the events scheduled in the scenario. The "simulation state" is defined as the set of variables and parameters describing the totality of a simulation at a given time step. This simulation state, which is available in a given data format, is sent periodically to the *Simulation Analysis Module*. The latter module must

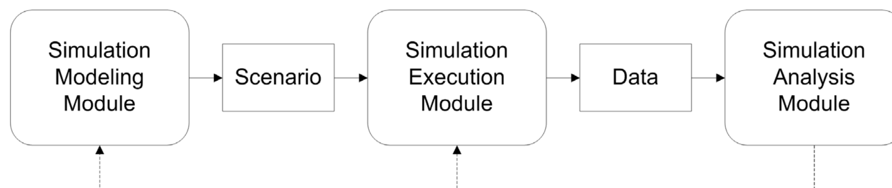

Figure 1: Interactive simulation process

provide the user with interfaces that allow him to visualize what is actually occurring in the simulation, and with tools that allow him to perform different types of analyses on the data such as statistical analysis or data mining (Schulz et al. 2006). A major difference between interactive simulation and batch run simulation is that, for interactive simulation, the user "closes the loop" by providing input to the models and to the *Execution Module* based on his interpretation of the results.

In this paper, we propose a design methodology that facilitates the implementation of user interaction with new and existing simulators. The methodology, presented as a conceptual framework that is a formal way of thinking, is generic and does not make any assumption on the architecture of the simulator. The paper demonstrates that the data pipeline must be designed carefully to ensure successful implementation of interactive simulations. Section 2 introduces a conceptual framework for this data pipeline, which comprises a data model and a generic data flow. Section 3 presents an instance of the conceptual model based on XML technology and software engineering design principles. Section 4 presents a sample implementation of the methodology and Section 5 includes a discussion on this implementation and an overview of future work.

2 CONCEPTUAL FRAMEWORK

This section presents a conceptual framework that sets up the foundation of a methodology for facilitating interaction with simulations. Figure 2 shows the key building blocks of the conceptual framework. Large rectangular boxes represent storage units, whereas rounded boxes represent processing units. Storage units encapsulate the state of data at a particular stage in the conceptual framework, whereas processing units transform input data and outputs the results.

2.1 Data model of the conceptual framework

The first block of the conceptual framework (Figure 2) is the *simulation scenario*, which contains the elements described in Section 1 (e.g. actors, models, interactions, scheduled events). The tools that are used for building the scenarios may range from sophisticated GUIs to simple text file editors. The "simulation scenario" box, exploded in Figure 2, illustrates the data model the scenario must comply. The scenario "document," which contains the data, is an instance of a "document model." On the other hand, the "document" validates against a "schema," a model defining the document syntax. The schema itself is an instance of the "schema model," a meta-model defining the content of a schema. Once the scenario document is built and validated against its schema, it must be converted to a format that is understandable by the simulator.

For that purpose, a "conversion engine" links the simulation scenario and *deserialized objects*, which are intermediate storage elements, by performing appropriate processing. The reason for feeding the simulator with deserialized objects instead of the scenario as such is that we want to keep the framework generic and independent of the architecture of the simulator. This genericity constraint has a direct impact on the technologies that need to be selected for ensuring smooth integration of the deserialized objects and the simulator. Therefore, the addition of the interactivity feature to a simulator should be transparent to its internal modules. It is worth noting that a single scenario usually generates several deserialized objects, each being an instance of a "class."

As shown at the bottom of Figure 2, we assume that simulation entities exist in the internals of a *simulator*, regardless of its architecture and implementation. These entities are of several types, e.g. actors, properties, interactions, agents. In object-oriented implementations, they are instances of a "class," which has a "class model" (Atkinson et al. 2003). The simulator executes interactions between entities, which generates data that must be serialized and converted back to a format that a user understands in order to be available for analysis.

