Process-oriented Discrete-event Simulation in Java with Continuations*
Quantitative Performance Evaluation

Antonio Cuomo[1], Massimiliano Rak[2] and Umberto Villano[1]
[1]*Università degli Studi del Sannio, Benevento, Italy*
[2]*Seconda Università di Napoli, Aversa, Italy*

Keywords: Discrete-event Simulation, Java, Continuations, Benchmark.

Abstract: In discrete-event simulation the process interaction view is appreciated in many different contexts, as it often provides the cleanest and simplest way to express models. However, this view is harder to implement than the more common event-oriented view. This is mostly due to the need for the simulation engine to support in a efficient way the coroutine-like semantics needed to implement the simulation processes. A common solution adopted in many Java-based simulators is the use of system threads to provide coroutines. This paper shows that this choice leads to unnecessary overheads and limitations, and presents an alternative implementation based on *continuations*. For many common models the continuation-based simulator shows significant performance gains compared to the most popular open source Java engines.

1 INTRODUCTION

Discrete-event simulation makes it possible to model a wide class of systems ranging from factory production lines to computer systems, from military operations to air-traffic control, just to mention a few. Support for the computer execution of discrete-event models dates back to the sixties, when simulation-oriented languages as Simula, GPSS and SIMSCRIPT were devised (Nance, 1996). A large research effort has been devoted to enrich mainstream languages as C, C++, Java, Python with simulation capabilities. The most common choice is to provide the additional simulation functionality through a *software library*. Independently of the architectural level at which they are provided (application, library, language), the simulation capabilities embody a *world view* (Derrick et al., 1989) for their users. The world view is essentially the set of concepts that constitute the basic elements available to the modeler to compose and to specify the simulation. The diverse world views are functionally equivalent, but differ in expressive power and in terms of computational efficiency. The most commonly used world views are the *event-oriented*, the *activity-scanning* and the *process-oriented* views. In the event-oriented formalism, the modeler describes the system in terms of *events* which are associated with an *event routine* in charge of handling event records, scheduling of future events, and evaluating conditional events. The resulting model logic is quite fragmented, as the scheduling and the evaluation of conditions are scattered throughout the event routines, but implementation can be made very efficient. In the activity scanning view, the modeler identifies various objects in the systems, the activities that these objects perform, and the conditions under which these activities take place. The simulation is composed of a time-scan (which determines the time increment for the system clock) and an activity scan (which determines which activities can be executed). The process-oriented view hinges on the concept of *process*, a sequence of events and activities through which a specific object moves. It enables the modeler to clearly grasp a model structure, since each object can be represented through a single, coherent process rather than multiple event routing.

Very often the process-oriented view is internally implemented on the top of an event-oriented kernel, due to the efficiency of the last approach. But the simulator design is not trivial: to implement through a sequential program the concurrent execution of simulation processes evolving in discrete time, these should take turns in their execution on a sequential machine (the use of parallel machines introduces further problems). One can argue that the notion of simula-

*This research is partially supported by MIUR-PRIN 2008 project "Cloud@Home: a New Enhanced Computing Paradigm".

Cuomo A., Rak M. and Villano U..
Process-oriented Discrete-event Simulation in Java with Continuations - Quantitative Performance Evaluation.
DOI: 10.5220/0004014500870096
In *Proceedings of the 2nd International Conference on Simulation and Modeling Methodologies, Technologies and Applications* (SIMULTECH-2012), pages 87-96
ISBN: 978-989-8565-20-4

tion processes as interacting "flows of control" which must be able to suspend themselves, to yield control to other processes and, later on, to restart from where they were suspended, is in close correspondence with the programming concept of *coroutines*. These are a generalization of subroutines, introduced in the sixties by Conway. Marlin (Marlin, 1980) best describes two key features of coroutines:

- the values of data local to a coroutine persist between successive calls;
- the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine later.

In the Java language, as there is no direct support for them, coroutines can be implemented through the built-in thread system: it is sufficient to associate a thread to each coroutine, and to perform yields through the basic synchronization primitives generally available with threads (e.g., `wait()` and `notify()`). However, many researchers have recognized that the use of threads just to hold some computation state is overkill, and can possibly lead to large overheads[2].

This paper illustrates an alternative design of a Java-based discrete event simulator, which implements the coroutine semantics through the less common concept of *continuations* (Reynolds, 1993). A continuation is a data structure that stores the computational process at a given point in the program execution (program counter, stack, ...). The data stored can be accessed later on programmatically: upon invocation of the instance of continuation, the process will resume execution from the control point that it previously saved.

Our Java-based discrete event simulator, JADES (which stands for JAva Discrete Event Simulator) uses continuations as the basic tool for providing the coroutine semantics. Our objective was to implement a process-oriented simulator not resorting to threading, and so one that could be possibly immune to the overheads and run-time limitations typical of thread-based Java simulators. This paper firstly presents the continuation-based design of JADES, and then tackles the problem of the quantitative evaluation of its performance in comparison to other currently-available Java simulators. This will require the design of suitable benchmarks and extensive experimentation, both widely documented here.

