A UML Tool for an Automatic Generation of Simulation Programs

L.B. Arief and N.A. Speirs
Department Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU, England
E-mail: {L.B. Arief, Neil.Speirs}@ncl.ac.uk

ABSTRACT

For sometime now, Unified Modelling Language (UML) has been accepted as a standard for designing new systems. Its array of notations helps system designers to capture their ideas in a way that is expressive yet easy to understand. One thing that UML lacks though, is a means for predicting the system's performance directly from its design. Performance prediction is a desirable feature that enables us to evaluate whether a particular design is worth implementing or not. This prediction can frequently be obtained by constructing a simulation program that mimics the characteristics of the new system. The information gathered from running the simulation will then allow us to estimate the performance. In this paper, we present a simulation framework that can be used to generate simulation programs straight from UML notations. We also present a tool has been built to demonstrate the feasibility of using this framework to perform such a transformation automatically.

Keywords

Model Design, Discrete Simulation, Program Generators, Process-Oriented, UML Extension.

1. INTRODUCTION

Design is an important aspect of the software industry: without a proper design, a software system may fail to deliver its intended service and quite often this can lead to some catastrophic consequences. It is therefore necessary for the software developers to undertake the design process thoroughly before implementing the system. With the emergence of the *Unified Modelling Language (UML)* [7, 10, 17] as an industry standard, the problem of not understanding other people's design is reduced substantially. UML provides a set of graphical notations for describing new systems in a clear and non-ambiguous way. There are also many tools available that support UML design notations and some of them can even generate a skeleton program from the design. What these tools are lacking though, is a way to evaluate whether a particular design will satisfy the requirements or not. There is no point in implementing a design that cannot meet the requirements and quite often the requirements concern about the performance of the system more than anything else. Yet we do not know what kind of performance a particular design will deliver until we built a prototype or a model based on the real system. Simulation facilitates the later approach and it is this kind of approach that we have investigated. Since building a simulation program requires a sound knowledge of some simulation technique - which is not often possessed by the system designers - it is desirable to have a tool that can automatically generate simulation programs directly from the design. We have identified some simulation components that are applicable to many simulation scenarios along with the actions that can be performed by them. Based on these components and their actions, we have developed a simulation framework called *Simulation Modelling Language (SimML)* [3, 4] to bridge the transformation from design to simulation program. A UML tool that supports this framework has been constructed in the *Java* [6] programming language using the *JFC/Swing* package [12]. The simulation programs are generated in the Java programming language using the *JavaSim* [9] package developed at the University of Newcastle upon Tyne. We decided to use the *Extensible Markup Language (XML)* [8, 19] for storing the design and the simulation data for two reasons. First, it allows us to store the information in a structure that is specific to our need by defining an appropriate *Document Type Definition* (DTD). Second, an XML document can be manipulated easily using some Java packages such as the *Simple API for XML (SAX)* [14] through IBM's *XML4J* parser [11].

The rest of this paper is structured as follows: Section 2 illustrates the design notations of UML that are relevant for generating a simulation. Section 3 introduces our simulation framework, the Java package we have constructed to support this framework and the simulation environment used (JavaSim package). Section 4 describes the process of putting these all together and Section 5 explains how we can use the XML notation to store the information that is relevant to both UML and SimML. We then provide a simple example in Section 6 to show the feasibility of using our UML tool in designing a system and predicting its performance through simulation before we conclude our paper in Section 7.

