Using SimML to Bridge the Transformation from UML to Simulation

L.B. Arief and *N.A. Speirs*

Department of Computing Science,
University of Newcastle upon Tyne,
Newcastle upon Tyne NE1 7RU,
England.
Email: {L.B.Arief, Neil.Speirs}@ncl.ac.uk

Introduction

The cost of building a new system is usually quite high and without a proper design, a mismatch might occur between the proposed system and the actual system delivered. Also, it is desired that the performance of the new system can be predicted beforehand, in order to see whether the proposed solution (architecture) does actually satisfy the requirements. One way to achieve this is through simulation. Simulation programs can be built to mimic the execution of the system and the data obtained from running the simulation enable the performance of the system to be calculated and analysed.

There are some drawbacks though. It is often difficult to transform the design into a simulation program without a 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. It would therefore be useful to have a tool that can automatically generate a simulation program from a design notation.

The currently available design tools, such as the Rational Rose's UML (Unified Modeling Language) tool, do not provide a way to do this. This shortcoming is the main reason that motivates our research. The work involves the investigation of the design method to be used (UML [1, 2]), a simulation environment (C++SIM [3, 4]), a syntax that can capture the simulation requirements (SimML [5]) and a parser to transform the design into simulation program (in the *perl* [6] scripting language).

What is SimML?

We have developed a syntax called *SimML* (Simulation Modelling Language) which provides a textual notation that can be transformed into a simulation program. SimML identifies simulation components which are applicable for many simulation programs as well as the interactions that can be performed among them:

1. **Basic Type Components**
   The basic type components represent the entities of the simulation and are defined as classes in the C++ term. There are four such components identified:
   a) *PROCESS component* represents the simulation process and different processes can be characterised by assigning different name, attributes and operations to them. The operations are represented as *actions* (see later) which specify the interaction among the active components in the simulation.
   b) *DATA component* is a simplified version of the PROCESS component and used only as a data storage. It does not represent a simulation process, hence it does not contain any operation.
   c) *QUEUE component* is used to represent the queueing mechanism, which is a very important aspect of simulation.
   d) *CONTROLLER component* is a special component which initialises the simulation, obtains the simulation parameters and later summarises the simulation.

2. **Auxiliary Components**
   They are useful for representing the instances of active objects as well as for specifying the simulation parameters and the collection of simulation statistics.

a) *OBJECT component* is an instance of the basic type component and during the 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) *INPUT component* allows the parameters for the simulation to be obtained from the user, which are then assigned to the appropriate PROCESS components (through the constructors).
c) *RANDOMS component* provides a way to model certain simulation parameters to various distribution functions.
d) *STATISTICS component* specifies what information/data to be collected from the simulation.

3. **Actions:** supporting the interactions among the PROCESS components.
The behaviour of a process is determined by its actions. These are specified inside the `+void Body()` member function of the PROCESS component. The summary of the actions allowed is as follows:
a) *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.
b) *wait:* reschedules the current process to be activated later after a specified time.
c) *activate:* activates another process; multiple activations are allowed (if the process instance is actually a group or array of many processes).
d) *sleep:* passivates the currently active process or another active object (if its name is supplied as a parameter).
e) *enqueue:* places an instance of PROCESS or DATA object to the tail of a queue.
f) *dequeue:* removes an object from the head of a queue.
g) *check:* passivates the process from which this action is invoked if there is no more item in the queue.
h) *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).
i) *update:* updates the value of a statistics variable.
j) *generate:* produces a number randomly, using a particular random number generator (as declared in the RANDOMS component).
k) *end:* terminates the execution of the current process.
l) *delete:* deletes an instance of the basic type created before.
m) *print:* useful for debugging, it allows specified simulation data to be printed during the simulation.
On top of these actions, there are some additional actions which are useful for specifying a more complicated simulation by adding a flow control feature:
n) *if:* specifies a condition that must be satisfied before certain action(s) can be performed. It is complemented by the *elsif* and *else* actions.
o) *while:* allows a loop to repeat the same action(s) until a certain condition is met.

More information about the syntax is available in a separate document [7].

SimML Parser and Example

A suitable parser has been constructed to automatically generate a C++SIM code from the SimML notation. This parser was built in *perl* (Practical Extraction and Report Language) and it also generates the appropriate *makefiles*, compiles the C++SIM code and runs the simulation automatically. This 2000-line script reads the SimML specification from a file, processes the information gathered and then generates the source code for the simulation.

This transformation expands the initial SimML code into a simulation code that is about 6 times in size (number of lines). This saves the system developer from manually writing the simulation program. The use of C++SIM as the simulation environment allows us to "fine-tune" the simulation program to fulfill some specific requirements that are difficult to include in the design notation, such as a complex condition statement or some debugging lines.

As an example, let us consider a simple queuing problem where jobs are generated from an Arrival process and placed into a queue to be processed by a Server process. The inter arrival time is distributed exponentially with mean 5 and the server's execution time is distributed uniformly between 1 and 5. We are interested to know how long it takes on average for a job to be completed. Here is the SimML notation that depicts this problem:

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

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

```
DATA Job
{
  +double arrTime
}

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 5
}

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

For the simulation length of 1000,000, the following results were obtained:

```
totalTime = 1.52947e+06
totalJobs = 200292
totalDone = 200291
avgTime = 7.63624
```

Different scenarios can be investigated easily by altering the simulation parameters or if needed, the structure of the system. For example, if two servers are to be used instead of one, only the following line needs to be changed:

`OBJECT s of Server` ⇒ `OBJECT s[2] of Server`.

The new structure gave these results for the same simulation length:

```
totalTime = 688537
totalJobs = 200292
totalDone = 200291
avgTime = 3.43768
```

Future Work

We are now investigating the feasibility of performing an automatic transformation from the UML's *class* and *sequence* diagrams into a simulation program through SimML. In a way, we have provided the "back-end" of the process, namely the transformation from SimML to C++SIM, so now we can concentrate on the "front-end". There are two possible strands of work here:

1. Use a standard UML tool and convert the information contained there into SimML.
2. Build a new UML tool that knows about SimML notation and hence can produce a simulation code straight from it.

The second approach is currently being undertaken and we are using Java/Swing to build a GUI tool for drawing the UML diagrams and performing the transformation in an implicit way.

References

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

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

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

[4] Arjuna Team, *C++SIM User's Guide*, Department of Computing Science, University of Newcastle upon Tyne, 1994.

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

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

[7] L.B. Arief, *The SimML User Manual* (in preparation).