The rest of this paper is organized as follows. Section 2 provides a description of the most relevant issues in designing process-oriented simulators and an overview of related work. Section 3 and 4 describe the design and implementation of the JADES simulator, respectively. Section 5 presents benchmarks for the evaluation of process-oriented discrete event simulators. In Section 6 a first application of the benchmarks is shown, using them to compare the performance of JADES to other state-of-the-art open source Java simulators. The comments on the test results and our conclusions are the object of Section 7.

[2]Overheads have been amplified by the progressive dismissal of 'green' threads and the use of native operating system threads by the Java Virtual Machine.

2 BACKGROUND AND RELATED WORK

The implementation of a general-purpose simulation system entails the provision of six fundamental features (Kiviat, 1969): 1) representation of simulated time; 2) management of simulated entities, including their creation, state and collections; 3) generation of uniform pseudorandom numbers; 4) generation of non-uniform random variates; 5) statistical data collection; 6) reporting facilities, for summary and/or detailed performance data.

If the simulator is to support the process-oriented view, additional features are needed, as pointed out in (Perumalla and Fujimoto, 1998):

F1 procedures can declare and use local variables.

F2 procedure calls can be nested.

F3 procedures can be recursive and re-entrant.

F4 primitives to advance simulation time can be invoked in any procedure.

F5 primitives to advance simulation time can be invoked wherever a conditional, looping or other statements can appear.

As noted in (Kunert, 2008), features F1 to F3 are directly provided by most general programming languages, while features F4-F5 are difficult to implement as they imply the possibility to suspend the execution of processes and continue it in a later moment.

In the context of Java simulators, we can distinguish between those that implement simulation processes as threads, and others that do not. Based on this distinction, the next subsections highlight the most relevant research contributions in each area. It is worth pointing out the existence of a plethora of discrete-event simulation libraries that do not offer direct support for the process-oriented world view, but provide only the activity scanning or, more commonly, the event-oriented view. On the other hand, there exist simulators that support the process-oriented view, but are not implemented in Java, as

CSIM for C/C++ and SimPy for Python. Both the classes of simulators mentioned above will not be dealt with in the discussion that follows, which will consider only Java-based simulators supporting a process-oriented world view.

2.1 Thread-based Simulators

SimJava (Howell and McNab, 1998) and JSim (Miller et al., 1997) are among the first implementations of the thread-based class of simulators. These early efforts pay particular attention to web-based simulation and to the Java Applet deployment model. Many simulators aim at replicating the functionality and design of Simula in Java. For example, Javasimulation (Helsgaun, 2000) follows the Simula design so close that coroutines are presented as the main mechanism for implementing simulation processes. However, coroutines are in their turn implemented by exploiting the Java threading.

DesmoJ (Lechler and Page, 1999) supports advanced process-oriented modeling features. These include capacity-constrained resources, conditional waiting and special process relationships as producer/consumer and asymmetric master/slave. SSJ (L'Ecuyer and Buist, 2005) is designed for performance, flexibility and extensibility. It offers its users the possibility to choose between many alternatives for most of the internal algorithms and data structures of the simulator.

As mentioned before, Java threads are a powerful resource, but using them just for saving an execution context for later resume is probably overkill, and introduces unnecessary overheads and limitations on the maximum number of simulated processes. Let us consider the two issues separately. As regards the overheads, it should be pointed out that most typically Java platforms implement threads as system threads. Hence threads are individually scheduled by the OS and managed through system calls. This makes it possible to exploit multiple processors, but also introduces a non-negligible overhead associated to thread management activities. As for the limitation on the maximum number of simulated processes, it can be observed that Java threads have a minimum size[3] that cannot be reduced due to the presence of guard pages on the stack. When the address spaces are not so large (e.g., in 32 bit systems), the maximum number of simulated processes is severely limited by the maximum number of threads that can be allocated in a memory where the heap is usually dominant due to the Java execution model. In practice, in a 32-bit Linux system, where the addressable space is 4 GB, of which 1 GB is typically reserved to the kernel, less than 3 GB are available for the rest. Under the reasonable assumption that 1.5 GB are used for the JVM text and data sections and the heap, only 1.5 GB remain for the stack. Even with the lowest setting of stack size for a thread (48 KB), the maximum number of allowed threads will be about 30,000. In fact, this huge number of threads turns out to be not sufficient for models of complex systems in which a considerable number of simultaneously active entities must be simulated (e.g., parallel computers, wide area networks, multitasking operating systems, sensors networks, ...).

[3]Platform dependent, 48 KB on a typical 32-bit Linux box.