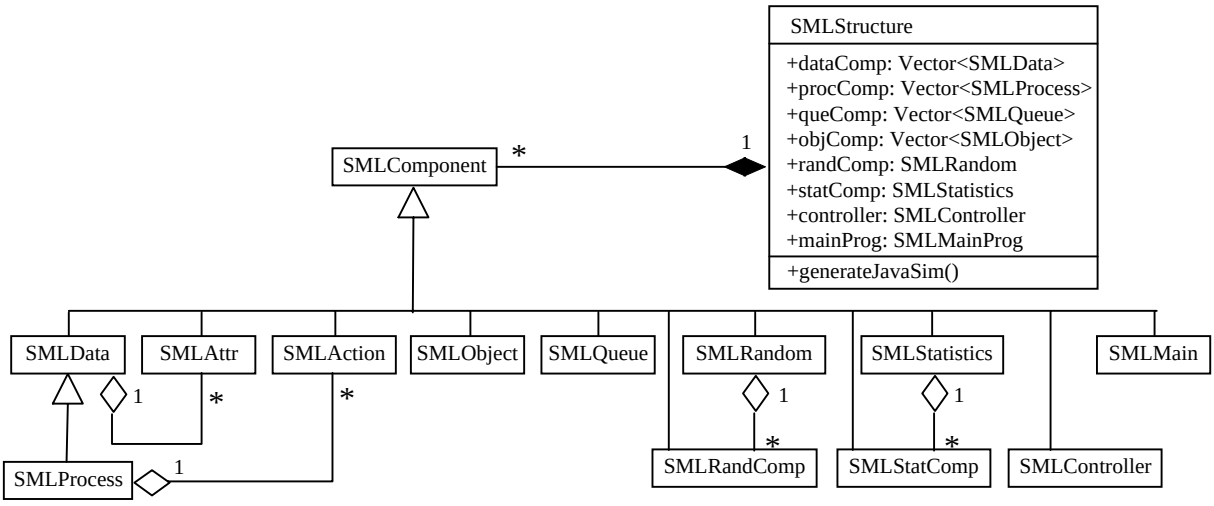

Figure 1: The class diagram for the SimML Components

2. USING UML FOR DESIGNING NEW SYSTEMS

Unified Modelling Language (UML) is a graphical language for visualising, specifying, constructing, and documenting the artefacts of a software-intensive system [7]. Our interest here lies with the *class diagram* and the *interaction* diagram.

We use the class diagram to model the static properties of a system. As we plan to transform the design into simulation, the classes defined in the class diagram serve as the simulation entities that we are going to model. In our view, the aspects of the class diagram that are relevant to simulation include the class name (for distinguishing one simulation entity from another) and the class attributes (for storing performance-related information).

The behavioural properties of a system can be represented as an interaction diagram, which consists of objects and their relationships, including the messages that might be sent from one object to another. An interaction diagram is also useful for constructing an executable system through forward engineering, which is exactly what we want to achieve with regard to automatic simulation generation. A *sequence diagram* is one kind of interaction diagrams that puts an emphasis on the time ordering of the message. This provides a clear visual perspective to the flow of control over time, which is why the sequence diagram is suitable to model a discrete-event process-based simulation.

Before we can translate the notations employed by the class and sequence diagrams into a simulation program, we must first investigate how to construct a simulation program in a generic way.

3. SIMULATION FRAMEWORK

In order to assist the construction of simulation programs, a framework called *SimML* (*Simulation Modelling Language*) has been constructed [3, 4]. This framework identifies the common simulation components and actions, which enables us to automatically transform the design notation into a simulation program.

Section 3.1 gives an insight into the simulation environment that we are going to use, namely the *JavaSim* simulation package [9]. Section 3.2 discusses the implementation of the SimML framework as a Java package to allow the construction of simulation programs in JavaSim environment.

3.1. Simulation Environment: *JavaSim*

JavaSim is a Java implementation of the original C++SIM simulation toolkit [5, 13], which supports the discrete-event process-based simulation where each simulation entity can be considered as a separate process [9]. The simulation entities are therefore represented by *process objects*, which are actually Java objects that possess an independent thread of control associated with them when they are created. These "active objects" then interact with each other through message passing and other simulation primitives in order to realise the operation path of the simulation.

In most cases, a simulation program needs to model the aspects of the real system to correspond to various distribution functions. JavaSim provides random number generators that follow five common distribution functions:

1. Uniform distribution
2. Exponential distribution
3. Erlang distribution
4. Hyper Exponential distribution
5. Normal distribution

These random generators, along with the simulation processes, constitute the core of JavaSim package. By using the JavaSim package, the effort required to construct a process-oriented simulation program in Java is substantially reduced.

