Object Oriented Simulation of Systems with Concurrency and Time Aspects

José M. Garrido  
Department of Computer Science  
College of Computing and Software Engineering  
Kennesaw State University  
jgarrido@kennesaw.edu

October 2015

Abstract

A simple problem is described and simulated with OOSimL. This problem consists of three-processes communicating among themselves and with the environment. The simulation traces produced by a simulation run can be used for partial verification.

The general goal is to precisely capture the timed interactions between the processes as software components and implement these with a simulation model. The final goal is to support the design of multi-processor systems and high-performance computing (HPC).

1 Introduction

The OOSimL simulation language applies the process interaction approach of discrete event simulation. A simulation model consists of a set of active and passive objects; the interaction among active objects mimics the dynamic behavior of small to large systems. In addition to the formal specification of concurrent systems, this paper focuses on illustrating the support of the simulation modeling of concurrent systems, synchronous communication, and timing aspects, as presented in [1], [2] and [3].

2 Simulation with OOSimL

The OOSimL Simulation Language supports object orientation and the process style of discrete-event simulation that facilitates the modeling and simulation of large and complex systems. A simulation model consists of a set of active and passive objects. Active objects have a life of their own and are implemented as thread objects. Passive objects only exhibit behavior when requested by another object.
2.1 General Statements in OOSimL

In OOSimL, a class defined for active objects is declared with the clause as process. This declaration indicates that class Sender inherits the OOSimL library class Process. For example, the header statement of the definition of class Sender is of the form:

```java
class Sender as process is
    ...
```

This OOSimL statement can also be written as follows:

```java
class Sender inherits psimjava.Process is
    ...
```

2.2 Synchronization Mechanism

Synchronous cooperation results among two or more processes when executing a joint activity during a finite time interval called the cooperation interval. For the processes to carry out the joint activity, the simultaneous participation of the processes is required during the time interval of cooperation. At the end of this joint activity, the processes will continue independently with their individual activities.

Synchronous communication of two processes occurs when a sender process and a receiver process are engaged in a joint activity of data transfer. Because this is synchronous communication, both processes need to participate simultaneously during the interval of cooperation.

In order for two processes to take part in a joint activity, one of the processes takes a more active role in the interaction and designated as the master; the other process is designated as the slave during the cooperation interval. The master is the dominant process and the slave behaves subordinated to the master process during the joint activity.

When the interaction starts, the slave process is suspended until the end of the cooperation interval. The master process then reactivates the slave process, and the two processes will continue their own independent activities. When the slave process is not available, the master process has to wait and is suspended until a slave process becomes available.

Figure 1 illustrates the master and slave processes cooperating in a joint activity. The figure shows the partial timelines of the two processes. The slave process becomes subordinated to the master process during the interval of cooperation.

The synchronization mechanism needed for the process cooperation discussed is provided by objects of class Waitq, known as cooperation objects. These objects support the cooperation of multiple slave processes and multiple master processes. Every cooperation object has two hidden queues: the slave queue and the master queue.

At the beginning of the interaction, the slave processes are suspended and placed in the slave queue until the end of the cooperation interval. A master
A slave process that requests cooperation with a master process, executes the \texttt{wait} statement. A master process that requests cooperation with a slave process, executes the \texttt{cooperate} statement. Figure 2 is a UML diagram that shows the process cooperation between a master process and a slave process using a cooperation object of class \textit{Waitq}.

A sender process attempts to send a message and is suspended if the receiver process is not willing to receive the message at that time. In the same manner, a receiver process is suspended if it attempts to receive a message but the sender is not available to cooperate at the same time.

Figure 3 shows the simulation activity diagram of two processes, P1 and P2, cooperating in a joint activity. Process P1 is the slave and process P2 is the master process. The two processes start with their own individual activities (\textit{Act.1a} and \textit{Act.2a}) then they initiate the cooperation by using the facilities provided by the cooperating object of class \textit{Waitq}. Process P1 gets suspended by the cooperating object. After executing the joint activity, process P2 reactivates
process P1. From this point on, both processes continue with their independent activities, Act.1b and Act.2b.

3 Example of Concurrent System

The following example is a concurrent system that interacts with the environment and that consists of a generic message process that manipulates messages (of class Msgproc) connected via channels to two time buffers (of class TBuffer). The timed buffers communicate with the environment via an input channel and an output channel. The system is illustrated in Figure 4.

