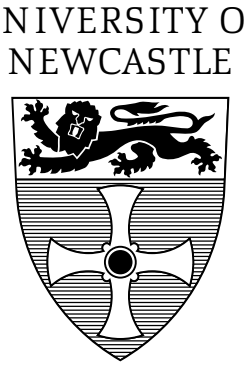
UNIVERSITY OF NEWCASTLE UPON TYNE

DEPARTMENT OF COMPUTING SCIENCE



A Framework for Supporting Automatic Simulation Generation from Design

by

Leonardus Budiman Arief

Ph.D. Thesis

July 2001

Abstract

Building a new software system requires careful planning and investigation in order to avoid any problems in the later stages of the development. By using a universally accepted design notation such as the *Unified Modeling Language (UML)*, ambiguities in the system specification can be eliminated or minimised.

The aspect that frequently needs to be investigated before the implementation stage can be commenced concerns the proposed system's *performance*. It is necessary to predict whether a particular design will meet the performance requirement - i.e. is it worth implementing the system - or not. One way to obtain this performance prediction is by using simulation programs to mimic the execution of the system. Unfortunately, it is often difficult to transform the design into a simulation program without some sound knowledge of simulation techniques. In addition, new simulation programs need to be built each time for different systems - which can be tedious, time consuming and error prone.

The currently available UML tools do not provide any facilities for generating simulation programs automatically from UML specifications. This shortcoming is the main motivation for this research. The work involved here includes an investigation of which UML design notations can be used; the available simulation languages or environments for running the simulation; and more importantly, a framework that can capture the simulation information from UML design notation. Using this framework, we have built tools that enable an automatic transformation of a UML design notation into a simulation program. Two tools (parsers) that can perform such a transformation have been constructed. We provide case studies to demonstrate the applicability of these tools and the usefulness of our simulation framework in general.

Acknowledgement

I would like to thank my supervisor, Dr. Neil Speirs for his guidance and useful feedbacks throughout my Ph.D. work. A lot of people at Newcastle University have also helped me towards the completion of this thesis, especially Prof. Isi Mitrani for explaining everything I need to know about simulation, Prof. Santosh Shrivastava and Dr. Mark Little for providing initial direction and one of the case studies, Prof. Cliff Jones for letting me finish my thesis while working as a research associate on his project, our librarian Ms. Shirley Craig for assisting me in finding references, and the Department of Computing Science for funding my Ph.D.

I would also like to acknowledge my colleagues for many fruitful and interesting discussions, in particular to Dr. Denis Besnard, Dr. Cristina Gacek, Mr. Tony Lawrie, Dr. Graham Morgan, Mr. Thomas Rischbeck, and Dr. Arnaud Simon. I am grateful to my friends, wherever they are now, for cheering me up and giving me comfort. Last but certainly not least, I would like to express my special gratitude to my Mom, without whose constant encouragement and prayer I would never finish this thesis.

Table of Content

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Overview

The construction of a new computer system is an inherently complicated process, hence it requires some careful planning and design. For example, in the area of distributed systems, a computer application usually needs to satisfy quite stringent requirements such as reliability, availability, security, etc. and the cost of building such an application will be quite high. It is therefore desirable to be able to predict the performance of the proposed system before the construction begins.

In order to solve this problem, it is important to evaluate the requirements of the new system and translate them into a specification for design purpose. The design process helps the system developers to understand the requirements better and to avoid misconceptions about the system. From the specification, a simulation program can be built to mimic the execution of the proposed system. The simulation run provides some data about the states of the system and from these data, the performance of the system can be predicted and analysed.

There is a drawback though as it is often difficult to transform the design specification into a simulation program. The specification is normally quite complex and different performance requirements (such as high availability, reliability, etc.), connection between components (network delay, partition, etc.) and fault tolerance (machine failures and repairs) require additional effort. On top of that, a new

simulation program needs to be built each time for a different specification, and the system developer must also be familiar with the simulation techniques - which is not always the case.

The currently available design methods/tools (such as the *Unified Modeling Language* or UML [18, 31, 63, 64]) do not provide a way to automatically generate simulation programs from the specification. There has been some effort in eliminating this shortcoming in areas outside UML, such as in the Rapide [53, 54] and DisCo [40] projects, but at the time this Ph.D. project started, there was no tool available yet that is able to transform a system specification in UML (with certain performance requirements) into a simulation program. Since then, there have been some effort trying to address the lack of performance prediction features in UML, such as the work by the PERMABASE project [1-3, 72], Pooley and King [62] and Bondavalli et. al [15].

The aim of this Ph.D. study was therefore to produce a tool or method that allows software performance prediction to be incorporated into the UML design specification. To achieve this goal, the tool should be able to generate simulation programs from UML design specification without the necessity of the system designers to know how to code a simulation program. This tool should be suitable for a variety of system applications and will allow different performance requirements to be incorporated in the simulation.

1.2. Related Work

There have been some projects trying to incorporate software performance prediction into system design/specification by tackling the problems from many different angles. In relevance to my Ph.D. work, these related projects can be divided into two main

categories: those that do not use UML at all and those that use UML (to some degree) for specifying a system. The second has only emerged recently (since around 1999) and the number of projects that fall into this group has grown quite rapidly.

1.2.1. Related Work outside UML

Rapide

Rapide is a computer language for defining and executing models of system architectures [53]. It introduces an *interface connection architecture* which means that all communication between modules is explicitly by connections between interfaces. Here, an architecture consists of a set of *specification modules* (*interfaces*), a set of *connection rules* (for defining the communication between the interfaces), and a set of *formal constraints* which determines whether a pattern of communication is legal or illegal.

A *component* consists of a *module* and its *interface*. A module either encapsulates an executable prototype of the component or it describes the hierarchical architecture of the component (when a component is composed of other components). An interface defines what a module requires-from or provides-to other modules and connections are defined between the features in interfaces.

The Rapide language is accompanied by a set of tools which help in the specification, design and testing of software modules and architectures. In general, the Rapide language is composed of five sublanguages:

1. *The Type Language*: describes the interfaces of components. It supports object oriented and abstract data type styles of defining interfaces as well as multiple interface inheritance.

2. *The Pattern Language*: provides a general language for defining event-based reactive constructs or dynamic architectures.
3. *The Executable Language*: is used for writing executable modules which are defined by a set of processes which observe and react to events.
4. *The Architecture Language*: models interface connection architecture and it defines dataflow and synchronisation between modules.
5. *The Constraint Language*: provides features for specifying formal constraints on the behaviour of components and architectures.

The feature provided by Rapide is very extensive and Rapide provides a simulation tool using an event-based execution model or POSET (partially ordered set of events).

DisCo

DisCo (Distributed Co-operation) is a formal specification method for reactive systems which incorporates a specification language, a methodology for using this language for building specifications, and a tool for supporting the methodology [40].

It focuses on the collective behaviour of the objects, i.e. how they cooperate with each other. As well as supporting an object oriented approach, DisCo is based on the joint action approach which concentrates on the *interaction* between different components instead of the components themselves. This increases the level of abstraction, i.e. the bias towards particular hardware and software architecture is minimised.

The DisCo language supports modularisation (due to its object oriented paradigm) and incremental specification (which means that the level of the specification's details can be gradually increased until the desired level is met). A

DisCo specification consists of a set of *layers* (which are composed of classes and actions). New layers can be constructed by composing two or more separate layers or by refining existing layers (*superposition*).

The DisCo tool provides a way to validate a specification by using animated simulations. Animation makes specifications much more understandable and promotes communication between the people involved. The DisCo tool also supports graphical representation of execution scenarios (*Message Sequence Charts*). A short tutorial on DisCo is available at [46].

AUTOFOCUS

AUTOFOCUS is a tool prototype for the formally based development of reactive systems [36] - mainly in the area of distributed systems. It supports system development by offering integrated, comprehensive and mainly graphical description techniques for specifying both different views and different abstraction levels of the system. There are four different description techniques provided to cover the different views on the system:

- *System Structure Diagrams* (SSDs): describe the static aspects of distributed systems by viewing them as a network of interconnected components with an ability to exchange messages over their communication channels.
- *Data Type Definitions* (DTDs): represent the types of the data processed by a distributed system in a textual notation.
- *State Transition Diagrams* (STDs): describe the dynamic aspects, i.e. the behaviour of a distributed system and its components.
- *Extended Event Traces* (EETs): provides extra behavioural view of a distributed system (on top of STDs) through exemplary runs from a component-based view.

AUTOFOCUS supports component-oriented development of systems, where a component represents a structural part of the system, possibly described by different views using the description techniques above, which allow different levels of system granularity to be specified.

AUTOFOCUS uses prototyping and simulation approach for observing and validating the properties of a system being developed. To support this approach, AUTOFOCUS provides a tool component called "SIMCENTER" that facilitates:

- the generation of executable prototypes of systems or parts thereof,
- the execution of these prototypes in a simulation environment,
- the visualisation of the runs using the same description techniques as used for designing the system, and
- an optional connection between the simulation environment and third-party front-ends such as multimedia visualisation tools or external hardware systems.

Both AUTOFOCUS and SIMCENTER are written entirely in Java programming language.

Architectural Modelling Box (AMB)

AMB is a modelling and design language that provides a unified basis for the design process as well as functional and quantitative analysis [44]. The aim of this project is to bring together the worlds of system designers and performance modellers by introducing a design language that includes quantitative properties of systems.

AMB is formed by a graphical language that models systems, their behaviour as well as other relevant data. In general, AMB models consist of two parts:

- *resource* or *entity* model: describes the *static* aspects of the system, i.e. its components, their physical properties and the way they can interact.

- *behaviour* model: describes the *dynamic* aspects, i.e. the processes or actions performed by the system, in terms of related activities.

Specifications written in the AMB design language can then be automatically translated into models for performance analysis (such as *graph models, timed Petri nets* or *hybrid models*) or for functional analysis. More "traditional" performance modelling formalisms (such as regular queueing networks or quantitative simulations) are not suitable for AMB's purpose because their expressive power is too limited to describe most of the aspects that AMB users are interested in.

1.2.2. Related Work using UML

PERMABASE

The Performance Modelling for ATM Based Applications and Services (PERMABASE) project [1, 3, 72] was carried out by British Telecom and the University of Kent at Canterbury. The aim of this project is to provide performance feedback as part of the object-oriented design process for distributed systems through an automatic generation of performance model directly from the system design model.

This project tries to address the lack of use of physical environment (hardware) specification in distributed system designs. There are four model viewpoints identified here:

1. *Workload Specification*: specifies the "work force" (human operators or other systems) that drives the system; this includes classes of components considered as external to the system.
2. *Application Specification*: specifies the classes of components that constitute the software or logical behaviour of the system, i.e. the system logic components (software, firmware or hardware logic).

3. *Execution Environment Specification*: identifies classes of components that are physical components providing the resources used by the application during system operation, e.g. processors, networks and other resources that the system operates over.
4. *System Scenario*: defines the instances of declared components (of the three above) and the connections between them, which form a specific system architecture or configuration.

By combining the specifications of each of the viewpoints above, a *Composite Model Data Structure* (CMDS) of the entire system is obtained. The CMDS can then be checked for consistency and translated into a discrete event simulation or performance model of the system.

This project tried to use, and adapt if necessary, the UML notations for the representation of the four domain specification areas. For example, the Execution Environment Specification can be described using the *class* diagram, while the System Scenario view can be represented using the *deployment* diagram.

As a follow on from the PERMABASE project, Akehurst and Waters propose a list of UML deficiencies with respect to performance modelling [2].

Pooley and King

The goal of this work is to integrate performance estimation with the system design process [62]. It was suggested that successful effort toward this goal most likely come from projects that attempt to build performance analysis directly into accepted design method such as UML.

The UML *sequence* diagrams were initially thought to have the potential to generate and display useful information relating to performance. In the end, it was

decided that the sequence diagrams are more suited to being a display format rather than a detailed behavioural specification format. It is mentioned that the sequence diagrams have been used as traces of events generated from a simple discrete event simulator. The work have also found a mapping from the *deployment* diagram to queuing models, and have built a simulation library around *collaborations* with *state machines.*

The challenge lies in setting the foundation for an integrated performance engineering approach on the whole UML notation. The use of collaboration diagrams with embedded state machines seems very promising, as well as its extension to incorporate collaborations within deployment diagrams.

In a follow on paper [48], King and Pooley describes how UML designs can be transformed systematically into Petri nets. The UML diagrams of interest here are the *collaboration* and *statechart* diagrams, and the target of the transformation is Trivedi's SPNP [24] tool's variant of stochastic Petri nets. Further exploration of systematic mappings from UML is undertaken by developing a graphical front end. The possibility of employing layered queueing networks (on top of Petri nets) as the targets of UML transformation is also being investigated.

In [60], Pooley outlines a roadmap that shows how software performance can be integrated with software engineering process. Performance engineering is described as an experimental approach to predicting the likely performance of systems. It can involve building and monitoring the system under the workloads of interest, or using models to represent the system. Modelling has many advantages over building/monitoring, which makes it used in most of the work of interest to software engineering.

The challenge is to bring the worlds of software engineers and performance analysts together, which can be achieved by embedding performance analysis techniques into design methods and tools. As UML is rapidly being adopted as a design standard, the focus of the work should be on expanding or augmenting UML to allow performance related information to be incorporated into its design notation.

Bondavalli et.al.

This work aims to extend UML design toolkits with automatic dependability analysis tools to evaluate the various dependability attributes of the system under design [15, 16]. It is part of a European ESPRIT project called *HIDE* (High-Level Integrated Design Environment for Dependability) and one of the tools created in this project deals with an automatic transformation of UML diagrams into Timed Petri Net (TPN) models, which allow the modelling of activities whose duration is a random time.

The UML diagrams of interest here include mainly the structural diagrams, such as the *use case*, *class*, *object* and *deployment* diagrams, as well as behavioural diagrams, such as the *statechart* diagrams. Since standard UML does not include a notation for dependability aspects, it is necessary to provide minor extensions through standard extension mechanisms: *tagged values* and *stereotypes*. Tagged values are pseudo attributes assigned in the form of a pair "tag = value", while stereotypes introduce a high-level classification (meaning/usage) of model elements. The transformation of the relevant UML design diagrams into TPN models is performed in two stages:

- *from UML to the Intermediate Model*

  First, the elements and relations of the UML design are projected into the Intermediate Model (IM), which is used to capture the dependability related

information. The IM is defined as a *hypergraph*, where each *node* represents an entity described somewhere in the UML structural diagrams, and each *hyperarc* represents a relation between the elements/nodes. IM nodes have a set of attributes attached to them, which describe their fault activation and repair process as well as the propagation process for a hyperarc. In other words, the IM is built by projecting the UML entities into IM nodes, and the structural UML relations into IM hyperarcs.

- *from the Intermediate Model to TPN models*

  The hypergraph representing the IM is examined, and from there, a set of subnets for each IM element is generated. The TPN model can then be derived from all these.

More details on this tool, especially on the generation of the TPN models and the suitable Petri Net environments/tools can be found in [15].

Kähkipuro

In [45], Kähkipuro presents a framework for creating, using and maintaining performance models of object-oriented distributed systems. The architecture of this performance modelling framework consists of four main elements:

- *The method of decomposition (MOD)*

  It provides the foundation of the framework by defining an algorithm for finding an approximate solution for performance models.

- *UML based performance modelling techniques*

  These techniques provide the means for modelling complex information systems by using abstraction for separating application level issues from the use of technical resources.

- *Performance modelling methodology*

  This provides a link to the software engineering process by indicating how the proposed UML modelling techniques can be used at different stages of system development to produce useful performance models for the system.

- *Object-oriented performance modelling and analysis tool (OAT)*

  The purpose of this tool is to automate some of the tasks required by the framework, such as:

  – the transformation of UML based performance models into a format that is solvable by the MOD algorithm,

  – the implementation of the MOD algorithm to produce an approximate solution for the performance model,

  – the conversion of the solution into a set of relevant performance metrics that will be used in the performance modelling methodology.

There are four performance model representations used, each with its own notation and the mappings between them are defined by the framework architecture:

1. *UML representation*: describes the system with UML diagrams.
2. *PML representation*: provides an accurate textual notation for representing performance related items in the UML diagrams. The purpose of this representation is for filtering out the UML information that has no significance for performance modelling.
3. *AQN representation*: describes the system in an augmented queueing networks format that may contain simultaneous resource possessions and allows the MOD algorithm to solve the model.

4. *QN representation*: consists of separable queuing networks with mutual dependencies that correlate them to the same overall system.

The UML notations employed in this work are the *class* diagrams (for deriving the resources or queues), as well as the *collaboration* and *sequence* diagrams, which are used in conjunction with one or more workload specifications to describe the behaviour of the application and the infrastructure of the system. It is mentioned that the state and activity diagrams could also be used for expressing the performance related information conveniently.

Cortellessa and Mirandola

The work by Cortellessa and Mirandola [26] also attempts to provide an automatic translation of UML diagrams into a queueing network based performance model. The goal is to complement the UML notation with a methodology that encompasses the performance validation task (which includes model generation and validation) as an integrated activity within the development process.

The target performance model is based on the Software Performance Engineering (SPE) methodology, and composed of two parts: the *Software Model* (SM) based on *Execution Graphs* (EG); and the *Machinery Model* (MM), based on *Extended Queueing Network Models* (EQNM). By combining SM and MM, a complete (parameterised) EQNM based performance model is obtained, which is then solved using some well-known techniques.

The methodology outlined in this paper uses three main UML diagrams for extracting performance related aspects and integrating them into performance model:

- *Use Case Diagram*: for deriving the user profile and the software scenarios,
- *Sequence Diagram*: for deriving an Execution Graph,

- *Deployment Diagram*: for deriving an EQNM and to identify the hardware/software relationships that improve the accuracy of the performance model.

This methodology is yet to be implemented as a tool, and to achieve this, the authors believe that there are two steps to take:

- choosing an appropriate syntax to represent the UML diagrams involved (and perhaps some supporting data structures),
- studying the underlying syntax of the existing SPE tools in order to ease the translation from UML notation to a performance model representation.

Hoeben

This work developed a prototype tool that is able to translate a UML model of a system into a queuing network representation, which is then used to calculate the system's response time and utilisation [33].

The UML *use case* diagrams are used for capturing the tasks that the system has to complete (i.e. its workload), and the tool will calculate response times for each of these tasks. The *interaction* diagrams (i.e. the *sequence* and *collaboration* diagrams) provide the translation of user tasks to hardware resources. In order to avoid the diagrams becoming too big, multiple interaction diagrams are used to get the entire decomposition, where each diagram represents the behaviour of a single method. The *class* and *component* diagrams are not of great importance for performance estimation, but they are used to model information that is later used to understand the dynamics of the system. The *deployment* diagram convey the properties of the processors and network connections, which are useful in performance estimation. More information needed for performance estimation can be supplied using the standard extension

mechanisms, i.e. *tagged values* and *stereotypes*, but this only shows that there needs to be a more robust way for incorporating performance related information into UML design.

de Miguel et. al.

This paper introduces UML extensions for the representation of temporal requirements and resource usage and their automatic evaluation [27]. The interest of this work is on the specification of architecture and requirements of real-time systems, which pays special attention to timeliness, performance and schedulability.

Standard UML extension techniques (constraints, stereotypes and tagged values) are used for specifying a UML *profile* where they collectively specialise and tailor UML for specific domain process. *Constraints* represent specific semantics of modelling elements with linguistic notations, *stereotypes* define new metamodel constructors, and *tagged values* identify new parameters or information associated with the modelling elements.

UML diagrams are used as input for the automatic generation of scheduling and simulation models. The diagrams employed include the *class*, *collaboration* and *activity* diagrams. There are two tools used: *Analysis Model Generator (AMG)*, which implicitly defines a middleware model that affects the scheduling analysis; and *Simulation Model Generator (SMG)*, which allows an automatic generation of OPNET model.

1.2.3. Relevance to this thesis

We believe that software performance can be predicted using simulation approach. This belief is also shared by some of the related work above, and more discussion on simulation concepts and methodology is given in Chapter 3.

The work by Bondavalli et. al. [15, 16] shows that the transformation from UML design notation into a performance model (in their case, it is a timed petri net) can be done in two stages. The first stage gathers information that is relevant for performance estimation from the UML notation, which is then stored into an *Intermediate Model*. The second stage converts the Intermediate Model into the desired performance model, which can then be evaluated and analysed to give the performance prediction. We consider our SimML framework (see Chapter 4) to serve the similar purposes as those of the Intermediate Model. The implementation of the SimML tools also follows the two staged approach, as illustrated in Chapter 5.

The SimML framework is also comparable to the PML (Performance Modelling Language) mentioned in [45]. The difference is that the goal of the SimML framework is for building simulation programs, while the PML framework aims to produce a performance model using queueing networks.

The importance of an ability to estimate the performance of a system (software) during its design stage is highlighted by the existence of workshops dedicated specifically for this topic, such as the one day workshop on *Software Performance Prediction extracted from Designs* held at the Heriot-Watt University in Edinburgh [47] and more recently, the *International Workshop on Software and Performance* (WOSP 2000) held in Ottawa, Canada [76]. These indicate that research on the approaches for incorporating software performance into software engineering/design is still an ongoing process and deemed useful so that it may provide many benefits for the software engineering community.

1.3. Chapters Plan

The rest of this thesis is arranged as follows:

- Chapter 2 gives a brief introduction on software design, with the emphasis on the Unified Modeling Language (UML), which has become the standard notation for specifying software system.
- Chapter 3 presents the basics of stochastic simulation, which can be used in predicting the performance of a system before its is being constructed. This chapter also introduces two simulation languages/packages called C++SIM and JavaSim, which provide the simulation environments for this project.
- Chapter 4 outlines a framework called the Simulation Modelling Language (SimML) that has been constructed in this Ph.D. study and represents the core of the work. The SimML framework serves as a bridge that enables an automatic generation of a simulation program directly from a UML design specification.
- Chapter 5 discusses the implementation of two tools that can perform the automatic construction of simulation programs from UML notations using the SimML framework. The first tool is a PERL parser that is able to transform a textual representation of UML using the SimML framework into C++SIM programs; this tool serves as a pilot project to test whether the ideas presented in the SimML framework are workable or not. The success of the first tool led to the implementation of the second tool in Java that allows graphical UML notations to be transformed into JavaSim programs.
- Chapter 6 shows some case studies that have been performed to evaluate the usefulness of the tools described in Chapter 5, especially the second one.
- Chapter 7 provides some concluding remarks of the overall project and outlines some areas into which the project could be extended or researched further.

Chapter 2

Designing a Software System

2.1. Introduction

The advancement of computer technology demands new systems to be built. New requirements are discovered and new, more efficient methods are available. The problem is that the requirements are usually presented in a plain language and hence they may contain many ambiguities. This may lead to a mismatch between the completed system and the proposed system as required by the customer.

It is therefore important to ensure that the system developer understands clearly what the system requirements are. To do this, the requirements must be transformed into a notation that can only be interpreted in one way and accepted universally. A formal method, such as VDM [43] or Z [50] is one possible candidate, but it is often too complicated and difficult to employ this method because it requires a good mathematical skill and a thorough analysis.

An alternative way is to use one of the design methods available in the area of Object Oriented Technology, such as the Coad/Yourdon [25], Booch [17], Object Modeling Technique (OMT) and OOSE methods (all briefly mentioned in [29]). The availability of several design methods brings the consequence that there is no standard design notation, i.e. a specification written/drawn in one method might not be understood by some system developers who normally use a different method.

In around 1995, a new design method was developed by the people behind the Booch, OMT and OOSE methods with an aim to standardise the design notation. This method is called the Unified Modeling Language (UML) [18, 28, 29, 31, 63, 64] and it has been accepted as the standard modeling language by the Object Management Group (OMG). The rest of this chapter introduces UML by discussing the design notations available, why UML becomes popular and identifies things that are still missing from UML.

2.2. Unified Modeling Language

Unified Modeling Language (*UML*) is a language for specifying, visualising, constructing and documenting the artifacts of software [63]. It uses graphical notations to illustrate a system specification, and since the specification is usually very complex, there are several diagrams available to provide different views of the proposed system:

1. Class diagram

A class diagram represents the static structure of a system which includes the static elements (*objects* or *classes*) of the system and the static relationships between them. A class represents a set of objects with similar structures (*attributes*) and behaviour (*operations*). Two or more classes can have a relationship between them and the relationship can be:

a) an *association*.

An association indicates the role a class plays in the relationship. On top of that, there are some additional notations available for the association, such as the *multiplicities* (which indicates how many instances a class can have in the association), *aggregation* (to show that one class is a collection of several

instances of the other class), *composition* (one class is a part of the other class) and *dependency* (to indicate that one class depends on the other).

b) a *generalisation*.

This captures the notion of *inheritance*; it shows the relationship between a more general element (the *supertype*) and a more specific element (the *subtype*). The subtype inherits the properties of its supertype and it may have some additional (more specific) information.

The notations for the class diagram can be seen in Figure 2-1.

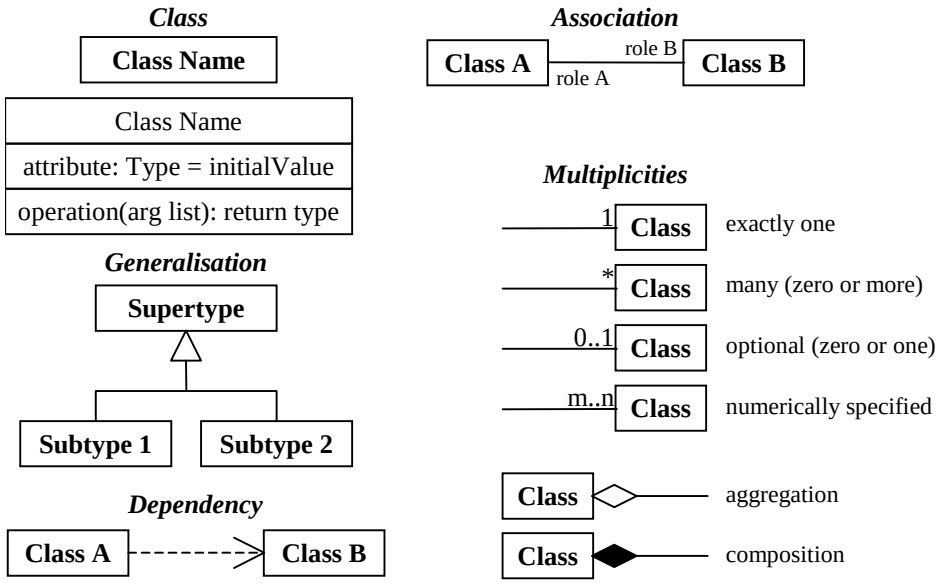


Figure 2-1: The Class Diagram notations

2. Use Case diagram

It is often important to investigate the relationships between a system and its users. A use case diagram describes the functional requirements of a system and the interaction between the *actors*, the system modelled and the *use-case*. The actors could be human users or other computer systems, and they are the ones who carry out the use cases. A use case is a set of sequences of actions performed by the system that yield an observable result of value to a particular actor [29]. The use

cases can have some relationships among them. The two most common ones are the *extends* and *uses* relationships: the *extends* relationship illustrates a use case that is similar to another use case, but it does a bit more, while the *uses* relationship indicates that one use case is using the features of the other use case (i.e. to avoid repeating the description of a same behaviour). The UML notation for the Use Case diagram is shown in Figure 2-2.

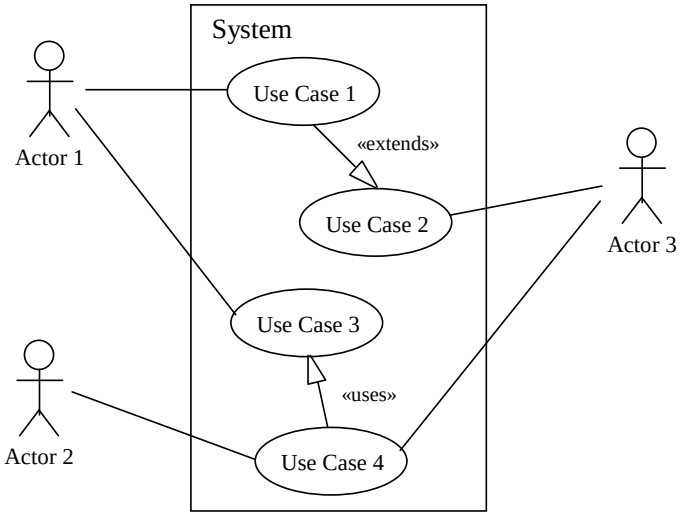


Figure 2-2: The Use Case Diagram notation

3. Interaction diagram

Interaction diagrams show the pattern of interactions between objects in a system. An *interaction* consists of *messages* that are exchanged among *objects* in order to achieve the desired result of an operation. There are two types of interaction diagrams:

a) *Sequence diagrams*: show the interactions in a time sequence.

