Technical Report TR-731

Submitted to the 2nd International Workshop on Software Performance (WOSP 2000), September 2000

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.

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)* [1-3] 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)* [4, 5] 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 [7]. The simulation programs are generated in the Java programming language using the *JavaSim* [8] package developed at the University of Newcastle upon Tyne. We decided to use the *Extensible Markup Language (XML)* [9, 10] 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)* [11] through IBM's *XML4J* parser [12].

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.

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 [3]. Our interest here lies with the *class diagram* (which denotes the static structure of a system i.e. its objects or classes) and the *interaction* diagram (which depicts the interaction pattern between the objects in the system). Of the two types of interaction diagrams, *sequence* and *collaboration* diagrams, have used the former for capturing the system's behaviour. The sequence diagram arranges the interactions in time sequence and this conforms neatly to the discrete-event process-oriented simulation paradigm that we are going to use.

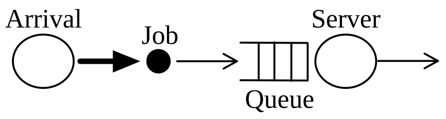


Figure 1: Simulation diagram of a simple queuing system

To illustrate how the UML can be used to design a system which may be simulated, let us consider a simple queuing system where jobs arrive into the system randomly with some probability distribution every certain interval. The jobs are then placed into a queue before being processed by a server. The server takes a certain time to process each job before the job is completed. The performance data consists of the time taken for a job to be completed. 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 1.

2.1. Static Structure: Class 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 (which is used to differentiate one simulation entity from another) and the class attributes (which are used to store the information that are necessary for predicting the performance). Therefore, the static characteristics of the queueing system shown in Figure 1 can be captured as UML class diagrams seen in Figure 2.

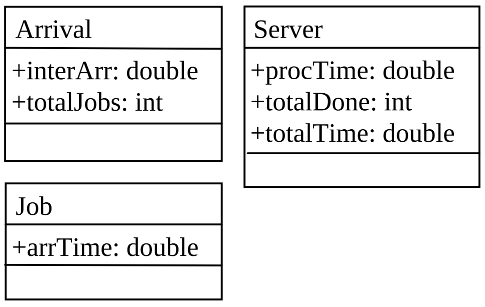


Figure 2: A Class diagram notation for a simple queueing system

This queueing system consists of three processes (Arrival, Job and Server), and each process can be represented as a class diagram in UML. The Arrival class has two attributes: a randomly generated inter arrival time (which determines when a new Job should be created) and a statistics variable to keep track of the number of Jobs created. The Job class only needs to know its arrival time, which is used later by the Server class to work out the time taken to serve all of the completed jobs. The counter for completed jobs is updated by the Server class as soon as it finishes processing a job, which depends on its randomly generated delay time.

The plus sign ('+') indicates a *public* visibility of the corresponding attribute. *Private* visibility is denoted by a minus sign ('-') while *protected* visibility is indicated by a hash ('#').

The behavioural properties of this system will be depicted as a sequence diagram in the following section.

2.2 Dynamic Structure: Sequence Diagram

UML provides an interaction diagram to model the dynamic aspects of a system. It 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 diagram 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.

The dynamic properties of the simple queueing system mentioned before can be transformed into a sequence diagram which is shown as Figure 3.