The *serializable objects* are data containers in which the simulator writes the simulation state. They own the same data model as deserialized objects. A conversion en-

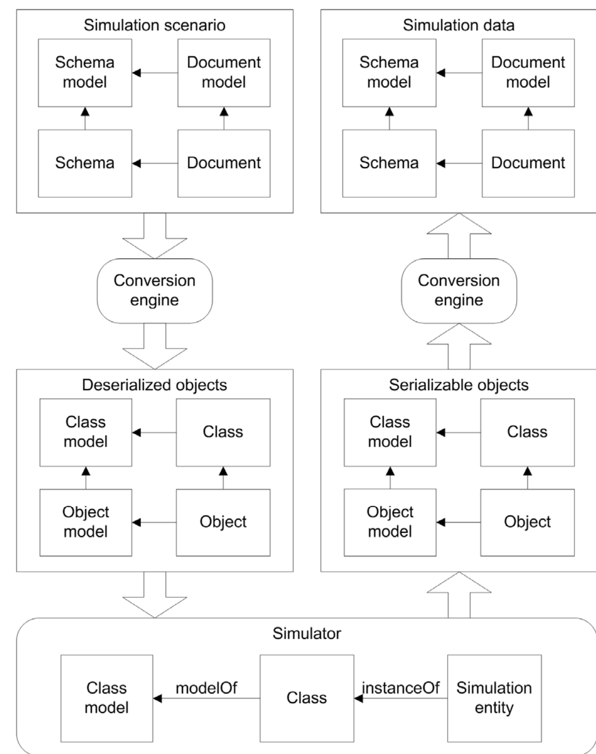

Figure 2: Technology-independent conceptual framework

gine translates serializable objects to *simulation data*, which shares the same data model as the simulation scenario.

2.2 Generic dataflow in the conceptual framework

Figure 3 shows a generic dataflow that is suitable for interactive simulation. The *simulation modeling* step shows that a user exploits a *scenario editor* to produce a *scenario document*. This document is usually a computer file. The elements contained in the scenario document are converted to deserialized objects and affected to the simulation state. The simulation engine performs calculations on elements composing the simulation state and updates involved entities of the simulation state accordingly. Then, the simulation state is copied to serializable objects.

We implemented a mechanism to meet the interactive simulation requirements; serializable objects can be saved to checkpoint files, which act as simulation state containers that a user can modify and reload back to the simulator. We chose the latter mechanism because it is simple to implement, sometimes already part of the features of a simulator, and compatible with many available simulation architectures (Brooke et al. 2003). On the other hand, the developer of the simulation modeling module of an interactive simulator needs to interface existing code with deserialized and serializable objects in order to load and save the simulation state.

A user is able to control the flow of a simulation with the *simulation analysis* step, which includes the following modules:

- A **data manager** whose role is to communicate with the simulation for retrieving simulation data into a stream, to manage incoming data, and to process user interaction;
- A **database** that stores selected information incoming from the simulator data streams;
- A **data stream query filter** that selects, according to the user's needs, information relevant to the simulation analysis;
- A **data analysis** module that is exploited by the user for exploring the simulation and acquiring knowledge of the phenomenon under study.

In summary, we propose in this section a framework that simulation practitioners should implement in order to convert existing software to an interactive simulator. The proposed framework allows the use of traditional simulation methodologies such as batch simulation because simulation data is stored in a database for later consultation.

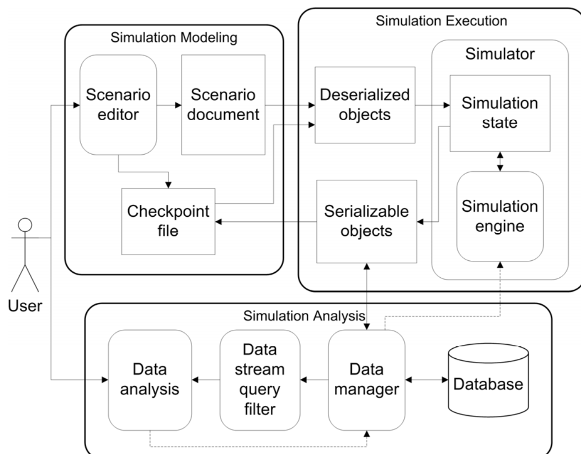

Figure 3: Generic dataflow for interactive simulation

3 XML AS A UNIFYING TECHNOLOGY

It is claimed that XML is a technology that is well adapted for the implementation of the conceptual framework described in Section 2. Figure 4 shows how the generic structure shown in Figure 2 can actually exploit XML technology to implement the conceptual framework. The suggested XML-based implementation makes the assumption that the simulator is implemented in object-oriented technology. We aim towards a design methodology that could be used by simulation practitioners to decrease the development effort when building an interactive simulator from existing software.