Section 3 presented the formal specification of a concurrent system that consists of a generic process that manipulates messages (of class Msgproc) connected via channels to two time buffers (of class TimedBuffer). This concurrent system interacts with an environment that generates messages and sends them to the system; the environment also receives and consumes messages from the concurrent system, as depicted in Figure 4. The other classes in the model are: Sender, Receiver, Channel, Message, Llist, PNode, and Concsys. The last one being the main class and a portion of which is shown on the following listing. This shows declarations of object references with these classes, except for class Message, which is used by the producer to generate messages.

// Channels
// one channel for each pair of processes
define static chin of class Channel //env to first tb
define static lch1 of class Channel //to msgproc
define static lch2 of class Channel // to second tb
define static chout of class Channel // to env rec

// references to active objects
define static send_obj of class Sender // send env
define static rec_obj of class Receiver // rec env
define static first_tbuf of class TimedBuffer
define static second_tbuf of class TimedBuffer
define static msgproc of class Msgproc

The following short listing shows a portion of the source code in class TimedBuffer. The first part the buffer receives a message from channel inchan and assigned to variable rmsg. In the second part, the buffer sends a message in variable rmsg to channel outchan.

    // get message and insert into list
    set rmsg = inchan.receive()
    . . .
    // send to output channel
    call outchan.send using rmsg
    . . .

At a high level of abstraction, a channel is the means of communication between a sender and a receiver processes. Therefore, the mechanism of process cooperation described is hidden in class Channel. This class provides appropriate methods (or member functions) for processes to call when there is need to send or receive a message. Class Message defines the structure and (passive)
behavior of messages. The example has four channels and these are modeled as passive objects. The messages are objects of class Message and are also passive objects.

The following listing shows a portion of the trace produced by a simulation run. This is a sequence of the events recorded during a simulation run.

OOSimL model: Concurrent system with Timed Buffers
Simulation date: 9/7/2015 time: 21:52

------------------------- TRACE ------------------------
0000.000 Env_Sender assembling Messg 0 Computer Science
0002.397 First Timed Buffer attempting comm via Chan_lch1
0002.489 Second Timed Buffer attempting comm via Chan_out
0004.529 Second Timed Buffer attempting comm via Chan_out
0004.844 First Timed Buffer attempting comm via Chan_lch1
0008.156 First Timed Buffer attempting comm via Chan_lch1
0008.736 Second Timed Buffer attempting comm via Chan_out
0010.867 First Timed Buffer attempting comm via Chan_lch1
0012.372 Second Timed Buffer attempting comm via Chan_out
0013.939 First Timed Buffer attempting comm via Chan_lch1
0015.761 Second Timed Buffer attempting comm via Chan_out
0016.496 First Timed Buffer attempting comm via Chan_lch1
0016.984 Env_Sender waiting to communicate via Chan_in
0016.984 Env_Sender sent message 0
0018.682 Env_Sender assembling Messg 1 College of Computing and Soft Engineering
0020.306 Second Timed Buffer attempting comm via Chan_out
1813.073 Second Timed Buffer attempting comm via Chan_out
1813.075 Env_Receiver attempting comm via Chan_out
1813.615 Env_Sender waiting to communicate via Chan_in
1813.615 Env_Sender sent message 96
1813.615 Env_Receiver received message 96 via Chan_out
1814.152 First Timed Buffer attempting comm via Chan_lch1
1816.034 Second Timed Buffer attempting comm via Chan_out
1816.417 Env_Sender assembling Messg 97 Kennesaw State University
1816.470 First Timed Buffer attempting comm via Chan_lch1
1819.793 First Timed Buffer attempting comm via Chan_lch1
1820.216 Second Timed Buffer attempting comm via Chan_out
1823.479 First Timed Buffer attempting comm via Chan_lch1
1825.265 Env_Sender waiting to communicate via Chan_in
1825.265 Env_Sender sent message 97

Verification of the formal specification can be carried out by analyzing the simulation trace. The complete simulation model, includes the OOSimL source files, Java files, byte-code files, and are archived in the file concsys.jar. Output test files of simulation runs are also included; these files and OOSimL software are available in the following web page.
4 Conclusion

The standard method of communication between components in an object-oriented architecture is the method invocation by which an object may request a service from another object if it knows the appropriate method name and object identifier.

Using object-oriented modeling and simulation provides a valuable and powerful approach to help improve the understanding of different aspects of computing such as concurrency and timing constraints.

OOSimL was developed as a high-level object-oriented programming and discrete-event simulation language for education and research in computing. The OOSimL simulation language has as its main advantages the flexibility and power for developing simulation models of large and complex systems.

References