In the sequence diagram, the objects are arranged horizontally (on top of the diagram) while the time line is shown vertically (normally proceeds down the page), one for each object. A message is represented by an arrow between the time lines of the objects, although in some occasion, an arrow can point to its own time line to indicate self invocation. An asynchronous message is

represented by a half-arrowhead. Conditions can also be included to indicate that the message is only sent if the conditions are satisfied. Figure 2-3 shows how the sequence diagram can be represented.

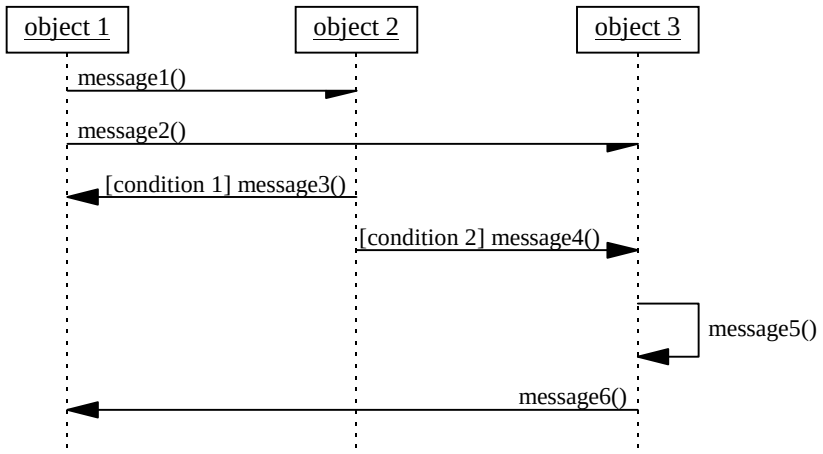


Figure 2-3: The Sequence Diagram notation

b) *Collaboration diagrams*: show the interaction in term of links between the objects.

As in the sequence diagram, objects are represented as icons (boxes with the objects' name underlined) and they interact with each other by exchanging messages. But there is no time line on the collaboration diagram, instead, the messages are given numbers to indicate their ordering. This makes it more difficult to see the sequence of messages, but on the other hand, the collaboration diagram makes it easier to see how the objects are linked together (Figure 2-4).

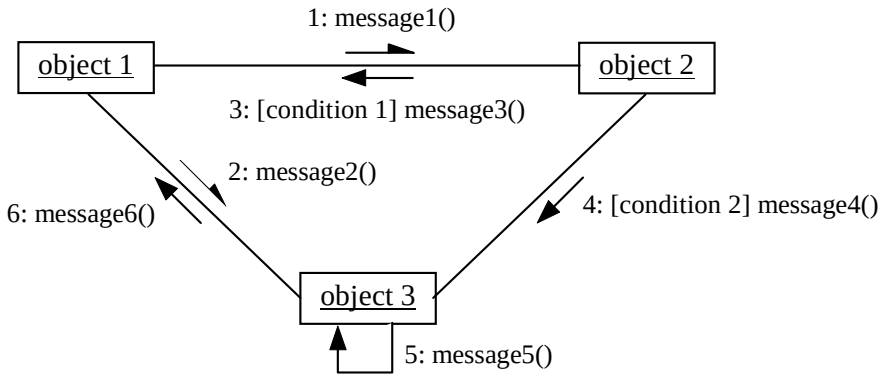


Figure 2-4: The Collaboration Diagram notation

4. State diagram

Every object has a *state* which can change if something (an event) happens to it. The state diagram describes the states that an object can get into and the interactions (*actions* and *activities*) that are involved to change the state. An action is associated with a *transition* (i.e. the changing from one state to another) while an activity is associated with the states. Figure 2-5 shows how a state diagram is represented in UML.

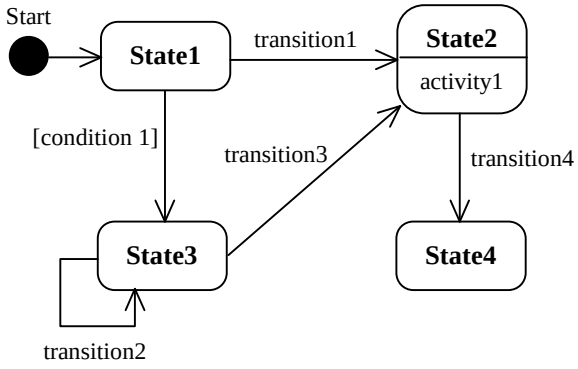


Figure 2-5: The State Diagram notation

5. Activity diagram

This diagram represent the activities that are triggered at the completion of an operation. An activity diagram is a variant of a state diagram but it emphasises on the actions, i.e. the activities that are performed to change the object states and the results of those activities (see Figure 2-6).

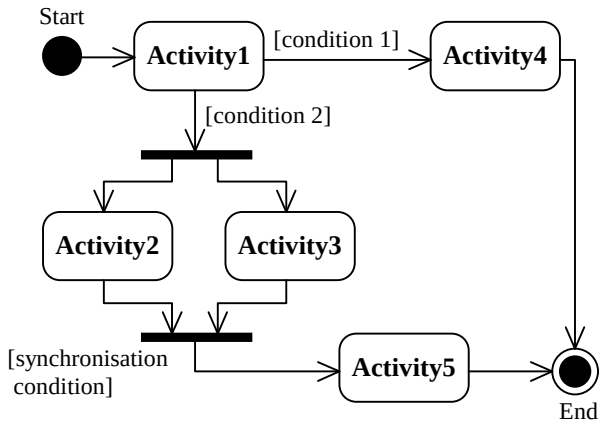


Figure 2-6: The Activity Diagram notation

6. Package diagram

As systems get more complex, it becomes more difficult to understand their structure, unless they are broken down into smaller manageable pieces or sub-systems. A package diagram shows how system's classes are grouped together, along with the dependencies among them (Figure 2-7). This kind of diagram is useful when there is a large number of classes that are closely linked or depend on each other, so that they might be bundled together into higher level units.

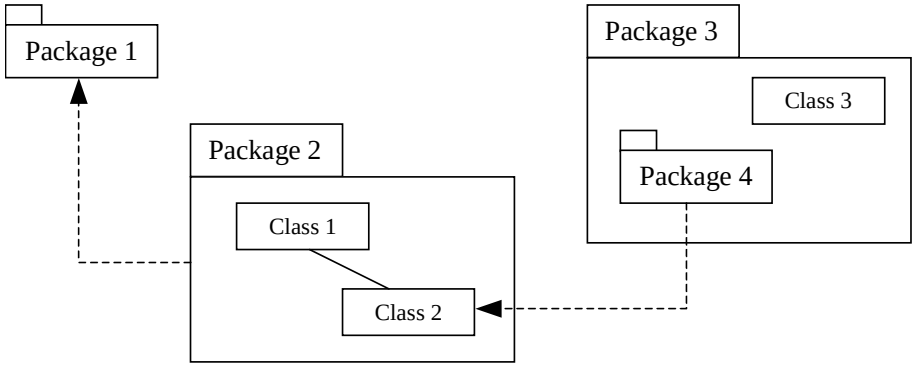


Figure 2-7: The Package Diagram notation

There are also some additional diagrams that can be used to provide more detailed explanation by adding notes or constraints (see Figure 2-8).

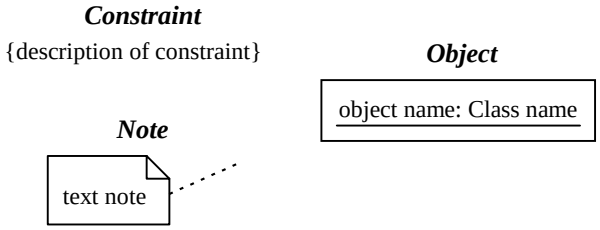


Figure 2-8: Other useful UML notations

UML has gained a wide acceptance since its introduction in 1995 and it has become a popular choice when it comes to selecting a design method. There are some reasons for this:

- The notations employed by the UML are reasonably simple yet they are powerful enough for complex specifications.

- It is an industry standard, so its notation will be understood by many people.

There are several tools available that make the job of drawing the UML diagrams easier, such as the *Rational Rose* tool [63], Together [59] and Argo/UML [71]. Each of these tools has its own benefits and drawbacks, but there is one thing that these tools - or UML in general - are still lacking, which is mentioned in the next section.

2.3. UML and System Performance

UML helps the system developer to understand the specification better through multiple design diagrams. But UML cannot answer a question on how a particular design will perform. This is because UML does not incorporate performance related issues into its diagrams/notations. The problem is, it is often necessary to know beforehand whether a design will deliver its performance requirement or not, or whether one scenario will give a better performance than another.

There has been some work done in an attempt to predict a system's performance before the system is built. The PERMABASE project [1, 3, 72] puts the emphasis on the UML Implementation diagrams, namely the Deployment and Component diagrams to derive a system's performance model. The performance model can then be simulated using either using a *discrete event simulation* (DES) engine of SES/Workbench, or Coloured Petri-nets.

The work by Pooley and King [62] addresses the need to integrate performance engineering into software engineering. The general approach adopted in this work is to use UML designs as directly as possible in performance evaluation. The performance evaluation itself is obtainable using simulation methods or queuing network modelling approaches.

In [15], Bondavalli *et. al.* describes a transformation from structural UML specification (mainly the Use Case and Deployment diagrams) into *Timed Petri Net* (TPN) models for the quantitative evaluation of dependability attributes.

These three are just some of the work attempting to incorporate software performance prediction into UML, more work that has been done along the similar line is outlined in Chapter 1.

The amount of related work done in the performance prediction area suggests that it is a useful thing to do with regard to UML. Our approach in tackling this problem is by using a simulation program to mimic the execution of the real system, hence enabling the system's performance to be analysed and calculated beforehand. This involves the generation of a simulation program from UML design specification, and this approach is discussed in the following three chapters. First though, it is appropriate to introduce the concept of simulation, which is covered in the next chapter.

Chapter 3

Simulation to Predict System's Performance

3.1. Introduction

Predicting the performance of a system that is yet to be built is not a trivial task. First, it is necessary to determine which aspects of the performance are to be measured, such as reliability, availability, response time, etc. This is important because different system applications demand different performance requirements, for example, a telephone system needs to be highly available while a flight control system has more emphasis on its reliability. From there, a *workload* that the system will face can be determined. A workload typically includes the operations that are needed to be performed by the system, the time needed for each operation to be completed and the frequency of each operation. This is not an exhaustible list as more aspects are often considered depending on the system's requirement.

A system's workload specification needs to be analysed in order to obtain an estimate for the system's performance. There are several ways to predict the performance of a system based on its workload specification. One way is through an analytical method where the system's workload is derived mathematically by constructing a number of parameterised mathematical equations that approximate the workload characteristics of the system's components. This approach is often called *modelling* and it is explained in great detail in [49, 56, 57]. An example of how modelling is used for examining workload specification analytically is the MaStA (**Ma**ssachusetts **St A**ndrews) cost model [65-67]. This was developed for modelling

the recovery of database systems. The MaStA model is used for comparing the performance (the cost) of several recovery mechanisms which are executed on different platforms to ensure that the performance is platform specific. The workload here is determined by the number of I/O operations involved in a particular recovery mechanism. The I/O operations can be divided into constituent independent I/O categories, such as data reads or commit writes, and the overall cost of a mechanism is equal to the sum of the costs of each category.

The analytical method requires a good mathematical knowledge and calculation, so it is valuable in a simple system but usually impractical or very difficult to use in a complex system. Another way of predicting system's performance is through *simulation*. This approach requires the construction of programs that capture the characteristics of the proposed system. Among other things, these characteristics include the workload of the system, which determines the simulation's results. The workload specifications can then be considered as the simulation's parameters, i.e. they must be known before the simulation can be performed. As with the analytical method, the simulation method requires some simplifying assumptions to make it easier to employ.

One feature of the simulation method is that it allows probabilistic nature of events or activities to be observed. This is a very important feature because a computer system, especially a distributed one, involves a lot of events that can happen at random, such as the system failures, the latency in the connections, job arrivals, the response times, etc. By knowing its probability or distribution function, an event can be modelled close to its real world counterpart.

Running the simulation program with the parameters from the workload specification provides some data about the states of the system. These data reflect how the system cope against a particular load, and by performing further calculation and analysis on the data, we can predict the performance of that system.

The rest of this chapter introduces the basic concepts of simulation, its benefits and drawbacks - which motivate the work presented in this thesis, as well as the different simulation types encountered. This is then followed by an overview of the simulation environments (languages or packages) available, in particular those which were used in this thesis.

3.2. Simulation Basics

3.2.1. What is simulation?

Simulation is "the imitation of the operation of a real-world process or system over time" [13]. The aim of simulation is to describe and analyse the behaviour of a system and explore the possible scenarios that can happen in the system (the "what-if" questions). Simulation allows different system designs to be investigated, and by analysing the simulation results, the performances of these systems can be compared. Therefore, simulation can be used as a decision making aid in choosing which system is to be implemented.

There are several underlying concepts in simulation, and in order to provide a better simulation understanding, these concepts or terminology are explained below:

- *model*: is a representation of an actual system.
- *system*: is a collection of entities which interact with each other, within some notional boundary, to produce a particular pattern of behaviour [14].

- *entity*: represents an object/component (with its interactions) that are considered important enough to be included in the model.
- *attribute*: a variable declared inside an entity/object, which describes the state of that entity.
- *system state variables*: the collection of all attributes of all entities. It is used to know what is happening within a system at a given point of time.
- *sample or operation path*: a realisation of system state variables. One simulation run generates one sample path and another simulation run of same system with different random numbers generates a different sample path of the same system.
- *event*: a change in the system state during the course of the sample path.
- *event time*: time at which an event occurs.

Simulating a system essentially means generating a sequence of event times with their appropriate events. How these event times are organised leads to several simulation types (see Section 3.2.3), but in general, the structure of a simulation run can be divided into three separate stages [14]:

1. **Initialisation.**

   This stage sets up the simulation model by obtaining the simulation parameters from the user. The system state variables (the variables for obtaining the simulation statistics) are declared and initialised in this stage.

2. **Dynamic stage.**

   The dynamic stage sets the model in motion, where the model is allowed to perform its dynamical behaviour until a fixed period of simulation time has elapsed or until a particular condition has been met (which is set in advance). Here, the system state variables are updated in a pre-determined way.

3. **Termination.**

   After the simulation run is completed (i.e. after the second stage above is finished), it is necessary to collect the system state variables, which are then analysed in order to work out the simulated system's performance.

These three stages serve as the skeleton for constructing simulation programs, whose details differ one from another according to the type of simulation they are meant to perform.

3.2.2. Advantages and Disadvantages of Simulation

Modelling a system through simulation means replacing that system with one that is simpler and/or easier to study, yet equivalent in all important aspects. This is not an easy task to do, so it is a good idea to compare the advantages and disadvantages of simulation before we proceed any further.

Advantages

There are many advantages of using simulation, some of which are listed here:

1. It allows us to perform a controlled experiment upon the system by varying the simulation parameters. We can therefore investigate how the system will react under certain circumstances without committing the resources for implementing that system (which in the end might not be suitable).
2. Time can be compressed or expanded within the simulation, which means that we can thoroughly examine the states of the system in a suitable time frame.
3. There is a separation from the real system, i.e. the simulation will not disturb the operation of a real system (if there is one already running).
4. It can serve as an effective training tool.

5. Simulation can be used to diagnose problems that might occur in the system (e.g. a bottle-neck in a particular stage of the operation), hence a remedial action can be taken.
6. It helps other people to understand how the system works.

Disadvantages

Inevitably, there are some disadvantages that must be faced:

1. Building a simulation program is quite a difficult task and it often requires some specialised skills.
2. Interpreting the simulation results is not easy either.
3. It could be a time consuming and expensive: a new simulation program needs to be built for a different system.
4. Model parameters may be difficult to initialise. It is often necessary to obtain them from some experimentations or time-extensive analysis.

There are efforts in eliminating these disadvantages, for example, there exist some specialised software for simulation (*simulators*). Most of these tend to be commercial software that come with output analysis and/or animation tools, so they are generally expensive to purchase.

The work discussed in this thesis tries to solve the first and third disadvantages above by providing a tool that can be used for transforming a design notation automatically into simulation programs.

3.2.3. Types of Simulation

Based on the ways the system states change during the simulation run, it is possible to categorise simulation model into three types:

1. *Discrete time*: the system is only considered at selected moments of time, which are usually evenly spaced. Only at these moments (called *observation points*) are the changes of the state recorded.
2. *Continuous time-continuous event*: the system states vary continuously with time; this kind of system is usually described by sets of differential equations.
3. *Continuous time-discrete event* (or more popularly termed just as *discrete event* simulation): the time is continuous but the state variables change only at those discrete observation points in time at which events occur. These observation points do not need to be equally spaced and can be of arbitrary increment. This kind of simulation is the one that is most commonly encountered.

The classification above takes into account the relation between the simulation events and the simulation time, which can be seen diagrammatically in Figure 3-1.

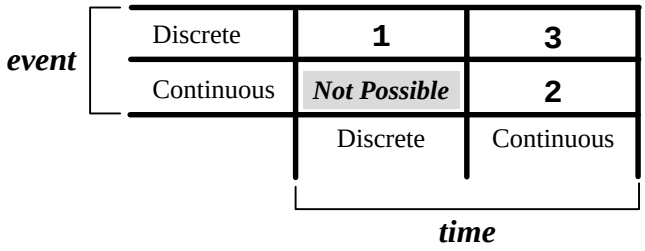


Figure 3-1: The classification of simulation type based on events and time

A simulation consists of a series of interacting events that constitute the sample or operation path. Therefore, a *simulator* can be defined as a program that is devoted to the generation of operation paths [58]. It permits the creation of events, whose interactions are controlled in a timely manner using the internal clock. The simulator keeps an *event list* - the list of events that are scheduled to be executed at certain times during the simulation. Based on the approach on how the simulator schedules the events in producing the operation path (i.e. how the event list is organised), there are two types of simulation:

1. *event-oriented* simulation.

   Each event has an *event notice* which contains both the event's *type* and *time*. Each event type is in turn associated with a procedure which performs the actions required to handle that particular type of event, which include the scheduling of the next event of this type as well as the collection of the statistics. This procedure is invoked every time an event of the relevant type occurs.

   There is also a *clock procedure* that controls the simulation by scanning the event list, finding the event notice with the smallest time (which is executed next), updating the simulation time and invoking the relevant event procedure. The event manipulations are therefore explicit. The operation path is obtained by taking a global view of everything that happens in the system.

2. *process-oriented* simulation

   A process is defined as a sequence of events where each event is accompanied by a set of actions. Each process is given an independent thread of control, so the management of events is implicit in the management of the processes. Hidden from us, there is still a clock that is advanced from event to event, as well as an event list which shows what is scheduled to happen at a particular time. The difference is that the event list now contains processes, ordered according to the time of the next events in their respective sequences. Each process keeps track on which action of its set is to be executed next.

Each time after the clock is advanced, the actions of the processes at the head of the event list are executed. If there are more than one process scheduled at the same time, the actions of all those processes are executed, in the order in which they appear in the list. The operation path of a process-oriented simulation is therefore obtained by

the interacting of a number of processes running in (pseudo) parallel. More explanation on this kind of simulation is given in the discussion of the C++SIM package (Section 3.3.1). The rest of this thesis refers to the *process-based discrete-event* paradigm when discussing simulation.

3.3. Simulation Languages/Packages

A simulation program can essentially be built in any general purpose programming languages but this usually requires more effort. A simulation language/package aims to make the process of writing a simulation code easier by providing some of the standard simulator components as its language primitives or types. Overall, there are five facilities that a simulation language needs to provide [58]:

1. Entity manipulation.

   This allows the creation and destruction of entities, as well as placing/removing them to/from their respective sets (stacks, queues, etc.).

2. Time and Event manipulation.

   The task involves the maintenance of a clock and an event list, plus all the scheduling operations, such as finding the next event, removing current event and inserting a new event.

3. Random Numbers.

   We talk about pseudo random numbers here, i.e. sequences of numbers that appear to be independent and randomly distributed. The simulation language needs to provide various discrete and continuous distributions and able to reproduce and change the sequences for different runs of the same simulation program.

4. Collection of Statistics.

This deals with the collection of data that can be used to work out the quantities of interest in the simulation.

5. Numerical computation.

   It is often necessary to be able to implement various numerical algorithm which is useful for analysing the simulation results and generating specific random numbers.

The task of writing simulation programs can be made easier by using certain languages or packages that support the essential simulation features outlined above. There are several simulation environments that can be employed, but many are commercial ones, which tend to be very complex and often come with some animation tools. What is required for this project is just a simulation environment that supports the process based, discrete event simulation paradigm, such as SIMULA [61], Modsim [21], SimJava [34, 35], C++SIM [11, 52] or JavaSim [51]. The work illustrated in this thesis uses C++SIM and JavaSim, so the basic concepts of these two packages are given in the following sub-sections.

3.3.1. C++SIM

C++SIM provides a discrete-event, process-based simulation facilities similar to SIMULA's [61] simulation class and libraries. It is written in standard C++ and since C++ compilers typically generate code which runs faster than similar SIMULA code, C++SIM would produce more efficient simulation codes.

The C++SIM environment uses active objects as the units of simulation. An *active object* is an object which has an independent thread of control associated with it, and it is used to convey the notion of 'activity' to the processes involved in the simulation. Active objects are created using threads (lightweight processes) and in C++SIM, they are used for:

1. Simulation Scheduler

Simulation processes (see later) are managed by a scheduler and are placed on a scheduler queue (the event list). Figure 3-2 shows how a tree structure is used to organise the scheduler queue. Each node represents a process and the nodes at the same level of the tree have the same simulation time. Here, the processes are executed in a pseudo-parallel mode, i.e. only one process is activated at any instance of real time, but the simulation clock is only advanced when all processes have been executed for the current instance of simulation time.

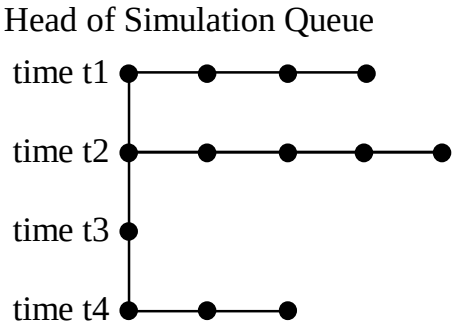


Figure 3-2: Simulation Queue

Inactive process are placed into the scheduler queue and when the currently active process yields control to the scheduler (either because it has finished or been placed back onto the scheduler queue), the scheduler removes the process at the head of the queue and activates it (Figure 3-3). When there is no process left in the scheduler queue, the simulation will terminate. Please note that every simulation must start one scheduler before the simulation can begin.

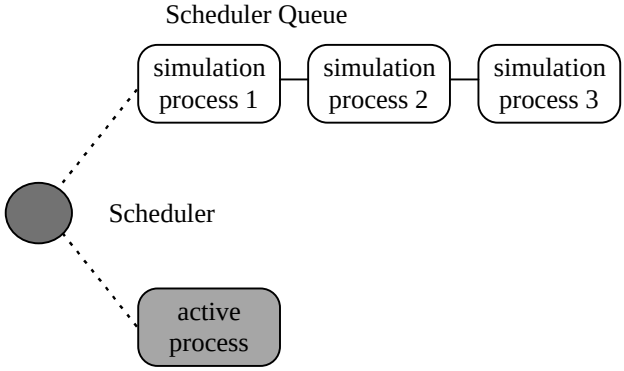


Figure 3-3: Scheduler-Process Interaction

2. Simulation Processes

C++SIM supports the process-oriented approach to simulation, i.e. each simulation entity can be considered as a separate process. These entities are represented by process objects: they are C++ objects which have an independent thread of control associated with them when they are created. Each process has a state and at any point during the simulation, a process can only be in one of the following states:

a) *active*: the process has been removed from the head of the scheduler queue and its actions are currently being executed.

b) *suspended*: it is on the scheduler queue and is scheduled to be active at a specified simulation time.

c) *passive*: it has been removed from the scheduler queue and if it is not brought back to the queue by another process, it will not execute anymore.

d) *terminated*: it is not on the scheduler queue and will not take any further part in the simulation.

C++SIM uses the object-oriented approach for developing the process objects by allowing classes to inherit the process functionality from a base class called `Process`. This class provides all required operations for the simulation system to control all of the processes in the simulation. The most important operations are:

(i) `Activate`: activates a process. This is invoked by the currently active process which passes the control to the activated process.

(ii) `Passivate`: removes the currently active process from the scheduler queue. Another process has to put this process back into the queue if it needs to be scheduled again in the future.

(iii) `idle`: returns true or false to indicate whether a process is actually on the scheduler queue or not.

(iv) `Hold`: reschedules the currently active process to be active a fixed units of time later.

(v) `Cancel`: removes a process from the simulation queue or suspends it indefinitely if it is currently active.

(vi) `CurrentTime`: returns the current simulation time which is useful for controlling action relative to a given time period.

Other operations and further explanation on the ones above are available in [11].

Any class derived from the `Process` class must supply a `void Body()` member function, within which its actions must be defined. These actions characterise the interactions among the processes in the simulation and these actions will be executed when the process to which they belong to is activated.

3. *Main System Thread*

This is a special thread which is used to initialise the threads used in the simulation. It is invoked in the main body of the simulation code and since this thread has the highest priority in the system, it is necessary to suspend it in order to allow other threads to run.

A more detailed description and some examples of C++SIM programs can be found in [11] and [52].

3.3.2. JavaSim

JavaSim is a Java implementation of the original C++SIM simulation toolkit, which supports the discrete-event process-based simulation where each simulation entity can be considered as a separate process [51]. 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.

Like in the C++SIM environment, there is a JavaSim *scheduler* that manages the simulation processes (the active objects) and places these processes onto a *scheduler queue* (the event list). The processes are also executed in a pseudo-parallel mode.

The difference between the C++SIM and JavaSim environments lies on how the processes are derived from its base class. JavaSim calls its base class `SimulationProcess` while the C++SIM calls its `Process` class. There is also a difference on where the processes' behaviour is specified. In C++SIM, the user needs to implement a `void Body()` member function on any classes derived from the `Process` class. The JavaSim code on the other hand requires its process class to implement a `void run()` member function. The rest of the syntax remains pretty much the same for both environments. More details on the JavaSim processes can be found in [51].

The advantage of JavaSim over C++SIM is that JavaSim is easier to install and manage on any operating system (due to Java's portability), and with the *jar* (Java ARchive) facility, it is even possible to put the whole JavaSim package into one *jar* file. The *jar* facility also allows more than one packages to be put into one *jar* file, which means that the JavaSim package can be integrated with other packages to provide more customised features.

3.4. Simulation Summary

Simulation can be used as a means for predicting a system's performance, but building a simulation program is not a trivial task and system developers usually are not familiar with the simulation concepts. It is therefore desirable to have a tool that can build a simulation program automatically from a design notation (which is easier to prepare and commonly understood by system developers).

The next chapter identifies a simulation framework that can serve as a bridge between design notation and simulation program. It will be demonstrated later that this framework can be used to build a tool that can automatically transform a design notation into a simulation program.

Chapter 4

From Design to Simulation

4.1. Introduction

It is often difficult to build a simulation program directly from a design specification. The specification is normally quite complex and in the case of the distributed system, different performance requirements (such as high availability over reliability, etc.), connection between components (network delay, partition, etc.) and fault tolerance (machine failures and repairs) require additional effort. On top of that, a new simulation program needs to be built each time for a different specification, and the system developer must also be familiar with the simulation techniques - which is not always the case.

A tool that can automatically transform a design notation into a simulation program is therefore desirable. Before such a tool could be built though, it is important to first identify the common simulation components along with the actions that can be performed among them. Based on these components and actions, a simulation framework can be specified, and this framework provides a foundation for an automatic generation of a simulation program from a design notation.

4.2. SimML Framework

We have constructed a framework called *Simulation Modelling Language* (SimML) [6, 7, 9, 10, 69] that classifies the simulation components into two main groups: *basic type* components and *auxiliary* components. The first group represents entities of a

simulation and they correspond to the classes of a simulation program; these entities are identified as PROCESS, DATA, QUEUE and CONTROLLER components. The second group is useful for representing the instances of active objects (processes) involved in the simulation - i.e. the instances of the simulation classes - as well as for specifying the simulation parameters and collecting the simulation statistics. This group includes the OBJECT, RANDOMS and STATISTICS components.