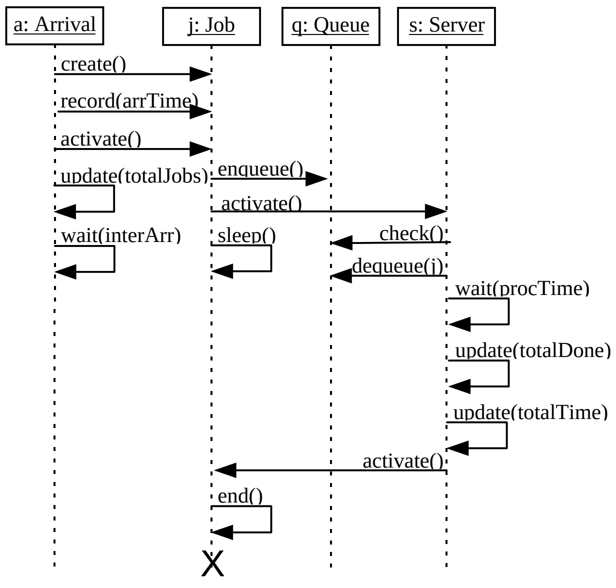


Figure 3: A sequence diagram notation for a simple queuing system

In Figure 3, the objects that participate in the interaction are identified at the top of the sequence diagram. The arrows indicate the direction of the messages which are drawn in order of increasing time from top to bottom. An arrow that points to itself represents a message that does not involve any other object. Each object has a name and must be of a specified type. For example, object "a" is an instance of an Arrival process which creates a new job, records the arrival time of the job and activates the job. It also updates "totalJobs" - a statistics variable that is used to keep track of how many jobs have been created. As the next job arrival happens at a random time, the Arrival process must wait for the inter arrival time (which is modelled to a particular distribution function) before it creates another job.

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. The following section will discuss the work done in order to attain this goal.

3. Simulation Framework

In order to assist the construction of simulation programs, a generic simulation framework has been specified. This framework enables us to perform the automatic transformation from design to simulation.

Section 3.1 introduces the simulation framework that has been developed during our research. This framework is then implemented as a Java package (Section 3.2) to allow simulation programs to be written in Java programming language. Section 3.3 gives an insight into the simulation environment that we are going to use, namely the JavaSim simulation package [8].

3.1. Simulation Modelling Language (*SimML*)

We have developed a framework called *SimML* (*Simulation Modelling Language* [4, 5]) which identifies generic simulation components that can be used for constructing process-based event-oriented simulation programs.

When designing this framework, we first identified some components which are commonly found in simulation programs:

- **PROCESS**
  This is the most important component since we are using a process-oriented approach for our

simulation. A PROCESS is used to represent an active object in the simulated system and different PROCESS'S are characterised by different names, attributes and operations. These static characteristics can be drawn out from the UML class diagram. A PROCESS can also interact with other PROCESSes during the simulation, and for that purpose, we have specified some *action*s (see later) that can be performed by a PROCESS. The UML sequence diagram can be used to convey these actions along with the PROCESSes involved.

- **DATA**
  It is a simplified version of the PROCESS component which does not need to be an active object (hence it takes less resources during the simulation). It can be used to represent passive simulation objects (i.e. just a data structure) and it can be derived from the UML class diagram.
- **QUEUE**
  A queuing mechanism is a very important concept in simulation and hence a way of specifying queues (for different types of object) must be provided. Here, a QUEUE component can be specified in the UML sequence diagram, usually next to PROCESS component which manipulates the queue.
- **OBJECT**
  An OBJECT is an instance of either a PROCESS, a DATA or a QUEUE component. In UML term, it corresponds to the name of the object specified on top of the sequence diagram, i.e. the instance name of a class. The OBJECT components therefore allow us to specify the objects that are involved in a particular interaction among the simulation components.
- **RANDOMS**
  Simulation programs need random numbers to model the notion of "time" according to a particular statistical distribution. Several distribution functions are supported by the SimML framework, such as the exponential, normal, uniform, hyper-exponential and erlang distributions.
- **STATISTICS**
  The STATISTICS component enables us to collect the information that is relevant to the performance of the simulated system, such as the average service time, number of processed jobs, number of jobs lost, etc.
- **CONTROLLER**
  This component is always required to be present in a JavaSim simulation. It acts as the main thread that initialises the simulation, sets up the simulation parameters and summarises the simulation results. Since the functionality of this component is always the same for every simulation, it was decided to make it as a template component.
- **MAIN PROGRAM**
  The MAIN PROGRAM is another template component that creates the CONTROLLER to start the simulation.

