Simulation of a Telecommunication System Using SimML

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

Abstract

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. One aspect that is important to be investigated prior to the system implementation is its performance. A simulation program could be built to obtain the performance characteristics of the new system, but constructing such a program is not a trivial task. Therefore, it is useful to have a tool that can generate a simulation program automatically from a design notation. We have developed a generic syntax based on the UML design notation which is transformable into a simulation program. A tool that performs the transformation automatically has also been built, and in this paper, we present our experience in designing a new telecommunication system using our syntax and tool.

1. Introduction

The provision of providing additional intelligent services is a pressing problem for the telecommunications industry. In this paper we discuss the problem of providing value added services within a global telecommunications system. As an example we consider adding call barring and blacklisting software services to a telecommunication system. We begin by providing a specification of the system in UML [1] and then use a simulation generation tool (SimML [2]) to automatically generate a simulation of the system. The simulation is then performed to see whether the proposed system can meet the performance requirements using current technology.

The paper is structured as follows. In section 2 we describe the problem and specify it in UML. In section 3 we describe our tool which generates simulations from specifications and discuss its applicability for the problem. In section 4 we present the results of the simulation and show that building such a large system is indeed feasible. Section 5 discusses the feasibility of using XML [3] as a bridge for generating the simulation program directly from the UML diagrams. Finally we present our conclusions in section 6.

2. System specification and design

2.1. The Requirements

We first outline a typical hardware specification for a future world-wide telecommunication system. Such a system might typically be comprised of a distributed system of computers, grouped into sites where each site comprises of approximately 10 machines. There maybe between 60 and 1000 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 and 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 [4].

We now consider a software architecture which implements call barring and blacklisting services. Call barring allows the telecommunication provider to disable all outgoing calls if for example the customer has failed to settle a bill. Blacklisting is a service

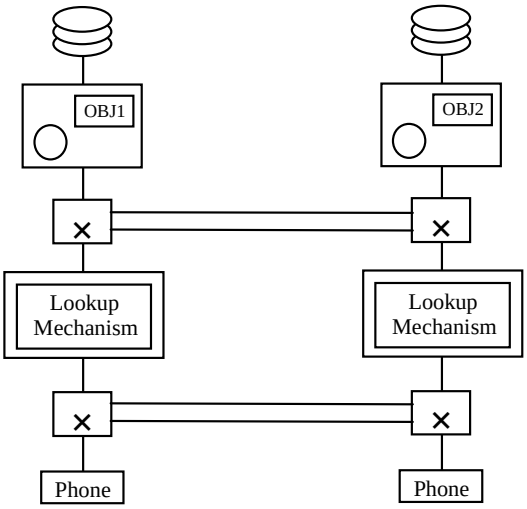

Figure 1: The architecture of the new system

provided to customers which allows them to reject calls from specified telephone callers. Customers can of course modify their blacklist of callers.

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 physical architecture of the basic system is shown in Figure 1.

