

Sequential and distributed simulations using Java Threads

Mostafa Azizi

DIRO, Université de Montréal
Montréal, Canada
azizi@iro.umontreal.ca

El Mostapha Aboulhamid

DIRO, Université de Montréal
Montréal, Canada
aboulham@iro.umontreal.ca

Sofiène Tahar

ECE, Concordia University
Montréal, Canada
tahar@ece.concordia.ca

Abstract

The purpose of this paper consists of demonstrating an implementation methodology of sequential and distributed simulations using Java programming: two specific algorithms based on Java threads (single-channel and multi-channel algorithms) are proposed. From this point of view, the events are timely ordered into events lists and controlled by threads with respect to clock cycles. Each thread possesses its event list. The threads are globally timed in the sequential case by one clock, meanwhile in the distributed case they are locally clocked. The main application that is targeted by this work is the simulation of hardware/software systems, where different components are described by threads and obey a multi-clocked system.

1. Introduction

Theoretically, the distributed simulation has more benefits than the sequential simulation, however its practical use is not obvious and depends heavily on the availability of software and hardware components which support this kind of techniques. Communicating threads in Java language provide such support and can be used to design distributed simulators. Since Java is platform independent, these simulators are very portable. Furthermore, the performance is an important issue in simulation, and a multi-threaded simulator can benefit from a powerful multiprocessor machine if available. In this case, independent threads are run in parallel upon many processors, while in the case of single processor architecture, all threads are scheduled upon the same processor. The gain of performance using a parallel architecture is related to the dependency between threads, and also to the operating system ability to optimally bind the threads over the available processors. This subject is not treated in this paper. However, the most important task in the distributed simulation is the management of the

events and time. The event handling must take into consideration the firing time and the clock evolution.

The two algorithms were implemented in Java2 using mainly communicating threads. We have constructed a package, called *coverification*, that must be imported. This package includes objects and methods, such as the clock and its methods, the queue and its methods, the scheduler and its methods, the channel, etc.

This paper is organized as follows. Section 2 presents the related works. Section 3 shows the notation used in this paper. Section 4 deals with the sequential simulation and its algorithm. Section 5 treats the distributed simulation and its algorithm. Section 6 describes the implementation of these algorithms using Java threads. Before concluding, Section 7 discusses the correctness of these algorithms.

2. Related works

Many papers in the literature dealt with the simulation process but we are especially interested in [1, 2]. These latter present the event-driven simulation theory from which we have inspired our idea to potentially insert Java threads [5, 6, 7, 8, 9] in that simulation domain. Java has been recently used to describe hardware/software systems in [3, 4] but the clocking and event-passing problems are not treated. So, from the point of view that a hardware/software system could be modeled as a set of intercommunicating concurrent threads [10], and in the hope to perform hardware/software coverification [11] in the future, we discuss in the current paper the two cases of simulation: sequential and distributed.

3. Notation

- T_i : a thread i
- L_i : the events list of T_i
- $M_i = (m, t)_i$: a message i whose content is m and its firing time is t .

- $(M_j, T_j) \in L_i$: means that the message $M_j = (m, t)_j$ must be sent to the thread T_j by the thread T_i (L_i belongs to T_i) with respect to its sending time t .
- $L_i.M_j$: the message M_j of the event list L_i of T_i .
- $C_{A \rightarrow B}$: the channel that leads the message from the thread sender A to the thread receiver B.
- Clock: the global clock of the system under simulation.
- Clock_i: the local clock of T_i .

4. Sequential simulation

The algorithm of a sequential simulation (SS) consists of simulating a system using a set of threads to describe its functionality and passing messages between threads via a single channel, which means that the communication process is serialized. All the messages belong to threads, and are scheduled according to a global clock. The algorithm is given below:

```
User_Constraint_or_end_of_SS;
Init {
    Clock=0;
    Declaring and creating the simulation channel:
    Channel_SS;
    Declaring and creating the different threads {T1,
    T2, T3,...};
    Timed_ordered_messages_list Li= {(Mj,Tj) /
    where the message Mj=(m,t)j consists of its content
    m and the time of its occurrence t, the thread Tj is
    the receiver (the sender by default is Ti)};
    // Li belongs to the Ti body, it is a local list.
}
while (not (end_of_SS))
{
    // Run all threads T1, T2,...
    Popping and transmitting message Mj=(m,t)j at
    time tj via channel_SS for each list Li of each
    thread Ti with respect to the global clock;
    // Clock ≤ tj
    Clock = tj;
}

```

Algorithm SS

Example 1

Let us assume that the system is modeled by three threads T1, T2 and T3. L1, L2 and L3 are their respective event lists. L1={ (M2, T2), (M3, T3)}; L2={}; L3={ (M1, T2)}; L1.M2=(5, 1); L1.M3=(5, 23); L3.M1=(19, 19); T1 is supposed to send M2 and M3 to respectively T2 and T3 at the times 1 and 23. The contents of M2 and M3 is 5. While T3 is expected to send M1 to T2 at the time 19. The whole execution is summarized in Table 1.