2.2 Beyond Threads

In light of all the above, the use of alternatives to threads for saving and resuming a context seems at least reasonable. In recent years, different approaches have been followed to provide a better implementation of the process interaction view, all of which try to deal with the lack of support for coroutines in the standard Java language and virtual machine. D-SOL (Jacobs et al., 2002) is based on the use of a process interpreter. This can be thought of as a virtual machine implemented in Java that executes the code of the process class. The interpreter takes care of pausable methods and is able to save their execution context. To avoid unnecessary overhead, the methods invoked by the process that do not lead to process suspension are directly executed through reflection. Currently, a D-SOL based interpretation engine is found in the current version of the above-mentioned SSJ simulator. Tortuga (Weatherly and Page, 2004) provides an implementation of coroutines in Java which is not based on threads. This is based on a modification of a non-standard Java virtual machine, the Jikes RVM, able to provide the state-saving mechanism.

Works as (Stadler, 2011) are paving the way for the integration of coroutines or of a continuation-like mechanism in the standard Java platform. In the meantime, the use of continuations is beginning to spread in discrete-event simulation. The MyTimeWarp simulator by Kunert (Kunert, 2008) hinges on the JavaFlow continuation library (which is also adopted for the JADES simulator presented in this paper) and focuses on process-oriented time warp optimistic parallel simulation. However, no performance tests and source code are available to make comparison with our work.

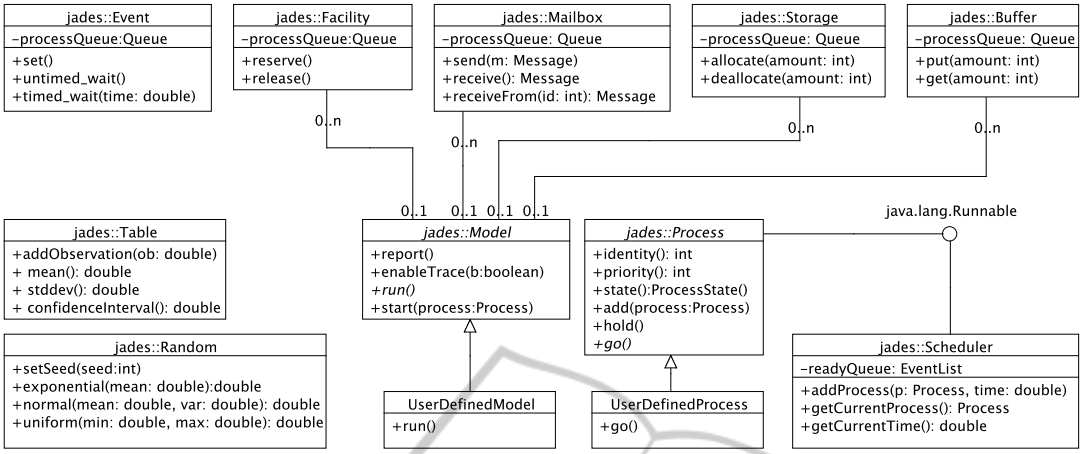


Figure 1: JADES class diagram.

3 JADES: SIMULATOR DESIGN

JADES (JAva Discrete Event Simulator) is a Java library that exploits continuations to support the construction of simulation models according to the process-oriented world view. The aim of JADES is providing modelers with an implementation of the process-oriented paradigm which is both *effective* (as it offers a wide range of building blocks to compose models) and *efficient* (allowing for the creation of large, complex models and their fast evaluation).

While the implementation of the simulator is quite innovative, the design of the JADES interface is inspired by the popular simulator CSIM (Schwetman, 2001), which provides one of the most complete process-oriented simulation APIs for C and C++. There exists a Java version of CSIM, which, unlike the coroutine-based C/C++ version, uses Java threads to implement the simulation processes. This design choice introduces all the drawbacks discussed in the previous section.

The design of JADES is illustrated in Figure 1. In our system, a simulation is composed of processes, which are active components able to act upon (and interact through) passive objects or resources. A model is a logical organization of simulation processes, together with a description of the simulation details, in terms of number of simulation runs, report generation and tracing. Modelers create their own simulation model as a subclass of `Model`: they have only to implement the `run` method, providing all the model-specific logic to create the initial simulation processes and the resources. A model is associated with a `Scheduler`, which manages the queue of ready processes and decides the next process that will run in order of simulation time and, to break ties, process priority. Besides defining the overall model, developers have also to specify the behavior of their processes: as for the model, this is done by subclassing a predefined `Process` class, which provides all the common basic functionalities a process needs. These include: *a*) *holding*, that is, waiting for the passage of a given amount of simulation time; *b*) adding other processes to the simulation (at the current simulation time, or later); *c*) making use of resources. The resources are passive objects that essentially provide higher-level, more useful abstractions than simple process queues. Predefined class of resources provided with the simulator are:

- `Facility`, which models all-or-none resources, as servers;
- `Storage`, which models partially allocatable resources, as the likes of memories, disks, ...;
- `Buffer`, a resource through which processes can communicate and synchronize by producing and consuming items;
- `Mailbox`, through which processes can interact by exchanging messages.
- `Event`, for conditional process synchronization. Process can wait for events to occur and declare events as occurred.

