AUTOMATIC GENERATION OF DISTRIBUTED SYSTEM SIMULATIONS FROM UML

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

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

ABSTRACT

Nowadays, an object-oriented approach is commonly used for building computer systems. The benefits of the object-oriented method, such as scalability, stability and reusability, make this method suitable for building complex systems, including those in the distributed system area. A distributed system 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 do this, it is important to evaluate the requirements of the new system and translate them into a specification (design). The design process helps the system developers to understand the requirements better as well as 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.

UML (Unified Modeling Language) is one example of the object-oriented design methods that has been widely used for specifying system requirements. There are also some object-oriented simulation languages/packages available, for example, SIMULA or C++SIM package, but it is often difficult to transform the system's requirements into a simulation program without sound knowledge of some simulation techniques. On top of that, a new simulation program needs to be built each time for different systems, which can be quite tedious. The currently available UML tools do not provide a feature to generate simulation programs automatically from UML specifications. In this paper, we describe a tool for constructing simulation programs in a generic way, based on a simple specification (preferably in a UML notation) by identifying the simulation components and their structure.

1. INTRODUCTION

The advancement of computer technology demands new systems to be built. New requirements are discovered and new, more efficient methods are available. There exist some difficulties though: as the requirements are normally presented in a plain language, they often contain many ambiguities, which in turn may lead to a mismatch between the completed system and the system proposed by the customer.

This is why it is important to carry out the design process before the commencement of the system's implementation. The aim of the design process is to transfer the system's requirements into a standard notation that can be understood by both the customer and the system developer. UML is one of the design methods that can be used for this purpose and more explanation about UML can be seen in Section 2.1.

Since complex systems are usually expensive to implement, it is often better to predict or estimate the performance they would deliver beforehand. This is where system modeling and simulation is needed. And bearing in mind that those systems would likely be built using an object-oriented programming language, such as C++ (Stroustrup 1997), it is sensible to do the simulation in an object-oriented manner as well. By using an object oriented simulation package, it would be easier to transfer the simulation program into the actual system, since they would employ the similar concepts. This is one of the ideas behind the development of the C++SIM package (Little and McCue 1993; Arjuna 1994) and consequently, the work presented here will use C++SIM for building simulation programs.

Building a simulation program is not a trivial task. The complexity of the proposed system often makes it difficult to know where to start and quite often people need to build a new simulation from scratch. The system developer also needs to know about some simulation techniques, which is not always the case. These difficulties can be solved by firstly identifying the common components of the simulation and their characteristics. Then, some interactions among those components can be defined to provide a way to mimic the behaviour of the proposed system. Based on the components and their interactions, it would be possible to construct a language/syntax which can be parsed to create simulation programs in the desired simulation language or environment, which is C++SIM in this case.

The syntax can be modeled to follow the UML notation (in a textual form), which enables automatic generation of the simulation program from UML-like specification. The simulation components, their interactions and the syntax used for the simulation specification will be described further in Section 2.3 and Section 2.4, while the feasibility of creating a parser for this syntax is discussed in Section 3. Section 4 contains a description of related work that has been done in this area and future work to be done.

2. FROM DESIGN TO SIMULATION

The work involved here includes the use of UML for specifying the system's requirements, the construction of simulation programs using C++SIM package, the analysis of the commonly used simulation components, and the invention of a syntax/language (using the simulation components) that can be used to automate the construction of C++SIM programs from UML specifications.

2.1. UML

UML (Unified Modeling Language) is a language for specifying, visualising, constructing and documenting the artifacts of software systems (Fowler and Scott 1997). 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:

- *Class diagrams*
  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:
  - 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).
  - 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 1.