Clock	L1	L2	L3	Fired message	Active thread
<1	(M2, T2) (M3, T3)	-	(M1, T2)	-	-
1	(M2, T2) (M3, T3)	-	(M1, T2)	L1:(M2, T2)	T1
19	(M3, T3)	-	(M1, T2)	L3:(M1, T2)	T3
23	(M3, T3)	-	-	L1:(M3, T3)	T1
>23	-	-	-	-	-

Table 1

5. Distributed simulation

In the case of the distributed simulation algorithm (DS), the threads are locally clocked and the message passing uses more than one channel. The multi-channel use allows a reduction of the simulation runtime of a distributed system by pipelining and paralleling system events. The algorithm is the following:

```
User_Constraint_or_end_of_DS;
Init {
    Declaring and creating the different threads {T1,
    T2, T3, ...};
    Timed_ordered_messages_list Li= {(Mj,Tj) /
    where the message Mj=(m,t)j consists of its content
    m and the time of its occurrence t. The thread Tj is
    the receiver (the sender by default is Ti).};
    // Li belongs to the Ti body, it is a local list.
    Initialize all the threads clocks clocki=0;
    // for all i=0 to n, n is the number of threads
    Declaring and creating channels {C1, C2,...} for
    all threads senders;
}
while (not (end_of_DS))
{
    // Run all threads T1, T2,...
    for ( each thread)
    {
        if (sending) {
            Popping and transmitting the earliest message Mj=(m,t)j at
            time tj via Ci for each list Li of each thread Ti with respect
            to the local clock clocki; // clocki ≤ tj
            clocki = tj;
        }
        if (receiving) {
            waiting for message Mj=(m,t)j;
            if ((Mj.m==cancel(Mp, Tp)) && (clocki<Mp.t))

```

```

clocki = tj;
}}}

```

Algorithm DS

Example 2

Just for comparison purposes we use the assembly line example presented in [1]. The processes Source, A, B, C and Sink are modeled as threads. Their event lists and communicating channels are listed below. Table 2 illustrates the expected execution.

Threads $T = \{\text{Source, A, B, C, Sink}\}$

$L_{\text{Source}} = \{((-, 5), \text{A}), ((-, 7), \text{A}), ((-, 30), \text{A}), ((-, 32), \text{A})\}$

$L_{\text{A}} = \{((-, 9), \text{B}), ((-, 19), \text{B}), ((-, 31), \text{B}), ((-, 37), \text{B})\}$

$L_{\text{B}} = \{((-, 21), \text{C}), ((-, 36), \text{C}), ((-, 38), \text{C}), ((-, 45), \text{C})\}$

$L_{\text{C}} = \{((-, 23), \text{Sink}), ((-, 39), \text{Sink}), ((-, 40), \text{Sink}), ((-, 49), \text{Sink})\}$

$L_{\text{Sink}} = \{\}$

Channels $C = \{C_{\text{Source} \rightarrow \text{A}}, C_{\text{A} \rightarrow \text{B}}, C_{\text{B} \rightarrow \text{C}}, C_{\text{C} \rightarrow \text{Sink}}\}$

Step	Messages to be sent	Senders
1	$((-, 5), \text{A}), ((-, 7), \text{A})$	Source
2	$((-, 9), \text{B})$ $((-, 30), \text{A})$	A Source
3	$((-, 19), \text{B})$ $((-, 21), \text{C})$ $((-, 32), \text{A})$	A B Source
4	$((-, 23), \text{Sink})$ $((-, 31), \text{B})$	C A
...

Table 2

We reproduce the same results as in [1] when applying the thread-based algorithm. The only problem that we faced was the clock blocking resulted from the task end of one of the threads before the whole simulation terminates. This point is discussed in Section 7.

6. Implementation

Both sequential and distributed simulations were implemented in Java, using communicating and concurrent threads. We have written a Java package, called *coverification*, that contains all the necessary simulation ingredients, such as clock, events list, message, signal, and others. This package must be imported in the main program to allow the use of its elements. In this implementation, each thread possesses each event list organized as two event sublists: one (ls) contains the events to be sent, and another (lr) collects the messages that have been received (Figure 1). The events to be sent are produced dependently of the local functionality described by the thread sender. The user of our modest tool has to define the behaviors of the system

entities by rewriting a declared method, called *behavior()*. This method belongs to a super class that defines abstractly the global communication between the threads and manages the storage of results.