Every resource is instrumented to gather statistics about its usage during the simulation (average queue length, waiting time, number of processes served). Additional statistics can be gathered programmatically using `Table`s, which allow to add modeler-defined observations of values at given points in the program. Almost all the events that happen during execution can be singularly traced, together with the

process to which they refer and the time of simulation when the event happened. Finally, the `Random` class allows the generation of multiple, independent streams of pseudo-random numbers.

4 JADES IMPLEMENTATION

This section will discuss some of the internals of JADES, with particular regard to its distinguishing feature, the use of continuations to implement the simulation processes. The description of the more "conventional" parts of the simulation library is omitted here for brevity's sake, and will be presented in a companion paper. Since continuations are not directly provided by the standard Java platform, JADES must resort to an external system which we call *continuation provider*. Several existing continuation providers have been evaluated for use in JADES. As one of our objectives was the use of a *standard* JVM (to allow easy integration of the simulator in other applications), we did not consider approaches based on modified VMs. Our analysis showed that currently the most mature project is Javaflow (Ortega-Ruiz et al., 2004), a component of the Apache Jakarta Commons Sandbox. Javaflow provides *asymmetric continuations*, in that it forces the programmer to specify the continuation to which he wants to pass control. The core JavaFlow API is found in the static methods of the `Continuation` class:

- `Continuation startWith(Runnable r)` makes it possible to construct a continuation from a `Runnable` object and execute its `run` method. Control passes to the continuation and goes back to the caller if `run` ends or if the `suspend()` method is invoked (in which case a valid continuation is returned).
- `Continuation continueWith(Continuation c)` resumes the execution of the continuation passed as parameter from where it left off.
- `void Continuation suspend()` stops the running continuation, creating a resumption point and giving control back to the method that called `startWith` or `continueWith`.

JavaFlow implements the continuations functionalities through bytecode rewriting. The bytecode of all the classes of the system is scanned for the invocation of the `suspend` method. When a method is found that contains such invocation, it (and all its potential invokers, recursively up to the start of the continuation), are instrumented to add the continuation-management code. This includes: *a*) code which must be executed when a suspension occurs, which includes switches that represent the intermediate points of the method being executed and calls to a library-managed stack that maintains the content of the stack frame; *b*) code for resuming the execution at the point of the suspension (following from the start the chain of switches) with the associated stack contents (popping the stack managed by the library). Examples of this process are shown in (Ortega-Ruiz et al., 2004; Kunert, 2008).

Let us now describe how the JavaFlow library is used in different part of JADES. When the first process is added to the simulation, it is added to the ready queue and a continuation is created for the scheduler code:

```
public void start (Process p){
    scheduler.addProcess(p, scheduler.getCurrentTime());
    Continuation.startWith(scheduler);}
```

When a process wants to add a new process to the simulation, it tells the scheduler to enqueue the new process and gives control to the scheduler continuation by suspending itself.

```
public void add (Process p, double delay){
    double currentTime = scheduler.getCurrentTime();
    scheduler.addProcess(p, currentTime+delay);
    scheduler.addToReadyProcesses(
        scheduler.getCurrentProcess(), currentTime);
    Continuation.suspend();}
```

The code for a process hold is shown below. If the current process will sleep past the wake-up time of the process at the head of the queue, the process is added to the ready queue, giving control to the scheduler continuation. Otherwise, the current process has to hold just to be made active immediately next: we can just advance simulation time to its wake-up time, avoid unnecessary rescheduling.

```
public void hold(double time){
    double wakeupTime = scheduler.getCurrentTime() + time;
    double nextWakeupTime = scheduler.peekNextTime();
    if (nextWakeupTime > wakeupTime)
        scheduler.setCurrentTime(wakeupTime);
    else{
        scheduler.addToReadyProcesses(this, wakeUpTime);
        Continuation.suspend();}}
```

The scheduler continuation executes the scheduling loop in which:

1. The process with smallest wakeup time is extracted from the ready queue. If there is no process available, simulation is finished. Otherwise:

2(a). if the extracted process is at its first schedule, a continuation is created for it and started through the `Continuation.startWith()` method;

2(b). else if the extracted process is not at its first schedule, its continuation is resumed through the `Continuation.continueWith()` method.

Table 1: A Benchmark suite for process-oriented discrete event simulators.

