OPTIMISTIC-PARALLEL, PROCESS-ORIENTED DES IN JAVA USING
BYTECODE REWRITING

Andreas Kunert
Institut für Informatik, Humboldt-Universität zu Berlin, Germany
email: kunert@informatik.hu-berlin.de
May 26, 2008

KEYWORDS
Discrete simulation, Process-oriented, Simulation, Simulation software, Modeling

Abstract
Since the very early beginnings of parallel discrete event-driven simulation (PDES) research, numerous general-purpose PDES frameworks have been implemented. These frameworks can be roughly classified by the utilized modelling view (process-oriented vs. event-oriented), kind of parallelism (conservative-parallel, optimistic-parallel), and implementation language.

Since the main motivation of parallel simulation is the acceleration of single simulation runs, most PDES frameworks have been written in rather low-level languages using the event-oriented modelling view. However, the drawbacks of this approach are modelling restrictions to the user and the impossibility to use concepts of higher-level programming languages or to interoperate directly with programs written in such languages.

In this paper, we review different approaches to implement general-purpose simulation processes in Java and discuss the appropriateness of them regarding optimistic-parallel simulations.

Afterwards, we present our current prototypical implementation of an optimistic-parallel simulation framework in Java providing simulation processes based on a bytecode rewriting approach.

INTRODUCTION
Combining optimistic-parallel simulation with the process-oriented modelling view is a task of combining two opposite design goals: While the main goal of (optimistic-) parallel simulation is the acceleration of simulation runs, the process-oriented modelling (i.e. building a simulation model based on simulation processes) focuses on the manageability of especially large and complex simulation models at the cost of simulation speed. However, there is a need for fast process-oriented simulation frameworks since especially the large and complex systems benefit most from simulation acceleration.

Unfortunately, simulation processes cannot be directly implemented in today’s most used programming languages C++ and Java due to missing concepts like semi-coroutines and continuations. One workaround in low-level languages like C and C++ is the direct manipulation of the program stack using pointers and memory copy routines (8). In pure Java this approach can not be used due to the lack of possibilities to directly access the stack. However, there exist other promising approaches, which we will discuss in this paper.

The main goal of our research is the implementation of an optimistic-parallel simulation framework which utilizes a specific simulation process implementation approach based on bytecode rewriting. The bytecode rewriter itself was taken from a web application framework and enhanced to fit into our simulation scenario.

This paper is structured as follows: first, we present related work in the field of sequential and parallel, event- and process-oriented simulation. A discussion of simulation process implementation alternatives and their applicability to optimistic-parallel simulations is following. Then, a specific implementation approach is particularized and, finally, we present our simulation framework prototype.

RELATED WORK
Domain-specific simulation implementations

The work most related to our research is a simulator for large and complex telecommunication networks implemented at the Georgia Institute of Technology described in (15). The authors developed a compiler-based approach to implement simulation processes described in a domain-specific language.

Their approach and the one we propose share the same main idea: automatically adding program code to store/restore the program stack element by element (instead of copying the whole memory area the stack occupies) and to implement some semi-coroutine behaviour (discussed later). However, due to its compiler-based nature their domain-specific approach is not directly
transferable to a general-purpose simulation framework in Java without having to implement a complete parser for Java which is undesirable for a number of reasons.

**General-purpose simulation frameworks**

Up to today numerous general-purpose DES frameworks have been implemented in Java. However, most of them are based on sequential simulation kernels utilizing the event-oriented modelling view exclusively.

Two sequential DES frameworks supporting the process-oriented modelling view are Desmo-J (12) and jDisco (11), both implementing simulation processes as alternately running Java threads. Another partially process-oriented sequential DES framework utilizing bytecode rewriting is JiST (5). Here bytecode rewriting is used to change the semantics of certain Java methods to introduce a simulation time. The reached speed gain is impressive and an inclusion of parallelization was already considered in the design phase, however, the project has been discontinued after the departure of the main developer.

SPaDES/Java (16) is the only parallel DES framework known to us supporting a process-oriented modelling view. However, it has been implemented using a conservative-parallel approach and therefore the authors did not have to cope with the problems associated with the backward jumps in simulation time in optimistic simulation.