3.2. Java package for SimML

Java supports modularity by grouping related classes into one *package*. We have built a package called `ncl.SML.Components` to represent the components and the actions of the SimML framework. This package is depicted as a UML class diagram in Figure 1. Some of the SimML components act as a container for other components. The `SMLData` may contain some `SMLAttr`, while `SMLProcess` (which inherits

from `SMLData`) may also contain some `SMLAction`. The `SMLRandComp` instances are contained by one `SMLRandom` component, just like the `SMLStatComp`s by one `SMLStatistics`.

Simulation information (which is derivable from the UML diagrams - see Section 4) can be loaded into the SimML framework before being transformed into a more suitable form. In our case, this would be a simulation program, and a function can be written to transform the simulation data into a simulation program for a particular environment, such as JavaSim (shown as `generateJavaSim()` function in Figure 1). In other words, the SimML framework acts as a bridge that enables an automatic generation of simulation programs from a UML design notation.

This framework can also be used to build simulation programs from a more formal (textual) notation such as XML. How this is achieved is discussed in Section 5.3.

4. INCORPORATING SIMULATION FRAMEWORK INTO UML

In the previous sections, we have laid out the foundation required in order to provide a mechanism that can automatically transform a UML design into a simulation program. A tool that can perform such a transformation has been built and this section will illustrate how this tool was constructed. The overall task can be split into several stages:

- investigation of possible solutions,
- building a Graphical User Interface (GUI) tool for UML, and
- generating simulation program.

When these stages are completed, the resulting tool can be used to assist the software designer to predict the performance from a UML design, which is the main issue of this paper.

4.1. Formulating a solution

There were several trails investigated before we came to one solution. In our previous work, we have built a parser that can transform a textual form of UML (using the SimML framework) into C++SIM simulation programs [3, 4]. We could therefore have adapted one of the UML tools available (such as the Rational Rose tool [16] or Argo/UML [18]) by extracting the UML information into a suitable textual format, which can then be parsed using our tool to generate a simulation program.

In the end, we decided to construct our own UML tool using Java's JFC/Swing package [12]. One of the reasons for taking this approach is because we need a tool that knows about simulation characteristics, and none of the tools mentioned above meets this demand. It is also more difficult to augment an existing tool, especially if this tool is very complex and extensive. By constructing our own tool, we aim to support the necessary design notations, and at the same time allowing the simulation characteristics of a system to be captured.

Figure 2 shows the possible paths that can be taken to generate a simulation program form a design notation. Later, we specified a formal textual notation that enables the necessary information to be interchanged among the components of our tool (Section 5).

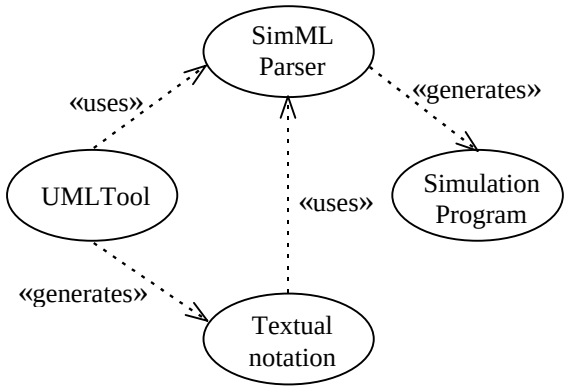

Figure 2: The use case diagram for our UML tool

4.2. Building a UML Tool

A GUI tool that brings together the UML design and the SimML framework has been created. It currently supports only the relevant UML diagrams (the class and sequence diagrams) and it allows simulation specific information to be identified in an easy and generic way. Four views are therefore supported:

1. **Class Diagram view**
   This view allows the user to draw class diagrams, specify their names and add the attributes and operations for each class (if any).
2. **Sequence Diagram view**
   The sequence diagram identifies the objects that are involved in the interaction and the messages that are sent between them. Only the SimML messages (i.e. those which are listed as SimML actions in [3, 4]) are treated in a special manner here. These SimML messages are used to construct the interactions between the objects (which represent simulation processes) hence they can be used to capture the dynamic aspects of the simulation.
3. **Random Variables view**
   The random variable names are automatically inferred from the WAIT and GENERATE messages [3, 4] of the sequence diagram. Each random variable is assigned to one of the five random distribution functions (see Section 3.1), and each distribution needs certain parameter(s) to be supplied, such as its mean and standard deviation. The random variables view allows the user to see all of the random variables used and to edit any of them to have the correct distribution function with the appropriate parameter(s).
4. **Statistics Variables view**
   The statistics variable names are created by the UPDATE messages [3, 4] of the sequence diagram. A statistics variable's type is either an *integer* or a *double* and the operations allowed are either *simple* (increment or decrement) or *calculation*. A parameter relevant to the statistics operation must also be defined, for example, the parameter for an integer-increment-by-one operation is "`+1`". As with the random variables view, the user is allowed to see all of the statistics variables and to edit them.