Originally, XML was developed as a subset of SGML, intended for web applications. It now describes data in

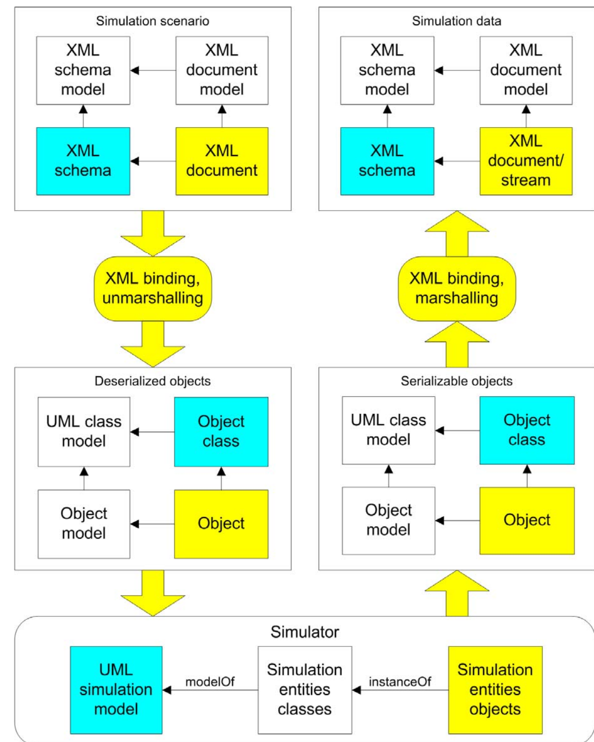

Figure 4: XML-specific conceptual framework. Blue boxes highlight a modeling framework, whereas yellow ones highlight a data pipeline

several application areas such as semantic web, mathematics, biological simulations, and military decision making (Wikipedia 2008a). Some authors propose guidelines to follow, so that researchers use XML technology only where it belongs (Boyno 2006).

Many tools exist in order to parse and validate XML files, bind XML entities to objects of different programming languages, store XML data in databases, visualize XML documents and schemas, transform XML documents, and query XML files (Wikipedia 2008a). An advantage of XML over other data formats is its self-description. XML describes its structure, field names and values. The integration of metadata in an XML stream is also straightforward. The resulting plain text is human- and machine-readable, and fully portable on different system architectures.

On the negative side, XML is verbose, which results in a waste of bandwidth when transmitted over a network. However, some binary XML formats compress data, making it less redundant and more efficient for processing. Also, every piece of data is a string, eliminating the intrinsic data type support that is available in most programming languages. For the conversion between XML and common data types, the marshalling operation transforms common data types (e.g. double, float, integer) to XML strings, whereas the unmarshalling operation transforms an XML string to common data types. Nonetheless, these conversions require considerable amount of processing time and need an XML schema that defines the node types and structure of a given XML file.

3.1 Detailed XML data model

Despite the weaknesses identified above for XML, its portability and simplicity make it an excellent choice for developing the framework for interactive simulation. In addition many tools for processing and handling XML data are available and reduce the development time. Figure 4 shows how XML is exploited in the conceptual framework described in Figure 2. More specifically, the scenario is stored in an XML document. This document must conform to a set of rules defined in a companion XML Schema. It should be noted that a schema validates only the syntax of an XML document, not the semantics. Therefore, a higher-level mechanism must take care of maintaining the coherence of the scenario.

XML node entities contained in a document are unmarshalled to objects, which are instances of object-oriented programming language classes (e.g. C++, Java, C#), via XML data binding that refers to the process of representing elements of a XML document as objects in computer memory. Through an automated procedure, XML binding libraries create "object classes" according to the schema of the XML document. Software libraries that bind and convert XML nodes to objects are available for many programming languages (Wikipedia 2008b).

When developing a simulation engine from scratch, designers should adopt sound software engineering practices by creating a UML static class model of the internal simulation state data model. In addition, the use of modern tools offering code generation functionality helps in saving development time. When existing simulation entity classes (bottom of Figure 4) do not have an accessible UML model, reverse-engineering tools can help in recovering this model.

A simple mechanism transposes *deserialized objects* to *simulation entities objects*. It is usually implemented by copying class attributes to corresponding fields in simulation entities objects. A similar mechanism transfers the state of the simulation from the simulator to *serializable objects*. Then, the marshalling process, included in the binding library, transforms objects back to XML format. It should be noted that deserialized objects and serializable objects do not necessarily share the same data model. However, adopting the same model for both concepts is recommended because of the resulting uniformity in processing. The same remark applies for the simulation scenario and the simulation data. Both can embody the totality or a fragment of a global and unique schema.

3.2 Design methodology

Figure 4 highlights important characteristics emerging from the use of XML as the basic building block of the data flow. Boxes with a blue background represent the data modeling methodology of the XML-based framework detailed at the left of Figure 5.