Name	Type	Parameters		Output
ProcessCreator	Micro	- *simTime* - *thinkTime*	total simulation time delay between successive process creations	- # of created processes - Execution time (ms)
PingPong	Micro	- *simTime* - *thinkTime* - *stackDepth*	total simulation time predetermined process delay depth of the stack (*additional* # of frames put on the stack when delay is invoked)	- Execution time (ms)
MM1Queue	Kernel	- *simTime* - *iarTime* - *srvTime*	total simulation time mean job inter-arrival time mean resource service time	- Execution time (ms)

Our impressions on Javaflow, after its adoption for JADES development, is that it is not the ultimate solution for providing continuations in Java, and that there is still room for improvements. In fact, in the literature are emerging other approaches that should allow a more efficient implementation of these constructs (Stadler, 2011), for example with the direct support of the virtual machine. Meanwhile, Javaflow offers the best support available for the continuations we wanted to exploit in JADES. The issue here is not to provide a better implementation of continuations, but simply to check if their use in the place of threads can lead to performance benefits in Java discrete-event simulations. This assessment is the goal of the next sections.

5 BENCHMARK DESIGN

To evaluate the efficiency of the proposed approach, the performance of JADES has been analyzed and compared to other open source process-oriented Java simulators. Since both JADES and its competitors are available for testing, we found direct measurement to be a suitable technique to obtain highly accurate performance comparisons. However, to the best of our knowledge, no standard benchmark exists for comparing process-oriented discrete-event simulators. In event-oriented simulators it is common to use the event processing rate as a measure of performance of the system under test, but this metric does not apply to process-oriented systems. The PHOLD model (Fujimoto, 1990) is suitable for comparing the performances of parallel discrete-event simulators which interact through message-passing, but it is unfit to evaluate sequential simulator behavior.

Our rationale in designing our own suite of benchmarks for comparision of process-oriented discrete event simulator is to provide models with small amount of computation and strong exercising of the process-oriented simulator mechanisms. The benchmarks are specified in Table 1 and described hereafter.

The `ProcessCreator` benchmark gives a measure of how many processes can be handled by the simulator. It runs until *simTime* is reached, spawning a new process every *thinkTime*. The created processes will perform a hold operation until *simTime*, in order to remain alive during the whole simulation and to avoid having their resources collected and reused. At the end of the run, there will be *simTime*/*thinkTime* processes alive. As `ProcessCreator` exercises only a single basic function, it can be classified as a microbenchmark.

`PingPong` is another microbenchmark, which measures the cost associated with process switch (what is called, in OS terms, a context switch). It consists of two processes that hold for *thinkTime* units of time. The first one is scheduled at time 0, while the second at time *thinkTime*/2. This leads to a strict alternation of the two processes, which generate a total of (2 * *simTime*/*thinkTime*) context switches. In order to measure switch time in different working conditions, a further parameter is used to supply the benchmark with the deepness of the stack at which the hold must occur. This has a direct impact on the quantity of "context" that must be saved and restored.

The `MM1Queue` benchmark is an implementation of a M/M/1 queuing model in which customers arrive according to a Poisson process with rate λ, service time is exponentially distributed with mean $\frac{1}{\mu}$ and there is 1 server. When customers find the server busy, they are added to a queue of infinite capacity. In the implementation, a `Generator` process loops, alternately spawning a `Job` and holding for a time exponentially distributed with mean *iarTime*. The `Job` processes compete for exclusive access to a resource, which is used for a time exponentially distributed with mean *srvTime*. When $\lambda > \mu$ the queue is unstable and the expected number of users grows steadily as sim-

ulation proceeds. In process-oriented simulation, an increasing number of users in the system corresponds to an increasing number of simultaneously active processes. Hence this benchmark can easily activate a huge number of processes if the queue is unstable. The `MM1Queue` benchmark can be classified as a kernel benchmark, as it is representative of the core behavior of more complex queuing network models.

6 EXPERIMENTS

For our tests, we have selected several state-of-the-art open source Java simulation engines, purposely neglecting non-Java frameworks, which typically have non-comparable performance. For every simulation engine, an implementation of the benchmark suite described in the previous section has been devised. The set of simulators to be evaluated, includes, in addition to JADES, the following:

- JADESThreads, a previous implementation of JADES based on threads;
- Javasimulation (Helsgaun, 2000), version 2.1, as a representative of the "barebone" process-oriented simulators, which provide a simple implementation of basic processes in the form of wrappers around the underlying thread implementation.
- Desmo-J (Lechler and Page, 1999), version 2.3.2, a process-oriented simulator whose thread-based implementation spans 10 years of maintenance.
- The D-SOL simulator, version 2.1 (Jacobs et al., 2002). This is representative of the (few) available simulation engines that are based neither on threads nor on continuations;
- The SSJ simulator (L'Ecuyer and Buist, 2005), version 2.4, a complete solution for discrete-event simulation. It is provided with both a thread-based implementation and an alternative interpretative mechanism which hinges on D-SOL. Only the former implementation has been chosen for our tests, as D-SOL is considered separately.