The components listed above represent the static properties of the simulation. On top of that, instances of the PROCESS component (i.e. the OBJECT components) may interact with each other and this interaction can be used to model the dynamic aspects of the simulation. These interactions are termed as *action*s that can be performed by the PROCESS components. There are several actions provided:

1. *create*: declares a new instance of DATA or PROCESS component.
2. *wait*: reschedules the current process to be activated later after a given time.
3. *activate*: activates another process.
4. *sleep*: passivates the current process.
5. *enqueue*: places an object (either of a DATA or PROCESS type) onto a queue.
6. *dequeue*: removes an object from the head of a queue.
7. *check*: passivates the process from which this action is invoked if there are no more items on the queue.
8. *record*: sets the value of an objects member variable to the current time or a specified value.
9. *update*: updates the value of a statistics variable (used in conjunction with the STATISTICS component).
10. *generate*: produces a number randomly, using a particular random number generator (as specified in the RANDOMS component).
11. *end*: terminates the execution of the current process. A terminated process will no further take part in the simulation.

There are two special actions that determine the flow of the interactions by allowing some conditions

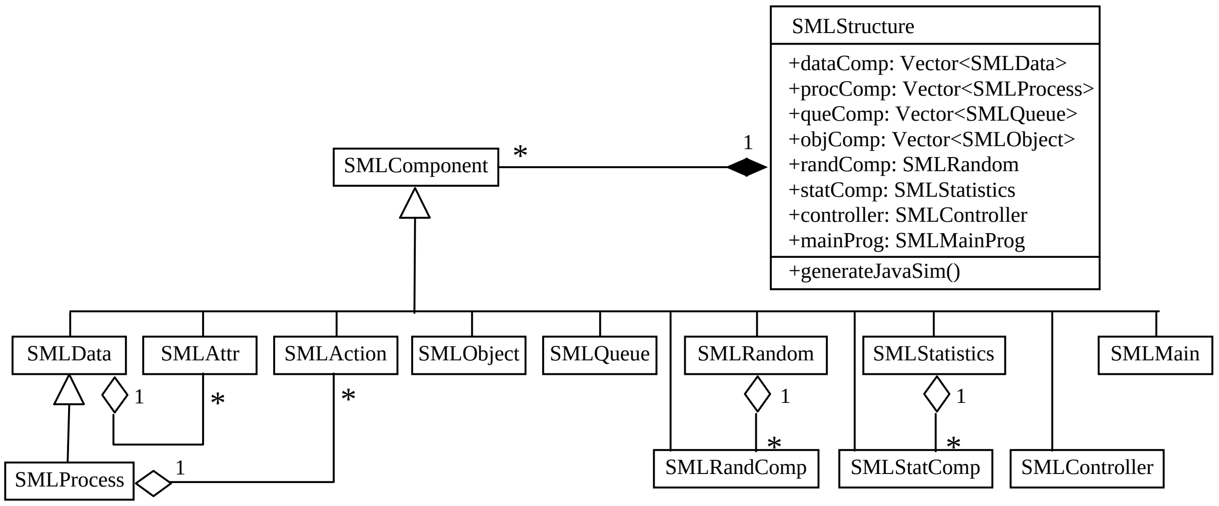


Figure 4: The class diagram for the SimML Components

to be specified in order for a particular action to be executed:

12. *if*: specifies a condition that must be satisfied before an action can be performed. It is complemented by *elsif* and *else* actions.
13. *while*: allows a loop to repeat the same action(s) until certain condition is satisfied.

Most of these actions require parameters and may involve other OBJECTs.

The SimML framework has been used to build a parser that can transform a textual representation of UML into JavaSim or C++SIM [13, 14] code. Further details about this framework, especially on the parameters of the actions can be found in [4, 5].

We have constructed a GUI tool that can do the transformation directly from the UML graphical notation to JavaSim simulation code. This required the tool to provide some drawing or graphics support customised to the shapes required by the relevant UML notation. To achieve this, we used the Java programming language since it comes with a JFC/Swing package which reduces the effort needed for constructing a GUI tool by providing standard re-useable GUI components. Since Java programs are portable on virtually any computer architecture, our tool is not platform dependent.

The SimML framework can then be implemented as a Java package as explained in the next section.

3.2. Java package for SimML

Java supports modularity by grouping related classes into one package. A package is a compilation unit that encapsulates several classes, interfaces and sub-packages into one file with an aim to improve the organisation of the program. It also helps to resolve naming problems and promotes software reuse.