In the case of sequential simulation, the process is timed just by one clock and the events are sent or received with respect to their firing times according to the considered clock. The earliest event is always sent first. By the earliest, we mean the message that has the smallest firing time among all the messages of all the sending event lists. These events are sequentially communicated between threads. In other words, they are serialized. Each thread can send a message to the others and receive from them (Figure 2).

In the case of distributed simulation, we consider for each thread an independent clock. The thread sender makes some local computations and eventually sends a message to another with respect to its local clock and to the other clocks (Figure 3).

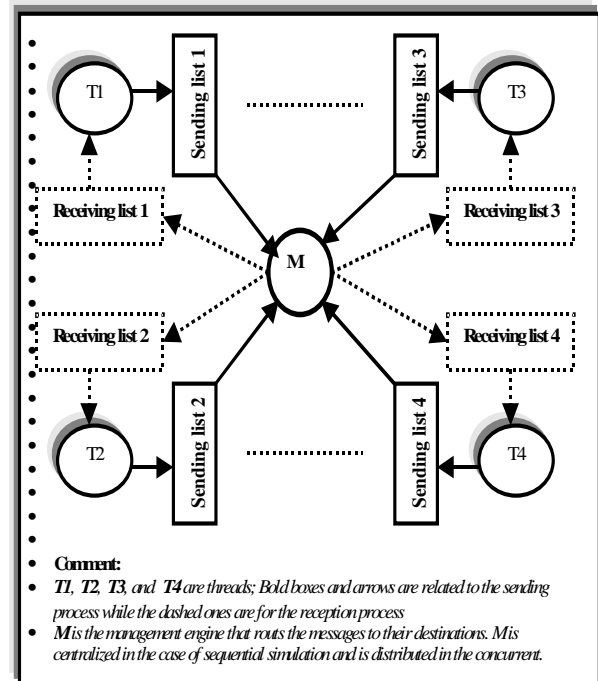


Figure 1. Global view of the implementation

7. Correctness discussion

The proofs of correctness of the two algorithms are given by some specific lemmas and theorems that are presented in [1]. By a simple projection, we concluded that the threads during simulation communicate messages between each other in a timely correct manner, both in sequential and distributed cases. Also, we demonstrated that when using those algorithms, the simulation is correct from two points of view: local (into-thread) or global (inter-thread). But some problems challenge at least in practice.

The exclusion problem is easily solved in Java. Threads cannot, at the same time, access together to a shared object when the keyword *synchronized* is inserted in the object method declaration or applied as an instruction on the object itself. We resolve the clock blocking by sending empty messages (with no data) in order to reach the simulation end, when that thread whose clock is stopped has no message to send/receive.

8. Conclusion

In this paper, we have presented a Java thread-based implementation of both sequential and distributed simulations. We have rewritten their specific classic algorithms by associating the processes to threads and exploiting the benefits of Java programming. The performing of the simulation is driven by event passing. We demonstrate within this paper the feasibility of both cases of simulation using Java threads. The obtained results are promising in the sense that this thread-based simulation can be used in the hardware/software coverification process. This latter checks the correctness of the event-passing mechanism between threads and their internal functions.

References

- [1]. J. Misra, "Distributed Discrete-Event Simulation", ACM/Computing Surveys, Vol. 18, No. 1, 1986
- [2]. D. Lungeanu, and C.-J. R. Shi, "Distributed Simulation of VLSI Systems via Lookahead-Free Self-Adaptative Optimistic and Conservative Synchronization", ICCAD'99, pp. 500-504, 1999
- [3]. T. Kuhn, W. Rosenstiel, and U. Keschull, "Description and simulation of hardware/software systems with Java", DAC'99, 1999
- [4]. R. Helaihel & K. Olukotun, "Java as a Specification Language for Hardware-Software Systems", ICCAD'97, pp. 690-697, 1997
- [5]. D. Lea, "Concurrent programming in Java", Addison-Wesley, 1996
- [6]. Scott Oaks & Henry Wong, "Java: Threads", O'Reilly & Associates, Inc., 1999
- [7]. J. Lowis & W. Loftus, Java : Software solutions, Addison-Wesley, 1998
- [8]. K. Arnold & J. Gosling, The Java™ programming language, Addison-Wesley, 1996
- [9]. J. Magee & J. Kramer, Concurrency : state models & Java programs, Wiley, Worldwide series in computer science, 1999
- [10]. M. Azizi, E.-M. Aboulhamid and S. Tahar, "Multithreading-Based Coverification Technique of HW/SW Systems", Proceedings of the International Conference of Parallel and Distributed Processing Techniques and Applications (PDPTA'99), CSREA Press, Vol. IV, pp. 1999-2005, Las Vegas (Nevada), USA, June 1999
- [11]. M. Azizi, E.-M. Aboulhamid and S. Tahar, "Properties Coverification in HW/SW Systems", Proceedings of the 2nd International Conference of Electronic Circuits and Systems (ECS'99), pp. 80-83, Bratislava, Slovakia, September 1999

```

import coverification.*;
public class Class1 //Sequential Simulation
{
public static void main (String[] args)
{
    Clock Clk=new Clock();
    Sim_Thread.Number_clocks(1);
    Event_List ls1=new Event_List();
    Event_List ls2=new Event_List();
    Event_List ls3=new Event_List();
    Event_List lr1=new Event_List();
    Event_List lr2=new Event_List();
    Event_List lr3=new Event_List();
    ...
    Message m=new Message();
    ...
    software T1=new software("T1",1,ls1,lr1);
    software T2=new software("T2",1,ls2,lr2);
    software T3=new software("T3",1,ls3,lr3);
    ...
    m.setMessage(1, "I", "T3");
    ls1.addMessage(m);
    ...

    T1.start();
    T2.start();
    T3.start();
}
}