The PROCESS components are able to interact with each other, and in SimML framework, these interactions are termed as *actions*. The behaviour of a PROCESS component is therefore determined by its actions, where an action might involve an instance of another PROCESS component. More details on the SimML components and actions are described in the following section.

4.2.1. SimML Components

There are seven simulation components identified in the SimML framework. These are divided into two main groups:

I. Basic type components

These components are essential in constructing a simulation program as they represent the classes or entities involved in the simulation. These components can be subdivided into four categories:

A. PROCESS component

A PROCESS component is used to represent the simulation process and different processes can be characterised by assigning different name, attributes and operations to them. The PROCESS's name is used as the name of the class constructed for the appropriate simulation program to represent this process. This class may have member

variables (with *public* or *private* visibility) as its attributes as well as some member functions for defining its operations. Since the PROCESS type inherits from the `Process` class (see Chapter 3, Section 3.3), it must specify the actions of the `Body()` member function (for C++Sim simulation) or the `run()` member function (for JavaSim simulation) derived from the `Process` class in order to provide interactions with other processes. A PROCESS class may also have some constructors, through which the simulation parameters specific for this process can be passed. The structure used for the PROCESS type is very similar to *the Class Diagram* used in the UML

B. DATA component

DATA is a simplified version of the PROCESS component, where it acts just as a data storage. It is useful for representing certain simulation entities which do not need to be active objects. This type does not inherit from the `Process` class and hence it takes up a lot less resources. DATA type has a name and attributes but it does not have any operation.

C. QUEUE component

A queueing mechanism is a very important concept in simulation, hence a way of specifying queues (for different types of object) must be provided. The queue ordering supported is *First In First Out* (FIFO), but a non pre-emptive priority ordering is possible using a SEQUENCE component where each queue object is given a sequence number - the smaller the number the higher priority the object gets. The objects

placed in a queue can either be a PROCESS or DATA component, and the queue is served by one (or more) PROCESS component(s).

D. CONTROLLER component

It acts as the main thread which initialises the simulation, obtains the simulation parameters and summarises the simulation. This is a standard component that must exist in any simulation, hence its properties can be defined in a template.

II. Auxiliary components

A. OBJECT component

It is an instance of a basic type component and during a simulation, there will be several, if not many, of such OBJECTs being created. Through these instances, the interactions among the simulation components can be achieved.

B. RANDOMS component

Many aspects of the real system that a simulation program tries to model (passed as simulation parameters) have properties which correspond to various distribution functions. C++SIM/JavaSim provides several random number generators to accommodate most of those distributions. The SimML framework therefore supports these random number distributions, which are shown (with their parameters) in Table 4-1.

C. STATISTICS component

Statistics collection is also an important part of a simulation. It is necessary to know beforehand what information is to be collected and

Table 4-1: Random Number Distributions supported by SimML

in SimML	in C++SIM/JavaSim	parameter 1	parameter 2
EXPONENTIAL	ExponentialStream	mean	-
ERLANG	ErlangStream	mean	standard deviation
HYPER	HyperExponentialStream	mean	standard deviation
NORMAL	NormalStream	mean	standard deviation
UNIFORM	UniformStream	bottom range	top range

where/when/how the collection should be done. This includes the identification of the simulation statistics variables and their types, and some mechanisms for updating their values appropriately. Each statistics variable is given a name and a type; the types allowed are numerical types (integer or double). There are two ways of updating the statistics values: a *simple* increment/decrement (by a factor) or a *calculation* (which might involve other statistics variables). When/where the updates are to be performed is specified by the PROCESS component (as *update* actions).

The seven components above serve as the foundation for designing simulation programs in a generic term. The dynamic properties of the simulation is specified as *actions* of the PROCESS components, and these are outlined in the next section.

4.2.2. SimML Actions

The behaviour of a PROCESS component is determined by its actions. These actions must later be transformed into program code inside the C++SIM's `Body()` member function or JavaSim's `run()` member function. There are 14 actions deemed necessary to support the interactions among simulation processes:

1. *create* : declares a new instance of the basic type (either a PROCESS or DATA component), which is to be used by the current process to perform the interactions.
2. *wait* : reschedules the current process to be activated later after a delay.
3. *activate* : activates another process if that process is idle; otherwise this action does not do anything (i.e. it will not interrupt the execution of that process).
4. *sleep* : passivates the currently active process.
5. *enqueue* : places an instance of a PROCESS or DATA object to the tail of 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 in the queue.
8. *record* : sets the value of an object's member variable to the current time (by default) or to a specified value/variable (with extra parameters).
9. *update* : updates the value of a statistics variable. This action is used in conjunction with the STATISTICS component (which specifies the statistics variables and how they should be updated).
10. *generate* : produces a number randomly, using a particular random number generator. A valid random number generator (as declared in the RANDOMS component) must be used.
11. *end* : terminates the execution of the current process. This action must only be invoked at the end of a process's lifetime, otherwise the rest of the simulation will not work properly.

12. *print* : prints the specified simulation variables as well as any text strings (enclosed in double quotes, without any white space). This is useful for debugging purposes.
13. *if* : specifies a condition that must be satisfied before subsequent actions can be performed (i.e. it simply guards the next actions). The condition must follow the C++/Java syntax, which allows complex conditions to be achieved using the `and` operator (`&&`) and the `or` operator (`||`). The *if* action is complemented by the *elsif* and *else* action.
14. *while* : allows a loop to repeat the same action(s) until certain condition is satisfied. It is very similar to the *if* action; the only difference is that the *while* action provides a repetition feature while the *if* action provides the selection feature of the simulation.

A PROCESS component may specify one or more actions, which are listed as a sequence: one action is followed by another. Generally (and by default), the whole sequence is repeated for the duration of the simulation. In some cases though, the PROCESS component only exist for one instance, i.e. it must terminate after all actions in its sequence has been executed. In that case, it is necessary to put an “end” action as the last action of the sequence.

4.3. Using the SimML Framework to Generate Simulation Program from Design

The SimML framework mentioned above can be used to bridge the transformation of a design notation into a simulation program. Before that could happen though, it is necessary to specify a syntax that must be followed in defining the components and actions of the SimML framework. This syntax provides a better understanding on how

the components and actions are grouped and interconnected. It also serves as the intermediate language which stores the necessary information derived from a design notation before this information is transformed into a simulation program.

4.3.1. SimML Syntax

We need a syntax which allows the SimML components and actions to be expressed in a semi-formal way that makes it possible to transform them readily into a simulation program. This syntax is represented in a textual format for an easy reading and parsing of the SimML specification.

A simplified version of the syntax is outlined here; this provides a general view on how such a notation can be specified. The syntax is shown in the `courier` font, where a **bold** typeface indicates that the word is a reserved word, an ALL CAPS typeface indicates a SimML component, and an *italic* typeface indicates a variable as specified by the user. Things in square brackets are optional.

The SimML components serve as the founding blocks of this notation and based on their granularity, the SimML components can be divided into two groups:

1. *Simple Components*: convey a simple/atomic information.

   The components that belong to this group (with their notations) are:

   QUEUE of *object_type*

   OBJECT *object_name* **of** *object_type*

   CONTROLLER *controller_name*

2. *Container Components*: contain a set of information which are necessary to be incorporated into one entity.

```
DATA data_name
{
  [attributes]
}
```

```
PROCESS process_name
{
  [attributes]
  [member_functions]
  +void Body()
  {
       <action_specifications ...>
  }
}

RANDOMS
{
  <random_type> random_name [parameters ...]
}

STATISTICS
{
  <type> statistics_name [expression ...]
}
```

The SimML actions are defined only inside the `Body()` part of the PROCESS component. Some of the actions require one or more parameters, as outlined below:

```
create object_name of object_type

wait delay

activate object_name

sleep [object_name]

enqueue object_name to queue_name

dequeue object_name from queue_name

check queue_name

record variable_name [as data]

update statistics_variable

generate random_number using random_generator

end

print variable_name ...

if conditions
{
      <other actions ...>
```

```
}
elsif conditions
{
      <other actions ...>
}
else
{
      <other actions ...>
}

while conditions
{
      <other actions ...>
}
```

The syntax above is better represented in the *Extended Backus-Naur Form* (EBNF), where `'*'` denotes an item with zero or more multiplicity, `'+'` denotes an item with one or more multiplicity, and `'?'` denotes optional item (see Figure 4-4 below).

```
SIMML-SPEC ::= (DATA-COMPONENT)* (PROCESS-COMPONENT)+
               (QUEUE-COMPONENT)+ (OBJECT-COMPONENT)+
               RANDOMS-COMPONENT STATISTICS-COMPONENT
               CONTROLLER-COMPONENT
DATA-COMPONENT ::= DATA-HEADER DATA-DEFINITION
DATA-HEADER ::= “DATA ” data-name
DATA-DEFINITION ::= “{” (ATTR)* “}”
ATTR ::= ATTR-VIS ATTR-TYPE “ ” attr-name
ATTR-VIS ::= “+” | “-”
ATTR-TYPE ::= NUMERIC-TYPE | CHAR-TYPE
NUMERIC-TYPE ::= “int” | “double”
CHAR-TYPE ::= “char” | “Text”
PROCESS-COMPONENT ::= PROCESS-HEADER PROCESS-DEFINITION
PROCESS-HEADER ::= “PROCESS ” process-name
PROCESS-DEFINITION ::= “{” (ATTR)* ACTION-BLOCK “}”
ACTION-BLOCK ::= “{” (ACTION)* “}”
ACTION ::= (CREATE | WAIT | ACTIVATE | SLEEP | ENQUEUE | DEQUEUE
           | CHECK | RECORD | UPDATE | GENERATE | END | PRINT
           | IF | ELSIF | ELSE | WHILE)*
CREATE ::= “CREATE ” object-name “ of ” OBJECT-TYPE
```

```
OBJECT-TYPE ::= data-name | process-name
WAIT ::= “WAIT ” DELAY
DELAY ::= numeric-value | random-name
ACTIVATE ::= “ACTIVATE ” object-name
SLEEP ::= “SLEEP ” (object-name)?
ENQUEUE ::= “ENQUEUE ” object-name “ to ” queue-name
CHECK ::= “CHECK ” queue-name
RECORD ::= “RECORD ” VAR-NAME (“ as ” VAR-VALUE)?
VAR-NAME ::= local-variable | stat-name
VAR-VALUE ::= numeric-value | VAR-NAME
UPDATE ::= “UPDATE ” stat-name
GENERATE ::= “GENERATE ” random-var “ using ” random-name
END ::= “END”
PRINT ::= “PRINT ” VAR-NAME
IF ::= “IF (” condition “)” ACTION-BLOCK
ELSIF ::= “ELSIF (” condition “)” ACTION-BLOCK
ELSE ::= “ELSE ” ACTION-BLOCK
WHILE ::= “WHILE (” condition “)” ACTION-BLOCK
QUEUE-COMPONENT ::= “QUEUE ” queue-name “ of ” OBJECT-TYPE
OBJECT-COMPONENT ::= “OBJECT ” object-name “ of ” OBJECT-TYPE
RANDOMS-COMPONENT ::= “{” (RANDOMS-VAR)+ “}”
RANDOMS-VAR ::= RANDOMS-TYPE random-name PARAM1 (PARAM2)?
RANDOMS-TYPE ::= “EXPONENTIAL” | “ERLANG” | “HYPER” | “NORMAL” |
                 “UNIFORM”
PARAM1 ::= numeric-value
PARAM2 ::= numeric-value
STATISTICS-COMPONENT ::= “{” (STATISTICS-VAR)+ “}”
STATISTICS-VAR ::= STATISTICS-TYPE stat-name numeric-update
STATISTICS-TYPE ::= NUMERIC-TYPE
CONTROLLER-COMPONENT ::= “CONTROLLER ” controller-name
```

Figure 4-4: The EBNF notation for the SimML Syntax

Later on, this syntax is also represented as a *Document Type Definition* (DTD) which enables the SimML specification to be written in the *Extensible Markup Language*

(XML) format, which allows the SimML information to be interchanged easily. More details on XML and the SimML DTD can be seen in Section 5.4.3 of Chapter 5.

4.3.2. Capturing SimML Notation from UML Design

The information needed for the SimML framework can be derived from some UML diagrams. The *class* diagram is very similar to the PROCESS and DATA components, while the *sequence* diagram can be used to capture the OBJECT and QUEUE components. The CONTROLLER component has a standard characteristics in the SimML, so it is not necessary to represent this in UML because its properties can be specified as a template.

Two of the most important SimML components that cannot be directly derived from any UML diagrams are the RANDOMS and STATISTICS component. They could be represented as a UML note, but it is not specific enough to support the characteristics of those components. The RANDOMS components need to know which random distribution they correspond to (one of the five listed in Table 4-1) and the STATISTICS components must know how they should be updated. It was therefore decided that for these two SimML components, it is necessary to augment the UML notation by adding two specific views which can capture the required information.

It was quite difficult to decide which UML diagram to use for depicting the SimML actions. UML provides some diagrams for capturing the dynamic aspects of the system through the use case, activity and interaction diagrams. With respect to SimML, we need to know which OBJECTs are involved with each action, so a UML diagram that can convey both object reference and dynamic properties at the same time is ideal. This suggests the *interaction* diagram as the best candidate to represent

the SimML actions. Of the two types of the interaction diagrams, the *sequence* diagram was chosen instead of the *collaboration* diagram because the former shows the interaction in a time sequence, which is a very important concept in relation with the simulation programs to be generated through the SimML framework. It is also because a GUI tool that can interpret the sequence diagram is easier to built, compared to that of the collaboration diagram (due to the more complex nature of the latter).

Note: At the time the decision on using the class and sequence diagrams was made, the version of the UML notation used was version 1.1. Since then, the sequence diagram notation has evolved to allow more elaborate execution control (e.g. better looping mechanism and handling of conditional statements).

4.3.3. Transforming SimML Notation into Simulation

The task of transforming one notation into another can be done using a *parser*. A parser must be able to read the information stored in the first notation and organise the data into a particular data structure for further processing. Typically, this involves the re-creation of the data into another form/notation. Therefore, the parser also needs to understand the syntax of the second notation in order to convert the information stored in the intermediate data structure into a valid representation.

In our case, the first notation is the SimML notation and the target notation is the simulation code. There is a mapping between the SimML framework and simulation programs, hence by using a suitable parser, simulation programs can be created generically from the SimML notation. The implementation of a parser capable of performing such a transformation is discussed in the next chapter.

Chapter 5

UML-to-Simulation Tool Implementation

5.1. Introduction

The process of implementing a tool for generating a simulation program from a (UML) design notation can be divided into two stages. One stage investigates the possibilities of generating simulation source code from a textual SimML notation; this stage is referred to as the *back-end* stage. The other stage (the *front-end* stage) concerns more on how to capture the UML information into the SimML notation. This stage includes the construction of a *Graphical User Interface* (GUI) tool for drawing the relevant UML diagrams (i.e. the class and sequence diagrams) and a formulation of a means for capturing the SimML information from those UML diagrams. The whole picture on the automatic transformation of the UML design notation into a simulation program can be seen in Figure 5-1 below.

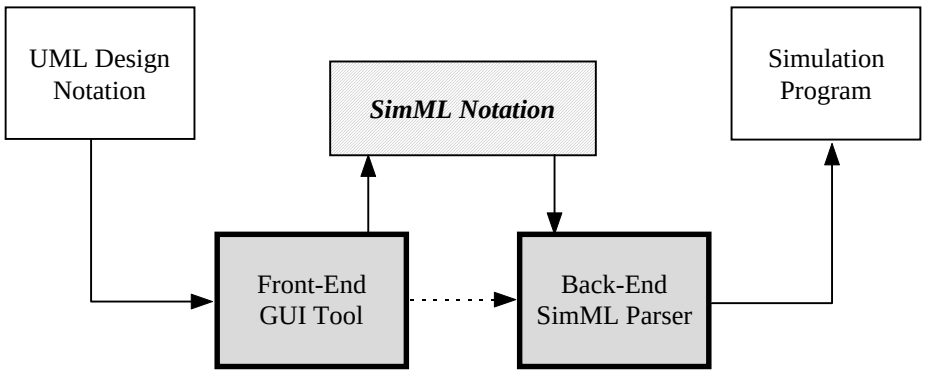


Figure 5-1: UML to Simulation path

The SimML framework (with its syntax - as illustrated in Chapter 4) makes it possible to automatically generate a simulation program from the SimML notation, therefore it can be used in implementing the back-end stage above. This back-end

stage can be achieved by writing a special parser that understands the SimML notation and is able to transform the information conveyed into an appropriate simulation program. During our research, we have developed two kinds of such a parser: one was written in *PERL (Practical Extraction and Report Language*) while the other was built in *Java*. The first parser is intended to generate simulation programs in the C++SIM environment, while the later parser produces JavaSim programs and employs *XML* (*eXtensible Markup Language*) as its intermediate language/notation. The second parser also provides the foundation for the construction of the complete tool that can transform a UML design notation directly into a JavaSim simulation program. It is possible though, to generate the simulation programs in any simulation language/environment. The components and actions identified in the SimML framework provide the building blocks for the construction of discrete-event, process-based simulation programs. Hence, the SimML framework can be used for generating simulation programs in other simulation languages (such as *SIMULA*) by modifying the program generator part of the SimML parser. The C++SIM was chosen because C++ compilers typically generate code which runs faster than similar SIMULA code, hence C++SIM would produce a more efficient simulation program. Since the second parser was written in Java, it was decided to use JavaSim for the simulation code to enable an easy interfacing task between the parser and the simulation program.

PERL was chosen for implementing the initial parser based on several reasons. First, PERL program can be modified and tested quickly, hence it is suitable for exploring the feasibility of implementing a SimML parser in the first place. Other reasons are due to PERL's features that allow an easy way to read data from file, to

store those data into a simple yet efficient data structure and later to write out the processed data into relevant (simulation source code) files.

The second (Java) parser was constructed after the first parser has been implemented and tested. The first (PERL) parser demonstrates that the idea of transforming the SimML framework (in a textual format) into a simulation program is workable. Unfortunately, PERL does not provide a support for implementing a graphical tool, therefore it can only be used in the construction of the back end stage of the UML to simulation process. Java, on the other hand, provides a full support on graphics, so the complete tool can be written in Java. Since the tool is built in Java, it is only appropriate that the simulation program is generated in Java programming language as well with the aid of the JavaSim package.

The rest of this chapter will discuss these two implementations in details, and brief introductions on the packages/languages used are provided. But first, it is important to outline the structure of a SimML parser along with its operations in general, which is independent of any programming language.

5.2. The Parser's Structure

A *parser* is "a computer program that breaks down text into recognized strings of characters for further analysis" [Miriam Webster's Dictionary]. For the SimML framework, the task of the parser is to interpret a text-based SimML specification file and discern the information conveyed into SimML components and actions. Some of these components will be container components (see Chapter 4, Section 4.3.1), for example, a PROCESS component will contain a list of PROCESS attributes (if any) and a sequence of SimML actions that are to be performed. Hence it is necessary to ensure that the SimML specification is read properly from file and the data are stored

in the right order before they are processed further. This requires the parser to keep a particular data structure to accommodate the SimML information. The two parser implementations that have been built employ different data structures, the first (written in PERL) uses a set of arrays while the second (in Java) uses a custom built class hierarchy (see Figure 5-11 in Section 5.4.2). These will be explained further in later sections. In general, the parser's functionality can be divided into two course of operations:

1. Reading the SimML information from a (textual) file and storing it into a particular data structure (in the memory).
2. Generating a simulation program using the information stored in that data structure (into files).

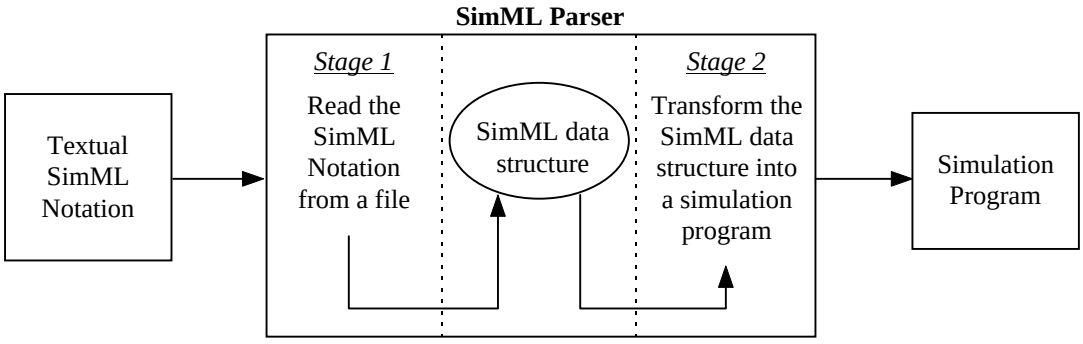


Figure 5-2: The two stages in parsing textual SimML notation into simulation

These two stages of operations convert a text-based representation of the SimML framework into a simulation program (Figure 5-2). The two implementations of this parser are explained in the following two sections.

5.3. PERL Implementation

The first implementation of a SimML parser was written in a scripting language called *PERL (Practical Extraction and Report Language)* [68, 73, 74]. PERL has some features which makes it an ideal language for retrieving data. Its *array* data structure

is flexible and can be manipulated easily, hence it is suitable for storing information of any arbitrary size. PERL also provides an easy way for reading and writing files. This is a very important feature since the SimML parser has to read a SimML specification from file, perform some transformation on the information read, and later write out the simulation code for that SimML specification.

A brief introduction on PERL syntax and its capabilities is laid out to give a flavour of the language. This is then followed by a discussion on how a SimML parser was constructed in PERL.

5.3.1. PERL Basics

PERL is an interpreted language optimised for scanning arbitrary text files, extracting information from those files, and printing reports based on that information [74]. It is UNIX based (although some emulators have been built to allow PERL to run on a PC) and its expression syntax corresponds quite closely to the C programming language's syntax. The most important part of PERL is its interpreter, which defines the syntax and the data types available in PERL, as well as other built-in functions for pattern matching, communication protocols, etc. Being a scripted language means that PERL source code is interpreted as it runs, i.e. there is no need to compile the source code. This can make a PERL program faster to debug, modify and test, but on the down side, can be more difficult to maintain the program's structure.

PERL Data Structure

PERL is not a strongly-typed language; unlike C++ or Java; there are no built-in types such as `int`, `double`, `char`, etc. Instead, PERL classifies its variables by what sort of data they can hold, whether the data is *singular* or *plural*. Based on that classification, there are three built-in data types defined in PERL: *scalars*, *arrays of*

scalars (or simply called “arrays”) and *associative arrays of scalars* (hash arrays). A *scalar* stores a single, simple value that is either a number or a string. It can be used as a building block for more complicated structures. An *array* is an ordered list of scalars where each scalar value is stored in a numerically indexed location (starting at zero) in the list. An array may contain numbers or strings, or a mixture of both, or even references to subarrays or subhashes. A *hash array* is similar to an array, the differences are that a hash array is not ordered and the scalars stored in it are indexed by some string values, not numbers. Also, each item in the hash array needs to contain both a *key* and a *value*, where the value is identified by its key.

As in other programming languages, PERL program stores its data in a set of *variables*. A PERL variable is prefixed by a certain character which indicates what kind of data structure it belongs to; this can be seen Table 5-1.

Table 5-1: PERL variable indicators

Type	Character	Explanation
Scalar	$	An individual value (number or string)
Array	@	A list of values, subscripted by number
Hash	%	A group of values, subscripted by string

Some examples on how PERL variables are declared and initialised are listed below:

```
$anInteger = 17;
$aRealNumber = 2.4342;
$aString = “Hello World”;
@anArray = (“value1”, “value2”, “value3”);
$anArray[0] = “value1”;
%aHashArray = (“key1”, “value1”, “key2”, “value2”);
$aHashArray{“key1”} = “value1”;
```

Sometimes it is necessary to have a more complex data structure. PERL provides a third scalar type called *reference* for achieving this. A reference is a form

of indirection where a variable refers to another variable, which can either be a scalar or an array. Since an array is composed of scalars, it is possible to have an array of references to other arrays, and this essentially constitutes a multi dimensional array. This kind of array is useful for storing multi-levelled information, such as that of the SimML components (this will be explained further in Section 5.3.2).

So, the data structures employed in the construction of the SimML parser are scalars, arrays and custom-built multi dimensional arrays (arrays of references). Some operations which are relevant for organising the information in those data structures are laid out below:

1. Array operations.

There are two operators that are needed by the SimML parser for adding and removing data in an array:

a) `push(@arrayName, $newValue);`

This operator allows a new value (or a list of new values separated by a comma) to be added as the last element of the array. The first parameter of this operator is therefore the name of the array upon which the new value(s) is to be stored. Please note that in PERL, an array's size is not fixed, i.e. the array will grow or shrink as needed.

The `push` operator is actually coupled with a `pop` operator, which takes the last element out of an array and returns it as a scalar variable. The `push` and `pop` operators enable a list to be represented as a *stack*, but this is not the data structure required in implementing the SimML parser. Instead an array is treated as a *queue*, which can be achieved by replacing the `pop` operator with a `shift` operator.

b) `shift(@arrayName);`

This takes the first element out of the named array and returns it as a scalar variable. It is therefore usual to have a variable name as a left hand side of this operation, for example:

```
$value = shift(@arrayName);
```

There is also an `unshift` operator which adds a new element as the first element of an array, but as with the `pop` operator, the `unshift` operator is not relevant for the SimML parser.

The two operators above facilitate a list that allows data to be added and removed from an array in a *First In First Out (FIFO)* order. On top of these two operators, there are two additional methods that are useful for accessing the values in an array:

- *subscript*

  PERL provides a subscripting operator (a pair of square brackets, i.e. "[]") to access an array element by its numeric index. An array element is treated as a scalar variable, therefore it is necessary to replace the `@` symbol before the array name with the `$` character for subscripting it, for example:

```
@array = (1, 2, 3); # initialise the array
$first = $array[0]; # obtain first array value
$array[0] = 0;      # set the value of the
                    # first array element.
```

  Note: The hash character (#), used as above, indicates that the rest of that line is a comment. This character is also used by the array syntax to obtain the last index of the array, as explained below.

- *index of last element*

  Trying to access an element beyond the ends of an array (i.e. an index of less than zero or greater than the last element's index) will return an undefined value. Therefore, it is necessary to know the length of the array before accessing its elements. The last element index of a PERL array `@array` can be obtained by using `$#array`, and since the array index starts with zero, this essentially means that the array's length is `$#array + 1`.

2. String Manipulation

PERL has a built-in feature for *regular expression* or pattern matching, which provides a great deal of help in searching and processing a large amount of data. There are several operators/functions available, but the most important one is the `split` operator. The `split` operator takes two parameters: a regular expression and a string; and splits the string up by returning parts of the string that do not match the regular expression into a list. It is also possible to store the values of this list directly into a named array by combining the `split` operator with the `push` operator. For example, to split a textual data into separate lines and store these lines into an array called LINES, the following PERL statement can be used:

```
push(@LINES, split(/^/, $data));
```

Each line can then be processed or split further into words:

```
$aLine = shift(@LINES);
push(@WORDS, split(' ', $aLine));
```

To process the whole data, it is necessary to employ two loops, one to go through each line and the other to go through each word of one line. Relevant operations are then performed based on each word; this could be storing the information into a

data structure (with reading of more words during that process if necessary) or validating that the information read follows a particular syntax. Figure 5-3 below shows the outline of this operation in PERL.

```
# obtain data first and store as “$data” variable.
$data = ...;
push(@LINES, split(/^/, $data)); # split into LINES.
while (@LINES)           # iterate for each line.
{
  $line = shift(@LINES);
  @WORDS = ();           # empty the WORDS array.
  # split line into WORDS (separated by a space).
  push (@WORDS, split(’ ’, $line));
  while (@WORDS)         # iterate for each word.
  {
    $word = shift(@WORDS);
    # process each word by storing it in an
    # appropriate array (with a validation check)
    ...
  } # end of WORDS loop.
} # end of LINES loop.
```

Figure 5-3: An outline for reading, evaluating and organising data

The code above actually represents the skeleton for the first stage of the SimML parser, namely reading the SimML notation and storing the data into an array data structure (Figure 5-2). However, it has not been mentioned how PERL reads data from a file. An overview on how file input and output operations are performed is given next.