The first step in the methodology consists of building a static UML simulation model. The UML model should contain classes associated to every simulation element (bottom of Figure 4). Each class should contain its attributes, the set of classes and attributes making what we call a *simulation state,* that includes elements part of the initial scenario (e.g. entities with their initial properties and links), as well as others essential at runtime (e.g. random number generators).

Figure 6 shows such a typical UML class diagram. The "Simulation" class is a root element that includes the "timeStep" attribute. It aggregates "InitialScenario," "Random," and "SimulatedEntity" elements. "InitialScenario" defines initial simulation entities as well as the terrain on

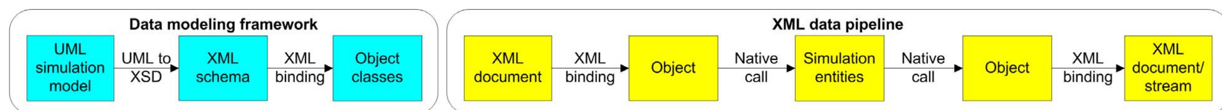

Figure 5: XML data modeling framework (left) and data pipeline (right)

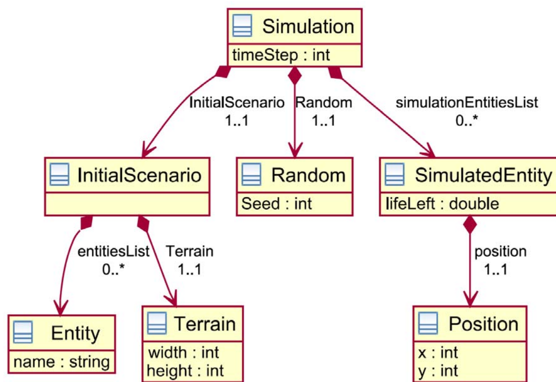

Figure 6: Sample static class UML diagram

which they evolve. Simulated entities store the state of entities being simulated during the process of checkpointing a simulation. The "Random" element includes information about random number generators that introduce randomness in a simulation. Saving this information is essential in order to obtain the exact same runtime results when loading back and resuming the execution of a checkpointed simulation.

The second step in the data modeling methodology is to generate an XML schema from the static UML class diagram. Tools exist that perform the conversion from UML to XSD (Carlson 2006), which conforms to the W3C XML schema syntax (Wikipedia 2008a). Figure 7 shows the schema resulting from the static UML diagram in Figure 6. It starts with the definition of elements, each associated with a class in the UML diagram. Then, every element is defined according to the static view specifications. The XML schema preserves the cardinality as well as types established in the UML diagram. It is thus the model of subsequent XML documents, such as initial scenarios or checkpoint files. It is also the template for object classes that constitute the data model inside the simulation architecture.

The third and final step in the data modeling methodology is the binding of the XML schema to an object-oriented programming language. This step consists of generating a class associated to every element defined in the schema. Several existing software suites or libraries can perform this binding task for various object-oriented programming languages (Wikipedia 2008b). Figure 8 shows the Java class that is produced by the Java XML Binding (JAXB) compiler using the XML schema presented in Figure 7. It shows protected attributes that are accessible via public get/set methods. The class also contains several annotations that help the Java runtime environment and compiler to perform object serialization and deserialization.

The methodology described above is an instance of Model Driver Engineering (MDE) (Kent 2002). The process starts off with a platform-independent, user-defined meta-model (the UML diagram) subsequently transformed to another platform-independent meta-model, the schema. The UML model will typically be a copy of an existing simulator execution model composed of simulation entity classes. If the latter does not exist, execution and data models should be developed in parallel in order to minimize inconsistencies. An alternative approach would be to inherit simulator classes from data model classes. The correspondence between attributes should then be trivial.

When the simulator source code is available, the UML diagram can be obtained by reverse engineering (IBM 2008). Otherwise, the success in applying the methodology depends on the possibility to link the internal simulator data model with the user-defined data model (e.g. through an Application Programming Interface or a network socket).