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

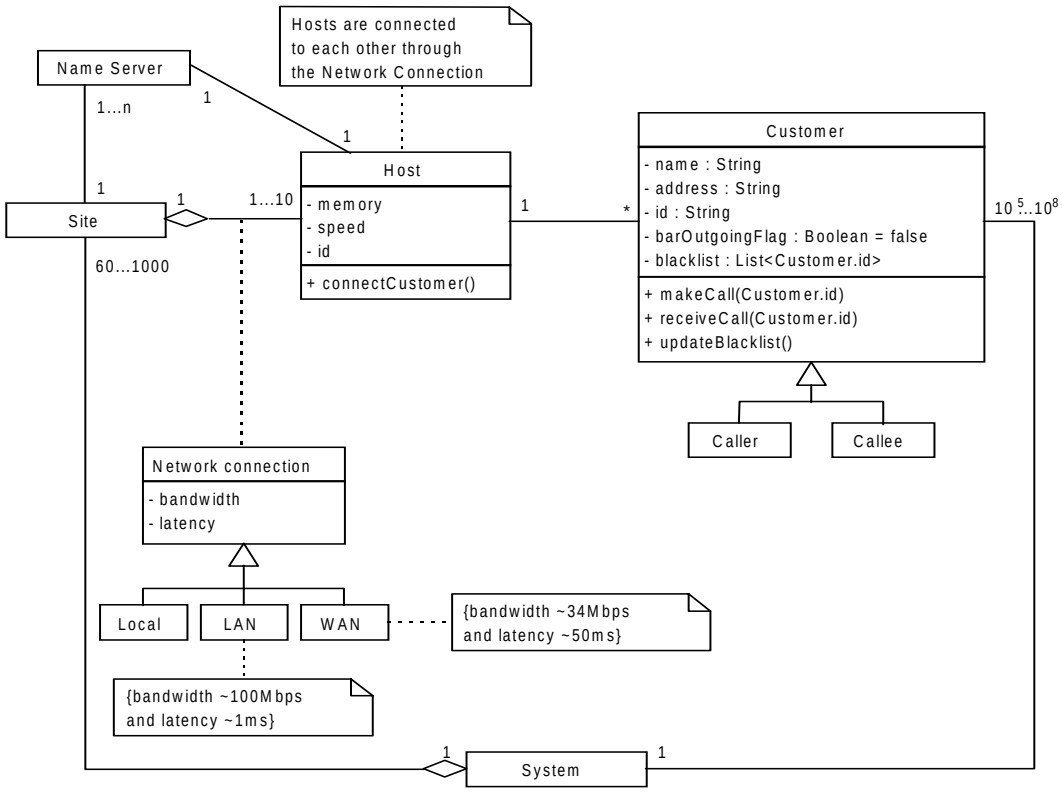

Figure 2: A class diagram notation for the system

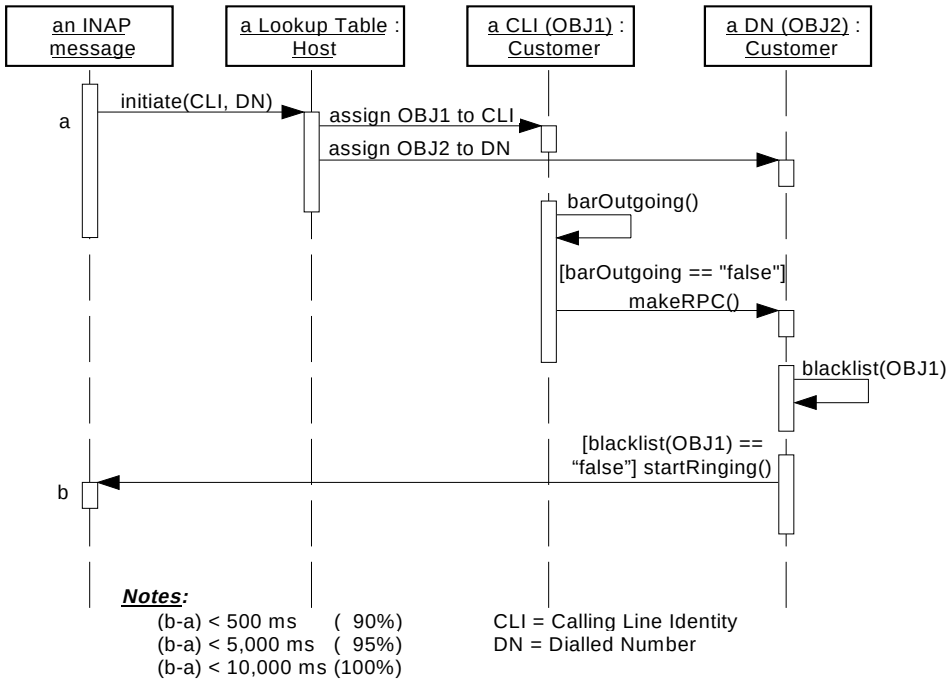

Figure 3: A sequence diagram notation for the system

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.

It is not immediately clear that a system meeting these stringent performance criteria can be constructed using today's technology. Our approach to the problem is first to specify it in UML [1] and then generate a simulation of the system directly from the specification using SimML [2] - a tool we have built to generate simulation code automatically.