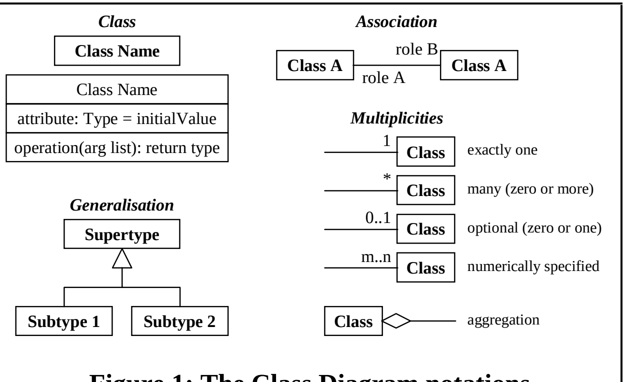

Figure 1: The Class Diagram notations

- *Use Case diagrams*
  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* (which can be a human user or another computer system), the system modeled and the *use-case* - a set of sequences of actions performed by the system that yield an observable result of value to a particular actor (Eriksson and Penker 1998).
- *Interaction diagrams*
  They 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:
  - *Sequence diagrams*: show the interactions in a time sequence.
  - *Collaboration diagrams*: show the interaction in term of links between the objects.
- *State diagrams*
  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 that are involved to change the state.
- *Activity diagrams*
  These diagrams 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.
- *Component diagrams*
  The structures of the implementations and the source codes are described in these diagrams. They show the software components and their dependencies to each other.

UML has some benefits which make it a popular choice for a design tool:

- It is an industry standard, so its notation will be understood by many people.
- The notations employed by the UML are reasonably simple yet they are powerful enough for complex specifications.

UML does not support automatic generation of a simulation from its specifications. Hence it is desired to provide a tool to build simulation programs based on a UML specification. But first, we need to have a look at a simulation language or environment that can be used to accommodate the simulation itself.

2.2. C++SIM Package

C++SIM provides a discrete-event, process-based simulation facilities similar to SIMULA's (Pooley 1987) 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 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.

   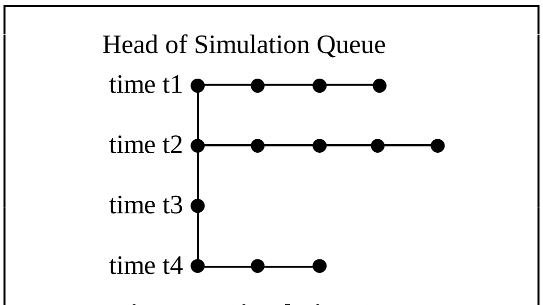

   Figure 2: Simulation Queue

   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). 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.

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.

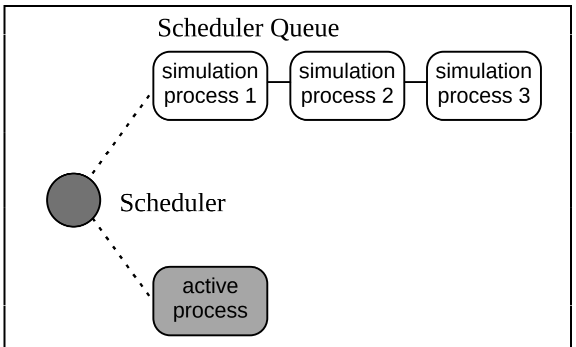

Figure 3: Scheduler-Process Interaction

Each process has a state and at any point during the simulation, a process can *only* be in *one* of the following states:

- *active*: the process has been removed from the head of the scheduler queue and its actions are currently being executed.
- *suspended*: it is on the scheduler queue and is scheduled to be active at a specified simulation time.
- *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.
- *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:

- `Activate`: activates a process. This is invoked by the currently active process which passes the control to the activated process.
- `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.
- `idle`: returns *true* or *false* to indicate whether a process is actually on the scheduler queue or not.
- `Hold`: reschedules the currently active process to be active a fixed units of time later.
- `Cancel`: removes a process from the simulation queue or suspends it indefinitely if it is currently active.
- `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 (Arjuna 1994).

Any class derived from the `Process` class must supply a `Body` part (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 (Little and McCue 1993) and (Arjuna 1994). From experience, we have observed that there are some basic components needed to construct a simulation program. The next section identifies those components which are applicable for many simulation programs.