The components and actions of the SimML framework can be represented as Java classes and grouped into one Java package. We call this package `ncl.SML.Components`; the structure of this package is represented as a UML class diagram in Figure 4. Some of these 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 `SMLStatComps` 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 4). 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. We now discuss the simulation environment that we used.

3.3. Simulation Environment: *JavaSim*

JavaSim is a Java implementation of the original C++SIM simulation toolkit [13, 14], which supports the discrete-event process-based simulation where each simulation entity can be considered as a separate process [8]. 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.

A *scheduler* manages the simulation processes (i.e. the active objects) and places these processes on a *scheduler queue* (the event list). Processes are executed in a *pseudo-parallel* mode: only one process is executed at any instance of real time, but many processes may be executed concurrently at any instance of simulation time. Only when all processes that are scheduled for a particular simulation time have been executed, can the simulation clock be advanced.

There is a `Process` base class, from which all process objects inherits their process functionality. This class defines all of the necessary operations to control the simulation and to allow the process objects to interact with each other. More details on the JavaSim processes can be found in [8].

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.

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 [4, 5]. We could therefore have adapted one of the UML tools available (such as the Rational Rose tool [15] or Argo/UML [16]) by extracting the UML information into a suitable textual format, which can then be parsed using our tool to generate a simulation program.

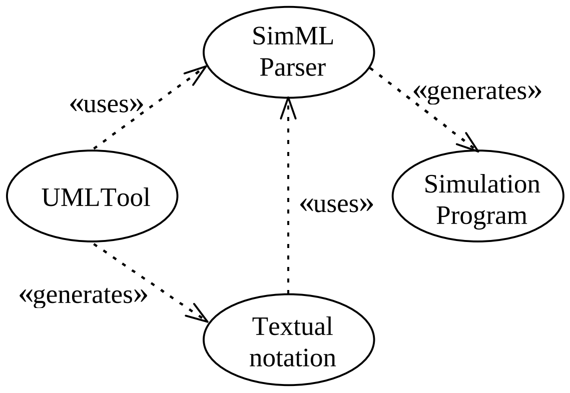


Figure 5: The use case diagram for our UML tool

In the end, we decided to construct our own UML tool using Java's JFC/Swing package [7]. 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 5 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 (see Section 5).

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 Section 3.1) 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 of the sequence diagram. Each random variable is assigned to one of the five random distribution functions (see Section 3.3), 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 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.
  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 6. 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