The next step is the design of a set of experiments to be performed on the simulators. A simple design was chosen, in which every benchmark-specific parameter has been varied to explore its effects. Apart from benchmark-specific parameters, the tests depend on the configuration of the Java Virtual Machine. The standard HotSpot virtual machine in server configuration has been used. Two parameters of the JVM are particularly relevant for the tests, maximum heap size and thread stack size. These were judiciously chosen for every benchmark. In the ProcessCreator and MM1Queue benchmark, which involve the creation of many processes, the heap was limited to 1 GB to ensure that thread-based simulators are not penalized. In the PingPong benchmark, with only 2 processes involved, the heap limit was raised to 1.5 GB. All the data points plotted in the figures are actually the average of 5 repetitions. We observed that the coefficient of variation (CV) was under 10%. Since many simulators are involved, error bars are not plotted in the following figures to avoid cluttering the graphs.

To allow for repeatability and results contextualization, Table 2 summarizes the hardware and software configuration of the test environment.

Table 2: Testbed configuration.

CPU	Intel Core 2 Duo T9500 2.5 Ghz
RAM	3 GB DDR2 PC5300 @667 Mhz
OS	GNU/Linux 2.6.38-2, 32-bit
Java version	Java™ SE 1.6.0_24-b07

6.1 ProcessCreator Results

Table 3 summarizes the results obtained for the process creation benchmark. For each simulator, two runs of the benchmark try to create 2^{12} (4,096) and 2^{20} (1,048,576) processes, respectively, measuring the total execution time. In the second run, as was to be expected, thread-based simulators fail after the creation of a few thousands of processes (figures in italic in the table). Reducing the thread stack size to 64 KB makes it possible to rise this limit, but on the machines used for our tests 2^{20} processes remain out of the reach of thread-based simulators. As discussed in Section 2, this limitation is linked to the maximum number of native threads that the platform allows to execute.

The only simulators that can manage the 2^{20} processes required by the `ProcessCreator` benchmark are JADES and D-SOL. However, the performance of the latter is very low (around 9 minutes of running time for the simulation, about 70 times the execution time of JADES). For 4,096 processes, it can also be observed that JADES is significantly faster than the thread-based simulators.

6.2 PingPong Results

Figure 2(a) shows the results of the base `PingPong` benchmark with no additional stack frames ($stackDepth = 0$). As D-SOL is much slower than the other simulator in this test, its results are not plotted in the figure for the sake of clarity, and are reported separately in Table 4. As in the figure the x axis scale is logarithmic, a linear function of x has been plotted to provide a reference. It can

Table 3: ProcessCreator benchmark results.

Simulator	Thread stack size	# proc. requested	# proc. created[a]	execution time (ms)[a]
JADES	N.A.	4,096	4,096	192
	N.A.	1,048,576	1,048,576	7,436
JADESThreads	N.A.	4,096	4,096	2,103
	256KB	1,048,576	**7,151**	**4,239**
	64KB	1,048,576	**23,292**	**29,390**
javasimulation	N.A.	4,096	4,096	2,291
	256KB	1,048,576	**6,709**	**2,929**
	64KB	1,048,576	**23,279**	**28,880**
Desmo-j	N.A.	4,096	4,096	3,354
	256KB	1,048,576	**7,288**	**5,558**
	64KB	1,048,576	**24,195**	**33,729**
D-SOL	N.A.	4,096	4,096	2,451
	N.A.	1,048,576	1,048,576	534,505
SSJ	N.A.	4,096	4,096	1,275
	256KB	1,048,576	**6,708**	**2,896**
	64KB	1,048,576	**23,342**	**28,052**

[a] Results in bold indicate the maximum number of processes created and the (partial) execution time when the simulator fails.

be observed that Desmo-J follows strictly its linear pattern. JADESThreads, Javasimulation and SSJ exhibit a very similar less-than-linear growth of execution time, but in the long run their execution time grows faster than JADES, which outperforms all the other simulators (by a factor of 2.5 when $simTime = 1,000,000$ and growing). Figure 2(b)

Table 4: PingPong execution times for D-SOL.

SimTime	Exec. time (ms)
1,000	1,347
10,000	3,917
100,000	17,066
1,000,000	147,372

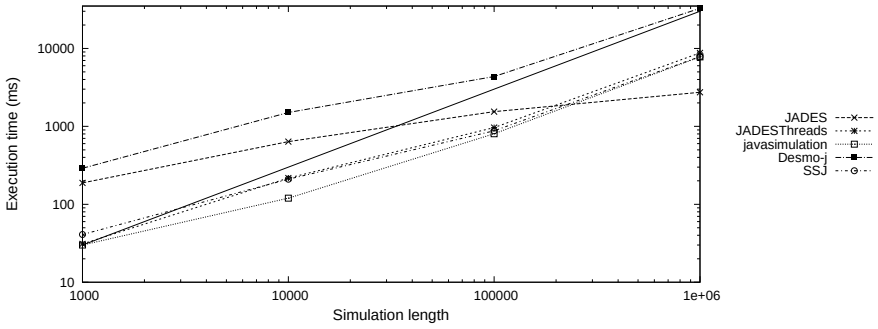
shows how the simulators behave when the stacks are deepened by using values of the benchmark parameter $stackDepth > 0$. It can be observed that while thread-based simulators are unaffected by the increase of stack depth, JADES execution times grow linearly with $stackDepth$. This phenomenon is due to the way JavaFlow implements continuations. As continuations involve saving and restoring of the stack, context switches introduce an overhead proportional to the stack size. On the other hand, thread-based simulators are not affected by the same problem, since every thread has its own private stack and so context switches are executed in constant time. Once again, the D-SOL results are not plotted to avoid breaking the graph scale: its execution time showed to be independent of the stack depth and very close to the last row of Table 4. Figure 2(b) makes it possible to observe that JADES shows better performance than the other simulators only for $stackDepth < 50$.