Finally there are optimistic-parallel DES implementations (e.g. FATWa (13)), usually utilizing the event-oriented modelling view.

Table 1 summarizes the mentioned general-purpose frameworks and their features. We are not aware of any general-purpose optimistic-parallel simulation framework providing the process-oriented modelling view.

<table>
<thead>
<tr>
<th>DES framework</th>
<th>parallelism</th>
<th>modelling view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desmo-J</td>
<td>sequential</td>
<td>process-oriented</td>
</tr>
<tr>
<td>jDisco</td>
<td>sequential</td>
<td>process-oriented</td>
</tr>
<tr>
<td>JiST</td>
<td>sequential</td>
<td>event-oriented</td>
</tr>
<tr>
<td>SPaDES/Java</td>
<td>conservative-parallel</td>
<td>process-oriented</td>
</tr>
<tr>
<td>FATWa</td>
<td>optimistic-parallel</td>
<td>event-oriented</td>
</tr>
</tbody>
</table>

**SIMULATION PROCESSES IN JAVA**

In (15), Perumalla and Fujimoto introduce five features of an ideal simulation process implementation shown in Table 2. While it is possible to ease implementation of the simulation kernel and/or increase simulation speed by not supporting some of the features, each missing one creates a restriction to the simulation modeller.

While the features F1 to F3 are standard features of any modern high-level programming language, the last two features, F4 and F5, are simulation specific ones. The mentioned primitives to advance simulation time are calls to simulation kernel methods interrupting the current event execution such as suspend, sleep, etc.

We will revisit the feature list when discussing different simulation process implementation approaches.

**Semi-Coroutines**

The main concept needed to implement a simulation process is basically a method that can interrupt itself (e.g. by calling a “magical” method like suspend()) and resume from exactly this point of interruption on its next call. Such an interruptable method is called a semi-coroutine (7).

In the following code fragment an example method is shown. If this method behaves like a semi-coroutine, on each call it prints out the number of calls so far.

```java
public void run() {
    int i = 1;
    while (true) {
        System.out.println("#call: " + i++);
        suspend();
    }
}
```

One well-known way to realize semi-coroutines is to implement each semi-coroutine in a separate thread. When a semi-coroutine shall be started for the first time, the according thread is started. A interruption-call from the semi-coroutine leads to a suspension of the underlying thread and when the semi-coroutine is to be resumed, the underlying thread is resumed. Whenever the semi-coroutine thread is running, the main program (being a thread itself) waits and vice versa.

The implementation in Java is straight-forward. First the semi-coroutine-code has to be implemented in the run() method of a class implementing the Runnable interface. Moreover, the class gets an attribute Thread thread that references the running semi-coroutine-thread.

```java
public class MySemiCoroutine implements Runnable {
    // already discussed above
    private Thread thread;
    public void run() {
        // coming next
    }
}
```
The implementation of the `resume` and the `suspend` methods are very similar. In each case the currently running thread sets itself into a waiting state immediately after notifying the other thread to continue working. It is important to notice, that even though both methods are implemented in the semi-corooutine class, `suspend` is exclusively called by the semi-corooutine thread, while `resume` is only called by the thread of the main program.

```java
public synchronized void resume() {
    // code
}

public void suspend() {
    // code
}
```

Once again introducing simulation processes, there is the problem that these process states also have to be saved at each event execution. However, this problem cannot be solved by the threading approach presented above.

### Continuations

The abstract concept of storing and later resuming the current execution state of a program or a program part (including the option of reresuming the same stored state over and over again) is called `continuation`. It is obvious, that it is possible to implement a semi-corooutine behaviour using continuations. However, as with semi-corooutines, Java does not provide a direct support for continuations. Moreover, it is impossible to implement generic continuations using pure Java when one design goal is to keep the end-user unaware of the continuation implementation. This is the main reason why there is some work in progress on a Java specification request (JSR) to include continuations in future releases of the Java Development Kit (6).