2.3. Components of Simulation

In general, simulation components can be classified into simulation's *basic types* (which represent entities of the simulation and are defined as classes in C++SIM) and *auxiliary components* (which are useful for representing the instances of active objects as well as for specifying simulation parameters and the collection of simulation statistics). The basic types are:

1. PROCESS
   A PROCESS type 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 in C++SIM to represent this process. This class may have member variables (*public* or *private*) as its attributes as well as some member functions for defining its operations. Since the PROCESS type inherits from the `Process` class (see Section 2.2), it must specify the actions of the `Body` member function derived from that 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 (Figure 1).
2. DATA
   DATA is a reduced version of the PROCESS type, where it actually 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.
3. QUEUE
   A queuing mechanism is a very important concept in simulation and hence a way of specifying queues (for different types of object) must be provided.
4. CONTROLLER
   It acts as the main thread which initialises the simulation, obtains the simulation parameters and summarises the simulation.

In addition to the components above, there are some auxiliary components to supplement the simulation system:

1. OBJECT
   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.
2. INPUT
   The parameters for the simulation are obtained from the user through the INPUT component which then assigns them to the appropriate PROCESS class (through the constructors).
3. RANDOMS
   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 provides several random number generators to accommodate most of those distributions.
4. STATISTICS
   Statistics collection is an important part of a simulation. It is important to know beforehand what are needed to be collected and 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.

Based on these components, a language can be created for specifying a simulation in a generic way, as illustrated in Section 2.4.

2.4. SML: A modeling language/syntax for specifying a simulation

In this section we describe SML (Simulation Modeling Language) which provides a way to specify a simulation (using the standard components described in Section 2.3) in a notation that is easy to understand. A *basic* component is declared as a type which must be followed by a name through which it can be referred. The characteristics of the simulation components can be specified in the `Body` part of the PROCESS component and are referred to as *actions*. The notation for the components and the actions of the PROCESS component are described below:

1. PROCESS component.
   The syntax allows the following to be defined:
   - constructor: denoted by a hash ('#') followed by the types of parameters passed (separated by a comma).
   - member variable: each member variable is declared in a separate line which contains the

visibility (‘+’ to indicate public and ‘-’ to indicate private) followed by its type and name.
- member function: the declaration begins with the visibility, followed by the return type, function name and the function parameters (within a pair of brackets, separated by commas, if any).

The lifetime of a process component is usually throughout the simulation run, i.e. its actions are repeated many times. Quite often, though, it is necessary to create a process that runs only once, for example, a process that has many instances (created dynamically) which are independent of each other. This kind of process is supported by SML and is denoted by adding a keyword ”once” after the process’s name.

2. DATA component.
   The syntax is the same as the that of the PROCESS component but it does not allow any member functions or actions to be declared.
3. QUEUE component.
   It is required to specify what type of object the queue would contain and this can be done by adding an “of” keyword followed by a named object type.
4. CONTROLLER component.
   Since a CONTROLLER is always required and its functionality remains almost the same, it is only necessary to provide a name for this component.
5. OBJECT component.
   An OBJECT is used to represent an instance of an active object used in the simulation. Many OBJECTs can be declared by giving a different name to each of them and specifying which active object it is an instance of.
6. INPUT component.
   This identifies the parameters and which PROCESS component’s member variable they should be assigned to.
7. RANDOMS component.
   The simulation parameters that are modeled to certain distribution functions are declared as random variables in this component. The distribution functions supported are: Uniform, Exponential, Erlang, HyperExponential and Normal distributions.
8. STATISTICS component.
   The STATISTICS component provides a way to specify statistics items (and their types) and how they should be updated. Where or when those items should be updated is specified in the definition of the `Body` function of the PROCESS component (as part of the interaction definition).

The *interactions* between the instances of the simulation’s active objects (i.e. OBJECTs) can be specified as the *actions* of the corresponding PROCESS component. There are some actions provided here:

1. *create*: declares a new instance of the basic type (PROCESS or DATA component).
2. *end*: terminates the execution of a process.
3. *wait*: reschedules the current process to be activated later after a specified time.

```
DATA Job
{
  +double arrTime
}

PROCESS Arrival
{
  +void Body()
  {
    wait interArr
    create j of Job
    record arrTime of j
    update totalJobs
    enqueue j to q
    activate s
  }
}

PROCESS Server
{
  +void Body()
  {
    check q
    dequeue j from q
    wait executionTime
    update totalDone
    update totalTime
    delete j
  }
}
```

```
QUEUE Queue of Job

CONTROLLER Controller
OBJECT a of Arrival
OBJECT s of Server
OBJECT q of Queue

RANDOMS
{
  interArr exponential 5
  executionTime uniform 2 4
}

STATISTICS
{
  double totalTime +now-j->arrTime
  int totalJobs +1
  int totalDone +1
}
```