2.2 Using UML to design a new system

Unified Modelling Language (UML) has become a standard for specifying, visualising, constructing and documenting the artefacts of software system [1]. It uses graphical notations to illustrate a system specification, and here, we are only interested in the *class* and *interaction* diagrams. The *class diagram* represents the static structure of a system, i.e. its static elements (objects or classes) and the static relationship between them; it can be used to denote the physical requirement of the new telecom system.

The *interaction diagram* shows the pattern of interaction between objects in the system. There are two types of interaction diagrams: a *sequence* diagram (which arranges the interactions in time sequence) and a *collaboration* diagram (which shows the interactions in terms of links between object). Our work here uses the sequence diagram to illustrate the behaviour of the system.

The UML notation for the requirement outlined in Section 2.1 can be seen in Figure 2 (as a class diagram) and Figure 3 (as a sequence diagram). It is desired that a simulation program can be generated automatically from the UML diagrams, but unfortunately, there is no tool available to perform such a task yet. We are working towards the construction of this kind of tool and currently we have a tool that takes a textual representation of the UML and generates a simulation program form it, as described in the next section.

3. Generating the Simulation

3.1. Introducing SimML

We have developed a tool called SimML (*Simulation Modelling Language* [2]) which automatically transforms UML like specifications into C++ code which can be used by C++SIM [5, 6] - a

```
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 <mean>
  exTime uniform <low> <high>
}

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

Figure 4: Specification of a simple queueing system in SimML

discrete-event process-based simulation facility 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. However, the tool could be adapted to generate Simula code if required.

When designing the tool we first identified the following 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 they must specify the actions of a Body function which determines the interaction amongst the active components of simulation.
- **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** - provides a way to model certain simulation parameters to various distribution functions.
- **STATISTICS** - used to specify what needs to be collected and where and when the collection should be done.

In Figure 4 we show the specification needed to simulate a very simple system where jobs are generated by an arrival process, are sent to a queue and are then processed sequentially by a server. This specification can then be transformed into a simulation program completely automatically. The specification we have used allows the following to be defined:

1. PROCESS component, consists of:
   - a *name*: used for identifying a PROCESS as well as for naming the C++ class.
   - *member variables*: each member variable is declared on a separate line which contains the *visibility* (+ for public, - for 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. There must exist a `+void Body()` function which contains the *actions* to be performed by this PROCESS.

2. DATA component
   The syntax is the same as that of PROCESS but no member functions are allowed.
3. QUEUE component
   The name of the queue is specified, followed by the word '**of**' and the type of object this queue will contain (either of the PROCESS or DATA component).
4. 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.
5. OBJECT component
   An object represents an entity used in the simulation. Objects must be named and their type specified.
6. RANDOMS component
   This identifies the statistical distributions and parameters of variables to be defined.
7. 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. *wait*: reschedules the current process to be activated later after a given time.
3. *activate*: activates another process.
4. *sleep*: passivates the current process.
5. *enqueue*: places an object (either PROCESS or DATA type) onto a queue.
6. *dequeue*: removes an object from the head of a queue.
7. *check*: passivates the process from which this action is invoked if there are no more items on the queue.
8. *record*: sets the value of an objects member variable to the current time or a specified value.
9. *update*: updates the value of a statistics variable (used in conjunction with the STATISTICS component).
10. *generate*: produces a number randomly, using a particular random number generator (as specified in the RANDOMS component).
11. *end*: terminates the execution of the current process.
12. *print*: useful for debugging, it allows specified simulation data to be printed during the simulation.

On top of these, there are some actions that are useful for simulating a more complicated system by allowing a flow control feature to be specified:

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

Most of these actions require parameters as shown in Figure 4.

We have built a parser that reads in an SimML specification and automatically transforms it into C++SIM code. This parser is written in the Perl [7] scripting language its operations can be divided into:

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

This tool also automatically generates appropriate *makefiles* for the C++SIM code generated. 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 so that any features of the simulation not supported by the parser can be added.

3.2. Using SimML to specify a telecommunication system

SimML can be used to specify the telecommunication system based on the requirements outlined in Section 2.1. Figure 5 shows the specification of this system, which can then be parsed

using our SimML tool into C++SIM code. Here we have three PROCESSes:

1. *Call*: represents a phone call, which goes through several stages of initiation process before both parties (phone objects) are connected.
2. *Arrival*: generates new Calls and randomly assign them to be either a local, LAN or WAN type calls (see Section 2.1).
3. *Lookup*: simulates the lookup mechanism in assigning the object uids for the objects involved in the calls.

The *actions* of the PROCESSes mimic the initialisation procedure of making a phone call, which is comparable to the sequence diagram in Figure 3. The RANDOMS component provides randomly