```

(a) Code

```

Clock( 1 ):- T1 is sending: M( 1, l, T3 )
Clock( 1 ):- T3 is receiving: M( 1, l, T3 )
Clock( 3 ):- T3 is sending: M( 3, d, T2 )
Clock( 5 ):- T2 is sending: M( 5, n, T1 )
Clock( 3 ):- T2 is receiving: M( 3, d, T2 )
Clock( 5 ):- T1 is receiving: M( 5, n, T1 )
Clock( 6 ):- T3 is sending: M( 6, s, T1 )
Clock( 6 ):- T1 is receiving: M( 6, s, T1 )
Clock( 10 ):- T1 is sending: M( 10, m, T2 )
Clock( 10 ):- T2 is receiving: M( 10, m, T2 )
Clock( 16 ):- T2 is sending: M( 16, l, T3 )
Clock( 16 ):- T3 is receiving: M( 16, l, T3 )
Clock( 35 ):- T2 is sending: M( 35, n, T1 )
Clock( 35 ):- T1 is receiving: M( 35, n, T1 )
Clock( 45 ):- T3 is sending: M( 45, k, T2 )
Clock( 45 ):- T2 is receiving: M( 45, k, T2 )
Clock( 69 ):- T1 is sending: M( 69, k, T3 )
Clock( 69 ):- T3 is receiving: M( 69, k, T3 )
T2: End of simulation ...101
T3: End of simulation ...101
T1: End of simulation ...101

```

(b) Execution

Figure 6.2. Code (a) and Execution (b) of an example of sequential simulation

```

import coverification.*;
class Class2 // Concurrent Simulation{
public static void main (String[] args)
{
    Clock Clk1=new Clock();
    Clock Clk2=new Clock();
    Clock Clk3=new Clock();
    Sim_Thread.Number_clocks(3);
    Event_List ls1=new Event_List();
    Event_List lr1=new Event_List();

    ...
    Message m=new Message();
    ...
    software T1=new software("T1",1,ls1,lr1);
    software T2=new software("T2",2,ls2,lr2);
    software T3=new software("T3",3,ls3,lr3);
    ...
    m.setMessage(1, "I", "T3");
    ls1.addMessage(m1);
    ...
    T1.start();
    T2.start();
    T3.start();
}
}

```

(a) Code

```

T2 starts ...
T3 starts ...
T1 starts ...
Clock1( 1 ):- T1 is sending: M( 1, l, T3 )
Clock3( 1 ):- T3 is receiving: M( 1, l, T3 )
Clock3( 3 ):- T3 is sending: M( 3, d, T2 )
Clock2( 3 ):- T2 is receiving: M( 3, d, T2 )
Clock2( 5 ):- T2 is sending: M( 5, n, T1 )
Clock3( 6 ):- T3 is sending: M( 6, s, T1 )
Clock1( 5 ):- T1 is receiving: M( 5, n, T1 )
Clock1( 6 ):- T1 is receiving: M( 6, s, T1 )
Clock1( 10 ):- T1 is sending: M( 10, m, T2 )
Clock2( 10 ):- T2 is receiving: M( 10, m, T2 )
Clock2( 16 ):- T2 is sending: M( 16, l, T3 )
Clock3( 16 ):- T3 is receiving: M( 16, l, T3 )
Clock2( 35 ):- T2 is sending: M( 35, n, T1 )
Clock1( 35 ):- T1 is receiving: M( 35, n, T1 )
Clock3( 45 ):- T3 is sending: M( 45, k, T2 )
Clock2( 45 ):- T2 is receiving: M( 45, k, T2 )
Clock1( 69 ):- T1 is sending: M( 69, k, T3 )
Clock3( 69 ):- T3 is receiving: M( 69, k, T3 )
T1: End of simulation ...101
T3: End of simulation ...101
T2: End of simulation ...101

```

(b) Execution

Figure 3. Code (a) and Execution (b) of an example of distributed simulation