We can conclude that the data modeling framework is better-suited for simulation software under development. When the source code of the simulator is available, the methodology can be easily adopted; otherwise, it is not likely to be successful. Nevertheless, the proposed methodology is the first step towards interactive simulation. The next

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Entity" type="Entity"/>
  <xsd:element name="InitialScenario" type="InitialScenario"/>
  <xsd:element name="Position" type="Position"/>
  <xsd:element name="Random" type="Random"/>
  <xsd:element name="SimulatedEntity" type="SimulatedEntity"/>
  <xsd:element name="Simulation" type="Simulation"/>
  <xsd:element name="Terrain" type="Terrain"/>
  <xsd:complexType name="Entity">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="InitialScenario">
    <xsd:sequence>
      <xsd:element ref="Terrain"/>
      <xsd:element     maxOccurs="unbounded"     minOccurs="0"
name="entitiesList" type="Entity"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Position">
    <xsd:sequence>
      <xsd:element name="x" type="xsd:int"/>
      <xsd:element name="y" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Random">
    <xsd:sequence>
      <xsd:element name="Seed" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="SimulatedEntity">
    <xsd:sequence>
      <xsd:element name="lifeLeft" type="xsd:double"/>
      <xsd:element name="position" type="Position"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Simulation">
    <xsd:sequence>
      <xsd:element name="timeStep" type="xsd:int"/>
      <xsd:element ref="Random"/>
      <xsd:element ref="InitialScenario"/>
      <xsd:element      maxOccurs="unbounded"     minOccurs="0"
name="simulationEntitiesList" type="SimulatedEntity"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Terrain">
    <xsd:sequence>
      <xsd:element name="width" type="xsd:int"/>
      <xsd:element name="height" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Figure 7: Sample XML schema file

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "SimulatedEntity", propOrder = {
    "lifeLeft",
    "position"
})
public class SimulatedEntity {

    protected double lifeLeft;
    @XmlElement(required = true)
    protected Position position;