File Input/Output Operations in PERL

PERL provides a "backquotes" facility which launches a (UNIX) shell command from within a PERL script. This starts off the specified command, waits for its completion, while obtaining the standard output as it goes along. Therefore, a file input operation can be performed easily in PERL by utilising the UNIX `more` command, which returns the content of a text file. All that is needed to do is to put the `more` command (with a file name parameter) within a pair of backquotes (` `) and store the returned result into a suitable variable, e.g.:

```
$content = `more data.txt`;
```

The data (stored in a PERL variable called `$content` in the above case) can then be processed as needed, which in SimML parser's case, is already outlined in Figure 5-3.

The output operation requires another PERL feature called *filehandle*. A filehandle is just a name that is given to a file, device, socket, or pipe, to hide the complexities of buffering to that destination; it is comparable to `stream` in C++. To create and attach a filehandle to a particular file, the `open` function is used. This function takes two parameters: the filehandle and the name of the file that is to be associated with it. Since the `open` function creates a filehandle that can be used for different purposes (input, output, piping), it is necessary to specify which operation is wanted. This is done by adding a special character in front of the filename, and for the output operation, the character used is the right angular bracket (>). This character signifies that a new file is to be created with the specified filename and the outputs are to be written to that file. Writing to that file is performed through the `print` command using the filehandler as the second parameter. When the output operation is completed, the file can be closed using the `close` function. For example:

```
open (MY_FILE, “>data.txt”);
print MY_FILE “Hello World”;
print ...
close MY_FILE;
```

The file input operation can be performed using a filehandle as well, but it is a lot simpler to use the backquote operation, especially in the UNIX environment.

Organising the code: Subroutines

Subroutines or functions provide a means for organising the source code into managable chunks. This considerably helps the programmer to understand the code better and provides efficiency by avoiding repeating the same code. In PERL, a subroutine can be defined in this syntax:

```
sub SUB_NAME
{
  statements ...
}
```

A subroutine can be called in several ways; the one that is used here puts an ampersand character (&) before the subroutine name, which is then followed by a list of parameter (if any, within a pair of brackets), for example:

```
&SUB_NAME(@PARAM_LIST);
```

In the subroutine definition, the parameter list becomes a special list called @_, which can be accessed in the same way as other lists. A PERL subroutine can also have a return value, this is the value of the last expression evaluated within the body of the subroutine on each invocation. For an example, let us consider a subroutine that calculates the sum of the numbers passed as its parameters:

```
sub SUM
{
```

```
    $sum = 0;
    while (@_)
    {
      $value = shift (@_);
      $sum = $sum + $value;
    }
    $sum;
}
# invoking the subroutine
$result = SUM;
print $result;            # This prints out 0.
$result = &SUM(5, 9, 16);
print $result;            # This prints out 30.
$result = &SUM(9, 0, -7, 12, 4, -10);
print $result;            # This prints out 8.
```

It can be seen that subroutines are essential in constructing a large program or a program that contains many repeated parts. This is the case with the SimML parser, where for example, several simulation program files are to be generated and each file is to be written out in a similar manner.

The PERL features illustrated above provide the necessary means for implementing the SimML parser. A more detailed explanation on how these features are used in constructing this parser is given in the next section.

5.3.2. From SimML to C++SIM using PERL

First, it was necessary to design the data structure for storing the SimML information. PERL's array is suitable for this purpose because each array can accommodate an arbitrary number of items, and based on the SimML framework, it is possible to work

out how many arrays are needed. There is one array each for the DATA, PROCESS, QUEUE, OBJECT, RANDOMS and STATISTICS components. Each array stores its information in a particular manner, depending on what sort of information (how many fields of data) is needed.

For example, the PROCESS array is required to store the information on all of the PROCESS components. Each PROCESS component contains a distinct name (through which the process is identified), a list of constructors (if any), a list of member variables (both public and private) and its member functions (complete with the action definitions for each function). Therefore, one PROCESS component will occupy several slots of the PROCESS array. In comparison, the QUEUE array only needs to store the information on all instances of the QUEUE component, which are just the QUEUE name and the type of object this queue will contain. Hence, the QUEUE array is a lot simpler compared to the PROCESS array. The list of the SimML component related arrays are given below, along with their further details:

1. @DATA_VARS

   This array stores the information for the DATA components, where each DATA component has a name and may have some attributes. Therefore, a multi-levelled array is used for storing the data. The root array will keep the DATA component's name, and this name can be used to point to another array which stores the attributes of that DATA component. This can be seen diagramatically in Figure 5-4.

   Later, when the parser tries to generate the C++SIM code, the DATA attributes are divided into two local arrays based on their visibility (*public* or *private*). These local arrays are declared inside a subroutine called CLASS (see

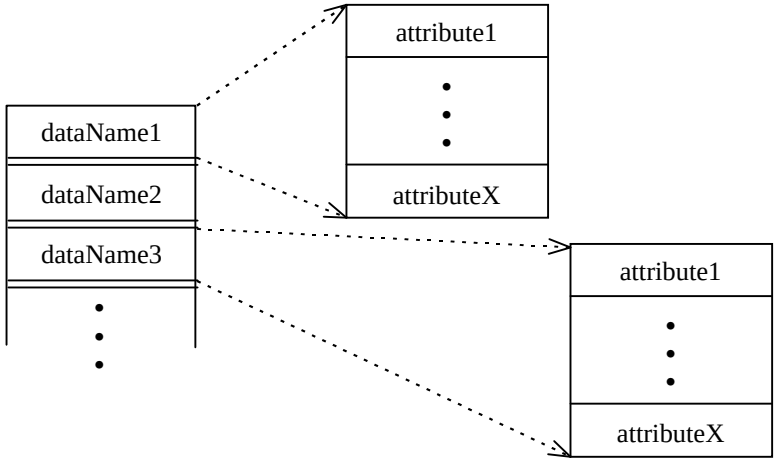


Figure 5-4: The DATA array

later) and they are used temporarily for storing the DATA attributes in a C++ syntax before being written into a file.

2. @PROC_VARS

As with the DATA components array, the PROCESS components array retains a multi-levelled array structure. Since a PROCESS component needs to have at least one function (the `+void Body()` function, which stores the information on its SimML actions), it is necessary to provide a slightly more complex array structure. The root array keeps the name of the PROCESS components, and each entry here points to another array that stores the PROCESS's attributes and function names. Each function name in turn points to another array which stores the details of the SimML actions associated with this function (Figure 5-5).

The PROCESS attributes are arranged in the same way as the DATA attributes. In fact, it could be said that the PROCESS component inherits from the DATA component. The difference is that a PROCESS component has at least one function defined, which contain some SimML actions specification. This indicates that the PROCESS component is an *active object* i.e. has its own thread of control, while the DATA component is only a passive data structure.

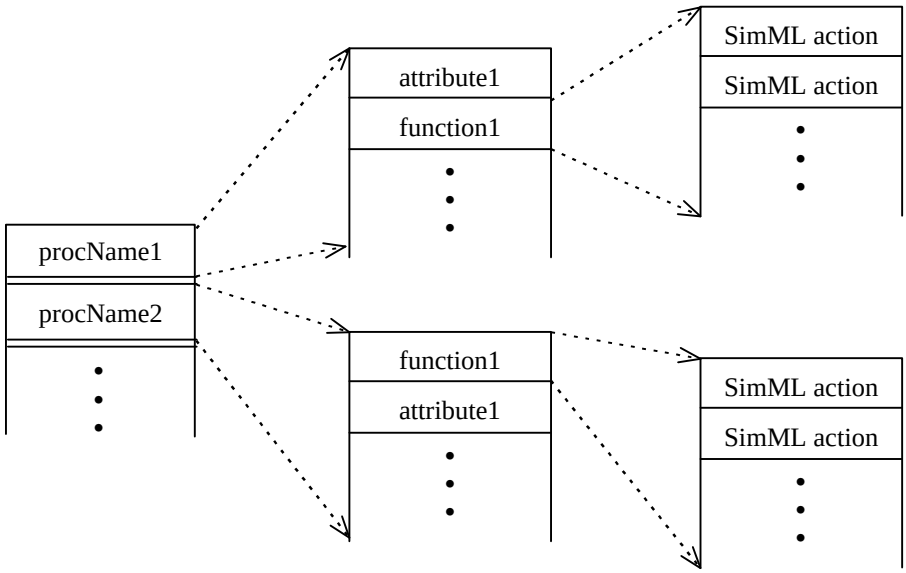


Figure 5-5: The PROCESS array

Since the DATA and PROCESS components are quite similar, it is more efficient to provide one subroutine to handle the generation of the C++SIM code for both components. This is done by a subroutine called CLASS, which is explained later in this section.

3. @QUE_VARS

   The QUEUE components are stored in a simple linear array, where each QUEUE component takes two slots in that array. The first slot keeps the QUEUE's name while the second holds the information on what type of objects are to be placed in this QUEUE. In other words, the first QUEUE component occupies the array at index number 0 and 1, the second QUEUE component takes slots number 2 and 3, etc. (see Figure 5-6).

4. @OBJ_VARS

   An object is an instance of a PROCESS component. The OBJECT array stores a set of three values for each OBJECT component: the OBJECT's type (which is essentially a PROCESS's name), its name and its multiplicity. The multiplicity is used for specifying a set of objects with the same type and characteristics; this

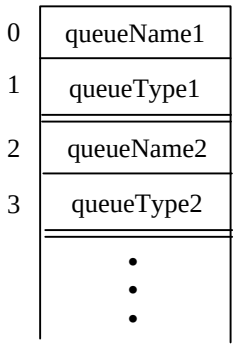


Figure 5-6: The QUEUE array

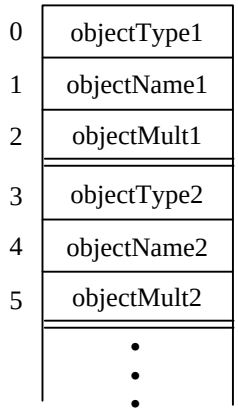


Figure 5-7: The OBJECT array

is a means of representing a multi-process. The arrangement of the OBJECT array is similar to that of the QUEUE array, i.e. the first OBJECT component occupies the array at index number 0, 1 and 2 (Figure 5-7).

5. @RANDOMS

A RANDOMS component is composed of random variables. Each random variable has a name and follows one of the five random distributions supported (see Table 4-1 in Chapter 4). It also has one or two parameters depending on what kind of distribution it belongs to. Sometimes, it also is necessary to specify the seed for generating the random numbers. Therefore, each random variable maintains up to five data values, which can be seen in Figure 5-8 (the elements within a pair of square brackets are optional).

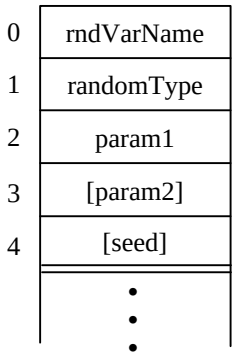


Figure 5-8: The RANDOMS array

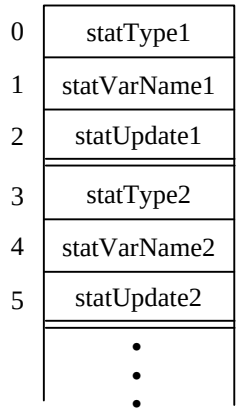


Figure 5-9: The STATISTICS array

6. @STATS

   Similar to the RANDOM component, a STATISTICS component is composed of statistics variables. Each statistics variable consists of three values: the statistics type (either an `int` or a `double`), its name and a description on how this variable should be updated (Figure 5-9). The update can be one of the two kinds:

   a). A simple increment or decrement.

   This kind of update is prefixed by a '+' or '-' sign (for increment or decrement), followed by the increment/decrement value, which can either be a numerical value or a C++ expression. This is useful for updating a counter-type statistics variable.

   b). A calculation involving other variables.

   It is prefixed by a '=' sign and followed by a mathematical expression involving other variables or literal values.

For the CONTROLLER component, there is no need to store the information in an array since a CONTROLLER's properties are well defined and a template can be constructed for it. It is therefore sufficient to store the name of the CONTROLLER component in a variable.

By using these array structures, the information obtained from the SimML specification (read from a text file) can be arranged and used properly.

Stage 1: Reading the SimML specification from file.

The PERL parser is designed to read a valid SimML notation from a text (ASCII) file. Any invalid syntax will be detected during the parsing and an error message will be

reported including the line number of the SimML notation file that contains the problem.

The parser initially reads all the content of the file into a PERL variable, which is then split into lines and stored in a PERL array called @LINES. Each line is then evaluated by splitting the words (separated by a white space) into an array called @WORDS (see Figure 5-3). The first word is a *keyword* which corresponds to one of the SimML components and it plays a major role in determining the organisation of the subsequent arrays. Any invalid keyword will cause the parser to display an error message and terminate. The valid keywords are:

i) `PROCESS`

This keyword indicates a SimML PROCESS component and is followed by a name through which this PROCESS can be referred to. There is an optional keyword after the name called `"once"`, which indicates whether the PROCESS's life-span is only for one execution (by default, the PROCESS's life-span is throughout the simulation).

Since PROCESS is a *container* component, it is followed by a block of statements (within a pair of curly brackets) that contains the PROCESS's attributes and functions. There must exist a `+void Body()` function, which contains the SimML action specifications. An example is given below:

```
PROCESS Server
{
    +double delay
    +void Body()
    {
        <SimML Action specification>
```

```
            ...
        }
    }
```

ii) `DATA`

It is similar to the PROCESS keyword, but it does not have a life-span nor any function, e.g.:

```
DATA Job
{
    +int id
    -double arrTime
}
```

iii) `QUEUE` or `SEQUENCE`

A QUEUE component is followed by the QUEUE's name, a keyword ("`of`") and the type of elements to be stored in that QUEUE. More than one QUEUEs can be specified for each SimML specification.

A SEQUENCE is a kind of queue that allows a simple priority ordering of its elements. There is a number associated with each element (stored as the element's member variable usually called `id`); the lowest the number is, the higher priority the corresponding element becomes. An element with the lowest `id` value will be placed at the head of the queue. The value of the `id` variable is assigned by a PROCESS component, usually an Arrival process.

The syntax for the SEQUENCE component is therfore slightly different: The `SEQUENCE` keyword is followed by the SEQUENCE's name, a keyword ("`of`"), the elements type, a keyword ("`using`") and the variable name that stores the priority number ("`id`"). It is necessary to ensure that the variable name

used for the priority number is actually declared as a public member variable inside the DATA/PROCESS component that meant to be placed on the SEQUENCE. An example is shown below:

```
QUEUE Queue of Job
SEQUENCE Seq of Job using id
```

iv) CONTROLLER

This component's property is pretty much fixed, so the parser has provided a template for its code. The only thing necessary to be supplied is the name by which this CONTROLLER component is known. There is only one CONTROLLER component per SimML specification. For example:

```
CONTROLLER MyController
```

v) OBJECT

An OBJECT keyword is followed by the name of the OBJECT, a keyword ("`of`") and the object type. Multiple objects (considered as a group) can be declared by adding the object's multiplicity after its name (surrounded by a pair of square brackets), e.g.

```
OBJECT singleServer of Server
OBJECT multServer[3] of Server
```

vi) RANDOMS

RANDOMS is another container component. It is followed by a set of RANDOMS variable, declared within a pair of curly brackets. Each RANDOMS variable is composed of a name, the type (which must be one of the five supported types as listed on Table 4-1), and its parameters:

```
RANDOMS
{
```

```
        randVar1 exponential 4
        randVar2 uniform 1 5
    }
```

vii) `STATISTICS`

This is also a container component and its structure is quite similar to that of the RANDOMS component. The STATISTICS component is composed of STATISTICS variables, each of which has a type (either `int` or `double`), a name and an expression on how this variable is updated. For example, if we would like to work out how long on average it takes to process a Job (which stores the information on its arrival time), we could declare three STATISTICS variables as follows:

```
STATISTICS
{
    int totalJob +1
    double totalTime +Time()-j->arrTime
    double avgTime =totalTime/totalJob
}
```

The updates are performed by the *update* actions which are declared inside the relevant SimML PROCESS(es).

After the SimML specification file has been read successfully, the information obtained are stored in the PERL arrays before the second stage of the parser is performed.

Stage 2: Generating the C++SIM program

The information stored in the SimML-customised arrays needs to be transferred into several C++ classes (stored as the C++ `.h` and `.cc` files) in order to build a

C++SIM program. There will be a class each for the DATA, PROCESS and QUEUE components, one for the CONTROLLER component plus a `Main.cc` file that is later compiled into an executable `Main` program.

Since PERL supports subroutines, several subroutines have been implemented to convert the SimML components information from the PERL arrays into their corresponding C++ code. These subroutines are:

1. ACTIONS

   This subroutine handles the transformation of SimML action notation into the corresponding C++SIM code. Each SimML action notation begins with a keyword that specifies which action is to be performed (one of the valid SimML actions listed in Section SimML Actions). Depending on the action keyword, there are some parameters that need to be specified. If the parameters do not match, the parser will print an error message and terminate.

2. CLASS

   The construction of the C++SIM code corresponding to the SimML DATA and PROCESS components is performed by this subroutine. This subroutine takes two parameters: the first one is just a string variable for determining whether this subroutine needs to construct a DATA or PROCESS component. This parameter is used in conjunction with the second parameter, which is an array containing the information of either the DATA or PROCESS component (through the @DATA_VARS or @PROC_VARS array).

3. CONT

   Since the CONTROLLER component's behaviour is relatively fixed, this subroutine takes care of everything that corresponds to the generation of the

C++SIM CONTROLLER class. This class organises the simulation run by obtaining the simulation parameters, activating the appropriate processes, waiting for the simulation time before terminating and summarising the simulation.

4. MAIN

   In order to run the C++SIM code, there needs to be a program that contains the `void main()` function, which essentially starts up the simulation program by initialising the threads and then passing the control to the CONTROLLER class. The purpose of the MAIN subroutine is therefore to create a simple program that is executable from the operating system.

5. QUE

   This subroutine creates the QUEUE classes that represent the SimML QUEUE components. The QUEUE elements must be of a certain type, which is represented by (associated to) the name of a DATA or PROCESS component. The QUEUE class uses a Linked-List data structure for storing its elements and supports the essential member functions, such as those for checking whether the queue is empty or not, adding/removing an element and obtaining the queue size.

In addition to these, there are more subroutines which support the features of the parser as well as for compiling and running the C++SIM program generated. Most of these subroutines are invoked from within the ACTIONS subroutine and their purposes are for the validation of the SimML actions' parameters.

1. CLEAN

   The data read from file need to have their new line character (`"\n"`) as well as their blank spaces removed, otherwise the information will not be interpreted

correctly. The task of cleaning the "raw" data is therefore performed by this subroutine.

2. VALID_OBJ

   It is used to determine whether a named object that participates in an action is actually valid (stored inside the @OBJ_VARS array) or not. An invalid object will cause the parser to display an error message and terminate.

3. VALID_STREAM

   The VALID_STREAM subroutine is used by the *wait* and *generate* actions. These actions require a random variable name as their parameter, and only those already declared inside the @RANDOMS array are allowed.

4. MULT

   If an object has a multiplicity that is greater than one (i.e. it is actually a group of objects of the same class treated as one item), it is necessary to handle it in a particular way by taking into account every member of the group. The notion of multiplicity is associated with the PROCESS component and normally used on the *activate* or *terminate* actions.

5. ONCE_ONLY

   In most cases, a PROCESS component's life-span is throughout the simulation length where it has a loop that repeats the same sequence of actions that needs to be performed. In some cases though, it is necessary to have a PROCESS component that executes its set of actions only once then terminates. The names of the PROCESSes that belong to the second category are stored in a simple array called @DYN_OBJS. This array is updated during the reading of the SimML specification from file, based on whether the PROCESS declaration contains a

`"once"` keyword or not. The purpose of the ONCE_ONLY subroutine is to make it easier to check whether a named PROCESS component need to have a forever loop inside its `+void Body()` method or not.

6. VALID_QUEUE

   The *check*, *dequeue* and *enqueue* actions use this subroutine to determine whether the queue name supplied as their parameter is a valid one or not. In case of the *dequeue* and *enqueue* actions, it is also necessary to verify whether the object to be added to or removed from the queue is of the valid type (i.e. the object is an instance of a valid DATA or PROCESS component and the queue stores elements of the same type).

7. CREATE_IMAKE

   C++SIM environment uses a UNIX program called `imake` to automatically generate the `Makefiles` (which are useful for building a group of programs like those for our simulation) from a template. The `imake` program reads an `Imakefile` configuration file that specifies the source and object files involved in a particular group of compilation. The purpose of the CREATE_IMAKE subroutine is therefore to generate the `Imakefile` based on the classes captured from the SimML DATA, PROCESS, QUEUE and CONTROLLER components.

8. CREATE_MAKE

   After the creation of the `Imakefile`, it is necessary to generate the `Makefile` that is used by the UNIX `make` command to build the final executable program. This automatic `Makefile` generation is performed by the CREATE_MAKE subroutine.

The transformation of the SimML information (stored in the PERL arrays) into C++SIM code is done by invoking the appropriate subroutines in a particular order along with their parameters (if any) as listed below:

i) &QUE(@QUE_VARS)

ii) &CLASS("DATA", @DATA_VARS)

iii) &CLASS("PROCESS", @PROC_VARS)

iv) &CONT($controllerName)

v) &MAIN

vi) &CREATE_IMAKE

vii) &CREATE_MAKE

The above subroutine calls are then followed by a UNIX shell invocation that executes the `make` command to generate the object (`.o`) files for the C++SIM code. This is then followed by a PERL `exec` function to execute the `Main` program which initialises the simulation run.

The execution of both the first and the second stage of the PERL SimML parser allows a complete transformation of a textual SimML notation into C++SIM program. After several name changes, it was decided to call this parser `runsim` and it is runable on the UNIX/Linux platform.

This concludes the section that describes the implementation of a SimML parser in the PERL scripting language. Some case studies on the application of this parser are discussed later in Chapter 7. The next section explains the second implementation of the SimML parser in Java, which has some added functionality for supporting a GUI tool for capturing the UML design notation (class and sequence diagrams) and for generating XML as well as JavaSim code.

5.4. Java Implementation

Java has emerged as a very popular programming language since 1996 when *Sun Microsystems* released their free Java compilers on the internet. Java is an object oriented programming language whose syntax is very similar to that of C++ but differs in many aspect to stand out as a language of its own. One major difference is that Java does not support *pointer* - a powerful and useful feature, yet quite difficult to understand and may lead to trouble - which might be missed by some C++ programmers. Java also provides an easier memory management by performing an automatic garbage collection, which can minimise memory leaks.

There are many reasons why Java has dominated the programming world. Being a free programming language helps Java's popularity a lot, but the biggest reason is probably due to Java's high portability which means that it can be installed and run on practically any machine. Java achieves this portability by compiling its source code into a byte code that is executed on a *Java Virtual Machine* (JVM) instead of directly on the host machine. JVM is Java's own personal machine which runs on the host machine and has been ported to a variety of platforms and operating systems. Java byte code for one platform can therefore be run on another platform because it is actually running on the same machine: the JVM.

Java also provides many features that allow programs with intensive *Graphical User Interface* (GUI) to be constructed easily. This benefit is one of the main reasons why Java was chosen as the language for building our UML tool (from now on, we call it a *UML/SimML* tool for reference's sake). As mentioned at the beginning of this chapter, the aim of this tool is to transform (relevant) UML notations into (JavaSim)

simulation programs. It consists of the *front end* GUI tool for drawing UML diagrams and the *back end* SimML parser for generating JavaSim programs (see Figure 5-1).

The rest of this section explains how the *UML/SimML* tool was implemented, and it begins with an introduction of the Java features necessary for building GUI programs. This is then followed by the formulation of approach in building the *UML/SimML* tool: the construction of the back end parser as well as the front end GUI tool, and the amalgamation of these two ends. The use of the *Extensible Markup Language* (XML) for storing SimML related information is also discussed. The XML notation was actually used in testing the back end parser before the GUI front end was developed. More detailed explanation on the Java language itself can be found in many references such as [12], [22], [23], [30], [37], [42], [75] and [77].

5.4.1. Java Foundation Class (JFC) and Swing

Java provides a generic, platform-independent windowing system called *Abstract Window Toolkit (AWT)* for writing *Graphical User Interface (GUI)* programs. AWT was the only/main GUI library for older versions of Java (i.e. those prior to Java 2); later versions support a new package called *Swing*, which is part of the *Java Foundation Classes (JFC)* that provide a better GUI facilities.

Java Foundation Classes (JFC) encompass a group of features to help people build graphical user interfaces (GUIs) [41]. This is achieved by providing standard GUI components, such as buttons, lists, menus, and text areas, as well as some container components for organising the GUI's layout. *Swing* itself was the codename of the project that developed these new components, which are subsequently called *Swing components*. The basic concepts of Swing are outlined below, which were later used in building the GUI part of the *UML/SimML* tool.

Swing Components and the Containment Hierarchy

Swing provides a containment hierarchy for arranging its components in a GUI. In order to appear on screen, every GUI component has to be part of a containment hierarchy. Based on the containment hierarchy, it is possible to separate the Swing components into three categories:

- *Top Level-Container*

  This kind of component exists mainly to provide a place for other Swing components to paint themselves. The commonly used top-level containers are frames (`JFrame`), dialogs (`JDialog`) and applets (`JApplet`).

- *Intermediate Container*

  Its only purpose is to simplify the positioning of the atomic components (see below). There are a few intermediate Swing containers, such as panel (`JPanel`), scroll panes (`JScrollPane`) and tabbed panes (`JTabbedPane`)

- *Atomic component*

  An atomic component is a component that exists as a self-sufficient entity that presents bits of information to the user. Most of the Swing components belong to this group, such as button (`JButton`), check box (`JCheckBox`), radio button (`JRadioButton`), list (`JList`), menu (`JMenu`), label (`JLabel`), text field (`JTextField`), text area (`JTextArea`), etc.

Each containment hierarchy has a top-level container as its root. In turn, each top-level container has a *content pane* that contains the visible components in that top-level container's GUI. Intermediate containers and atomic components are placed inside the content pane, which is accessible using the `getContentPane()` method of the top-level container.

Containers have a method called `add`, which has at least one parameter: the component to be added into the container. In order to make a nice layout, the components are ordered according to a particular arrangement, which is specified by one or more *layout managers* (explained next). The layout managers dictate how the components are added and placed inside a container. There are various forms of the `add` method, and one of the forms also takes a layout manager as a parameter as well.

Layout Management

Layout management is the process of determining the size and position of components. There is a *layout manager* for each container, whose task is to perform the layout management for the components within that container. Java provides five commonly used layout managers: `BorderLayout`, `BoxLayout`, `FlowLayout`, `GridBagLayout`, and `GridLayout` (details on how they differ from one another can be found in [41] or any Java books)

Components can provide their size and position information to layout managers, but in the end, it is up to the layout managers to decide on the size and position of those components. After the components have been added to the relevant containers, the top-level container needs to invoke the following two lines in order to display the GUI:

```
pack();
setVisible(true);
```

The `pack()` method causes the size of the top-level container to be determined and its contents to be laid out using the layout managers. The `setVisible(true)` method makes the window (where the top-level container resides) visible and usable as a GUI.

Event Handling

To make an interactive GUI, some of the components need to be able to respond to external events, such as the user pressing a mouse button. These events are managed by *event handlers*: they are Java classes that implement a *listener interface* (one of those declared in the Swing library, such as `ActionListener`, `MouseListener`, `MouseMotionListener`, `KeyListener`, `ListSelectionListener`, etc.) and register one or more *event listener* on the appropriate *event source* (usually a Swing component). For example, if we would like to handle a mouse-clicked event on a button, the following skeleton code could be used (the **bold** typefont indicates the event handling code):

```
public class MyHandler implements ActionListener
{
    // declare a button component
    JButton myButton = new Jbutton();
    ...
    // register an action listener for this button
    myButton.addActionListener(this);
    ...
    public void actionPerformed(ActionEvent event)
    {
      // ...code that reacts to the action
    }
}
```

The event listeners are implemented by the GUI programmers to perform whatever needs to be done in response to a particular event.

For more information on the Swing package, see [41] or [75].

5.4.2. Our Approach in Building the *UML/SimML* Tool

Chapter 4 identifies a framework called SimML that can be used to bridge the transformation of a UML design notation into a simulation program. So far we have seen how this framework is applied in building a PERL parser (called `runsim`) that can generate C++SIM simulation programs automatically from a SimML notation (see Section 5.3).

We would like to extend this idea by adding some features that allow simulation programs to be generated directly from graphical UML notations. To achieve this goal, we split the overall task into several stages:

- investigating possible solutions or available tools,
- extracting SimML information from UML design,
- generating simulation programs from SimML information.