Our UML tool is composed of several Java classes, with a core class called `UMLEditor`. This class is supported by several Java classes, which can be divided into two categories according to their functionality:

- The classes that provide GUI facilities.

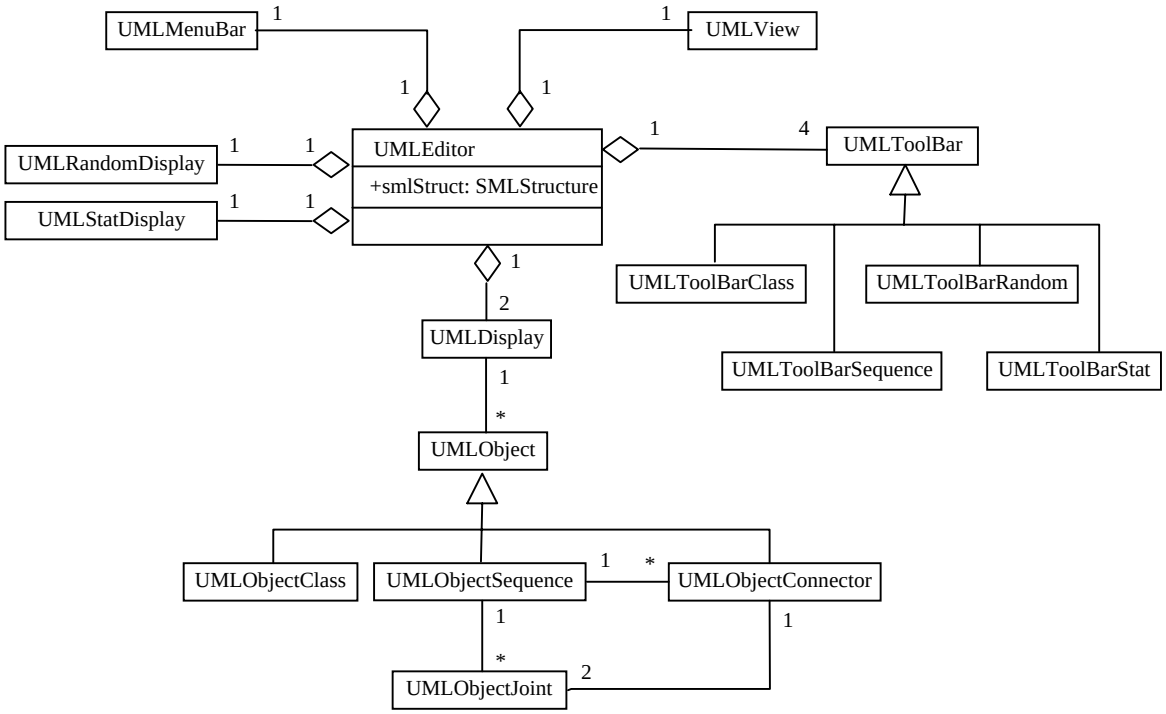

Figure 3: The structure of our UML tool

These classes allow the user to draw the class and sequence diagram notations and to select one of the four supported views mentioned above.

- The classes that support the SimML framework.
  Included in this category are the classes that store the information of the class and sequence diagrams (with their messages), as well as those for obtaining the random and statistics data for the simulation. The `UMLEditor` class has a member variable called `smlStruct` (which is an instance of the `SMLStructure` class) for storing this set of information.

The organisation of the classes used can be seen as a UML class diagram in Figure 3. By using the `SMLStructure` class for storing the information conveyed by the four views above, we are able to generate a simulation program automatically from a design notation, as explained in the next section.