There are two main problems to be solved when implementing continuations in a stack-based programming language (like Java): The first problem is how to store and restore (fragments of) the stack. This is important since it is the stack where the current local variable values as well as the complete procedure call hierarchy are stored. Once revisited Table 2, a stack backup automatically provides features F1 to F3. The second problem is the provision of entry points (the points in the program code where a continuation continues) to provide feature F4 and F5.

Both problems can be solved by hand when implementing a specific continuation. To solve the first problem, the continuation programmer has to assure that each local variable’s value is stored on a global stack before calling another method. The second problem can be solved in a similar way as converting a process-oriented simulation to an event-oriented one. The method gets an additional parameter representing the point in the code where to continue and the method itself has to provide entry-points to be jumped to as dictated by the parameter (this is usually realized by an all-embracing switch-case statement).

As mentioned above, it is impossible to implement continuations using pure Java when the end-user does not provide some help solving the problems mentioned above. However, apart from the simulation community, Java-based continuations have been implemented utilizing bytecode rewriting, i.e. modifying the bytecode produced by the Java compiler prior to execution by the JVM.

### Threaded simulation processes and the timewarp

In optimistic-parallel simulation, the simulation framework has to provide the necessary data structures and methods to undo all effects resulting from each event if the event becomes invalid due to a backward jump in simulation time (the timewarp). This “undo” is usually implemented by saving simulation model states after every event execution while the simulation time goes forward (9). If a timewarp occurs, the current state is set to the last valid state and all further saved states are deleted.

The threaded implementation approach including various implementation alternatives is particularized in (10). The approach is simple to implement and has been used in a couple of process-oriented DES frameworks (e.g. Desmo-J (12) and jDisco (11)). Moreover, as already discussed in (15), the threaded simulation process implementation supports all features listed in Table 2.

However, the threading approach has one disadvantage: once a thread is resumed, its state before the interruption is lost. This is not a problem in sequential or even conservative-parallel simulation but becomes a huge problem when implementing an optimistic-parallel one.

### Threaded simulation processes and the timewarp

In optimistic-parallel simulation, the simulation framework has to provide the necessary data structures and methods to undo all effects resulting from each event if the event becomes invalid due to a backward jump in simulation time (the timewarp). This “undo” is usually implemented by saving simulation model states after every event execution while the simulation time goes forward (9). If a timewarp occurs, the current state is set to the last valid state and all further saved states are deleted.
CONTINUATION IMPLEMENTATION BY BYTECODE REWRITING

Continuation frameworks provide some “magic” methods (similar to semi-coroutine implementations) like suspend and resume, which end-users can use like any other method. In a continuation framework based on bytecode rewriting, a bytecode rewriter analyzes the compiled Java code and adds additional bytecode statements that realizes the continuation behaviour.

The following code shows a simple continuation example using the later discussed continuation framework Javaflow:

```java
public void run() {
    while (true) {
        Continuation.suspend();
    }
}
```

When compiling this code example using the java compiler javac the following bytecode is generated. Since bytecode is a binary code, we are using the according jasmin assembly representation introduced in (14) in the following code examples.

The ```while``` loop is compiled into a backward jumping goto statement and an according Label (Label0). Moreover, the code example shows how the java compiler did not modify the suspend call, not knowing the additional behaviour it shall imply.

```java
.method public run()V
    .limit locals 1
Label0:  getstatic System/out LPrintStream;
      ldc "#call";
      invokevirtual PrintStream.println(LString;)V
      invokestatic Continuation/suspend()V
      goto Label10
.end method
```

After Javaflow has modified the bytecode, the code looks as follows (the highlighted code was added by the bytecode rewriter):

```java
.method public run()V
    .limit locals 1
    .limit stack 2
    invokestatic StkRec/get()LStkRec;
    dup
    astore_1
    ifnull Label170
    aload_1
    getfield StkRec/isRestoring Z
    ifeq Label170
    aload_1
    invokevirtual StkRec/popObject()J
    astore_0
    ifnonnull Label106
    invokevirtual PrintStream/println(LString;)V
    aload_1
    invokevirtual StkRec/popReference()LObject;
    invokestatic Continuation/suspend()V
    goto Label175
Label106:
    invokevirtual Continuation/suspend()V
    astore_1
    ifnonnull Label137
    invokevirtual StkRec/popObject()LObject;
    invokevirtual Continuation/suspend()V
    goto Label175
Label137:
    invokevirtual PrintStream/println(LString;)V
    goto Label170
.end method
```

The new code in the first highlighted block combines the stack reconstruction and the jumping to specific entry points (stack reconstruction takes place after Label140 and Label159 while the entry points are the labels 70, 75 and 106). The last two highlighted code blocks (following the println and the suspend) do the stack backup.