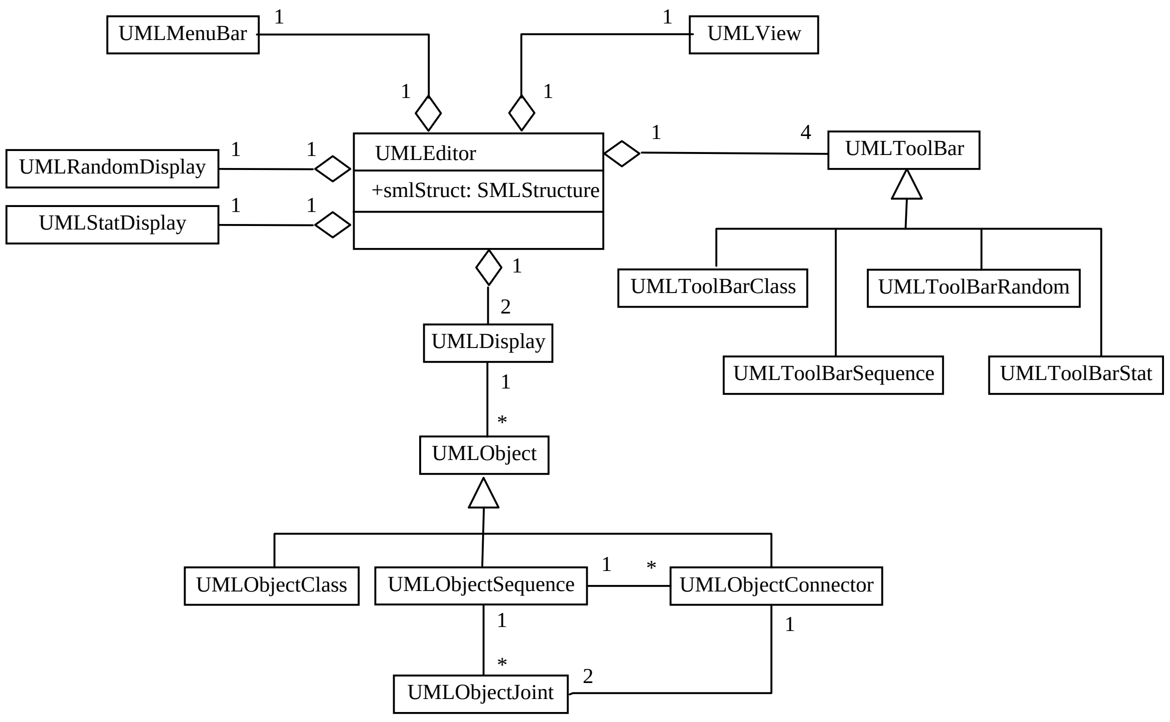


Figure 6: The structure of our UML tool

SimML components and actions are represented as Java classes that inherit from a parent class called `SMLComponent` (Figure 4). 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 [10].

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

```
<!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 7: A snapshot of the SimML DTD

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 7. The complete SimML DTD is available at [17].

Note that the element names are case sensitive. An element is declared using the `<!ELEMENT...>` construct that specifies the name of the element and its content. The content of an element is either some other elements or a plain text (indicated as `#PCDATA`). If an element needs to have some attributes, it must have an attribute list declared using the `<!ATTLIST...>` syntax.

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. XML provides a method to indicate the multiplicities of each element. Element that may be present *zero or more* times are marked by a star sign (‘*’), while a plus sign (‘+’) indicates those that can occur for *one or more* times. Optional elements are indicated by a question mark (‘?’).

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) [11], 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.

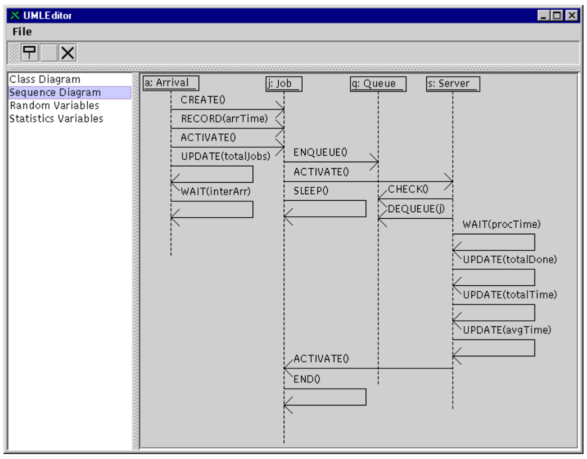
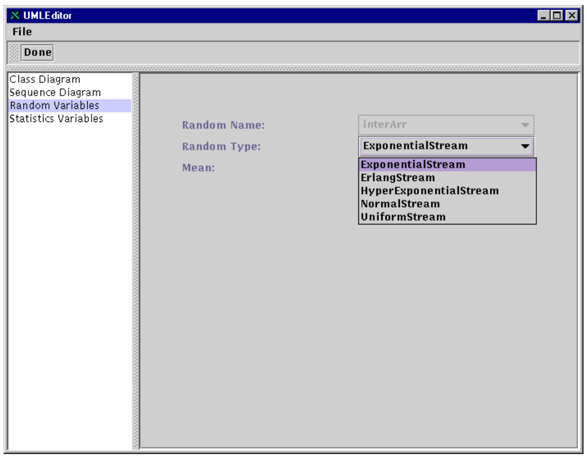
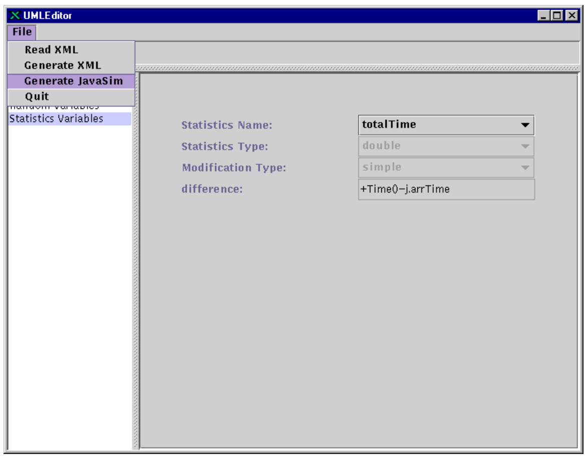


Figure 8: The UMLEditor's sequence diagram view of a simple queuing system



Random Name: interArr
Random Type: ExponentialStream
Mean:

Figure 9: The Random Variable view



Statistics Name: totalTime
Statistics Type: double
Modification Type: simple
difference: +Time()–j.arrTime

Figure 10: The statistics variable view and the UMLEditor's File menu

6. Simple Example and Results

The static characteristics of the simple queueing system problem mentioned in Section 2 can be represented as a class diagram as seen in Figure 2. Here, we show how we represent the dynamic aspect of the same system (as a sequence diagram) using our UML tool (Figure 8).

The random variables and statistics information views are displayed in Figure 9 and Figure 10; the later also shows the File Menu of our UML tool for reading/writing the data from/into an XML file and for generating a JavaSim program. An XML representation of this queuing system can be seen in Figure 11.

The random variables set up for this system indicate that the inter arrival time is exponentially distributed with mean 5, while the Server's processing delay is exponentially distributed with mean 6. Later, the processing delay is reduced to 4 in order to compare the results and to analyse the performance of the system. 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