Figure 4: An example of a specification written in SML

4. *activate*: activates another process.
5. *sleep*: passivates the currently active process or another active object (if its name is supplied as a parameter).
6. *enqueue*: places an object (either of PROCESS type or DATA type) into the tail of a queue.
7. *dequeue*: removes an object at the head of the queue.
8. *check*: passivates the process from which this action is invoked if there are no more items in the queue.
9. *update*: updates the value of a statistics variable.
10. *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).
11. *generate*: produces a number randomly, using a particular random variable (as declared in the RANDOMS component).
12. *print*: useful for debugging, it allows specified simulation data to be printed during the simulation.

On top of these, there are some actions useful for specifying more complicated simulations by adding flow control feature:

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

Each control action is followed by a block of actions (enclosed in ‘[‘ and ‘]’) which determines the appropriate actions for each condition.

Most of the actions above require some parameters, as can be seen in Figure 4.

The concept outlined in this section can be used as a foundation for building a parser which interprets the SML specification and transforms it into C++SIM codes. One example of such a parser is discussed in the following section.

3. AN IMPLEMENTATION EXAMPLE

A tool (parser) written in the Perl scripting language (Wall and Schwartz 1990) is used to transfer the specification written in SML into C++SIM program. It is beyond the scope of this paper to explain this parser in detail, only the principal concepts will be discussed here. The operations can be divided into two parts:

1. Reading the SML specification from a file and storing the information into some Perl arrays to be processed later.
2. Generating the header (.h) and implementation (.cc) files for the C++SIM program from the data stored in the array.

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 an arbitrary size. Perl also facilitates the reading and writing from/to files, which is useful for reading in the specification and writing out the code for that specification.

We need to design the structures of the arrays used beforehand, which is modeled on the SML component specifications. There is one array each for the PROCESS, DATA, QUEUE, OBJECT, INPUT, STATISTICS and RANDOMS components and these arrays will contain different sets of information[†]. 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). In comparison, the QUEUE array needs to store the information on all instances of the QUEUE component, which are just the QUEUE name the type of object this queue will contain.

Based on these array structures, the information obtained from the SML specification (read from a text file) can be arranged and used properly. This information needs to be transferred into several C++ classes in order to build a C++SIM program. Since Perl supports *subroutines*, several subroutines can be implemented to convert the information of different simulation components (stored in the arrays) into their corresponding C++ codes.

There were some difficulties encountered during the development of this simulation tool. We had to bear in mind that this tool must not contain a too complicated syntax which hinders the prospective users from using it in the first place.

The most difficult problem was in deciding how the interactions among the simulation processes should be administered. This involved the determination of the actions that can be performed by a process and how the parameters for those actions are to be passed. It was decided that (for now) the interactions are specified in the `Body` part of each PROCESS component using the actions described in Section 2.4.

[†] There is no need to store much information about the CONTROLLER component since there is only one controller for each simulation and its behaviour is similar. A template for the CONTROLLER component is therefore provided.

Statistics collection was not a trivial task either, especially on how a particular statistical variable should be updated. This is due to the fact that some complex calculation might be required to update the values properly.

Further effort has been made to enable this tool to perform an automatic compilation and execution of the simulation programs generated from the specification. This involves the creation of the appropriate *makefiles* which are needed to compile the generated C++ code (on *Linux* platform), the actual compilation itself and the invocation of the resulting simulation program by the tool.

So far, several non trivial simulation programs have been generated using this tool, such as the Voltan (Brasiliero et al. 1996) fault tolerant system and the Intelligent Network specification of the British Telecom (Arief et al. 1999).