Existing continuation frameworks

There are two well-known continuation frameworks for Java: Javaflow (3) and Rife/Continuations (4). Both have their origins in the field of web application development. The main use-case for continuations here is to simplify the implementation of user interactions: Typically, a web application starts showing the user a web page in the user’s browser requesting some input. After the user responded, the application uses this input for some calculation and a new web page is created containing results, additional information and/or a new request.
for input. Again the user can respond and gets a new web page and so on.

This interaction between web application and user works quite as well as long as the user keeps responding as expected. However, there is one browser-specific detail, that makes this approach complicated: the back button. Since a browser (and sometimes a user as well) cannot distinguish between static web pages and dynamically created pages from a web application, the user might press the back button and expect the last action to be undone.

The simple and actually used but unsatisfying approach to solve this problem is to programmatically disable the back button. A more sophisticated approach is to implement the user interaction using continuations and to save the continuation’s state before each input request. When the user now hits the back button and expects the last action to be undone, the last valid continuation state will become the current one and all later ones will be deleted.

**Rife/Continuations**

Rife/Continuations is the continuation providing part of the open-source web application framework Rife (4). It was implemented using the bytecode rewriting library ASM (1) and has an additional convenient feature: it also saves and restores the attributes of the object a continuation is implemented in (i.e. information stored on the heap). In terms of simulation processes this means that automatically all simulation process variables are (re)stored, too.

However, up to the current version, Rife/Continuations has a drawback which disqualified its usage for simulation purposes: since the added bytecode only saves the topmost stack segment, continuations implemented using Rife/Continuations are only allowed to suspend in their main method (in our example the run method), i.e. calls to suspend from subsequent methods are not allowed. This means that simulation processes based on the Rife/Continuations implementation would neither provide feature F4 nor F5 from Table 2. This is a serious drawback since process-oriented simulations based on such restricted simulation processes are in fact “process-wrapped” event-oriented simulations as shown in (15).

**Javaflow**

Javaflow is a subproject of the apache-commons-project and it is based on the Bytecode Engineering Library (BCEL) (2) which is an Apache project itself.

Unlike Rife/Continuations Javaflow allows calls to its suspend methods also from nested and recursive called methods, thereby simulation processes based on Javaflow provide all desired features from Table 2.

Since Javaflow provides the semi-coroutine as well as the continuation behaviour which guarantees the provision of all desired features to a simulation process implementation, we chose Javaflow for our simulation framework implementation.

**OUR JAVA PDES FRAMEWORK**

We implemented a prototype of an optimistic-parallel simulation framework including simulation processes based on Javaflow continuations. As most process-oriented simulation frameworks, ours is built on top of an event-oriented one. The parallelism is directly based on Java threads, which makes synchronization very simple. Moreover, we do not have to deal with certain optimistic-parallel simulation problems arising from more distributed solutions such as delayed messages or messages arriving in a different order than sent.

**End-user view**

For the end-user, the continuation part is nearly invisible. The end-user implements a simulation process by deriving from an abstract class SimProcess which demands the definition of two methods: a run method which contains the behaviour of the simulation process and a clone method which is responsible for cloning all time variable attributes (regarding simulation time). The latter method is necessary since Javaflow does not contain the convenient feature of storing/restoring attributes. Moreover, it is a good idea to let the simulation modeller decide which attributes are time variable to avoid the time- and memory-consuming backup of time constant attributes (e.g. references to databases or files, fields filled with constant values, ...).

**Simulation kernel implementation**

Our simulation kernel consists of an arbitrary number of logical processes, which can send each other messages about future events by directly calling a target the receive method of a logical process. Each logical process runs in a separate Java thread and contains a message input queue where messages are sorted by the simulation time the according events shall occur.