In conclusion, we can deduce that only simulation models in which the level of recursion is low (or, equivalently, call nesting is shallow) will benefit from the use of continuations.

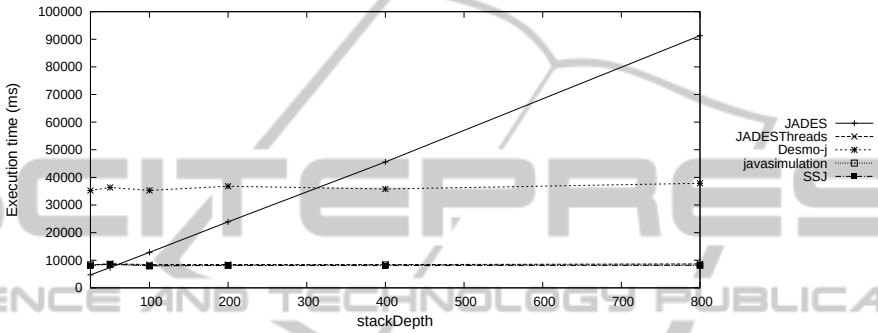
6.3 MM1Queue Results

Figure 2(c) and 2(d) show the plot of execution times for the M/M/1 benchmark. The arrival rate has been set to 1.0. Service rate has been set to 2.0 for obtaining the stable queue behavior, and to 0.1 for the unstable one. The stable queue produces a model in which the number of active processes in the system is small, whereas the unstable one implies a growing number of processes as simulation time flows. Figure 2(c) shows the behavior of the four fastest simulators with a stable queue. D-SOL and Desmo-j have not been included in the plot as their execution times are considerably higher than the others. SSJ and JADES are the fastest simulators for this benchmark. The latter gains advantage as simulation time grows.

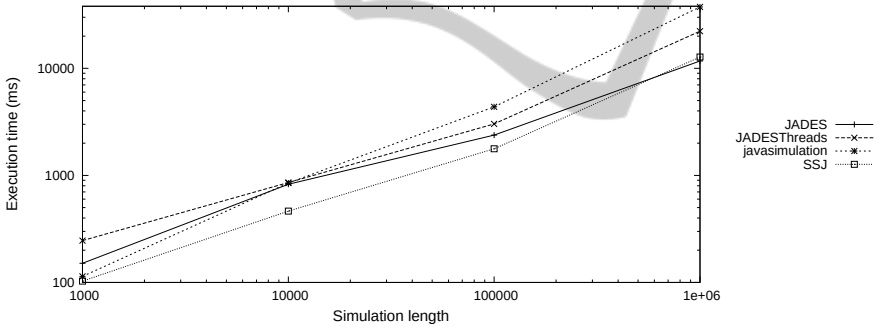
Figure 2(d) shows execution times in unstable conditions. Once again, Desmo-j had to be omitted from the graph as it is slower than the others by a factor of 4. The best performance is exhibited by D-SOL (typically very slow for other benchmarks) and by JADES, which is the clear winner here, thanks to the very slow growth of execution times as $simTime$ increases. It should also be pointed out that for $simTime > 20,000$, the thread-based simulators crash rapidly, as they cannot create the required



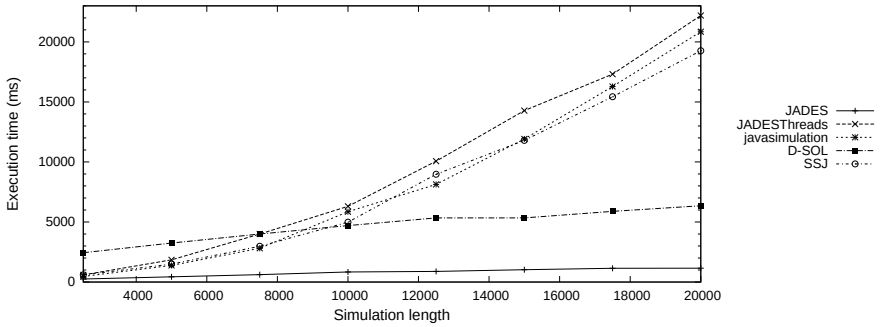
(a) `PingPong` benchmark: execution time vs. Ssimulated time.