    public double getLifeLeft() {
        return lifeLeft;
    }

    public void setLifeLeft(double value) {
        this.lifeLeft = value;
    }

    public Position getPosition() {
        return position;
    }

    public void setPosition(Position value) {
        this.position = value;
    }
}
```

Figure 8: Sample Java class

section shows how the use of XML throughout the pipeline facilitates data integration.

3.3 XML-specific data pipeline

The data modeling methodology presented in the previous section can be used for setting up a data pipeline. The right-hand side of Figure 5, composed of the yellow boxes shown in Figure 4, suggests such a pipeline. It starts off with an XML document containing the user-defined initial scenario (or a checkpointed simulation). Then, with the XML binding library's unmarshaller, XML elements are automatically converted to objects that are instances of XML-bound classes. These intermediate storage elements format depends on the chosen programming language. Then, through native calls, simulation entities are filled with data on which they will execute mathematical operations. The next step consists of copying, with native calls, data from simulation entities to serializable objects. Then, the XML binding library marshaller converts these objects to XML in the form of streams or documents. The checkpoint operation becomes trivial; it consists of writing the entire simulation state to an XML document.

A user can experiment interactivity with the simulation by modifying the checkpointed simulation file via a user interface or an XML editor, and by loading back the file in the simulator. The implementation of such a user interface is simple since a XML file adopts a tree structure.

4 SAMPLE USE OF THE METHODOLOGY

The proposed methodology was applied successfully to the implementation of an interactive simulator using existing open-source software. The system is described below.

4.1 Pythagoras as a simulator

Pythagoras is a free, open source, agent-based simulator that models agent entities having behaviors as well as several properties (e.g. life left, position, side color), and evolving on a terrain having its own properties (e.g. size, presence of buildings, movement factor) (NPS 2008). This software was part of Project Albert, which aimed at using high performance computing in order to "understand the unexpected" in a military context. Pythagoras employs brute force computing in order to investigate the proposed problems. It also includes a utility for batch simulation, but does not offer facilities for interactive simulation steering. Pythagoras is implemented in Java and already exploits XML binding technology to load the scenario into its kernel.

4.2 Applying the methodology

Pythagoras lacks several features that are needed to make it interactive. Using Pythagoras original source code and through a thorough reverse engineering process, we implemented various functionalities, in compliance with Figure 9. According to the diagram in the data modeling framework (left of Figure 5), UML modeling should be the first step in the methodology. However, since the simulator was already programmed, the approach of reverse engineering from source code to UML, then UML to XML schema, would have slowed down the design process. Hence, the XML schema was edited by adding essential elements for dumping a simulation state to a file. The

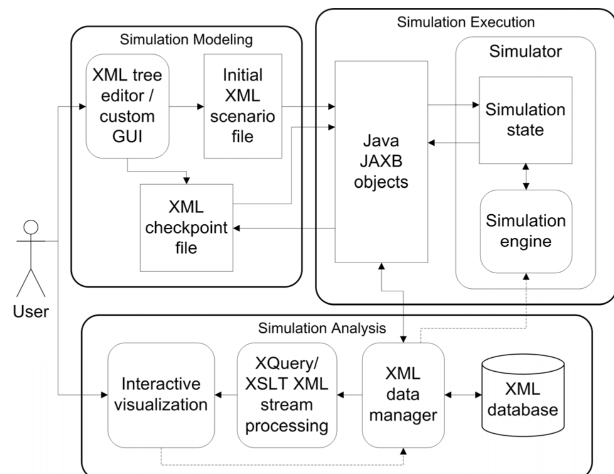

Figure 9: XML and Java technology-specific dataflow

XML schema compilation thus produces one JAXB class for every additional element. The new classes own the same attributes as their counterparts in the simulation kernel, allowing for the implementation of "save" and "restore" methods, which copy data back and forth from JAXB objects to existing simulation models, in a few lines of code.

In order to modify minimally the existing simulator, we implemented a network bridge between Pythagoras and a tailored simulation analysis module, allowing for the communication between a user interface and the simulator. The bridge sends commands such as "start," "pause," "checkpoint," and "load" and receives feedback from the simulator, such as the current time step and a custom simulation data stream. In the current implementation, the "Pythagoras simulation viewer" is used to visualize the execution of a simulation. In the future work section, we discuss the implementation of a generic viewer.

Pythagoras includes a graphical scenario editor for generating the initial XML scenario file. This editor does not need any modification for transforming Pythagoras to an interactive simulator. However, a user interface is needed for modifying an XML checkpoint file. We thus built a tree-based XML editor that allows for changing XML leaf nodes values straightforwardly (Figure 10). The dialog shows available parameters on the left along with their values. The user chooses the parameters of interest and inputs new values in the corresponding text boxes.

Based on the above tools, we propose a procedure that, once completed following the correct order in its various steps, leads to interactive simulation. When the user notices an interesting event he would like to explore with mode detail, he should:

1. stop the simulation;
2. save the current simulation state to an XML file;
3. open the file with the simulation parameters editor dialog;
4. perform the changes he wants to bring to the course of action in the simulation;
5. save the file;
6. load it back into the simulator;
7. resume the execution of the simulation.