It is also the receive method that detects causality errors and subsequently triggers timewarps. Since the receive method is marked synchronized, parallel access to the input queue of a logical process is automatically serialized.\(^3\)

The following Java code example shows the main loop of a logical process:

```java
public void run() {
    while (true) {
        synchronized (this) {
            if (!inputQueue.isEmpty()) {
                // restore already processed messages
            }
            if (!inputQueue.isEmpty()) {
                // remove first message from inputQueue
                // advance in simulation time
            }
        }
    }
}
```

\(^3\)All other methods accessing the input queue are also marked synchronized.
As seen in the code example, timewarps are not handled asynchronously but in a synchronized block in the beginning part of the main loop. This guarantees that all external requests to the logical process are delayed until the timewarp has finished.

Another detail to point out is that the main loop does not differ between event-oriented and process-oriented simulation since the process specifics are dealt with in the evaluateMessage method. The following unabridged code shows the implementation of this method to demonstrate how Javaflow eases the implementation.

```java
private void evaluateMessage() {
    if (currentMessage != null) {
        evaluateMessage(); // discussed later
        yield();
    }
    synchronized (this) {
        // backup processed message
        if (inputQueue.isEmpty() &&
            !timeWarpOccurred) {
            // wait
        }
    }
}
```

Each logical process in our implementation contains two hash tables: While the processStackMap stores a sorted map for each simulation process containing continuation objects (preserving process state information stored on the stack), the processHeapMap stores a sorted map for each process containing process objects (preserving process state information stored on the heap).

When a process is to be started for the first time, two new sorted maps (t1 and t2 in the code example) are created and stored in the according hash tables, in either case using the process object itself as a hash key. Then the process (being a continuation as described above) is started. After the process execution has finished, the resulting continuation object (representing the state to continue with on the next call) is being stored as well as the process object (in its current state).

When a process is to be resumed, the last saved continuation object is restored to continue with. Moreover, the last saved process object is restored, but instead of directly resuming with this object, a cloned object is made to preserve the current process state for a potential later timewarp.\(^4\) Then the process is resumed using the restored continuation object and the cloned process object. Afterwards continuation and process objects are stored as described above.

Experiences and future work

Our prototype already works with simple simulation models and the first simulation experiments reflect the expected behaviour: a large speed gain for simulation models consisting of encapsulated, non-interacting processes; a speed loss for strongly linked simulation models (in both cases compared to a sequential simulator).

However, up to now, we are restricted to small simulation models because we have not implemented a fossil collection yet. I.e. at the moment our implementation does not remove no longer used process states, which leads to a continuously growing memory consumption on each simulation run.

Our next step is the missing implementation of the fossil collection including the calculation of the global virtual time. Afterwards, we plan to implement some larger simulation models that shall demonstrate the speed gain by the parallelism as well as the convenience of the process-oriented modelling view.

CONCLUSION

In this paper we have discussed the two main concepts needed for an implementation of simulation processes in the field of optimistic-parallel simulation.

First, we have introduced the semi-coroutine concept (i.e. methods that can be suspended and resumed)

\(^4\)We do not have to clone the continuation object as well due to the semantics of Javaflow where a new continuation object is returned on each resumption.
that is needed to implement the behaviour of suspendable simulation processes and showed the typical semi-coroutine implementation approach. Afterwards, we introduced the continuation concept which provides the ability to store and later resume the current execution state of a process (and can be used to implement semi-coroutine behaviour, too).

Using continuations, we can implement simulation processes that can go backward in simulation time by resuming from earlier saved process states as needed in optimistic-parallel simulations. We have shown a bytecode rewriting approach to implement continuations in Java and presented two existing continuation frameworks coming from the field of web application development.

Finally, we have presented our prototypical simulation framework using such a bytecode-rewriting continuation framework. The experiences with this prototype so far are promising as there is a speed gain compared to sequential simulation. Future implementation work will show how far this speed gain is affected by the completion of the simulation framework. Once finished, a simulation modeller using our framework will benefit from the ability to build and execute optimistic-parallel simulation models using the process-oriented modelling view.

References