```
QUEUE Queue of Call

PROCESS Call once
{
  +int type
  +double netDelay
  +double arrTime
  +void Body()
  {
    wait netDelay
    enqueue this to ltq
    activate lt
    sleep
    wait netDelay
    wait readTime
    wait netDelay
    wait searchTime
    wait netDelay
    update totalTime
    update totalDone
    if type == 1
    [
      update localTime
      update localDone
    ]
    elsif type == 2
    [
      update lanTime
      update lanDone
    ]
    else
    [
      update wanTime
      update wanDone
    ]
    end
  }
}

PROCESS Arrival
{
  +void Body()
  {
    wait interArrivalTime
    create c of Call
    generate rndVal using rndCallGen
    if rndVal < 0.33
    [
      record type of c = 1
      generate rndDelay using localDelay
      record netDelay of c = rndDelay
      update totalLocal
    ]
    elsif rndVal < 0.67
    [
      record type of c = 2
      generate rndDelay using lanDelay
      record netDelay of c = rndDelay
      update totalLan
    ]
    else
    [
      record type of c = 3
      generate rndDelay using wanDelay
      record netDelay of c = rndDelay
      update totalWan
    ]
    update totalCalls
    record arrTime of c
    activate c
  }
}

PROCESS Lookup
{
  +void Body()
  {
    check ltq
    dequeue call from ltq
    wait lookupTime
    activate call
  }
}

CONTROLLER Controller
OBJECT ltq of Queue
OBJECT a of Arrival
OBJECT lt of Lookup

RANDOMS
{
  interArrivalTime exponential 7
  lookupTime exponential 7
  rndCallGen uniform 0 1
  localDelay exponential 0
  lanDelay exponential 1
  wanDelay exponential 50
  readTime exponential 0.00023
  searchTime exponential 0.0057
}