4.3. Generating JavaSim code

The SimML framework enables a direct transformation from UML design notations into simulation programs. Section 3.2 outlines the implementation of the SimML framework as a package in the Java programming language. The SimML components and actions are represented as Java classes that inherit from a parent class called `SMLComponent` (Figure 1). The difference is, the SimML components act as containers within which the SimML actions can be defined.

Both the SimML component classes and the SimML action classes are referred to as the components of the `ncl.SML.Components` package. Each component of the `ncl.SML.Components` package provides a mapping to transform the information stored in it into a (segment of) JavaSim program. The container components (i.e. those which represent the SimML components) have a `write()` member function to perform this transformation. The `write()` functions of all container components are then invoked by the `generateJavaSim()` function (of the `SMLStructure` class) to obtain the complete JavaSim program. These `write()` functions can be adapted to generate simulation programs in other simulation environments, such as C++SIM or SIMULA. The modifications that need to be made are limited to the language specific syntax as the SimML framework is generic to almost all process-based simulation requirements.

5. DATA INTERCHANGE

The information conveyed in the UML diagrams needs to be stored into a file so that it can be retrieved again later. As demonstrated in Section 3 and Section 4, this information can be kept in a structured way by using the SimML framework. A widely used technique for filing a structured document is through the *Extensible Markup Language (XML)*. This section discusses the applicability of XML for storing UML and SimML related information, which can then be used to supplement our UML tool.

5.1. Extensible Markup Language (XML)

The *Extensible Markup Language (XML)* is designed to make it easy to interchange structured documents over different application programs [8]. XML is based on the idea that a structured document is made of a series of *entities* where each entity can contain one or more *logical elements*. Each element is distinguished by its *name*, and may have a *content* and/or a list of *attributes*. XML clearly marks the start and the end of each element by a pair of tags. The start tag is composed of the element name followed by its attribute list (if any), enclosed in a pair of angle brackets (<…>). The end tag is similar but the name is preceded by a forward slash character ('/') and it does not include the attribute list. The content of the element is defined in between

these two tags, and it is possible for an element to have an empty content.

XML does not have a predefined set of tags; instead, it is up to the user to define their own tags set in a formal model known as the *Document Type Definition (DTD)*. Since the XML tags are based on the logical structure (not presentational style) of the document, it is easier for a computer application to understand and to process them.

5.2. DTD for SimML

In order to define a set of tags that can be used to capture the SimML structure, we must create a DTD that formally identifies the relationships between the various components of the SimML framework. These SimML component are therefore regarded as XML elements and some of these can be seen in Figure 4. The complete SimML DTD is available at [2].

```
<!ELEMENT SPEC (DATA*, PROCESS+, QUEUE+,
  OBJECT+, RANDOMS, STATISTICS)>
<!ELEMENT DATA (ATTR*)>
<!ATTLIST DATA name CDATA #REQUIRED>
<!ELEMENT PROCESS (ATTR*, ACTION)>
<!ATTLIST PROCESS name CDATA #REQUIRED
                  span (ONCE | FOREVER)
                  "FOREVER">
<!ELEMENT ATTR (#PCDATA)>
<!ATTLIST ATTR visibility (public |
  private | protected) "public"
                type CDATA #REQUIRED>
<!ELEMENT ACTION (CREATE | WAIT
 | ACTIVATE | SLEEP | ENQUEUE | DEQUEUE
 | CHECK | RECORD | UPDATE | GENERATE
 | END | IF | ELSIF | ELSE | WHILE)*>
...
```

Figure 4: A snapshot of the SimML DTD

The SimML DTD dictates that a valid XML document must start with a `<SPEC>` tag (and consequently end with a `</SPEC>` tag). A specification is composed of `DATA`, `PROCESS`, `QUEUE`, `OBJECT`, `RANDOM` and `STATISTICS` elements, which in turn are composed of smaller elements.

5.3. XML Parser for SimML using SAX

A suitable parser is required to read the information stored in an XML document. For that purpose, we have built an application program that parses an XML document written to follow the SimML DTD. This program was written as a Java package (`ncl.SML.Parser`) to allow other application programs (such as the UML tool described in Section 4) to re-use its parsing features.

