Proceedings of the IASTED International Conference
Applied Modelling and Simulation
September 1-3, 1999, Cairns, Australia

Simulation Generation from UML Like Specifications

L.B. Arief and N.A. Speirs

Department of Computing Science, University of Newcastle upon Tyne, England

E-mail: L.B.Arief@ncl.ac.uk Neil.Speirs@ncl.ac.uk

Abstract

We describe a tool which transform UML specifications (in a textual form) into C++ code which can be used by C++SIM - a discrete-event process-based simulation facility. A tool written in the Perl scripting language is used to perform the automatic transformation from specification into C++SIM code. As an example we show how the tool is used to generate simulations of a non-trivial fault tolerant distributed computing system. The system was specified in about 180 lines of UML like notation and automatically generated a simulation program of about 1100 lines of C++.

Keywords: automated simulation, discrete event simulation, generating simulations, distributed systems

1. Introduction

In complex dependable systems it is desirable to be able to predict or estimate the performance before the system is built. This is typically achieved by the use of system simulation. However, building a simulation program is not a trivial task. The complexity of the proposed system often makes it difficult to begin and quite often a new simulation needs to be built from scratch. The system developer also needs to know about some simulation techniques which is not always the case. These difficulties can be overcome by first identifying the common components of a simulation and their characteristics. Then, some interactions between these components can be defined to provide a way to mimic the behaviour of the proposed system. Based on the components and their interactions, it is possible to construct a language/syntax which can then be parsed to create simulation programs

The syntax we have used follows 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 is described in Section 2. Section 3 discusses the implementation of the parser and in Section 4 we discuss how a simulation of a dependable system (Voltan Fail-Silent nodes [1]) can speedily be generated from a UML specification.

2. From Design to Simulation

The work described here uses UML for specifying the system's requirements. UML [2] is a language for specifying, visualising, constructing and documenting the artifacts of software systems, as well as for business modelling and other non-software systems. It uses many different graphical notations to illustrate a system specification. However, our work is largely based upon the Class diagram which represents the static structure of the system. UML is an industry standard so its notation is understood by many people. In addition, the notations employed by UML are reasonably simple yet powerful enough for complex specifications of dependable.

We shall transform UML specification into C++ code which can be used by C++SIM - a discrete-event process-based simulation facility [3] which is similar to Simula's simulation class and library. We selected C++SIM since more programmers are familiar with C++ programs than with Simula programs and because C++SIM programs runs much faster than their Simula counterparts.

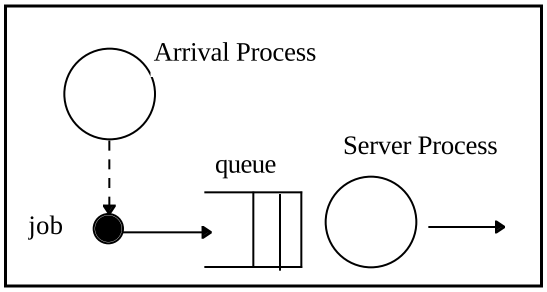
We now identify various simulation components which are applicable for many simulation programs.

- PROCESS - a PROCESS is used to represent an active object in the simulated system and different PROCESS'S are characterised by different names, attributes and operations. Instances of a PROCESS will be transformed into a C++SIM class and must specify the actions of a Body function which is invoked automatically.
- DATA - stores Simulation entities which do not need to be active objects.
- 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.
- CONTROLLER - acts as the main thread which initialises the simulation, obtains the simulation parameters and summarises the simulation results.

In addition to these components there are some auxiliary components to supplement the simulation system.

- OBJECT - an instance of a basic type component. Through these instances, the interaction among the simulation components is achieved.
- RANDOMS - the simulation parameters and their distributions and parameters are specified.
- STATISTICS - used to specify what needs to be collected and where and when the collection should be done.