5.4.2.1. Formulating a solution

There were several possible lines of approach investigated before we came to one solution. Since there already exists a parser that can transform a (textual) UML-like notation into C++SIM program (see Section 5.3 and [6, 7, 10]), we could have adapted one of the UML tools available (such as the Rational Rose tool [63] or Argo/UML [71]) by extracting the UML information into a suitable textual format. This textual notation can then be applied to the above parser in order to generate the simulation program.

In the end though, it was decided to construct our own UML tool using Java's JFC/Swing package (see Section 5.4.1 above and [41]). 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 extremely

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.

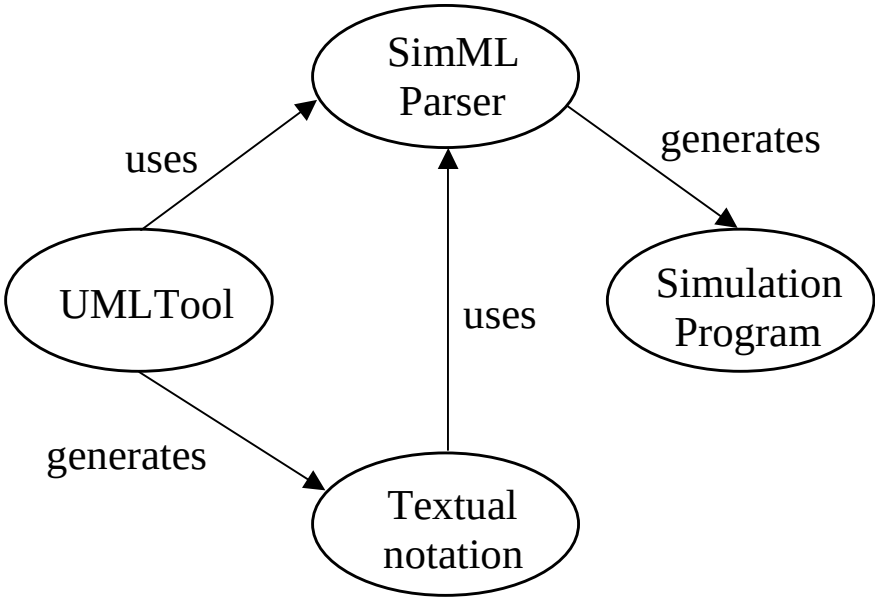


Figure 5-10: The paths for generating simulation from UML

Figure 5-10 shows the possible paths that can be taken to generate a simulation program from the UML design notation.

The first attempt is focused on how to represent the SimML framework in Java environment. This is a very important task because we need the SimML framework to act as a bridge that connects the UML design and the simulation program. In other words, the SimML framework is used by *both* the front end (GUI) part and the back end (simulation generator) part of the complete *UML/SimML* tool.

The SimML framework representation in Java was achieved by building a data structure composed of several sub structures that represent the relevant SimML components. The next section illustrates how this data structure is constructed as a Java package.

5.4.2.2. 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 [37]. 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 5-11. (***Note:*** Initially, our simulation framework was called `SML` instead of `SimML` because we were not aware of a programming language with the same name. Subsequently, we changed the framework's name to `SimML` to avoid name conflict, but the Java package's name remains unchanged for simplicity).

The SimML data structure in Java offers a better and clearer organisation of components compared to that of the PERL implementation. Every SimML component is represented by a class and it inherits from a base class called `SMLComponent`. The inheritance allows necessary methods (such as those for setting and getting the component's name and parameters as well as that for generating JavaSim) to be declared uniformly (i.e. with consistent method signature). Each SimML component then implements its own version of these method suited to its requirement.

As mentioned in Section 4.3.1, the SimML components can be classified into *simple* (or *atomic*) components and *container* components. This is also the case with the Java implementation of our SimML tool. This classification can seen clearly in the UML class diagram representation of the SimML framework (Figure 5-11), where container components are shown as aggregation of one or more simple components.

There are three Java classes for the SimML simple components: `SMLQueue`, `SMLObject` and `SMLController`. There is also an extra class that can be considered as a simple component called `SMLMainProg`, which creates the part of the simulation program that contains the executable "`main`" method.

The container classes are composed of sub-elements, and these sub-elements are not any of the simple components above. There are four container classes defined:

- `SMLData`

  `SMLData` contains zero or more `SMLAttr` (a class that represents attributes of the SimML DATA component), which are stored as elements of a `Vector` variable inside the `SMLData` class called `attrInfo`. (***Note:*** `Vector` is an implementation of array data structure that allows dynamic size allocation).

- `SMLProcess`

  This class inherits from the `SMLData` class so it also contains zero or more instance of `SMLAttr`. On top of that, `SMLProcess` also contains zero or more `SMLAction`, which represents SimML actions. The instances of SMLAction are stored in another Vector variable called `actInfo`.

- `SMLRandom`

  `SMLRandom` class is the container for the `SMLRandComp` instances, which store the details of each random variable (including its name, numerical type and distribution function). These instances are stored in a Vector called `randInfo`.

- `SMLStatistics`.

  This is very similar to the `SMLRandom` container, the only difference is it contains instances of statistics variables (`SMLStatComp`), which are stored in `statInfo` Vector.

There is a special class called `SMLStructure` that holds the information on all the SimML components (as well as the SimML actions in the case of the SimML PROCESSes). As can be seen in Figure 5-11, this class is composed of instances of the `SMLComponent` class, and these instances are stored either as elements of some Vector member variables or just as simple member variables.

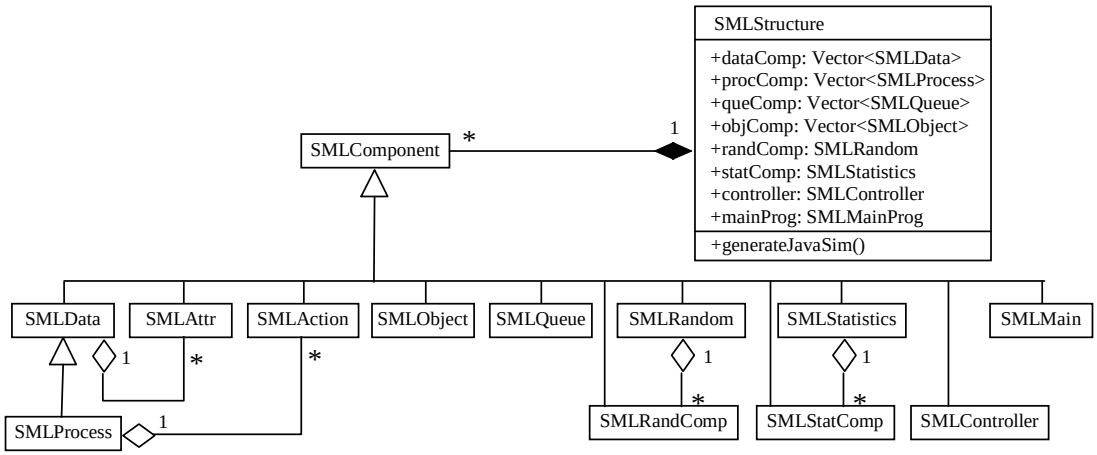


Figure 5-11: The class diagram for the SimML Components and Actions

Simulation information (which is derivable from the UML diagrams - see Section 5.4.2.3 below) is loaded into the `SMLStructure` before being transformed into a simulation program. A member function can be added to the `SMLStructure` class to perform this transformation for a particular simulation environment. The work illustrated here uses JavaSim as the target simulation environment, therefore a member function called `generateJavaSim()` has been added to the `SMLStructure` class above. Details on this member function implementation can be seen in Section 5.4.2.4.

5.4.2.3. Deriving SimML information from UML diagrams

UML allows both static and dynamic characteristics of a system to be shown by using the appropriate diagrams and notations. This is useful for the SimML framework because simulation information can also be divided into static and dynamic properties

amongst other things (the random and statistics properties are other points of interests in simulation). Therefore, capturing SimML information from UML can be done as follows:

- Static Properties

  The SimML framework specifies the static characteristics of simulation inside the DATA and PROCESS component. These characteristics include a name through which the DATA/PROCESS component can be referred to, as well as member variables belonging to that component.

  The UML *class diagram* is an ideal place for capturing these static properties. This is because the class diagram has a field for storing the class's name (which is comparable to the DATA/PROCESS name in the SimML framework) along with a field for specifying its member variables.

- Dynamic Properties

  SimML actions represent the dynamic characteristics of simulation. SimML actions are invoked *only* from within PROCESS components but they might involve instances/objects of DATA, QUEUE or other PROCESS component. Each of these instances is given a name, through which the interactions are specified. These SimML actions must also be executed in a particular sequence or order (within one PROCESS component), since they affect the interactions of the objects involved in the simulation.

  There are many notations in UML that can be used to convey the dynamic aspects of a system, but one notation that comes closest to the SimML requirement is the *sequence diagram*. The sequence diagram allows objects to be specified (on the top side of the diagram) and each object has a time-line associated to it. The

interactions among the objects are achieved by passing *messages* (indicated as a labeled arrow) from one time-line to another. The order of the messages is determined by their position on the time-line: the lower down the diagram they are, the later they are executed.

The 14 SimML action keywords (see Chapter 4, Section 4.2.2) can be represented as the labels of the messages. These labels follow the notation of a standard "function call", which might include some values passed as its parameters. For some of the SimML actions that need an interaction with another object (such as *activate*, *enqueue* and *dequeue*), the message's arrow points to the time-line of the relevant object. Otherwise, the arrow can just point to its own time-line to indicate a self-invocation.

The random and statistics simulation properties are not quite easy to derive from any UML notations though. This is due to the fact that there is no UML notation that is directly applicable to or sufficiently capable of conveying the characteristics of these two properties. It is possible to attach the random and statistics properties as UML notes, but this approach weakens their special case since UML notes tend to be too expansive. It was therefore proposed to extend UML by adding two extra views that allow the random and statistics properties to be specified consistently:

- Random Properties View

  This view consists of the *random variable names* along with their appropriate *types* and *parameters*. To ensure that only a valid random distribution function is used as the type, it is necessary to provide a list (represented as a pull-down menu) showing the five supported random distribution functions (as specified in Table 4-1) for the user to select. The number of parameter(s) is either one or two

depending on the distribution function, so one or two field(s) should be given accordingly to contain the values of the parameter(s). The user should also be able to select a different distribution function from the first one selected if necessary, and to modify the parameters (i.e. the random properties should be editable).

- Statistics Properties View

  The statistics view contains the statistics variable names, each of which also conveys its type (either an `int` or a `double`, shown as a pull-down list), modification mode (either *simple* increment/decrement or *calculation*) and a mathematical expression on how this variable is to be updated (which depends on the modification mode). Similar to the Random properties, the Statistics properties should be editable.

The four views above enable the components and actions of the SimML framework to be captured in a pretty straightforward manner. Therefore, in order to derive SimML information automatically from UML diagrams, it is necessary to provide a means for supporting these views. Two of the views (static and dynamic properties) are already supported by the standard UML notation through the class and sequence diagrams, but the other two (random and statistics properties) are not. This lack of support from UML notation was taken into account when we design and build the *UML/SimML* tool, which aims to support all the four views in one tool. Another feature that this tool also needs to support is the JavaSim program generator, and this is explained in the following sub-section.

5.4.2.4. Generating JavaSim code

The SimML components and actions can be mapped directly into simulation entities and interactions, and these mappings pave a way for building a parser that can

automatically transform the information stored in the SimML framework into a simulation program. The process of transforming the SimML data into simulation program can be made easier by organising the data into a well-defined structure, for example by using a *modular* approach. Each module knows how to write the information stored in it into a relevant piece of JavaSim code (into a file), therefore a complete JavaSim code can be obtained by performing the write operation on all the modules.

As mentioned in Section 5.4.2.2, the SimML framework can be implemented 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 5-11). 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 from its data to a segment of JavaSim program.

Some of the SimML component classes (`SMLData`, `SMLProcess`, `SMLQueue`, `SMLController`, and `SMLMainProg`) contain a `write()` member function to perform the intended transformation. These classes are what we meant by *modules* in the first paragraph above. Each `write()` member function attempts to write out the information stored in this module into a file, which is named according to the name associated to that module. Some of these modules must also take into account the information stored in other SimML classes related to it. For example, the `SMLData` and `SMLProcess` component need to incorporate the data from the `SMLAttr` objects associated to them before writing out the file. On top of that, a `SMLProcess` class also has to deal with its `SMLAction` instances, which in turn

might need to obtain some information from the `SMLRandom` or `SMLStatistics` classes. The modular arrangement of the SimML components can be seen in Figure 5-12 below. The module classes are shown to be derived from an imaginary class called `SMLModule`, which in reality is the same as the `SMLComponent` class. This alias is necessary to illustrate how different classes are considered in our modular approach. A module that needs to obtain further information from another class is denoted by a *dependency* association to the relevant class. In Figure 5-12, we can also see that the `SMLAttr`, `SMLAction`, `SMLObject`, `SMLRandom` and `SMLStatistics` are not module classes, but they are used by the module classes to obtain the whole information of the SimML framework.

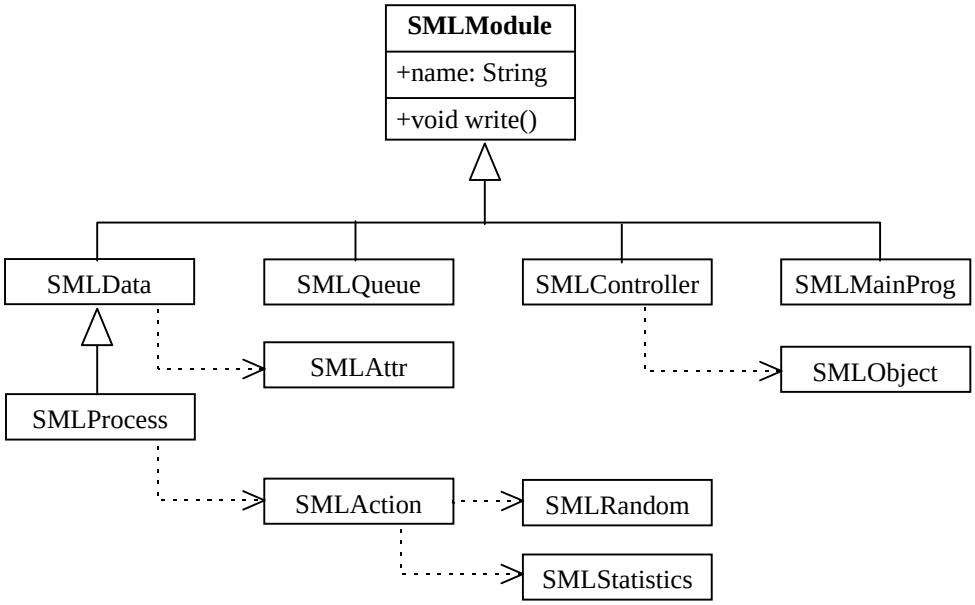


Figure 5-12: Modular organisation of the SimML Components

The `write()` functions of all the modules are then invoked by the `generateJavaSim()` function (of the `SMLStructure` class) in order to generate 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 to be made are limited to the language specific syntax as the SimML framework is generic to almost all process-based simulation requirements.

5.4.2.5. Building the UML/SimML Tool

A GUI tool that brings together the UML design and the SimML framework has been created. This tool is called the *UML/SimML* tool, and it supports only the relevant UML diagrams (the class and sequence diagrams) and allows simulation specific information to be identified in an easy and generic way. As a recap, the four views supported are:

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 4.2.2) 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 Table 4-1), 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 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.

The *UML/SimML* 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 specify the random and statistics properties. They also allow the user to select one of the four supported views mentioned above to be displayed when necessary. All of the classes shown in Figure 5-13 belong to this category.

- The classes that support the SimML framework.

  These are bundled together in the `ncl.SML.Components` package explained in Section 5.4.2.2. The classes of this package is used to store the information of the class and sequence diagrams, as well as the data contained in the random and statistics views. 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 in the *UML/SimML* tool can be represented as a UML class diagram as seen in Figure 5-13. By using the `ncl.SML.Components` package for storing the information conveyed by the four views above, we are able to generate a simulation program automatically from a UML design notation.

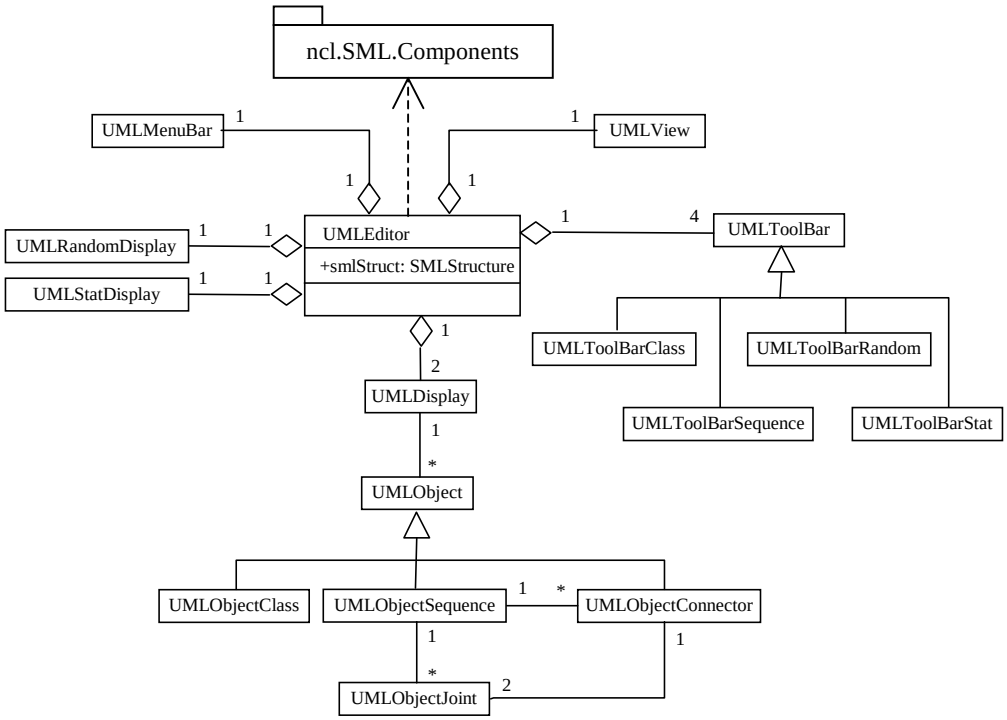


Figure 5-13: The Structure of our UML Tool

The design layout of the *UML/SimML* tool can be seen graphically in Figure 5-14. The classes that support the GUI features of this tool are explained as follows:

- `UMLEditor`

  This is the top level container of the whole *UML/SimML* tool, so it inherits from (or extends) the `JFrame` component. The `UMLEditor` maintains the layout of the tool (which should look like that shown in Figure 5-14) by using a pair of `JSplitPane`s, and keeps the information on the instances of other GUI components as well as an instance of the `SMLStructure` (to enable an automatic generation of a simulation program). This class also contains some useful member functions that might be used by other GUI classes as well, such as those for

splitting a text into separate lines, removing the blank spaces of a text, and preparing the SimML related information for further use.

- `UMLDisplay`

  This class serves as a whiteboard for drawing the UML *class* and *sequence* diagrams. There is one `UMLDisplay` instance for each diagram (and they are declared inside the `UMLEditor` class), so the characteristics of the `UMLDisplay` class should be generic enough to be useable in both cases. The `UMLDisplay` class extends the `JLayeredPane` class (to enable multiple diagrams to be contained in the same area), implements the `MouseListener` interface (to handle mouse events associated with drawing the diagrams) and also implements the `Printable` interface (so that the diagrams can be printed either to a printer or a postscript file).

- `UMLObject`

  This is the parent class for the classes that represent or draw the UML class and sequence diagrams. It stores the identity, size and location (within the `UMLDisplay` area) of the diagrams and implements the `MouseListener` and `MouseActionListener` interfaces to enable the drawings to be moved, modified, etc. Specific handling of these interfaces are performed by the derived classes, which are as follows:

  – `UMLObjectClass`

    The `UMLObjectClass` represents the UML class diagram and it needs to provide three fields for the user to specify the name of the class diagram, its attributes and member functions/operators. These fields are represented by classes called `UMLObjectClassName`, `UMLObjectClassAttr` and

`UMLObjectClassOper` respectively. They extend the standard `JTextArea` class and implement the `KeyListener` interface (to handle the user input from the keyboard). These three classes are encapsulated within a class derived from the `JPanel` called `UMLClassHolder` which handles all the mouse events, including those for clicking, moving and deleting the diagram.

– `UMLObjectSequence`

The shape of the UML sequence diagram dictates a slightly different approach in representing it from the `UMLObjectClass` above. A sequence diagram is represented by a box with an object name on top and a vertical time-line (which is expandable) below that box. The box can be represented using a similar approach as the `UMLObjectClassName`, and in this case, it is called the `UMLObjectSequenceHead`. The dashed time-line is used in conjunction with arrowed messages (implemented as `UMLObjectConnector` below) to represent the interactions among the objects in the sequence diagram. All in all, the representation of the sequence diagram is encapsulated within a `UMLObjectSequenceHolder` class, which also handles all the mouse events similar to those of the class diagram. The messages involved in the sequence diagram are represented using these two classes:

* `UMLObjectConnector`

  This class also extends the `UMLObject` class because it performs some drawing as well. It draws an arrowed line from one sequence diagram time-line to another (or to the same time-line for a self invocation message) and contains a `UMLObjectConnectorText` class (which extends the

`JTextArea` class and implements the `KeyListener` interface) for denoting the message and its parameters (if any).

An instance of the `UMLObjectConnector` is associated directly with one sequence diagram (a `UMLObjectSequence` that performs as the interaction source), but it may also be associated with another sequence diagram (if it is *not* a self invocation message) through the `UMLObjectJoint` (see Figure 5-13 for a clearer view on this relationship).

* `UMLObjectJoint`

  This class is used as a data structure for storing the relationships between the messages and the objects of a sequence diagram. This class also provides some member functions for processing the messages contents and translating them into the corresponding SimML actions. Only messages that match the SimML action keywords will be treated in this special manner, the rest will just be ignored.

  A `UMLObjectSequence` instance keeps a `Vector` for storing the `UMLObjectJoint`s connected to it, hence all the SimML actions performed by (or involving) that object (which is essentially an instance of a SimML PROCESS component) can be captured from the diagram, and later used to generate the dynamic characteristics of the simulation program.

- `UMLRandomDisplay`

  The random property of a specification is captured from the UML sequence diagram through messages with *wait* or *generate* keywords. The `UMLRandomDisplay` class automatically extracts the random variable names (which are passed as parameters in the messages with the two keywords above)

from the sequence diagram and displays them one at a time using a pull-down list. This class also provides a pull-down list of the five random distribution functions (those outlined in Table 4-1) to allow the user to indicate which distribution function a random variable should belong to. Depending on the random distribution function, there is one or two field(s) displayed for the user to specify the parameters for this random variable. The `UMLRandomDisplay` inherits from the `JPanel` class and implements the `ItemListener` interface.

- `UMLStatDisplay`

  The purpose of this class is very similar to that of the `UMLRandomDisplay` class; it extracts the statistics variable names from the UML sequence diagram, which are indicated by the messages that contain the keyword *update*. The `UMLStatDisplay` class shows the list of statistics variable names in a pull-down list, its numerical type (either an `int`eger or a `double`), whether a simple increment/decrement or a calculation operation is involved, and an expression for the update operation.

- `UMLView`

  The four views of the *UML/SimML* tool are managed by this class, which inherits from the `JList` class and implements the `ListSelectionListener` interface. This interface detects if a particular view is selected and updates the main display area to show the appropriate display or panel. Table 5-2 shows the four *UML/SimML* views and the GUI classes that correspond to them.

- UMLToolBar

  Depending on the view selected, there are four kinds of toolbar that provide some clickable buttons to enable more operations to be performed:

Table 5-2: The four views of the UML/SimML tool

UML/SimML view	The corresponding GUI display class
UML Class Diagram view	`UMLDisplay`, containing `UMLObjectClass`
UML Sequence Diagram view	`UMLDisplay`, containing `UMLObjectSequence` and `UMLObjectConnector`
Random Variables view	`UMLRandomDisplay`
Statistics Variable view	`UMLStatDisplay`

– `UMLToolBarClass`

This toolbar is shown when the UML Class Diagram view is selected. It provides buttons to draw the basic class diagram shape, reset the display area (i.e. remove all the class diagrams shown) and to exit the tool.

– `UMLToolBarSequence`

When the Sequence Diagram view is chosen, the `UMLToolBarSequence` will be shown on the ToolBar strip of the *UML/SimML* tool. This toolbar is similar to the `UMLToolBarClass` above, but it has a button to draw the basic sequence diagram shape instead.

– `UMLToolBarRandom`

The Random Variables view is associated with the `UMLToolBarRandom`, which displays two buttons: one for editing the random properties of a selected random variable (the “Edit” button) and the other for quitting the tool. After the “Edit” button is pressed, the random properties fields become editable (apart from the random variable name, which is derived automatically from the sequence diagram). This allows the user to change the parameters of the random variable currently selected. The “Edit” button itself is then transformed into a “Done” button, which upon pressing commits the changes made to the random properties and transforms the button back into an “Edit” button.

– `UMLToolBarStat`

This toolbar has very much the same features as those of the `UMLToolBarRandom` class, bar its association with the Statistics Variables view.

- `UMLMenuBar`

The `UMLMenuBar` provides features related to "file operations" such as reading from or writing to file (in XML format, see Section 5.4.3 later), printing the class and sequence diagrams, as well as a feature for generating the simulation program.

These Java classes provide the implementation of the GUI features of the *UML/SimML* tool. Used in conjunction with the `ncl.SML.Components` package, these classes constitute the complete *UML/SimML* tool that allows an automatic generation of JavaSim simulation program from UML design. Figure 5-14 below shows how the this tool would look like.

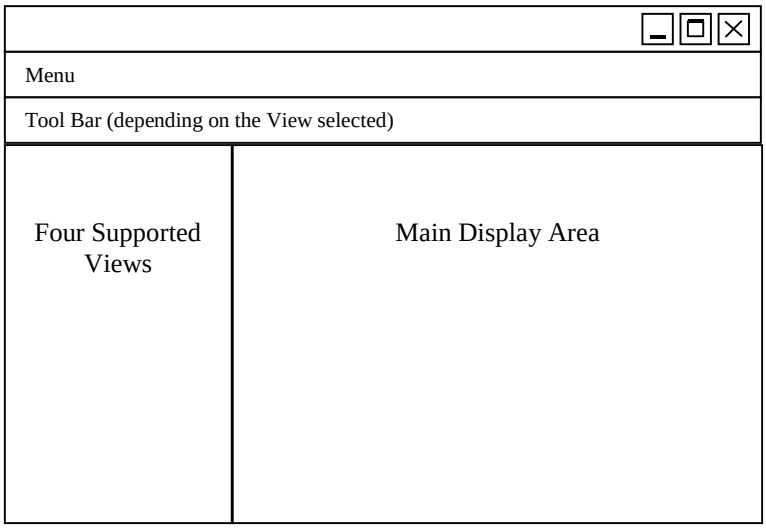


Figure 5-14: The proposed layout of the *UML/SimML* tool

The data captured from the UML diagrams needs to be stored into a physical storage or file so that they can be retrieved again later. As demonstrated in Section 5.4.2.2, this information can be kept in a structured way by using the SimML framework; therefore we need to find a proper way to store the SimML data into a

file. A widely used technique for filing a structured document is through the *Extensible Markup Language (XML)*. The following section discusses the applicability of XML for storing UML and SimML related information, which can then be used to supplement the *UML/SimML* tool.

5.4.3. Using XML for storing SimML data

The *Extensible Markup Language (XML)* is designed to make it easy to interchange structured documents over different application programs [20]. 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.