We can see from the previous paragraphs that the conversion of Pythagoras from a batch simulator to an interactive simulator involves some changes in the original code and the implementation of several utility programs. On the other hand, the native use of XML as a data format in Pythagoras greatly facilitated the use of the proposed methodology. Also, XML libraries were easily found since there are numerous available implementations for the Java programming language in which Pythagoras is programmed.

4.3 Lessons learned

The following lessons were drawn from the experience of applying the proposed methodology with the Pythagoras simulator as a test bed:

- *Open source code facilitates the application of the methodology*. The more control a user has on software, the easier it will be for him to modify existing functionalities and add new ones. For the current demo, the learning curve was steep. However, reverse engineering tools help developers to get a better hand on complex software architectures.
- *MDE generally applies if a model is already available*. In completing the current work, we did not use MDE, but rather did modify the XML schema manually. It was easier to do so because the advantage of using current software engineering tools over manual techniques was not clear. Future work section discusses the use of automated tools for solving that problem.
- *The use of standard data formats facilitates integration.* Since XML was initially used by Pythagoras as the basic data format, the integration with the proposed methodology was straightforward. Other data formats should provide serializa-

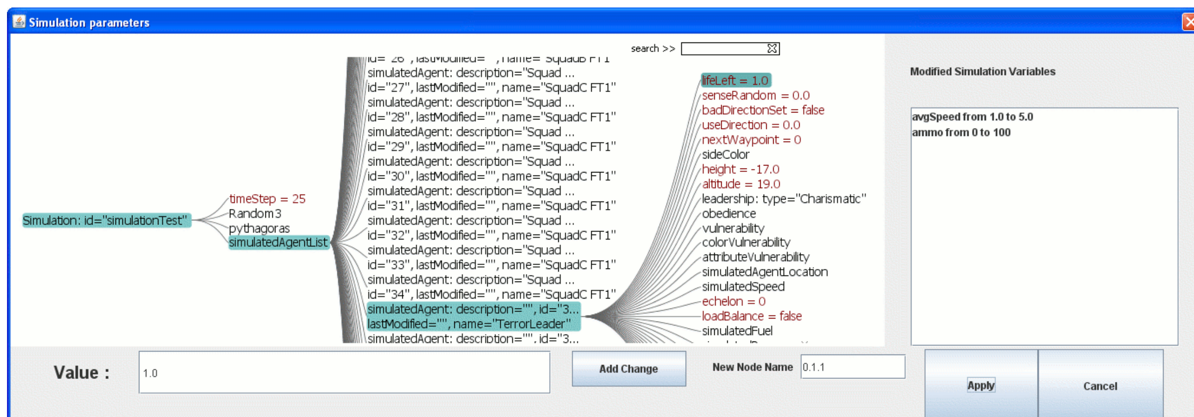

Figure 10: Simulation parameters configuration dialog

tion and deserialization methods in order to allow for the proposed methodology to be exploited successfully. Several data formats other than XML exist that meet these requirements. However, XML offers additional functionalities that other data formats are lacking (e.g. data manipulation with XQuery and XSLT, direct data binding, simple visual representation).

- *The compiled schema semantics during XML binding is limited.* The XML schema binding compiler that we used in the current implementation does not support relations other than “aggregation” and “attribute” between elements. Relations such as “inheritance” were added through the development of a compiler plug-in that offers a richer semantic to be added to the data model.
- *The methodology requires a minimal set of essential functionalities to be supported by the simulator.* The only essential feature that a simulator must implement is the ability to checkpoint and resume simulations. Other features ease the development of advanced interaction, but are not mandatory.

The lessons demonstrate that, despite limited functionality of the demo application described in this section, the methodology was applied successfully; a non-interactive simulator was transformed to an interactive one. We discuss issues that can be improved in the future work section.

Methodologies similar to the one proposed in this paper are found in the literature. For example, (Harrison et al. 2005) suggest a process called KARMA, which aims at capturing the knowledge of experts into reusable and interoperable models that can persist over several applications. It is based on software engineering concepts, tools, and best practices to guide modelers in the development of models through a modular, small-grained configuration based on XML. This work compares to the one presented here because a methodology is proposed for developing a generic framework for modeling and simulation. However, our work focuses on the issues of analysis and interaction during a simulation rather than on modeling and interoperability issues.

(Kurtev et al. 2005) propose a method for transforming XML documents into an application-specific model. General transformation concepts that form our generic conceptual data model were borrowed from this work. For the current applications, XML is sufficient, but sometimes XML schema semantics lacks expressiveness.

Several software libraries allow for the implementation of interactive simulation tools. (Brooke et al. 2003) developed the RealityGrid computational steering library. A user must add instrumentation code at well-defined locations in order to update parameters, retrieve simulation results and control the execution of the simulation. This approach is therefore less flexible than ours, because one must update the source code every time an additional parameter is added. SciRun (Parker et al. 1997) adopts a component-based approach for developing simulation code. This method involves more work when a simulator is already implemented, as the original simulation software has to be decomposed into simple components.

In the current implementation, XML was chosen for several reasons, but the main rationale was the possibility of exploiting a single technology throughout the entire data pipeline: from visual representation to data binding or stream transformation. Several other data formats could have been used, such as HDF5 or NetCDF, which are binary data formats (McGrath 2003). However, the latter technologies do not offer as much flexibility and ease of use as XML does. Also, the conversion from a technology-independent data model (e.g. UML) to a technology-dependent data model (e.g. XML schema) is not as straightforward as for the XML technology.

(Shi et al. 2002) propose a data pipeline that allows for the extraction of knowledge from simulation results. They use principal components analysis and rough sets theory in order to extract knowledge from the data flow coming from a simulator. Our approach is more generic as it does not impose any analysis technique on the data pipeline processing.