In Figure 1 we show the specification needed to simulate a system where jobs are generated by an arrival process a, are sent to a queue and are then processed sequentially by a server s. This specification can then be transformed into a simulation program completely automatically.



```
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 exTime
    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
  exTime uniform 1 10
}

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

Figure 1 Specification of a Simple Queueing System

The specification we have used allows the following components to be defined and used.

- PROCESS component: each member variable is declared on a separate line which contains the visibility (+ for public, - for private, = for a constructor) followed by its type and name. For member functions the declaration begins with the visibility followed by the return type, function name and the function parameters.

- DATA component: the syntax is the same as that of PROCESS but no member functions are allowed.

- QUEUE component: the type of object contained in the queue is required and is specified by the word '**of**' followed by a named object type.

- CONTROLLER component: a controller component is always required and its functionality remains almost the same so it is only necessary to give this component a name.

- OBJECT component: an object represents an entity used in the simulation. Objects must be named and their type specified.

- RANDOMS component: this identifies the statistical distributions and parameters of variables to be defined.

- STATISTICS component: the STATISTICS component provides a way to specify statistics items (and their types) and how they should be updated. Where and when the items should be updated is specified in the Body function of the PROCESS component.

The interactions between the instances of the simulation's active objects can be specified in the operation definition of the PROCESS type. We present some identified actions below:

1. *create*: declares a new instance of the basic type.
2. *delete*: deletes an instance of the basic type previously created.
3. *wait*: reschedules the current process to be activated later after a given time.
4. *activate*: activates another process.
5. *sleep*: passivates the current process.
6. *enqueue*: places an object (either PROCESS or DATA type) onto a queue.
7. *dequeue*: removes an object from the head of a queue.
8. *check*: passivates the process from which this action is invoked if there are no more items on the queue.
9. *update*: updates the value of a statistics variable.
10. *record*: sets the value of an objects member variable to the current time or a specified value.
11. *print:* prints a diagnostic message
12. *generate*: produces a number using a particular random number generator.
13. *end*: terminates the execution of the current process

Most of these actions require parameters as shown in Figure 1. Control flow "if" and "while" statements are also permitted.

3. The Parser

A tool called SML [4] written in the Perl scripting language [6] is used to automatically transform specification into C++SIM code. The operations of the parser can be divided into:

1. Reading the specification and storing the information in Perl arrays for processing.
2. Generating header (.h) and implementation (.cc) files for the C++SIM program using the data stored in the Perl array.

The parser also automatically generates appropriate makefiles for the C++SIM code which it generates. The parser allows any system configuration which consists of queues and servers to be specified and simulation code to be automatically generated. Systems where many queues feed into a server and where pipelines of servers are required can also be specified. It is of course possible for the user to insert their own code into the C++SIM code generated. This is necessary to add features to the simulation not supported by the parser.

4. Example Application

To show that the above tool can be used to generate simulations which provide performance information about a non-trivial system, we selected Voltan fail-silent nodes as an application [1]. Voltan fail-silent nodes provide a software implemented technique of creating the abstraction of a fail-silent processor i.e. one which either works correctly or fails by becoming silent. We chose this as an example system because we have already simulated [5] and implemented [1] the system and so know the performance characteristics.

Briefly, a Voltan system is comprised of two computers, a leader and a follower. Correct output messages are signed by both leader and follower to guarantee their authenticity and correctness. Messages arrive at the leader node and are selected for processing in any desired order. A copy of the message selected for processing is then sent to the follower node and the messages are then processed by the nodes. Then each node sends a signed output messages to the other and the messages produced by the two nodes are compared. If the messages are the same, the received message is countersigned and output from the node. If a discrepancy is detected the node attempts to stops processing. No incorrect doubly signed messages are ever produced (see [1] for more details).

The structure of the Voltan software design uses queues to pass messages between processes and so can easily be specified (and hence simulated) using the above techniques. The facilities described above enabled us to simulate the behaviour of a Voltan system within a day whereas coding the simulation by hand took an Masters student (with no prior knowledge of C++SIM) several weeks of effort [5]. The simulation was generated from the specification completely automatically. The system was specified in about 180 lines of UML like notation (see figure 2 below) and generated a simulation program of about 1110 lines of C++.

```
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
    ]
    else
    [
      #similar code for each
      #message type
    ]
    end
  }
}

PROCESS Arrival
{
  +void Body()
  {
    wait interArr
    create msg of Message
    create msg2 of Message
    update totalMsgs
    record id of msg=totalMsgs
    record arrTime of msg
    record type of msg = 1
    activate msg
# and similarly for msg2
# but with type = 2
  }
}

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
  }
}

#Similarly for Follower Server

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

#Similarly for Follower Voter

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

RANDOMS
{
  interArr exponential 10
  procDelay exponential 1
  localDelay exponential 0
  netDelay exponential 0.1
}

STATISTICS
{
  double totalTimeL +now-
               msg->arrTime
  double totalTimeF +now-
               msg->arrTime
  int totalMsgs +1
  int totalDoneL +1
  int totalDoneF +1
}
```

Figure 2 Voltan System Specification

The specification consists of an arrival process to generate new messages, a process for each message transmission in the system and a server and voter process for both the leader and follower. The code for the follower processes is not shown in Figure 2 but is similar to that shown for the leader. `The SEQUENCE Queue of Message using id` statement specifes that items stored in the queue are ordered by the identifier `id`. The `PROCESS Message once` statement indicates that the process is only run once and is terminated when the word `end` is encountered

5. Conclusions

We have described a tool which allows simulations to be generated automatically from specification in a UML like notation. The performance of complex dependable systems can thus be estimated as soon as they have been specified rather than having to write complex simulation code by hand.

6. References

[1] F.V. Brasileiro, P.D. Ezhilchelvan, S.K. Shrivastava, N.A. Speirs and S. Tao, Implementing Fail-Silent Nodes for Distributed Systems, *IEEE Transactions on Computers*, 45 (11), November 1996, 1226-1238.

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

[3] M.C. Little and D.L. McCue, Construction and Use of a Simulation Package in C++, *Computing Science Technical Report TR437*, University of Newcastle upon Tyne, July 1993.

[4] L.B. Arief and N.A. Speirs, *Proc. 13th European Simulation Multiconference*, Volume 1, 1999, 85-91.

[5] L.G. Jardin, Simulation of Fault Tolerant Distributed Systems, *M.Sc. Thesis*, University of Newcastle upon Tyne, September 1998.

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