```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE SPEC SYSTEM "http://www.cs.ncl.ac.uk/
 people/l.b.arief/home.formal/SimML.dtd">

<SPEC>
  <PROCESS name="Job">
    <ATTR visibility="public" type="double">
      arrTime</ATTR>
    <ACTION>
      <ENQUEUE to="q">this</ENQUEUE>
      <ACTIVATE>s</ACTIVATE>
      <SLEEP></SLEEP>
      <END />
    </ACTION>
  </PROCESS>

  <PROCESS name="Arrival">
    <ACTION>
      <CREATE type="Job">j</CREATE>
      <RECORD of="j">arrTime</RECORD>
      <ACTIVATE>j</ACTIVATE>
      <UPDATE>totalJobs</UPDATE>
      <WAIT>interArr</WAIT>
    </ACTION>
  </PROCESS>

  <PROCESS name="Server">
    <ACTION>
      <CHECK>q</CHECK>
      <DEQUEUE from="q">j</DEQUEUE>
      <WAIT>procTime</WAIT>
      <UPDATE>totalDone</UPDATE>
      <UPDATE>totalTime</UPDATE>
      <UPDATE>avgTime</UPDATE>
      <ACTIVATE>j</ACTIVATE>
    </ACTION>
  </PROCESS>

  <QUEUE of="Job">Queue</QUEUE>
  <OBJECT type="Arrival">a</OBJECT>
  <OBJECT type="Queue">q</OBJECT>
  <OBJECT type="Server">s</OBJECT>

  <RANDOMS>
    <EXPONENTIAL mean="5">
      interArr</EXPONENTIAL>
    <EXPONENTIAL mean="6">
      procTime</EXPONENTIAL>
  </RANDOMS>

  <STATISTICS>
    <SIMPLE type="double" diff="+Time()-
      j.arrTime">totalTime</SIMPLE>
    <SIMPLE type="int" diff="+1">
      totalJobs</SIMPLE>
    <SIMPLE type="int" diff="+1">
      totalDone</SIMPLE>
    <CALC type="double" expr="=totalTime/
      totalDone">avgTime</CALC>
  </STATISTICS>
</SPEC>
```

Figure 11: An XML notation for a simple queueing system

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).

This simple example demonstrates the feasibility of using our UML tool to design a system and to derive its performance prediction automatically through a simulation.

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 [18], which puts the emphasise on the UML Deployment and Component diagrams to derive a model of the system. Work by Pooley and King [19] 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] M. Fowler and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.

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

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

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

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

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

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

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

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

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

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

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

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

[14] M. C. Little 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.

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

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

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

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

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