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 5-15. The complete SimML DTD is available at [4].

```
<!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 5-15: A snapshot of the SimML DTD

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 (‘`?`’).

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 5.4.2) to re-use its parsing features.

There is an *Application Programming Interface (API)* called *SAX (Simple API for XML*) [55], which is essentially another Java package that provides a skeleton for parsing any XML document. The parsing feature of this package is achieved through yet another package called XML4J [38]. 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 the 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.

A class called `XMLReader` has been added to the *UML/SimML* tool to support the feature for reading the UML and SimML data from an XML document/file. The `XMLReader` class does the following tasks:

- It reads the information related to the SimML DATA and PROCESS components and prepares the corresponding UML class diagrams to be displayed on the *UML/SimML* tool.
- For the SimML PROCESS components, the `XMLReader` also loads their SimML actions specification and convert them into messages in the UML sequence diagram. Together with the information from the SimML DATA, PROCESS and QUEUE components, the complete sequence diagram can then be constructed.
- It reads the SimML RANDOMS and STATISTICS properties and upload the data into the relevant views of the *UML/SimML* tool (i.e. the random and statistics variables view).

`XMLReader` is incorporated into the `UMLEditor` class and provides the feature of the *UML/SimML* tool to read the SimML specification from file, which also means to regenerate the UML class and sequence diagrams from the SimML XML notation.

The complement of the “read from an XML file” feature is the “write to an XML file” feature that allows the UML notation to be stored into an XML document. The “write to an XML file” operation is more straightforward than the “read from an

XML file" operation because it is very similar to the operation for generating the JavaSim program. There are two steps taken to achieve this:

- As mentioned earlier in Section 5.4.2.4, the `SMLStructure` class has a member function called `generateJavaSim()`, which transforms the information stored in that class into a JavaSim program. A new member function similar to the `generateJavaSim()`function can be created to perform the transformation from the same set of data into an XML representation. This member function is called `generateXML()`, and it is used in conjunction with the member functions of some SimML component classes mentioned below.
- The `SMLData`, `SMLProcess`, `SMLAction`, `SMLObject`, `SMLQueue`, `SMLRandom`, `SMLRandComp`, `SMLStatistics` and `SMLStatComp` classes are modified by augmenting them with a member function called `xmlRep()`, which translates the information stored in its respective class into a piece of XML representation. The complete XML representation is obtained by invoking the `generateXML()` function (of the `SMLStructure` class), which essentially executes all of the `xmlRep()` member functions of these relevant classes.

The "read" and "write" features allow the *UML/SimML* tool to retain the necessary information of a particular UML design into a stable storage, which means that the same specification can be retrieved and recreated easily from file.

5.5. Summary

This chapter discussed the implementation of two tools that can automatically transform UML design notation of some sort into simulation program. Both of these tools use the SimML framework as the cornerstone for writing their implementation;

this means that the SimML framework serves as a bridge that connect the UML notation and simulation program.

The first tool is called `runsim` and it was built using a scripting language called PERL. The `runsim` tool reads a textual UML notation (that follows the SimML syntax) from a file, generates the C++SIM source code files from the information read, compiles these files and runs the simulation. At this stage, this tool only allows a textual representation of the UML to be used and it serves as a feasibility study before we decided to build a graphical tool.

The second tool was implemented in Java with a main program called `UMLEditor`. This tool allows graphical UML notations to be incorporated in the forms of *class* and *sequence* diagrams, which represent the static and dynamic properties of simulation. These properties are then stored in a structured way within the SimML framework before being transformed into JavaSim simulation program. This tool is subsequently called the *UML/SimML* tool and it was implemented by several Java classes that support the *Graphical User Interface (GUI)* for drawing the UML diagrams as well as a Java package called `ncl.SML.Components` that supports the SimML framework.

The next chapter provides some case studies and results obtained from using the *UML/SimML* and `runsim` tools for designing systems and predicting their performance through simulation.

Chapter 6

Case Studies

6.1. Introduction

There are some case studies carried out in order to evaluate the usefulness of the *UML/SimML* tool (as well as the SimML framework in general) as an aid in predicting a system's performance through process-based, discrete-event simulation. The kind of simulation we are interested here involves *processes* and *queues*, in which the *performance time* is one of the most important aspect we would like to measure. A system is composed of several sub-systems which interact with each other and constitute the behaviour of the whole system. In the simulation, a sub-system is represented by a process which model the sub-system's execution time (delay) as well as its interactions with other sub-systems or processes. The queues are used to make sure that the interactions are performed in a correct manner/order. Jobs (or requests) arrive at the queue with a certain rate and the relevant process takes a job from the head of the queue to be executed. It is the execution time of the process that is normally of interest, together with the queueing mechanism used.

Three case studies are presented here. One is on a set of simple systems where their performance can be easily predicted even without a simulation. The aim of this exercise is to validate that the simulation results are close to the expected values. The other two case studies represent more complex systems and their purpose is to prove that the *UML/SimML* tool and the SimML framework can be used in helping system

designers to predict whether a particular system would deliver the required performance or not.

Each case study is discussed in a separate section, and each section follows the same structure: it begins with a description of the system being studied, followed by the UML/SimML specification of the system, the simulation of that system (or a particular part of the system that is of the greatest interest), and completed by a short comment or analysis of that case study.

The first case study represents simple queuing systems where a determined number of servers process the jobs delivered by a client (or several clients) [7, 10]. The second case study focuses on the `makeCall` operation of the British Telecom's Intelligent Network (IN) application [5, 8, 69], while the last case study investigates the use of the *SimML* framework in predicting the performance of a fault tolerant system called Voltan [19, 39, 70].

As mentioned in Chapter 3, we are interested in *discrete-event, process-based* simulations, and the examples shown here represent this kind of approach.

6.2. Simple Queueing Systems

Queueing system is a classic topic in simulation studies, therefore it is appropriate to start the case studies with some examples of this kind of system. Only simple queuing systems are considered here, the purpose of this approach is to allow an easy and straightforward validation effort on the simulation results: we can logically predict the performance of the system based on the time delays at the server(s) and the arrival rate of the jobs.

The performance of this kind of system depends on how quick the server can process the jobs, but we must bear in mind that a faster server will cost more than a

slower one. The server takes a certain time or delay to complete each job, and by altering the length of this delay (to simulate different servers), 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.

6.2.1. Description of the Queueing Systems

There are three examples considered here. The first one is a simple queueing system with one arrival process and one server process. The arrival process generates jobs at a certain rate and places them into a queue. The server process takes one job at the time from (the head of) the queue, performs a time delay to simulate the execution of the job and updates the statistics concerning the overall performance of the system. By keeping track on 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 queueing system can be represented as a simulation diagram as seen in Figure 6-1.

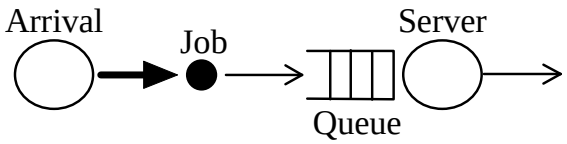


Figure 6-1: A system with one Arrival and one Server

The second queueing system builds on the first queueing system by considering multiple Server processes to handle the requests. The rationale used here is that a better performance can be obtained by adding more (slow but cheap) servers instead of replacing it with a faster (yet more expensive) one. Figure 6-2 gives a diagrammatical illustration of the second queueing system.

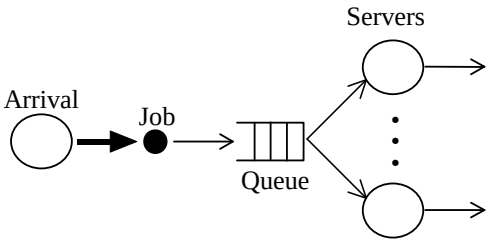


Figure 6-2: A system with multiple Servers

The third queueing system builds on the second queuing system further by having multiple Arrival process as well (Figure 6-3). The aim is to evaluate until what point a fixed number of servers can handle the stream of jobs from an increased number of arrival processes with a reasonable performance time.

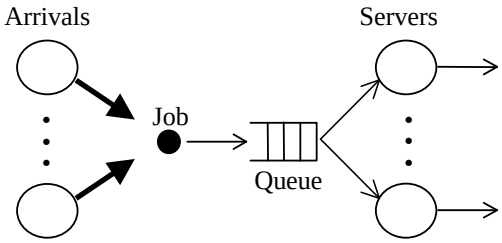


Figure 6-3: A system with multiple Arrivals and Servers

In all cases, we are interested to know how a system fares under different scenarios. This information can then help in making the decision on how to compose the system (i.e. which scenario to pick) based on the tradeoff between performance requirements and available resources. In order to obtain this information, the three system scenarios above are specified using the *UML/SimML* tool, which is then able to generate simulation programs for those systems automatically.

6.2.2. Queueing Systems Specification using UML/SimML

The static and dynamic characteristics of the three queueing systems outlined in Section 6.2.1 (see Figure 6-1, Figure 6-2 and Figure 6-3) can be represented as UML *class* and *sequence* diagrams using the *UML/SimML* tool. This tool also provides two extra views for specifying the *random* and *statistics variables*.

The *UML/SimML* notation for these three queueing systems are given below. Each system is covered separately, but since they are similar, only the incremental changes are shown for the second and the third systems. The class diagram for each system differs slightly, but the sequence diagram and the statistics variable view will

be the same for all three. Only the first system will have a different random variables view among the three.

One Arrival, One Server queueing system

This system represents the basic configuration, hence all of the four views of the *UML/SimML* tool are given. Figure 6-4 shows the class diagram, which depicts the entities of the system, namely the `Job` data structure, the `Arrival` process and the `Server` process. We would like to know how long each `Job` spends in the system, so we must store the information on its arrival time (as a member variable of the `Job` class, in this case it is called `arrTime`).

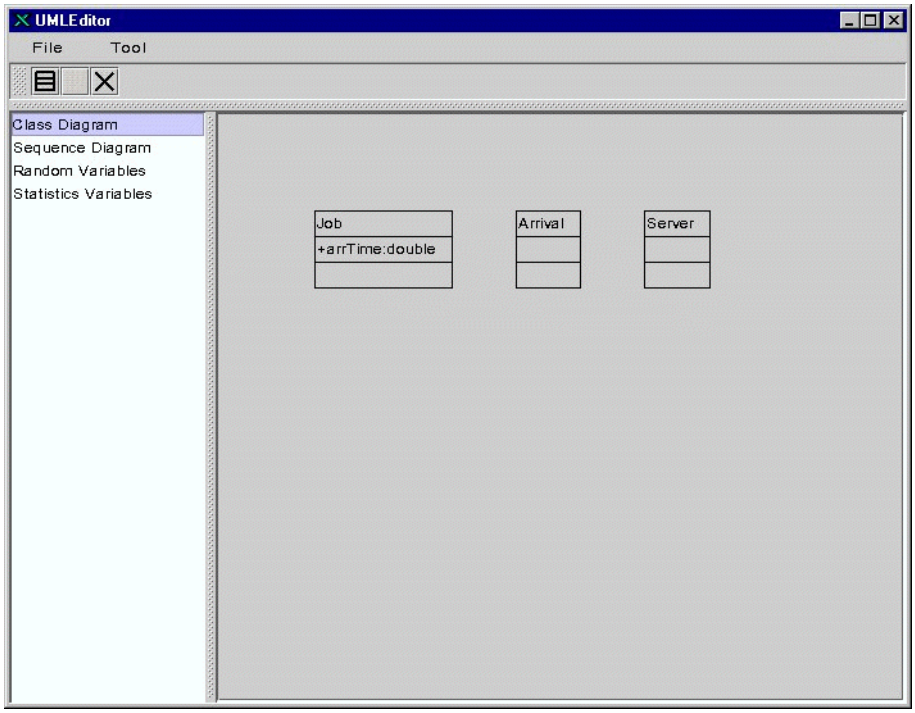


Figure 6-4: The Class Diagram for "One Arrival, One Server" system

The sequence diagram (Figure 6-5) shows the interaction of the system's objects (instances of the classes described in the class diagram). It also shows the `Queue` object, which is used to marshal the instances of the `Job` class before they are being processed by the `Server`. The messages shown here represent the SimML actions that are required for generating the simulation program.

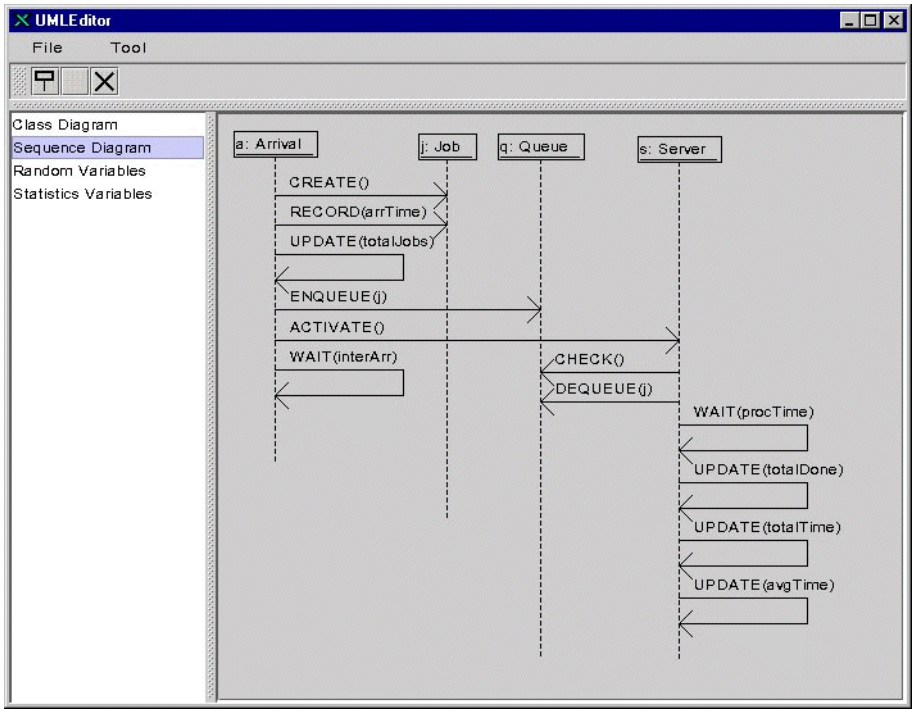


Figure 6-5: The Sequence Diagram for "One Arrival, One Server" system

The random variables view (Figure 6-6) allows the random variables for the simulation to be specified. In the case of this simple queueing system, these are the inter arrival time of the `Job`s (`interArr`) and the processing time (i.e. delay) of the `Server` (`procTime`), and both are exponentially distributed with mean 5.

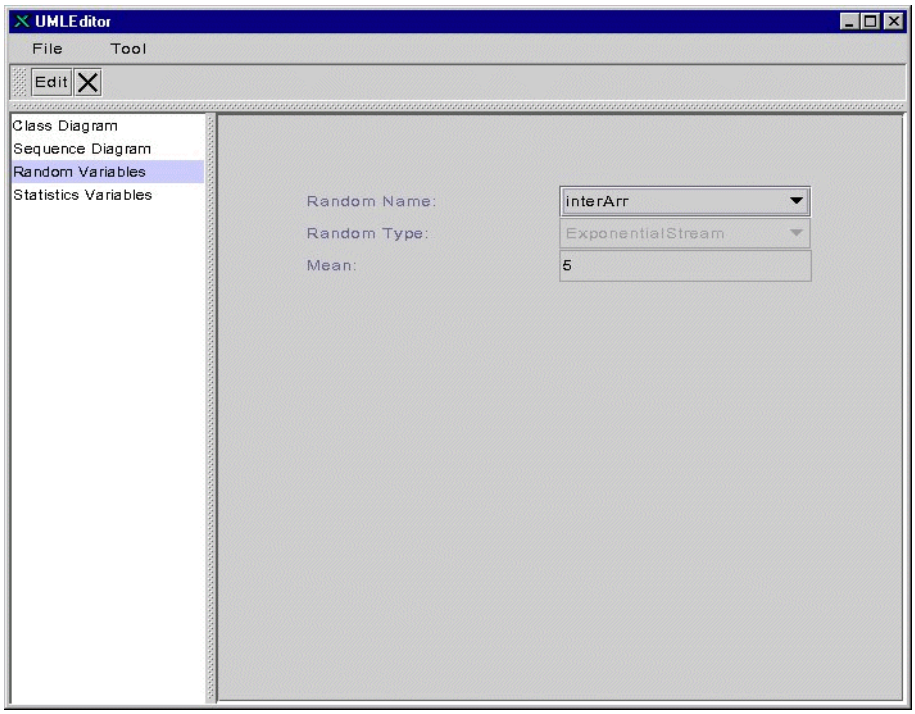


Figure 6-6: The Random Variables View for "One Arrival, One Server" system

The last view shows the statistics variables that we would like to collect (Figure 6-7). These variables are updated during the simulation run, and the updates include incremental update (e.g. for `totalJobs`) as well as calculation (e.g. for `avgTime`).

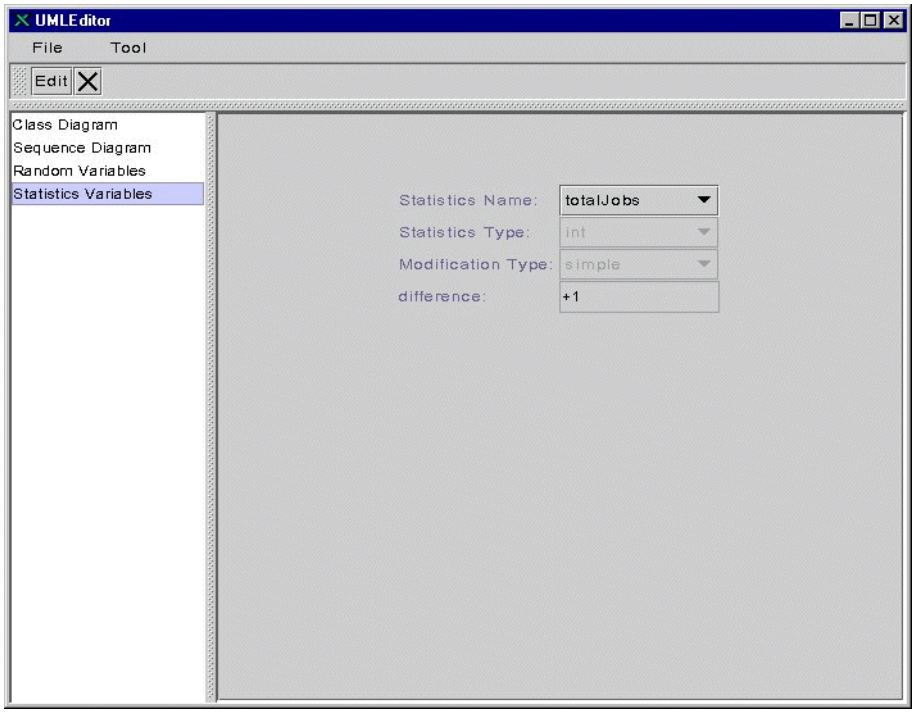


Figure 6-7: The Statistics Variables View for "One Arrival, One Server" system

These four screen dumps are some of the views available from the *UML/SimML* tool for representing the first simple queueing system. For the other two queueing systems, the views are very similar, so only the differences will be shown.

One Arrival, Multiple Servers queueing system

The multiplicity of the `Server` processes is denoted by a *tagged value* in the class diagram. This tagged value is placed just under the class name in the format of `{mult=X}`, where `X` is an integer value indicating how many instances of this class should be created (Figure 6-8).

Changing the multiplicity of the `Server` is therefore very straightforward: one just needs to replace the value of `X` by an integer literal; the SimML framework will take care of the rest (for generating the appropriate simulation program later).

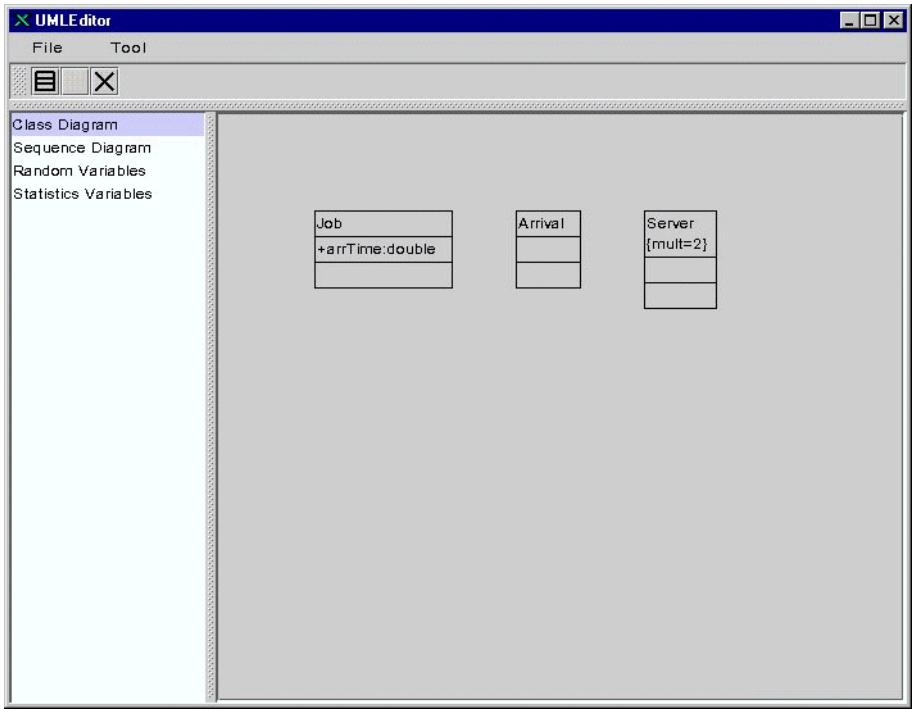


Figure 6-8: The Class Diagram for "One Arrival, Multiple Servers" system

Another change made here concerns the parameter of one of the two random variables. The `procTime` variable now has a mean of 12, to represent a slower `Server` used in this system. The rest is the same as the "One Arrival, One Server" system scenario.

Multiple Arrivals, Multiple Servers queueing system

To obtain multiple `Arrival` processes, the only thing to do is to supply the `{mult=X}` tagged value in the appropriate class diagram. The rest are the specification is the same as the "One Arrival, Multiple Servers" one, so we do not need to elaborate it further here.

6.2.3. Simulation of the Queueing Systems

From each of the specifications outlined in Section 6.2.2, a simulation program is generated using the "Generate JavaSim" feature of the *UML/SimML* tool. Each simulation is then executed with a long simulation time (say, 100,000 units) in order to generate sufficient jobs for bona fide statistical data. The parameters of interest are

the *inter arrival time* of the jobs (`interArr`), the *processing time* of the Server process (`procTime`), as well as the multiplicity of each process. These parameters are random variables that follow an exponentially distributed function.

One Arrival, One Server queueing system

The parameters represent the mean of the exponentially distributed random variables. For the first queueing system, they are as follows:

```
interArr = 5
procTime = 5
```

Since `interArr` and `procTime` are the same (5 units), we expect that the system will be saturated, since the offered load (ρ = `procTime` / `interArr`) is equal to 1. The results obtained from the simulation are as follows:

```
$ java Main 100000
totalJobs: 19991
totalDone: 19968
totalTime: 5839619.843022785
avgTime: 292.44891040779174
```

Running the simulation for a longer time (1,000,000 units) highlights the build up of the jobs waiting in the queue as the system becomes more saturated. As a consequence, the average processing time gets worse and there is a higher number of unfinished jobs, since the system cannot cope with the load.

```
$ java Main 1000000
totalJobs: 200411
totalDone: 200332
totalTime: 8.707705766162425E7
avgTime: 434.6637464889496
```

We can see that the simulation runs produce results as expected.

This is an example of the **M/M/1** queueing system. For this kind of system, the performance of interest (e.g. the average response time) can be calculated easily using some equations:

A = inter arrival time (`interArr`)

B = server's processing time (`procTime`)

ρ = offered load = B / A

W = average response time = B / (1 - ρ)

More simulation runs were conducted by giving different parameter values to `interArr` and `procTime`. The results were then compared to the expected results, which were calculated using the equations above.

- `interArr = 6`

  `procTime = 5`

```
$ java Main 100000
totalJobs: 16667
totalDone: 16663
totalTime: 468685.94052131195
avgTime: 28.12734444705707
$ java Main 1000000
totalJobs: 166972
totalDone: 166962
totalTime: 4946078.361097511
avgTime: 29.62397648026204
```

Expected result:

```
ρ = procTime / interArr
  = 5/6

W = procTime / (1-ρ)
  = 5 / (1 - 5/6)
  = 30
```

- `interArr = 5`

  `procTime = 4`

```
$ java Main 100000
totalJobs: 19991
totalDone: 19988
totalTime: 373642.7350252167
avgTime: 18.693352762918586
```

Expected result:

```
ρ = procTime / interArr
  = 4/5

W = procTime / (1-ρ)
  = 4 / (1 - 4/5)
  = 20
```

```
$ java Main 1000000
totalJobs: 200411
totalDone: 200397
totalTime: 3972644.332626486
avgTime: 19.823871278644322
```

These simulation results conform to the calculated results, therefore we can be confident that the simulation produces the right results.

One Arrival, Multiple Servers queueing system

Here we replace the server with a slower one but keep the inter arrival time to the same value. So, the parameters are:

```
interArr = 5
procTime = 12
```

In order to be able to keep up with the load, there is a condition that must be satisfied: $\rho < n$, where n is the number of servers. For the parameters above, we obtain:

$\rho = 12/5 = 2.4$

We therefore could estimate beforehand that we would need at least three servers if we want to get an acceptable performance.

Calculating the performance of this kind of system (**M/M/n** system) is more difficult, because we need to use some complex equations:

λ = arrival rate = 1/A μ = processing rate = 1/B

$$P_0 = \left[\sum_{j=0}^{n-1} \frac{\rho^j}{j!} + \frac{\rho^n}{(n-1)!\,(n-\rho)} \right]^{-1}$$ (Probability of the server being idle)

$$L = \frac{\lambda}{\mu} + \left[\frac{(\lambda/\mu)^n \lambda\mu}{(n-1)!\,(n\mu-\lambda)^2} \right] P_0$$ (Average number of jobs in the system)

$$W = \frac{L}{\lambda}$$ (Average response time)

For example, if the inter arrival time is 5 (i.e. $\lambda = 1/5$) and the server's processing time is 12 (i.e. $\mu = 1/12$), the following values are obtained (sparing the calculation details):

$\rho = 2.4$

$P_0 = 0.05618$

$L = 4.98877$

$W = 24.94$

Using the same parameters for the inter arrival time and processing time, several simulation runs were performed (each with a simulation length of 100,000), with an increasing number of servers. In all cases, the number of jobs generated will be 19,991. The results of the simulations can be seen Table 6-1 below.

Table 6-1: Simulation results of One Arrival, Multiple Servers systems

Number of Servers	totalDone	totalTime	avgTime
1	8326	2.385E8	28646.28
2	16715	1.378E8	8181.79
3	19987	469270.50	23.48
4	19987	279053.62	13.96
5	19987	248312.51	12.42

It can be seen that when there is only one (slow) server used, the jobs overflow the system, hence the response time is very poor: the jobs spend most of their time waiting in the queue before being processed. Adding more servers alleviates the problem, and a reasonable performance can be obtained by using three servers.

It should also be noted that when there are three servers used, the average performance time found from the simulation (`23.48`) is pretty close to the calculated value (`24.94`). Simulating the same configuration with a longer simulation time (1,000,000 units) produces an even closer result:

```
$ java Main 1000000
totalJobs: 200411
```

```
totalDone: 200397
totalTime: 4953181.88039389
avgTime: 24.71684646174289
```

This indicates that the simulation method can be used to predict the performance, thus saving the effort of performing the complex calculations.

Multiple Arrivals, Multiple Servers queueing system

We use the same parameters as in the "One Arrival, Multiple Servers" system and we fix the number of servers to five. We then add the number of arrival processes one at the time until the system cannot cope with the load anymore.

Table 6-2: Simulation results of Multiple Arrivals, Multiple Servers systems

Number of Arrivals	totalJobs	totalDone	totalTime	avgTime
1	19979	19976	240153.74	12.02
2	40232	40182	2242744.50	55.81
3	60097	41918	6.38E8	15225.39

Table 6-2 shows that as the number of arrival processes is increased, the total jobs generated also increases (in the same ratio). In this case, the system can only cope with two arrival processes, adding more arrival processes will overflow the system and degrade the performance considerably.

6.2.4. Summary of the Queueing Systems case study

This case study shows that the *UML/SimML* tool can be used for generating simulation programs for simple queueing system specifications. Simulation parameters can be changed easily to enable a quick and convenient way of exploring different scenarios. It also shows that the results obtained from the automatically generated simulation program are logically correct and the tool can be used for the specification of more complex systems, including multiple queues and pipelining.

6.3. BT Intelligent Network (IN) Application

The *British Telecom* (BT) *Intelligent Network* (IN) application aims to make the telephone system smarter by allowing it to be expanded with extra features such as call barring, call forwarding, etc.

6.3.1. Description of the BT IN Application

New features for call handling (e.g. credit call charging, call barring, etc.) in intelligent networks (INs) are typically delegated to computer systems that are attached to switches [5]. A switch passes the incoming call request that needs special processing to the local computer system to be dealt with properly before the connection is set up. These computer systems maintain data pertaining to the customers. The processing must be done fairly quickly (less than a second for most of the calls), otherwise the customer might not be happy. Two operations provided for each customer are `makeCall` and `receiveCall`; the former maintains information relating to outgoing calls (e.g. should the call be barred) and the latter maintains information concerning incoming calls (e.g. does the receiver wish to receive calls from the caller).

Physical Architecture

The basic system structure is shown in Figure 6-9. Processing at a switch site is performed by a high-performance computing cluster, comprising of about ten hosts. There maybe between 60 and 1,000 sites distributed throughout the world. Communication within a site would typically be carried out via a Local Area Network (LAN) with ~100Mbps bandwidth and a latency of ~1 millisecond. Sites would be connected by Wide Area Networks (WANs) with bandwidth of ~34Mbps and a latency of ~50 milliseconds. The total number of customer objects (i.e. telephone

callers) in the system is estimated to be in the range 10^5-10^8 with an approximately equal allocation of objects to hosts.

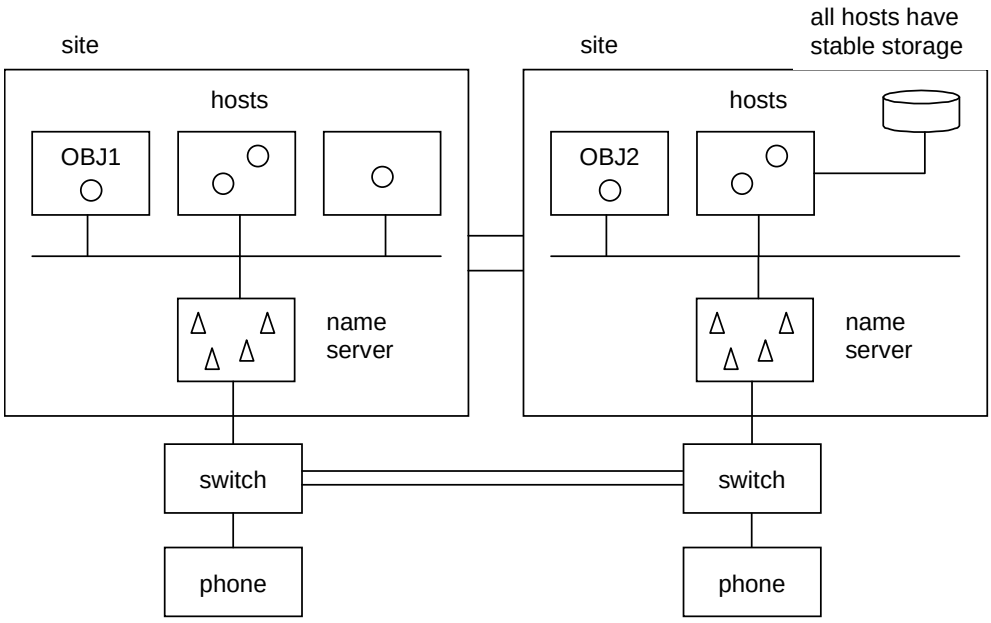


Figure 6-9: The Architecture of the BT IN Application

Processing Requirements

Processing is initiated via messages from the physical switch which contain two parameters - the *calling line identity (CLI)* and the *dialled number (DN)*. Between 3,000 and 3,000,000 messages per second are expected. Each message is first handled by a name location mechanism which assigns object uids to the CLI (`OBJ1`) and DN (`OBJ2`). This may be achieved by, for example, a dedicated name server. The object uids are unique identifiers which contain the address and host number of the appropriate caller (`OBJ1`) and callee (`OBJ2`) objects. The `makeCall` method of `OBJ1` is then invoked, passing `OBJ2` as a parameter. This typically takes place on a different host on the same site (though it may be in the same process, or a different process on the same host or, exceptionally, on a different site). Marshalling routines pack and unpack inter and intra site messages. The `makeCall` method of `OBJ1` checks to see that the `barOutgoing` flag is not set and it then makes an RPC

(*Remove Procedure Call*) to the `receiveCall` method of `OBJ2`. The `receiveCall` method checks whether `OBJ1` is in the `blacklist` of `OBJ2` and sends back a `startRinging` reply if the call is to be accepted. The performance requirements imposed on the `makeCall` method are that it must service 90% of calls in at most 500 milliseconds, 95% of calls in at most 5,000 milliseconds and 100% of calls in at most 10,000 milliseconds.

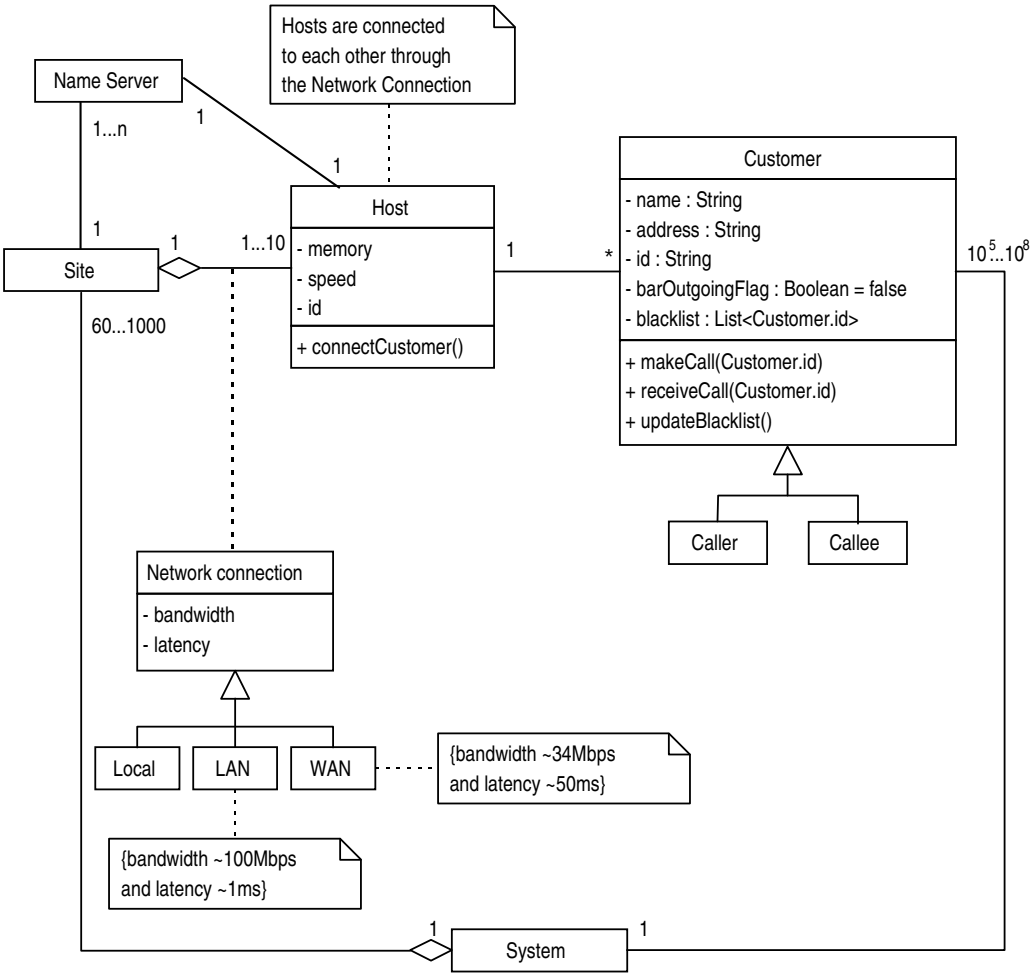


Figure 6-10: The Class Diagram of BT IN Application

6.3.2. BT specification using UML/SimML

The elements of the architecture and the requirements above can be translated into several UML diagrams:

- *The physical architecture*

The main details of the underlying hardware (the machines and the connection between them) as well as the application objects can be represented as a *class diagram* as shown in Figure 6-10.

- *Application logic*

  Customers can only make a call if they are not barred from doing so by the telephone company (e.g. due to unpaid bills). Another requirement is that the caller is not on the blacklist of the called party. The `makeCall` operation tries to connect two customers through a series of operations that involves several checks to ensure that both requirements are satisfied. As a customer can have more than one phone number (in different locations), it is necessary that `makeCall` accesses the name server to find the latest binding indicating where the called party can be reached.

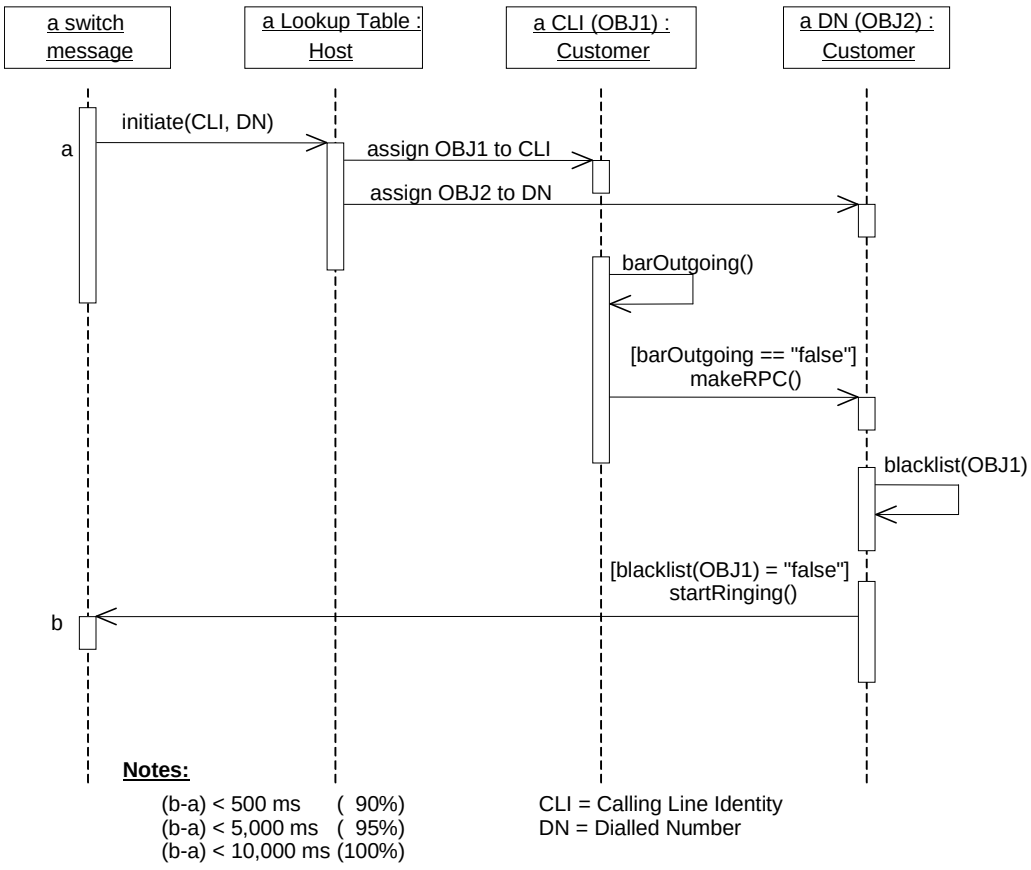


Figure 6-11: The Sequence Diagram of BT IN Application

Figure 6-11 illustrates the *sequence diagram* which represents the `makeCall` operation. This diagram also indicates the time constraints of the `makeCall` operation. It should be noted that the standard UML notation cannot describe the probabilistic constraints except in comments or notes.

In addition to the two diagrams above, an *activity diagram* can also be used to elaborate the `makeCall` operation by showing a possible scenario when a customer tries to call another customer using this system (Figure 6-12).

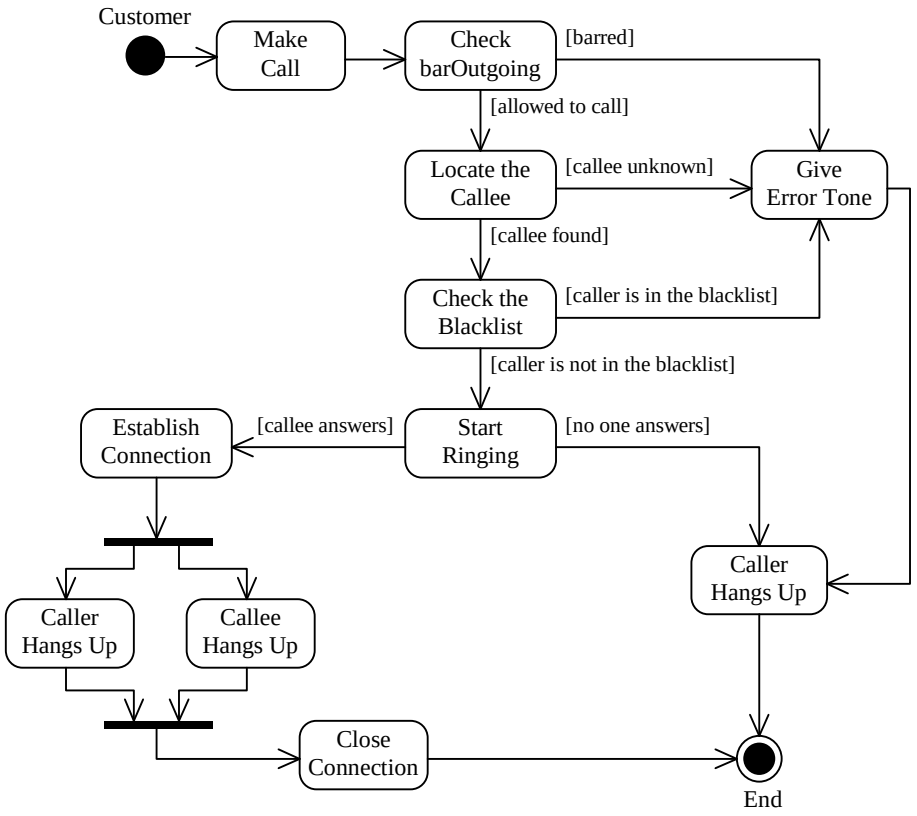


Figure 6-12: An Activity Diagram for `makeCall`

It is the performance of the `makeCall` operation that we are interested in investigating. The *UML/SimML* tool can be used to predict the performance by generating a simulation program for the `makeCall` operation. We first need to obtain more detailed information on which processes (entities) are involved and what actions do actually happen within each process when a `makeCall` operation is invoked.

There are three processes needed to simulate the `makeCall` operation: the `Arrival`, `Call`, and `Lookup` processes. They are represented as classes in the *UML/SimML* Class Diagram view (Figure 6-13). The `Arrival` process simulates the initiation of a phone call by generating a `Call` object. Since the location of the `Call` object within the phone network can be *local*, *LAN* or *WAN*, it is necessary to assign appropriate properties to the call, such as its network connection latency.

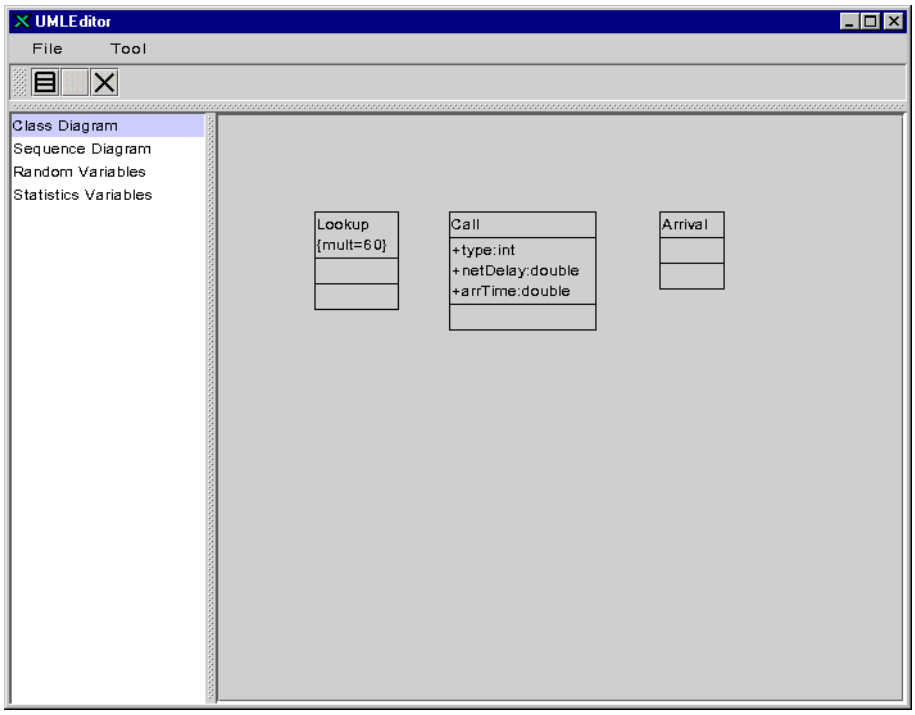


Figure 6-13: The Class Diagram for the `makeCall` operation

The `Call` process represents a phone call request. When such a request is initiated, it is necessary to perform several operations (cf. Figure 6-11):

- getting caller's and callee's identities from the Name Server (a *lookup* operation),
- checking the `barOutgoing` flag of the caller (a *read* operation),
- checking the `blacklist` of the callee (a *search* operation),

Each operation takes a certain time to complete and any operation sent across the network must also take into consideration the delay imposed by the network latency based on the locality of the call.

The `Lookup` process represents the Name Server and it keeps a queue for handling the call initiation requests in order. Since there are many sites (between 60 and 1,000 according to the *Physical Architecture* specification, see Section 6.3.1 and Figure 6-10) and each site has a Name Server, this means that there are multiple instances of the `Lookup` process as well. Here we took 60 as the multiplicity of the `Lookup` process.

The interaction between the `Arrival`, `Call` and `Lookup` processes can be seen in the Sequence Diagram view of the *UML/SimML* tool (Figure 6-14).

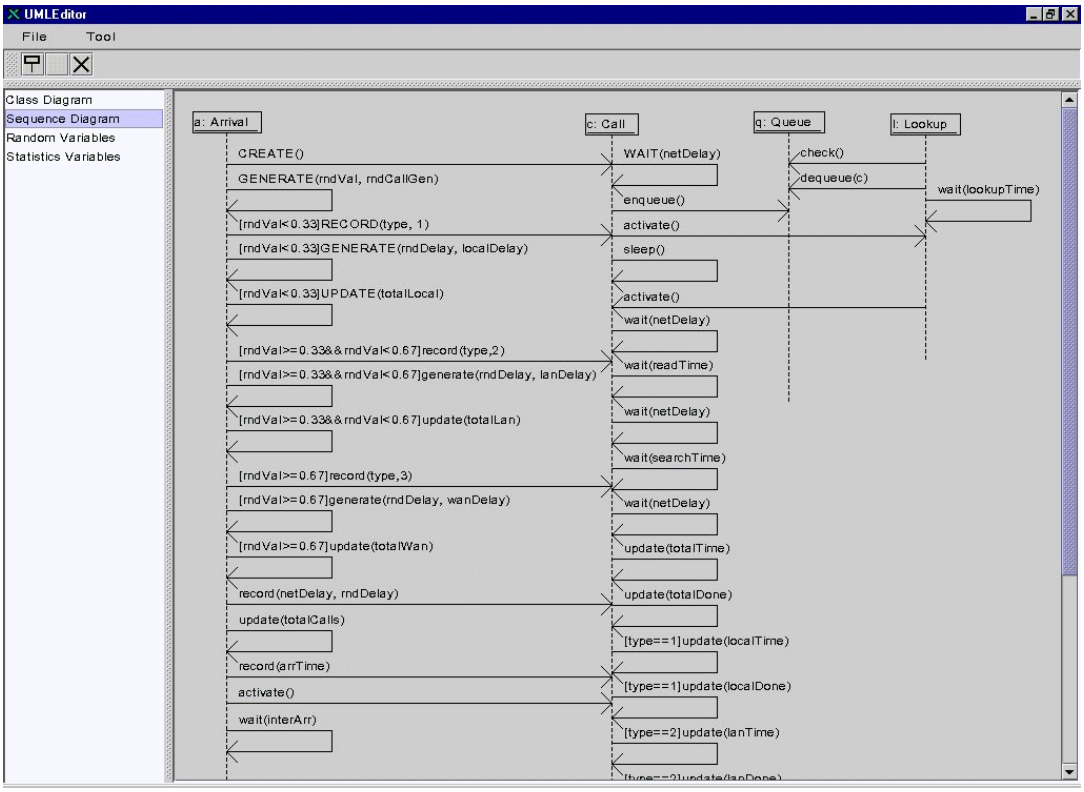


Figure 6-14: The Sequence Diagram for the `makeCall` operation

The Random and Statistics Variables views of the *UML/SimML* tool are not shown here, but these variables are outlined in the simulation sub-section that follows.

6.3.3. Simulation of the *makeCall* operation

The specification of the `makeCall` operation using the *UML/SimML* tool is then converted into a JavaSim simulation program. There are eight random number

variables, which are shown in Table 6-3 below, including their types and some explanation on what they are used for.

Table 6-3: The Random Variables used in the makeCall simulation

Name	Type	Explanation
interArr	Exponential	The inter arrival time of the calls
lookupTime	Exponential	The time taken by the Name Server to lookup for call identities
readTime	Exponential	The time taken to perform the `barOutgoing` flag evaluation
searchTime	Exponential	The time taken to check the `blacklist`
localDelay	Exponential	The network latency for local call objects
lanDelay	Exponential	The network latency for LAN call objects
wanDelay	Exponential	The network latency for WAN call objects
rndCallGen	Uniform	Used for randomly generating the local/LAN/WAN call types

The statistics to collect can be divided into four categories: those for the local, LAN and WAN calls, as well as the aggregate of them all. For each category, the information gathered concerns the number of generated calls, the number of completed calls and the total time of processing the calls. The average processing times of those four categories are also calculated. All the units used here is in *millisecond* (ms).

Obtaining simulation parameters

Before the simulation can be performed, it is necessary to obtain the parameters for the random variables. The *Processing Requirements* specification (Section 6.3.1) mentions that there are between 3,000 and 3,000,000 calls made per second. For simplicity, we took the lower boundary value of 3,000 calls per second, i.e. the inter arrival rate is 0.33 ms. The local, LAN and WAN delays are specified to be zero, 1 ms and 50 ms respectively (Figure 6-10).

The parameters for the *read* and *search* operations were obtained from experiments where random access and binary search for a record (among 1,000,000

records) are executed. These experiments were conducted on a Pentium II/133 Mhz running Linux with a low load. From the results, we set the `readTime` to be 0.00023 ms and the searchTime to be 0.0057 ms.

We would like to investigate what parameter is needed for the Name Server's *lookup* operation in order to fulfill the requirements where most (90%) of the calls must be processed within 500 ms. (Here, we would base the comparison with the average processing time of the WAN calls, which have the worst performance).

Simulation Results

Several simulations were performed, each for a simulation length of 100,000 ms. The parameter for the `lookupTime` is modified until an acceptable performance is obtained. An example of a report generated from one simulation run (with `lookupTime` parameter of 20 ms) is shown below:

```
totalLocal: 100070
localDone: 99043
localTime: 5.1879070461091794E7
avgLocalTime: 523.8035041455912
totalLan: 102853
lanDone: 101801
lanTime: 5.354958759335911E7
avgLanTime: 526.0222158265549
totalWan: 99926
wanDone: 98759
wanTime: 7.109652840283486E7
avgWanTime: 719.8992335162857
totalCalls: 302849
totalDone: 299603
```

```
totalTime: 1.7652518645728326E8
avgTime: 589.1969922106363
```

We can see that if the `lookupTime` has a mean of 20 ms, the performance of the `makeCall` operation is worse than what the requirement asks. Changing the `lookupTime` parameter to 19 ms gave the following results:

```
totalLocal: 100070
localDone: 100050
localTime: 2041272.6611432696
avgLocalTime: 20.402525348758317
totalLan: 102853
lanDone: 102831
lanTime: 2506776.6377693936
avgLanTime: 24.37763551622948
totalWan: 99926
wanDone: 99725
wanTime: 2.1875343049229436E7
avgWanTime: 219.35666131089934
totalCalls: 302849
totalDone: 302606
totalTime: 2.6423392348142095E7
avgTime: 87.31945945599921
```

Here, a drastic performance improvement is obtained. To find the minimum parameter value for `lookupTime` that satisfies the requirement, a small decrement of 0.01 ms was applied to the parameter value, starting at 20 ms. The results are shown in Table 6-4. Only the information of main interest are shown here, i.e. the average times for each call-type and the summary of all calls. In all cases, there are 302,849 calls made in total for each simulation run.

Table 6-4: Simulation results of the makeCall operation

lookup	average times			all calls		
Time	local	LAN	WAN	done	time	avgTime
20.00	523.80	526.02	719.90	299603	1.77E8	589.20
19.99	499.13	501.38	695.36	299748	1.69E8	564.58
19.98	474.39	476.76	670.77	299889	1.62E8	539.93
19.97	449.67	452.12	646.22	300053	1.55E8	515.29
19.96	424.88	427.49	621.68	300223	1.47E8	490.65
19.95	400.05	402.79	597.00	300353	1.40E8	465.91
19.94	375.25	378.10	572.33	300511	1.33E8	441.19
19.93	350.44	353.33	547.61	300650	1.25E8	416.42
19.92	325.60	328.58	522.87	300804	1.18E8	391.64
19.91	300.71	303.76	498.13	300945	1.10E8	366.83
19.90	275.83	278.94	473.33	301091	1.03E8	341.99

It was found that in order to satisfy the performance requirement (i.e. the WAN calls should be processed within 500 ms on average), the upper limit of the `lookupTime` is 19.91 ms.

6.3.4. Summary of BT case study

The BT case study demonstrates how the *UML/SimML* tool could be used in specifying (a part of) a complex system. The interest lies on the `makeCall` operation, which is invoked when a phone call is made.

The performance of the `makeCall` operation is influenced by many factors, such as the location of the caller/callee (in relation to the Name Server), the number of Name Servers used, the number of calls initiated per second, etc., but the most important factor is the time delay that the Name Server takes to process or initiate each call (the `lookupTime` delay).

The simulation program (generated automatically from the specification) enables us to estimate what parameter is needed for the Name Server in order to satisfy the requirements of the system. This is very helpful, for example in deciding

how much resources should be spent on the Name Server by evaluating the possible scenarios.

6.4. Voltan

This project addresses the problem of preventing faulty hardware processors from causing application programs to fail by developing a family of “fail-controlled” nodes, the *Voltan nodes*. These nodes are *fail-silent* nodes, which means that they either function correctly or stop functioning after an internal failure is detected.

The construction of fail-silent nodes can be achieved using one of the two approaches available: *hardware-implemented* or *software-implemented*. A hardware-implemented fail-silent node requires special-purpose hardware components, such as fault-tolerant clocks, comparators and bus interface circuits. On the other hand, a software-implemented fail-silent node can be constructed simply by using standard “off-the-shelf” components. There are many advantages offered by the software-implemented nodes, such as easy upgradeability and more robustness against transient failures; these can be seen in [70] and [19].

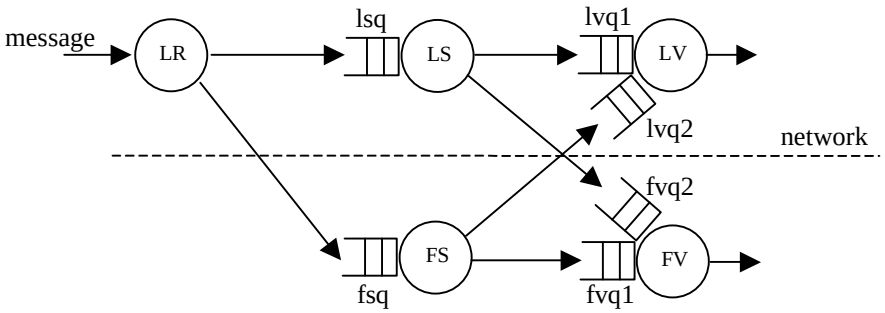
The Voltan project is an example of how fail-silent nodes can be constructed using the software-implemented approach. In the software-implemented fail-silent nodes, the non-faulty processors of the node need to execute a message order protocol (for keeping in step) as well as a comparison protocol (for checking each other). More details on the Voltan system are given below.

6.4.1. Description of the Voltan system

A *Voltan node* is composed of a number of conventional (“off-the-shelf”) processors on which application level processes are replicated to achieve fault tolerance [19, 70].

The processors in Voltan nodes are connected via communication links (network and local connections). Within a node, the processors execute message agreement and ordering protocols to guarantee that correct replicas of application processes will receive and process input messages in identical order.

A basic Voltan node can be constructed of two processors: the LEADER and the FOLLOWER. The LEADER determines the order of processing messages by choosing a message and sending a copy of that message to the FOLLOWER. The LEADER keeps a counter, which is used to assign a unique identifier to input messages. Both the LEADER and FOLLOWER then process the message and cross-send the results to their respective voters in order to validate/compare the results. A diagrammatical representation of a Voltan node can be seen as Figure 6-15.