There is an Application Programming Interface (API) called *SAX* (Simple API for XML) [14], which is essentially another Java package that provides a skeleton for parsing any XML document. SAX is an event-based API, which means that it reports parsing events (such as the start and end of elements) directly to the application. The application must therefore implement a handler to deal with different events; three of the most important ones are listed here:

- *startElement* event
  This event is raised by the parser when it detects the beginning of every element in an XML document. The application handler must then obtain the element's name and if any, the list of its attribute. For SimML handler, the information gathered from this event is used to initialise the appropriate components of the SimML structure.
- *endElement* event.
  When the end of an element is reached, the handler must update the named element, which for SimML handler means updating the right component of the SimML structure.
- *character data* event
  In between the startElement and the endElement events, the parser returns all character data as a single chunk of information. This chunk actually represents the content of the element, therefore it must be stored by the corresponding SimML component.

An application program based on the SAX approach has been built to read any XML notation that conforms to the SimML DTD. The information read is then stored as a SimML structure, which is transformable into a JavaSim simulation program. We have also augmented our UML tool to support the same feature by implementing a class called the `XMLReader`. This class is incorporated into our `UMLEditor` tool and it allows the user of our tool to read a SimML XML file and display the information as class and sequence diagrams. The random variables and statistics information is also loaded into the appropriate views of this tool.

6. SIMPLE EXAMPLE AND RESULTS

As an example, let us consider a simple queuing system where jobs arrive into the system randomly according to an exponential distribution with mean 5. The jobs are then placed into a queue before being processed by a server. By keeping track of how many jobs are completed and the total time spent by these jobs in the system, we can work out the average response time. This problem can be represented as a simulation diagram in Figure 5, and as a UML sequence diagram in Figure 6 (the class diagram, random and statistics views of our UMLEditor tool are not shown here).

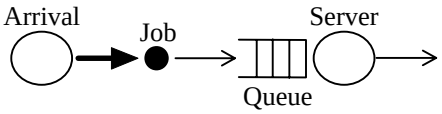

Figure 5: Diagram of a simple queuing system

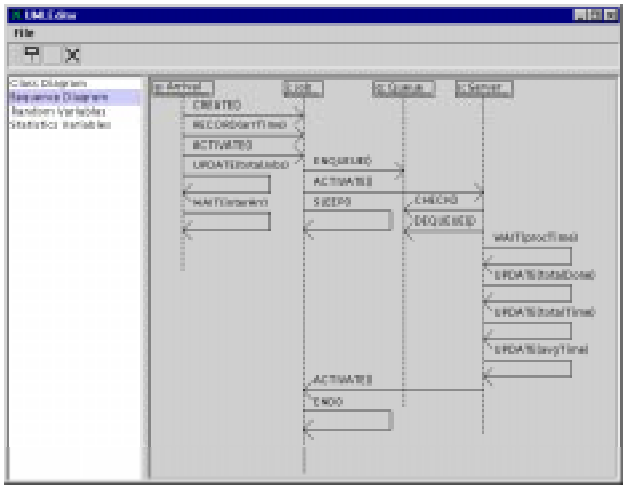

Figure 6: Sequence diagram of a simple queueing system

The performance of this system depends on how quick the server can process the jobs; a faster server costs more than a slower one. The server takes an exponentially distributed time to complete each job, and by altering the mean of this delay, different results will be obtained. These results reflect the performance of the system for each scenario, hence we can choose one which satisfies the requirements with a minimal cost.

A simulation program is generated using our UMLEditor tool and two simulation experiments were conducted using different server delays: one with a mean processing delay of 6 and the other of 4. Each simulation is run for a length of 10,000, which generates 1929 jobs. The results obtained from the two simulations are compared in Table 1.

Table 1: Simulation results

processing delay	total proc. time	total jobs completed	avg. proc. time
6	1364886.5	1613	846.2
4	7995.9	1928	4.1

The results show that the system performs badly if the Server takes too long to process each job. When the processing delay is 6 (which is greater than the inter arrival time of 5), the queue grows quite rapidly and the jobs spend most of the time waiting in the queue. This is shown by the high number of uncompleted jobs and the high average processing time. On the other hand, when the processing delay is 4 (i.e. it is less than the inter arrival time), virtually all of the jobs is completed and the average response time is very close to the expected value (4.1).