5 FUTURE WORK AND CONCLUSION

We showed that the methodology presented in this paper can be successfully applied for developing a specific application. However, it can be improved in several ways. First, the data modeling framework could be fully automated (left of Figure 5). In fact, using a stereotype on appropriate classes, the UML diagram could be converted to XSD, the XSD compiled to source code and methods that copy data to/from objects automatically generated. This process is relevant for a new simulator design and one that was reversed engineered.

Also, we are currently designing and implementing a generic visualization environment that will allow its users to manipulate data in an immersive virtual reality environment. We plan on integrating our entire data pipeline, so that multiple simulation instances can be visualized simultaneously.

Finally, the transformation of Pythagoras from a batch run type of simulator to an interactive simulator is the beginning of a long term project. We plan on modifying several additional simulators and experiment whether or not users perform better in the understanding of a complex system model using the interactive version.

ACKNOWLEDGMENTS

This work was supported in part by a post-graduate scholarship from the Natural Science and Engineering Research

Council of Canada (NSERC), the “Fonds de recherche sur la nature et les technologies” (FQRNT), and the IMAGE project at DRDC-Valcartier. This support is gratefully acknowledged.

REFERENCES

Atkinson, C., and T. Kuhne. 2003. Model-driven development: a metamodeling foundation. *Software, IEEE* **20** (5): 36-41.

Boyno, E. A. 2006. XML: What, What, Who and Where. *ISECON*.

Brooke, J. M., P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning, and A. R. Porter. 2003. Computational Steering in RealityGrid. *UK e-Science All Hands Meeting*: 2–4.

Carlson, D. 2006. Semantic Models for XML Schema with UML Tooling. *2nd International Workshop on Semantic Web Enabled Software Engineering*.

Harrison, N., B. Gilbert, A. Jeffrey, R. Lestage, M. Lauzon, and A. Morin. 2005. KARMA: Materializing the Soul of Technologies into Models. *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*.

IBM 2008. Rational Rose. Available via <http://www.rational.com> [accessed June 23, 2008].

Kent, S. 2002. Model Driven Engineering. *Integratted Formal Methods. Third International Conference, IFM*: 15-18.

Kurtev, I., and K. van den Berg. 2005. Building adaptable and reusable XML applications with model transformations. *Proceedings of the 14th international conference on World Wide Web*: 160-169.

McGrath, R. E. 2003. XML and Scientific File Formats. *2003 Seattle Annual Meeting*.

NPS 2008. Pythagoras. Available via <http://harvest.nps.edu> [accessed June 23, 2008].

Parker, S. G., D. W. Weinstein, and C. R. Johnson. 1997. The SCIRun computational steering software system. *Modern software tools for scientific computing table of contents*: 5-44.

Schulz, H. J., T. Nocke, and H. Schumann. 2006. A framework for visual data mining of structures. *Proceedings of the 29th Australasian Computer Science Conference*-Volume 48: 157-166.

Shi, X., J. Chen, H. Yang, Y. Peng, and X. Ruan. 2002. A Novel Approach to Extract Knowledge from Simulation Results. *The International Journal of Advanced Manufacturing Technology* 20 (5): 390-396.

Wikipedia 2008. Extensible Markup Language (XML). Available via <http://en.wikipedia.org/wiki/XML> [accessed June 23, 2008].

Wikipedia 2008. XML Data Binding. Available via <http://en.wikipedia.org/wiki/XML_data_binding> [accessed June 23, 2008].

AUTHOR BIOGRAPHIES

FRANÇOIS RIOUX received the B.Eng. degree in Electrical Engineering at Laval University in 2003. He received the M.Eng. degree from McGill University in 2005. He is completing his Ph.D. degree in Electrical Engineering at Laval University. He performs research in collaboration with Defence R&D Canada – Valcartier on the topics of interactive simulation and visualization applied to solving complex systems. His email address is <frioux@gel.ulaval.ca>.

FRANÇOIS BERNIER received the B.Eng. degree in Engineering Physics and the M.Sc. and the Ph.D. degree in Electrical Engineering at Laval University in 1997, 1999, and 2008 respectively. In March 2003, he joined the Defence R&D Canada – Valcartier as a Defence Scientist in the Simulation and Comprehension of Complex Situations group, in the System of Systems (SoS) Section. He has been involved in SoS related research on capability engineering, lead a project on combat identification and set up the Virtual Immersive Facility, which explores the potential of advanced visualization and interactive simulation for the DND/CF. His email address is <francois.bernier@drdc-rddc.gc.ca>.

DENIS LAURENDEAU received the B.Eng. degree in Engineering Physics and the M.Sc. and Ph.D. degrees in Electrical Engineering at Université Laval in 1981, 1983 and 1986 respectively. He joined the Department of Electrical and Computer Engineering at Laval University in 1987 where he is now a full-time professor and director of the Computer Vision and Systems Laboratory. He is also director of REPARTI, a NATEQ-funded (Nature and Technology Funding Agency, Province of Quebec) research center conducting research on distributed intelligent environments. Dr. Laurendeau’s research work focuses on range sensor design, modeling and analysis of 3D data, modeling for virtual and augmented reality, biomedical applications of virtual reality, and simulation. His email address is <laurend@gel.ulaval.ca>.