4. RELATED AND FUTURE WORK

Other projects, e.g. Rapide (Luckham et al. 1995) and DEPEND (Goswami et al. 1997) use similar techniques. Rapide provides a set of tools which help in the specification, design and testing of software modules and architectures; it is composed of five sub-languages. We are more interested in its *Executable Language*, which is used for writing executable modules defined by a set of processes that observe and react to events. DEPEND, meanwhile, is a functional simulation tool which provides an integrated design and fault injection environment for system level dependability analysis. Some techniques for reducing the simulation time explosion are outlined, which are useful for generating accurate simulations in a reasonable time.

Note that the work presented here is *not* a full programming language, as compared to MODSIM III (CACI 1996), for example. Instead, it is a tool that can be used to generate simulation programs from specifications based on the components described in Section 2.3 and Section 2.4. Theoretically, this syntax can be applied for generating simulation program in many process-oriented simulation environment (SIMULA, MODSIM III, etc.), but a parser must be built to perform the transformation. We have provided a parser/tool for C++SIM environment, as outlined in Section 3. Using this tool, the performance of the system to be built can be analysed, and many scenarios can be investigated easily.

As a summary, the syntax and tool described in this paper can be used to produce process-oriented simulation programs from UML-like specifications. They can be applied for any systems which involve queues and servers which can then be easily refined to satisfy more specific simulation requirements. At the moment, the notation used by SML is not identical to that of UML; some work is to be done to investigate a way to get the SML syntax closer to the UML notation.

REFERENCES

Arief, L.B.; M.C. Little; S.K. Shrivastava; N.A. Speirs and S.M. Wheater. 1999. “Specifying Distributed System Services”. *BT Technical Journal - Special Issue* (Apr.).

Arjuna Team. 1994. *C++SIM User’s Guide*. Department of Computing Science, University of Newcastle upon Tyne (included in the C++SIM Package, available at *http://cxxsim.ncl.ac.uk/*).

Brasiliero, F.V.; P.D. Ezhilchelvan; S.K. Shrivastava; N.A. Speirs and S. Tao. 1996. “Implementing Fail-Silent Nodes for Distributed Systems”. IEEE Transactions on Computers, Vol. 45, No. 11 (Nov.): 1226-1238.

CACI Products Company. 1996. *MODSIM III: The Language for Object-Oriented Programming (Tutorial)*. CACI Products Co. (Dec.).

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

Eriksson, H. and M. Penker. 1998. *UML Toolkit*. John Wiley & Sons, Inc.

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

Goswami, K.K.; R. K. Iyer and L. Young. 1997. “DEPEND: A Simulation-Based Environment for System Level Dependability Analysis”. *IEEE Transactions on Computers*, Vol. 46, No. 1 (Jan.): 60-74.

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

Luckham, D.C. *et al*. 1995. “Specification and Analysis of System Architecture using Rapide”. *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, (Apr.): 336-355.

Mitrani, I. 1982. *Simulation Techniques for Discrete Event Systems*. Cambridge University Press.

Pooley, R.J. 1987. *An introduction to Programming in SIMULA*. Blackwell Scientific Publications.

Stroustrup, B. 1997. *The C++ Programming Language*. Addison-Wesley.

Wall, L and R.L. Schwartz. 1990. *Programming Perl*. O’Reilly & Associates.

BIOGRAPHY

Leonardus B. Arief received his B.Sc. in Computing Science with a 1st class honours from the University of Newcastle upon Tyne in 1997. He is currently a Ph.D. student at Newcastle University with a scholarship from the Department of Computing Science. His research interests include distributed system, simulation, specification languages, and automatic code generation from software specification.

Neil Speirs obtained a 1st class Honours degree in Mathematics from the University of Newcastle upon Tyne in 1980 and a doctorate in Theoretical Physics from the University of Durham in 1985. For 2 years he worked for Sagesoft Ltd. writing many commercial packages. For two years he worked for Mari Applied Microelectronics Ltd., where he was a project leader on the Esprit Projects Concordia and Delta-4, both of which were concerned with the design and implementation of Fault-Tolerant Distributed Computer systems. Since 1987, he has been a lecturer in Computing Science at the University of Newcastle upon Tyne. His main research interests are in fault-tolerance, reliability and distributed systems. He was the deputy project manager on the Esprit Delta-4 project. He has since led the implementation effort on Voltan - a project to built fail-controlled computing nodes using off the shelf components.