7. CONCLUSIONS AND FURTHER WORK

The work presented in this paper enables us to evaluate whether a particular system design will deliver its performance requirement or not. The tool that has been constructed allows the system developer to design a new system using the UML Class and Sequence diagram notations. These diagrams, along with some random and statistics information, can then be used to generate a simulation program to mimic the execution of the proposed system. From the simulation run, we can predict the kind of performance that the system will deliver, and hence we can decide whether it is worth it or not to implement this design.

Other work that has been conducted in this area includes the Parmabase project [1], which puts the emphasise on the UML Deployment and Component diagrams to derive a model of the system. Work by Pooley and King [15] argues that the UML sequence diagram is more suitable as a display format rather than for detailing the behaviour of a system. Although this is true to some extent, we have shown that it is possible to use the sequence diagram to specify the behavioural characteristics of a simulated system with an aid of the SimML framework.

The UML tool that we have constructed can be improved in several ways. First, the class diagram can be utilised more to allow the random properties of the static objects to be specified. This can be achieved by attaching a “note” to the appropriate class. The associations between the classes can also be used to indicate the multiplicity of the objects involved in the system. This is useful, for example, to evaluate the effect of adding another server to process a queue. We are currently investigating the use of this tool in more complex situations.

REFERENCES

[1] Akehurst, D. H. and A. G. Waters, “UML Specification of Distributed System Environments”, Computing Laboratory, University of Kent at Canterbury, Technical Report 18-99 (May, 1999).

[2] Arief, L. B., “DTD for the SimML Framework”, (http://www.cs.ncl.ac.uk/~l.b.arief/home.formal/SimML.dtd).

[3] Arief, L. B. and N. A. Speirs, “Automatic Generation of Distributed System Simulations from UML”, *Proc. 13th European Simulation Multiconference (ESM'99)*, Warsaw, Poland, pp. 85-91 (June, 1999).

[4] Arief, L. B. and N. A. Speirs, “Simulation Generation from UML Like Specifications”, *Proc. IASTED International Conference on Applied Modelling and Simulation*, Cairns, Australia, pp. 384-388 (September, 1999).

[5] Arjuna-Team, “C++SIM User's Guide”, Department of Computing Science, University of Newcastle upon Tyne (1994).

[6] Arnold, K. and J. Gosling, *The Java Programming Language*, Addison-Wesley (1996).

[7] Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley (1999).

[8] Bryan, M., “An Introduction to the Extensible Markup Language (XML)”, The SGML Centre (http://www.personal.u-net.com/~sgml/xmlintro.htm).

[9] Computing-Laboratory, “The JavaSim User's Manual”, Department of Computing Science, University of Newcastle upon Tyne (1999).

[10] Fowler, M. and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley (1997).

[11] IBM, “XML Parser for Java - XML4J”, IBM Alpha Works (http://www.alphaworks.ibm.com/tech/xml4j).

[12] Java-Team, “Creating a GUI with JFC/Swing”, (http://java.sun.com/docs/books/tutorial/uiswing).

[13] Little, M. C. and D. L. McCue, “Construction and Use of a Simulation Package in C++”, Department of Computing Science, University of Newcastle upon Tyne, Technical Report 437 (July, 1993).

[14] Megginson, D., “SAX: The Simple API for XML”, (http://www.megginson.com/SAX/index.html).

[15] Pooley, R. J. and P. J. B. King, “The Unified Modeling Language and Performance Engineering”, *IEE Proceedings on Software*, Vol. 146, No. 1, pp. 2-10 (February 1999).

[16] Rational, “Rational Rose”, (http://www.rational.com/products/rose/).

[17] Rumbaugh, J., I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley (1999).

[18] UCI, “Argo/UML - Providing Cognitive Support for Object-Oriented Design”, (http://www.ics.uci.edu/pub/arch/uml/).

[19] W3C, “Extensible Markup Language (XML)”, (http://www.w3.org/XML/).