STATISTICS
{
  double totalTime +now-this->arrTime
  double localTime +now-this->arrTime
  double lanTime +now-this->arrTime
  double wanTime +now-this->arrTime
  int totalCalls +1
  int totalLocal +1
  int totalLan +1
  int totalWan +1
  int totalDone +1
  int localDone +1
  int lanDone +1
  int wanDone +1
}
```

Figure 5: Specification of the telecom system in SimML

generated numbers based on a certain distribution function. These numbers are used to simulate the network delays (local, LAN and WAN), the read delay (for performing the check on the *barOutgoing* flag), the search delay (for checking the blacklist), the lookup delay (for assigning the object uids of the calls), and the inter arrival time of the calls.

We are interested to know how long it takes for a call to be processed on average. Therefore, the statistics to be collected are the total processing time and the total number of calls completed. A more detailed statistics on the type of calls (local, LAN and WAN) are also obtained.

4. Simulation Results

We have simulated the problem described in Section 2 where incoming calls arrive, the identity of the calling and receiving objects are found, the caller object is invoked and a check on the boolean value of the *barOutgoing* flag is made. The caller object then invokes the receiver object which checks to see that the caller is not on the blacklist whereupon the call is accepted.

We assume that network delays have an exponential distribution with a mean of 1 millisecond for a LAN and 50 milliseconds for a WAN. The time taken to invoke an object on the same host is assumed to be zero. The time to inspect a *barOutgoing* flag is exponentially distributed with mean 0.23 microseconds and the time to search a blacklist is exponentially distributed with mean 5.7 microseconds. These mean figures were established by experimentation with a simple implementation of a customer object. The lookup time for each object (i.e. the time to map a calling line identifier onto on object address) was assumed to be exponential with mean 7 milliseconds. This figure was the best we could achieve and was derived experimentally based upon a system with 1 million objects evenly distributed over 1000 hosts using our new name location mechanism.

Table 1: The simulation results for the inter arrival time of 10 ms.

	Local	LAN	WAN	Total
Number of Calls Processed	3332	3315	3311	9958
Avg. Response Time (ms)	13.4	19.1	219.8	84.1

Finally, we assume that the probability of an object being local or on the same LAN or on the same WAN are equal. Table 1 shows the simulation results for a run of 100,000 milliseconds with a mean inter arrival time of 10 milliseconds.

These figures show that the system will easily meet the performance criteria given in Section 2. The system is relatively insensitive to variations in the inter arrival time until it is comparable to the mean name lookup time. With a mean inter arrival time of 7 ms the simulation behaves as shown in Table 2.

Table 2: The simulation results for the inter arrival time of 7 ms.

	Local	LAN	WAN	Total
Number of Calls Processed	4731	4724	4715	14170
Avg. Response Time (ms)	53.8	62.9	258.2	124.4

These figures are still acceptable but it is clear that the system is approaching its capacity. Hence for this system to have the desired performance, the crucial parameter is the lookup time of the name location service and not the performance of the network or the objects in the system.

5. Current and Future Work

At the moment, the SimML tool can only take a textual representation of the specification, i.e. it is not yet possible to automatically generate the simulation code from the UML notation. We are currently working towards the provision of a tool that can perform the transformation in a direct manner.

One way to achieve this is by capturing the information contained in the UML diagrams into a standard notation that can be parsed easily. With the acceptance of the *Extensible Markup Language* (XML) [3] as a standard for data/information interchange, it is thought that having a tool that accepts an XML notation would be an advantage. Also, there are some standard parser packages available for reading the XML notation, such as the IBM's XML4J [8] parser for Java [9]. The XML4J package provides the libraries for parsing an XML document and can be used in conjunction with the SAX [10] interface to construct an application that reads in an XML document and transform them into a desired code. Since the parser is written in Java, it is

only appropriate that the simulation program is to be written in Java as well. The JavaSim [11] package suits this requirement well because it is derived from the C++SIM package.

We have created a *Data Type Definition* (DTD) for SimML that allows a simulation specification to be written in XML. We have also constructed an application parser that reads an XML specification and transforms it into a JavaSim program. This application parser can be used later as the back end of the ultimate tool that transform the UML notation straight into a simulation code. We are planning to use the UML class diagram to capture the static structures (classes) of the simulation system and the sequence diagram to represent the interactions among the objects involved in the simulation (i.e. the *action*s in SimML term).

The future work therefore involves the construction of a front end that serves as a UML design tool which is compatible to the application parser above. There are some UML tools that can be modified to suit this need, such as Argo/UML [12], but it is also feasible to construct a new tool specifically for this purpose. We are still investigating both possibilities at the moment.

6. Conclusions

We have described a methodology for assessing the performance of complex distributed telecommunications applications based upon specification in UML and the automatic generation of a simulation based upon this specification. We have shown how the methodology can be applied to a sample telecommunications problem and have deduced the crucial parameters in the design. This allows the system designer to concentrate upon the parts of the system which most affect the performance of the system. This can be done very early in the design process and so is a cost effective solution to the problems of complex system design.

Acknowledgements

We would like to thank Paul Martin of British Telecom for describing the telecommunication system which was used as the study case in this paper.

References

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

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

[3] W3C, "Extensible Markup Language (XML)", online material, available at *http://www.w3.org/XML/*.

[4] L.B. Arief, M.C. Little, S.K. Shrivastava, N.A. Speirs and S.M. Wheater, "Specifying Distributed System Services", BT Technical Journal - Special Issue, April 1999, pp. 126-136.

[5] 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.

[6] Arjuna Team, "C++SIM User's Guide", Department of Computing Science, University of Newcastle upon Tyne (available at *http://cxxsim.ncl.ac.uk/)*, 1994.

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

[8] IBM Alpha Works, "XML Parser for Java-XML4J", online material, available at *http://www.alphaworks.ibm.com/tech/xml4j*.

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

[10] D. Megginson, "SAX: The Simple API for XML", online material, available at *http://www.megginson.com/SAX/index.html*.

[11] M.C. Little, "JavaSim", online material, available at *http://javasim.ncl.ac.uk/*.

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