(b) `PingPong` benchmark: effect of stack depth on execution time.



(c) `MM1Queue` benchmark: stable queue (`iarTime` = 1.0, `srvTime` = 0.5)



(d) `MM1Queue` benchmark: unstable queue (`iarTime` = 1.0, `srvTime` = 10.0).

Figure 2: `PingPong` and `MM1Queue` benchmark results.

number of threads. In the same 20 seconds needed by the other simulators to simulate 20,000 time units, JADES is able to simulate 2,000,000 time units, an improvement of two orders of magnitude.

7 CONCLUSIONS

Threads are not the only option when implementing process-oriented discrete-event simulations in Java. Our tests have shown that the continuation-based JADES simulator is a viable alternative, which can lead to significant performance gains in many cases. In particular, the use of continuations turns out to be advantageous for applications with stack of moderate depth. On the other hand, large stacks to be saved and restored are managed more efficiently by thread-based simulators. In fact, this behavior is partly due to the limited efficiency of the continuation library Javaflow used for JADES development. Things are likely to change if optimized continuations are introduced into the standard Java platform: the chances of this introduction get higher and higher as the language and its ecosystem evolve (Stadler, 2011).

To the best of our knowledge, this paper is the first contribution in the literature that proposes a benchmark suite for discrete-event process-oriented simulators and makes a comparative evaluation of threads and continuations as means for the implementation of such simulators in Java. Our future work will focus on devising parallelization strategies for the simulator, and on the implementation of the continuation-based simulation library in C language. Application-level benchmarks with complex models will also be added to the base benchmark suite presented here. JADES will be made publicly available soon with open source license at http://deal.ing.unisannio.it/perflab/projects/jades/.

REFERENCES

Derrick, E., Balci, O., and Nance, R. (1989). A comparison of selected conceptual frameworks for simulation modeling. In *Proc. of the 21st Winter Simulation Conference*, pages 711–718.

Fujimoto, R. M. (1990). Performance of Time Warp under synthetic workloads. In *Proc. of 22nd SCS Multiconference on Distributed Simulation*.

Helsgaun, K. (2000). Discrete Event Simulation in Java. http://akira.ruc.dk/∼keld/research/JAVASIMULATION/JAVASIMULATION-1.0/docs/Report.pdf.

Howell, F. and McNab, R. (1998). SimJava: a discrete event simulation package for Java with applications in computer systems modelling. In *Proc. of the First International Conference on Web-based Modelling and Simulation*.

Jacobs, P., Lang, N., and Verbraeck, A. (2002). D-SOL; a distributed Java based discrete event simulation architecture. In *Proc. of the 34th Winter Simulation Conference: exploring new frontiers*, pages 793–800.

Kiviat, P. (1969). *Digital computer simulation: computer programming languages*. Rand Corp.

Kunert, A. (2008). Optimistic parallel Process-Oriented DES in Java using Bytecode Rewriting. In *Proc. of MESM 2008*, pages 15–21.

Lechler, T. and Page, B. (1999). DESMO-J: An object oriented discrete simulation framework in Java. In *Proc. Simulation in Industry '99 - 11th European Simulation Symposium '99*, pages 119–124. SCS publ.

L'Ecuyer, P. and Buist, E. (2005). Simulation in Java with SSJ. In *Proc. of the 37th Winter Simulation Conference*, pages 611–620.

Marlin, C. (1980). *Coroutines: A Programming Methodology, a Language Design and an Implementation*, volume 95 of *Lecture notes in computer science*. Springer.

Miller, J., Nair, R., Zhang, Z., and Zhao, H. (1997). JSIM: A Java-based simulation and animation environment. In *Simulation Symposium, 1997. Proc.. 30th Annual*, pages 31–42. IEEE.

Nance, R. (1996). A history of discrete event simulation programming languages. In *History of programming languages—II*, pages 369–427. ACM.

Ortega-Ruiz, J., Curdt, T., and Ametller-Esquerra, J. (2004). Continuation-based mobile agent migration. http://hacks-galore.org/jao/spasm.pdf .

Perumalla, K. and Fujimoto, R. (1998). Efficient large-scale process-oriented parallel simulations. In *Proc. of the 30th Winter Simulation Conference*, pages 459–466.

Reynolds, J. (1993). The discoveries of continuations. *Lisp and symbolic computation*, 6(3):233–247.

Schwetman, H. (2001). CSIM19: a powerful tool for building system models. In *Proc. of the 33nd Winter Simulation Conference*, pages 250–255. IEEE.

Stadler, L. (2011). Serializable coroutines for the HotSpot^{TM} Java virtual machine. Master's thesis, Johannes Kepler University Linz, Austria.

Weatherly, R. and Page, E. (2004). Efficient process interaction simulation in Java: Implementing co-routines within a single Java thread. In *Proc. of the 36th Winter Simulation Conference*, pages 1437–1443.