```
LR = Leader Receiver        lsq  = Leader Server Queue
LS = Leader Server          lvq1 = Leader Voter Queue from LS
LV = Leader Voter           lvq2 = Leader Voter Queue from FS
FS = Follower Server        fsq  = Follower Server Queue
FV = Follower Voter         fvq1 = Follower Voter Queue from FS
                            fvq2 = Follower Voter Queue from LS
```

Note: `Server` and `Voter` are actually threads within the same processor.

Figure 6-15: A simplified architecture of a single Voltan node

There is a network delay associated with the task of sending a message between the LEADER and the FOLLOWER. There is also a delay for processing a message, and within each process, there is a (very small) local delay between different actions (e.g. from processing to comparing).

We would like to investigate the performance of a Voltan node, more specifically on the comparison between the performance times of the LEADER and the FOLLOWER processes. In the following sub-section, a specification of the Voltan system is given in a textual SimML framework. This specification is then converted using `runsim` (which is one of the two SimML parser implementation outlined in Section 5.3.2) into C++SIM simulation program.

6.4.2. Voltan system specification using the SimML framework

An older SimML parser implemented in Perl (`runsim`) is used here because Voltan system specification requires a priority ordering on its queues, and this is not yet supported in the *UML/SimML* tool. Here each message is given an identification number (stored as a member variable called `id` in the `Message` class) and they are queued according to that number. This means that messages with smaller `id` numbers will be placed closer to the head of the queue.

The older SimML framework syntax (which is parseable using `runsim`) supports a way to achieve this requirement. It has an extra SimML component called SEQUENCE, which is a specialised form of QUEUE component that takes into account the `id` variable of the queued object.

The complete SimML notation for the Voltan system can be seen in Figure 6-16. As we can see, it is a fully textual notation, but `runsim` is able to automatically transform this notation into C++SIM program for performance evaluation.

```
SEQUENCE Queue of Message using id

PROCESS Message once
{
  +int id
  +int type
  +double arrTime
  +void Body()
  {
    if type == 1
    [
      wait localDelay
      enqueue this to lsq
      activate ls
      sleep
    ]
    elsif type == 2
    [
      wait netDelay
      enqueue this to fsq
      activate fs
      sleep
    ]
    if type == 11
    [
      wait localDelay
      enqueue this to lvq1
      activate lv
    ]
    elsif type == 12
    [
      wait netDelay
      enqueue this to fvq2
      activate fv
    ]
    elsif type == 21
    [
      wait netDelay
      enqueue this to lvq2
      activate lv
    ]
    elsif type == 22
    [
      wait localDelay
      enqueue this to fvq1
      activate fv
    ]
    end
  }
}

PROCESS Arrival
{
  +void Body()
  {
    create msg of Message
    create msg2 of Message
    update totalMessages
    record id of msg = totalMessages
    record id of msg2 = totalMessages
    record arrTime of msg
    record arrTime of msg2
    record type of msg = 1
    record type of msg2 = 2
    activate msg
    activate msg2
    wait interArrivalTime
  }
}

PROCESS LeaderServer
{
  +void Body()
  {
    while lsq->IsEmpty()
    [
      sleep
    ]
    dequeue msg from lsq
    wait procDelay
    create msg2 of Message
    record id of msg2 = msg->id
    record arrTime of msg2 = msg->arrTime
    record type of msg = 11
    record type of msg2 = 12
    activate msg
    activate msg2
  }
}

PROCESS FollowerServer
{
  +void Body()
  {
    while fsq->IsEmpty()
    [
      sleep
    ]
    dequeue msg from fsq
    wait procDelay
    create msg2 of Message
    record id of msg2 = msg->id
    record arrTime of msg2 = msg->arrTime
    record type of msg = 21
    record type of msg2 = 22
    activate msg
    activate msg2
  }
}

PROCESS LeaderVoter
{
  +void Body()
  {
    while lvq1->IsEmpty() || lvq2->IsEmpty()
    [
      sleep
    ]
    while lvq1->First()->id != lvq2->First()->id
    [
      sleep
    ]
    dequeue msg from lvq1
    dequeue msg2 from lvq2
    update totalTimeL
    update totalDoneL
  }
}

PROCESS FollowerVoter
{
  +void Body()
  {
    while fvq1->IsEmpty() || fvq2->IsEmpty()
    [
      sleep
    ]
    while fvq1->First()->id != fvq2->First()->id
    [
      sleep
    ]
    dequeue msg from fvq1
    dequeue msg2 from fvq2
    update totalTimeF
    update totalDoneF
  }
}

CONTROLLER Controller
OBJECT lsq of Queue
OBJECT fsq of Queue
OBJECT lvq1 of Queue
OBJECT lvq2 of Queue
OBJECT fvq1 of Queue
OBJECT fvq2 of Queue
OBJECT a of Arrival
OBJECT ls of LeaderServer
OBJECT lv of LeaderVoter
OBJECT fs of FollowerServer
OBJECT fv of FollowerVoter

RANDOMS
{
  interArrivalTime exponential 10
  procDelay exponential 1
  localDelay exponential 0.01
  netDelay exponential 0.1
}

STATISTICS
{
  double totalTimeL +now-msg->arrTime
  double totalTimeF +now-msg->arrTime
  int totalMessages +1
  int totalDoneL +1
  int totalDoneF +1
  double avgTimeL =totalTimeL/totalDoneL
  double avgTimeF =totalTimeF/totalDoneF
}
```

Figure 6-16: The SimML notation of the Voltan system

The simulation of this specification is discussed in the next sub-section.

6.4.3. Simulation of Voltan system

There are several parameters that need to be supplied for the random variables used. These can be seen in the "RANDOMS" component of the SimML specification shown in Figure 6-16. This specification is saved into an ASCII file called "voltan" and by running the `runsim` script, the corresponding C++SIM simulation program is generated, compiled and run. Figure 6-17 illustrates the execution of the `runsim` parser and the simulation results produced.

```
bash$ runsim voltan

Simulation maker
----------------
Removing the old C++SIM files...
Read 179 lines of source code...
Creating a SEQUENCE of Message with name Queue using id
Creating a PROCESS class called Message
Creating a PROCESS class called Arrival
Creating a PROCESS class called LeaderServer
Creating a PROCESS class called FollowerServer
Creating a PROCESS class called LeaderVoter
Creating a PROCESS class called FollowerVoter
Creating a CONTROLLER called Controller
Creating the Main program...

Creating the Imakefile...
Creating the Makefile...
Executing the Makefile...

Running the program...

Enter the simulation length: 1000000
totalTimeL = 178048
totalTimeF = 177330
totalMessages = 100272
totalDoneL = 100272
totalDoneF = 100272
avgTimeL = 1.77565
avgTimeF = 1.76849
```

Figure 6-17: Screen dump obtained from running `runsim` parser

The performance time of the FOLLOWER process is better than that of the LEADER. This is because the LEADER has to start first and subsequently it has to wait for the result from the FOLLOWER before it can compare both of the results (in the Leader Voter). On the other hand, the FOLLOWER process would usually have had the result from the Leader Server already in the Follower Voter queue (`fvq2`) before (or at around the same time as) the Follower Server puts its result in `fvq1`. This observation agrees with that of an M.Sc. thesis carried out in 1998 that studied and compared the behaviour of fail-silent and TMR nodes through simulation [39].

Increasing the network delay not only makes the average performance time worse, but also widens the gap between the performance times of the LEADER and FOLLOWER processes. To investigate this further, several simulation runs were performed with the network delay incremented by 0.1. Each simulation was given a simulation length of 100,000, and the results can be seen in Table 6-5.

Table 6-5: Voltan simulation results with incrementing netDelay

mean of	average proc. time		proc. time
netDelay	LEADER	FOLLOWER	difference
0.1	1.77541	1.76586	0.009548
0.2	1.91205	1.88006	0.031986
0.3	2.06357	2.00105	0.062526
0.4	2.22748	2.12677	0.100713
0.5	2.39783	2.25833	0.139498
0.6	2.57757	2.38926	0.188310
0.7	2.76821	2.52798	0.240230
0.8	2.96302	2.67023	0.292786
0.9	3.16979	2.81513	0.354660
1.0	3.37338	2.96305	0.410327

The data presented in Table 6-5 indicate that the differences between the LEADER and FOLLOWER processing times do not grow in a linear fashion. By plotting these differences against the `netDelay` values, a clearer picture is obtained (Figure 6-18).

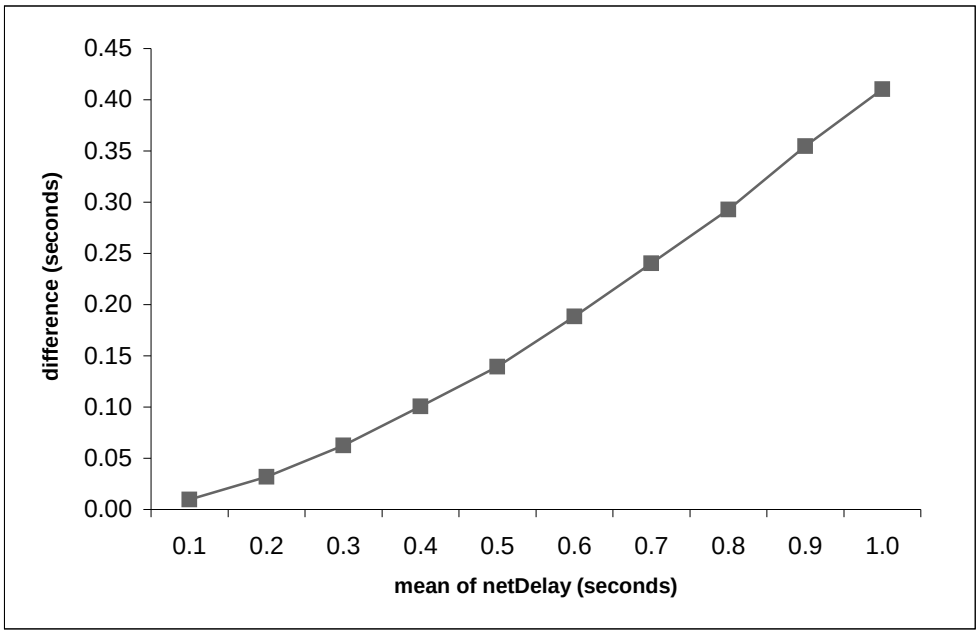


Figure 6-18: Effect of netDelay on performance difference between LEADER and FOLLOWER

This graph shows that the difference between the performance times of the LEADER and FOLLOWER processes grows *exponentially* when the network delay is incremented by a fixed value. This indicates the significance of the network performance in affecting the performance of the Voltan system.

6.4.4. Summary of Voltan case study

Although the *UML/SimML* tool does not provide sufficient support for the Voltan specification, it has been shown that the other SimML parser - `runsim` - is up to the task. This is because `runsim`, which is the older parser of the two, has a richer semantics at an expense of no graphical notation (i.e. its notations are purely textual). Further work can be done to include the extra semantics into the *UML/SimML* tool.

6.5. Some remarks

The three case studies described and simulated in this chapter demonstrate the usefulness of the SimML framework in assisting system designers in choosing a design/scenario that would satisfy the requirement with minimum cost.

The two parser implementations for the SimML framework (the *UML/SimML* and `runsim` parsers, see Chapter 5) have their own advantages and disadvantages. While the *UML/SimML* tool offers graphical notations to be incorporated into the design, it lacks some semantics which makes it a bit difficult for specifying complex systems like Voltan. The `runsim` parser requires its users to understand its syntax well but it offers more features such as easier loop/condition statements and an extra component called SEQUENCE for priority queues.

In both cases, appropriate simulation programs can be generated automatically from the parsers, and if necessary, they can be “fine-tuned” to allow more specific information to be incorporated into the simulation. This include some debugging messages, more complex loops or control sequences and extra statistics collection.

All in all, this chapter has shown that the idea of constructing simulation programs for performance prediction directly from a design notation is workable and proven to be beneficial. Some work still needs to be done to make the tools better and to allow more complex systems to be specified and their performance predicted.

Chapter 7

Conclusion

7.1. Analysis of the SimML Tools

The *Simulation Modelling Language* (SimML) framework is supported by two tools (parsers) that allow the appropriate specification to be transformed into simulation programs in a generic manner. The implementation of these tools is covered in detail in Chapter 5, and some examples of their usage are described in Chapter 6.

The first parser (`runsim`) is able to transform textual design specification (that conforms to the SimML syntax) into C++SIM simulation program. This serves as a prototype for exploring the feasibility of deriving simulation program directly from design specification. In a sense, the SimML framework acts as a bridge that connects the specification and simulation, and this framework proves to be the core of this Ph.D. project.

The second parser (*UML/SimML* tool) combines *Graphical User Interface* (GUI) features and the SimML framework for enabling an automatic generation of simulation program (in JavaSim) from the relevant *Unified Modeling Language* (UML) design notations. The UML notations used here are the *class* and *sequence* diagrams, where a direct mapping between these diagrams and the SimML components and actions have been obtained. Two important aspects of simulation not supported in UML are random properties and statistics specifications. The

UML/SimML tool tries to address this deficiency by providing two extra views for those specifications.

The two parsers above have their own strong points and weak points. While the `runsim` parser only supports textual notation, it allows richer syntax and has some extra features that were not implemented yet in the *UML/SimML* tool (such as the PIPELINE and SEQUENCE components). On the other hand, the *UML/SimML* tool allows graphical notation to be used, but it is more difficult to incorporate more detailed SimML features (such as the *if-elsif* actions) into the notation. As a summary, some comparisons between these two tools are outlined in Table 7-1.

Table 7-1: Comparisons between `runsim` and *UML/SimML* tools

Feature	**`runsim`**	***UML/SimML***
Notation supported	textual	graphical
Simulation program generated	C++SIM	JavaSim
PIPELINE, SEQUENCE components	supported	not supported
SimML actions defined in	`Body()`	`run()`
Control actions supported	*if, elsif, else*	*if*
A block of conditional actions	supported	not supported

One of the advantages of generating simulation programs separately from the design framework or tools is the possibility of fine tuning the simulation to suit more detailed requirements. Here, the tools provide the starting blocks for the simulation program (i.e. the necessary classes and their interactions, data structures, random variables, statistics variables, etc.) and this helps the software designer a lot since normally they are not familiar with simulation techniques. These tools also save us from always building simulation programs from scratch, which can be quite tedious and time consuming.

The most prominent drawback of using simulation approach in predicting systems' performance concerns the amount of resources (machine's memory and processing time) that the simulation program needs to take. This becomes more apparent when long simulation runs are to be performed. The simulation may fail to finish due to the fact that there is not enough memory available to spawn another thread for the active objects. This problem can be alleviated by trying to keep the number of active objects created to a minimum, e.g. by using the SimML DATA component instead of PROCESS component for representing the jobs that the system tries to simulate.

One problem that was found (when using the C++SIM or JavaSim packages) concerns the random numbers generation. The random number generator requires the seeds to be modified when each random distribution class is created, otherwise the starting position of that sequence will always be the same. This results in the generation of the same sequence of numbers, which is not desired. To prevent this from happening, each random stream class has an additional parameter for one of its constructors to indicate the offset in this sequence from which sampling should begin.

7.2. Further Work

During the implementation stage of this project, there are some UML deficiencies uncovered. It is extremely difficult to incorporate *control structures* (such as loops and condition statements) efficiently into design. Although UML provides a *constraint* notation - with the conditions enclosed in a pair of square brackets - it is yet to be formulated if there are multiple actions to be performed upon the fulfilment of those conditions (***Note***: the UML notation used in this thesis was version 1.1. Since

then, the UML notation has evolved a lot and the current version has supports for more complex control structures).

One possible further work is on combining or incorporating the SimML framework with/into an existing UML tool such as Argo/UML [71]. Although the *UML/SimML* tool provides the necessary GUI features for designing and generating simulation program for a system specification, it is not a complete UML tool and its features are limited. The *UML/SimML* tool only offers a subset of UML notations, so it is useful to provide a translation of standard UML patterns into this subset.

It might also be useful to add an extra feature to the *UML/SimML* tool that allows a range of experiments to be conducted. So, instead of investigating only one scenario at a time, it should be possible to specify a range of values upon which several scenarios can be automatically predicted (even compared) and summarised.

Another area that can be investigated further is on how to reduce time and/or resources taken by the simulation runs. This would be useful for generating accurate simulations in a reasonable time. There is a functional simulation tool called DEPEND [32], which presents some techniques for reducing the simulation time explosion. The techniques mentioned there include those on hierarchical simulation, general time acceleration mechanism, as well as variance reduction.

There is also a need to have a standard for incorporating performance related information into UML notations. The amount of work done in this area indicates that there is a wide audience on this topic, and collaboration with other people/project working on the similar line would be beneficial.

Bibliography

[1] Akehurst, D., G. Waters, P. Utton, and G. Martin, "Predictive Performance Analysis for Distributed Systems - PERMABASE position", *Proc. One Day Workshop on Software Performance Prediction extracted from Designs*, Heriot-Watt University, Edinburgh (25 November 1999).

[2] Akehurst, D. H. and A. G. Waters, "UML Deficiencies from the Perspective of Automatic Performance Model Generation", *Proc. OOPSLA '99 Workshop on Rigorous Modelling and Analysis with the UML: Challenges and Limitations* (November 1999).

[3] 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).

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

[5] Arief, L. B., M. C. Little, S. K. Shrivastava, N. A. Speirs, and S. M. Wheater, "Specifying Distributed System Services", *British Telecom Technical Journal - Special Issue*, pp. 126-136 (1999).

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

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

[8] Arief, L. B. and N. A. Speirs, “Specification of Distributed Intelligent Network Systems using UML and Extensions”, BT Deliverable (February 1998).

[9] Arief, L. B. and N. A. Speirs, “A UML Tool for an Automatic Generation of Simulation Programs”, *ACM Proc. 2nd International Workshop on Software Performance (WOSP 2000),* Ottawa, Canada, pp. 71-76 (17-20 September 2000).

[10] Arief, L. B. and N. A. Speirs, “Using SimML to Bridge the Transformation from UML to Simulation”, *Proc. One Day Workshop on Software Performance Prediction extracted from Designs*, Heriot-Watt University, Edinburgh (25 November 1999).

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

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

[13] Banks, J., *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*, John Wiley & Sons, Inc. (1998).

[14] Bennett, B. S., *Simulation Fundamentals*, Prentice Hall (1995).

[15] Bondavalli, A., I. Majzik, and I. Mura, “Automated Dependability Analysis of UML Designs”, *IEEE Proc. 2nd International Symposium on Object-oriented Real-time distributed Computing (ISORC'99),* Saint-Malo, France, pp. 139-144 (1999).

[16] Bondavalli, A., I. Majzik, and I. Mura, "Automatic Dependability Analysis for Supporting Design Decision in UML", *IEEE Proc. 4th High Assurance System Engineering Symposium (HASE'99)*, Washington D.C. (17-19 November 1999).

[17] Booch, G., *Object Oriented Design with Applications*, Benjamin/Cummings Publishing Company, Inc. (1991).

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

[19] Brasiliero, F. V., P. D. Ezhilchelvan, S. K. Shrivastava, N. A. Speirs, and S. Tao, "Implementing Fail-Silent Nodes for Distributed Systems", *IEEE Transactions on Computers*, Vol. 45, No. 11, pp. 1226-1238 (1996).

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

[21] CACI, *MODSIM III: The Language for Object-Oriented Programming (Tutorial)*, CACI Products Co. (1996).

[22] Campione, M., K. Walrath, and A. Huml, *The Java Tutorial Continued - The Rest of the JDK*, Addison-Wesley (1998).

[23] Chan, P., *The Java Developer Almanac 1999*, Addison-Wesley (1999).

[24] Ciardo, G., J. Muppala, and K. Trivedi, "SPNP: Stochastic Petri Net Package", *Proc. 3rd International Workshop on Petri Nets and Performance*, Kyoto, Japan, pp. 142-151 (1989).

[25] Coad, P. and E. Yourdon, *Object-Oriented Analysis, 2nd Edition*. Englewood Cliffs, N.J., Yourdon Press (1991).

[26] Cortellessa, V. and R. Mirandola, "Deriving a Queuing Network based Performance Model from UML Diagrams", *ACM Proc. 2nd International*

Workshop on Software Performance (WOSP 2000), Ottawa, Canada, pp. 58-70 (17-20 September 2000).

[27] de-Miguel, M., T. Lambolais, M. Hannouz, S. Betgé-Brezetz, and S. Piekarec, "UML Extensions for the Specification and Evaluation of Latency Constraints in Architectural Models", *ACM Proc. 2nd International Workshop on Software Performance (WOSP 2000)*, Ottawa, Canada, pp. 83-88 (17-20 September 2000).

[28] Douglass, B. P., *Real Time UML: Developing Efficient Objects for Embedded Systems*, Addison-Wesley (1998).

[29] Eriksson, H.-E. and M. Penker, *UML Toolkit*, John Wiley & Sons, Inc. (1998).

[30] Flanagan, D., *Java in a Nutshell: a Desktop Quick Reference*, O'Reilly (1997).

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

[32] Goswami, K. K., R. K. Iyer, and L. Young, "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis", *IEEE Transactions on Computers*, Vol. 46, No. 1, pp. 60-74 (1997).

[33] Hoeben, F., "Using UML Models for Performance Calculation", *ACM Proc. 2nd International Workshop on Software Performance (WOSP 2000)*, Ottawa, Canada, pp. 77-82 (17-20 September 2000).

[34] Howell, F. and R. McNab, "SimJava", Institute for Computing Systems Architecture, Division of Informatics, University of Edinburgh (online at http://www.dcs.ed.ac.uk/home/hase/simjava/).

[35] Howell, F. and R. McNab, "simjava: a discrete event simulation package for Java with applications in computer systems modelling", *Society for Computer*

Simulation Proc. First International Conference on Web-based Modelling and Simulation, San Diego, CA (January 1998).

[36] Huber, F., S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch, "Tool supported Specification and Simulation of Distributed Systems", *IEEE Computer Society Proc. International Symposium on Software Engineering for Parallel and Distributed Systems*, pp. 155-164 (1998).

[37] Hunt, J., *Java for Practitioners - An Introduction and Reference to Java and Object Orientation*, Springer (1999).

[38] IBM, "XML Parser for Java - XML4J", IBM Alpha Works (online at http://www.alphaworks.ibm.com/tech/xml4j).

[39] Jardin, L. G., "Simulation of Fault Tolerant Distributed System", Department of Computing Science, University of Newcastle upon Tyne, M.Sc. Dissertation (August 1998).

[40] Jarvinen, H.-M. and R. Kurki-Sunio, "DisCo Specification Language: Marriage of Actions and Objects", *IEEE Computer Society Press Proc. 11th International Conference on Distributed Computing Systems*, Arlington, Texas, pp. 142-151 (May 1991).

[41] Java-Tutorial-Team, "Creating a GUI with JFC/Swing" (online at http://java.sun.com/docs/books/tutorial/uiswing).

[42] Java-Tutorial-Team, "The Java Tutorial" (online at http://java.sun.com/docs/books/tutorial/).

[43] Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall International Series in Computer Science (1986).

[44] Jonkers, H., W. Janssen, A. Verschut, and E. Wierstra, “A Unified Framework for Design and Performance Analysis of Distributed Systems”, *Proc. 3rd Annual IEEE International Computer Performance and Dependability Symposium (IPDS’98)*, Durham, NC, USA, pp. 109-118 (September 1998).

[45] Kähkipuro, P., “UML Based Performance Modeling Framework of Object-Oriented Distributed Systems”, *Proc. One Day Workshop on Software Performance Prediction extracted from Designs*, Heriot-Watt University, Edinburgh (25 November 1999).

[46] Katara, M., “A short tutorial on DisCo” (online at http://www.cs.tut.fi/ohj/DisCo/BEST98/tutorial/Tutorial.DisCo92.html).

[47] King, P. and R. Pooley, “Software Performance Prediction Workshop”, Heriot-Watt University, Edinburgh (25 November 1999, online at http://www.cee.hw.ac.uk/~pjbk/umlworkshop/sppw.html).

[48] King, P. and R. Pooley, “Using UML to Derive Stochastic Petri Net Models”, *Proc. 15th UK Performance Engineering Workshop (UKPEW’99)*, Department of Computer Science, University of Bristol, pp. 45-56 (July 1999).

[49] King, P. J. B., *Computer and Communication Systems Performance Modelling*, Prentice Hall (1990).

[50] Lightfoot, D., *Formal Specification Using Z*, Macmillan Press Ltd. (1991).

[51] Little, M. C., “JavaSim” (online at http://javasim.ncl.ac.uk).

[52] 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).

[53] Luckham, D. C., “Rapide: A Language and Toolset for Simulation of Distributed System by Partial Orderings of Events”, DIMACS Partial Order Methods Workshop IV, Princeton University (July 1996).

[54] Luckham, D. C., J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, “Specification and Analysis of System Architecture using Rapide”, *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, pp. 336-355 (1995).

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

[56] Mitrani, I., *Modelling of Computer and Communication Systems*, Cambridge University Press (1987).

[57] Mitrani, I., *Probabilistic Modelling*, Cambridge University Press (1997).

[58] Mitrani, I., *Simulation Techniques for Discrete Event Systems*, Cambridge University Press (1982).

[59] Object-International, *Together: Visual UML modeling with simultaneous round-trip engineering*, Object International Software Ltd. (1998).

[60] Pooley, R., “Software Engineering and Performance: A Road-map”, *ACM Proc. The Future of Software Engineering*, pp. 191-199 (2000).

[61] Pooley, R. J., *An Introduction to Programming in SIMULA*, Blackwell Scientific Publications (1987).

[62] 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 (1999).

[63] Rational-Rose, “UML Resource Center” (online at http://www.rational.com/uml).

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

[65] Scheuerl, S., R. C. H. Connor, R. Morrison, J. E. B. Moss, and D. S. Munro, "MaStA - An I/O Cost Model for Database Crash Recovery Mechanisms", ESPRIT BRA Project 6309 FIDE2, Report FIDE/95/128 (1995).

[66] Scheuerl, S., R. C. H. Connor, R. Morrison, J. E. B. Moss, and D. S. Munro, "The MaStA I/O Cost Model and its Validation Strategy", *Proc. 2nd International Workshop on Advances in Databases and Information Systems (ADBIS'95)*, Moscow, Russia, pp. 165-175 (1995).

[67] Scheuerl, S. J. G., "Modelling Recovery in Database System", School of Mathematical and Computational Sciences, University of St. Andrews, Ph.D. Thesis (August 1997).

[68] Schwartz, R. L., *Learning Perl*, O'Reilly & Associates (1993).

[69] Speirs, N. A. and L. B. Arief, "Simulation of a Telecommunication System using SimML", *IEEE Proc. 33rd Annual Simulation Symposium*, Washington D.C., pp. 131-138 (April 2000).

[70] Speirs, N. A., S. Tao, F. V. Brasileiro, P. D. Ezhilchelvan, and S. K. Shrivastava, "The Design and Implementation of Voltan Fault-Tolerant Nodes for Distributed Systems", *Transputer Communications*, Vol. 1, No. 2, pp. 93-109 (1993).

[71] UCI, "Argo/UML - Providing Cognitive Support for Object-Oriented Design" (online at http://www.ics.uci.edu/pub/arch/uml/).

[72] Utton, P., G. Martin, D. Akehurst, and G. Waters, “Performance Analysis of Object-Oriented Designs for Distributed Systems”, Computing Laboratory, University of Kent at Canterbury, Technical Report 17-99 (March 1999).

[73] Wall, L., T. Christiansen, and J. Orwant, *Programming Perl*, 3rd ed, O'Reilly & Associates (2000).

[74] Wall, L. and R. L. Schwartz, *Programming Perl*, O'Reilly & Associates (1990).

[75] Winder, R. and G. Roberts, *Developing Java Software*, Second ed, John Wiley & Sons, Ltd. (2000).

[76] Woodside, M., “WOSP2000: Second International Workshop on Software and Performance”, Ottawa, Canada (17-20 September 2000, online at http://www.sce.carleton.ca/wosp2000/).

[77] Wright, C., *Java*, Second ed, Hodder & Stoughton (2000).