Proceedings of the 2004 Winter Simulation Conference
R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds.

SINGLE-THREADED SPECIFICATION OF PROCESS-INTERACTION FORMALISM IN JAVA

Peter H.M. Jacobs
Alexander Verbraeck

Department of Systems Engineering
Delft University of Technology
Delft, THE NETHERLANDS

ABSTRACT

In order to support the conceptualization and specification of simulation models of complex systems, several *formalisms* or *world views* exist. Petri nets, differential equations, discrete event system specification and process interaction are typical examples. Throughout the last decade many have attempted to implement the *process interaction* formalism in Java. These initiatives mostly resulted in multi-threaded simulation languages in which a `Process` extends a `Thread`. These threads are then sequentially suspended and resumed. The article "Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?" (Sun Microsystems 1999) implicitly ended most of these deadlock prone initiatives. This paper introduces a unique *single-threaded* implementation of this world view by introducing a Java-based Java *interpreter*, which is used only to interpret pausable processes. This interpreter supports all Java programming constructs and hopefully serves as a cornerstone for renewed development of process oriented Java based simulation languages.

1 INTRODUCTION

As introduced by (Kiviat 1967) and (Fishman 1973), simulation models have two orthogonal types of structure: a *static* and a *dynamic* structure. Where the static structure represents the *state* of the simulation model, the dynamic structure represents its *time-dependent* state transitions.

(Nance 1981) introduces a small set of basic definitions which carefully distinguish the time and state relationships. He argues that with this given set of definitions any simulation model representation can be constructed. Nance's starting point is the object-oriented system description of (Kiviat 1969). A simulation model is considered to be comprised of objects that are described in terms of their attributes and values.

According to (Nance 1981), any simulation model must have an *indexing attribute*, that is, an attribute that enables state transitions in the model. In time-based simulation, *system time* (or *simulation time*) is used as such an attribute. Based on this definition, Nance defines the following concepts:

An *instant* is defined as a value of system time at which the value of an attribute can be altered. An *interval* is the duration between two successive instants and a *span* is the concatenated succession of intervals. The *state* of an object is the enumeration of all attribute values of an object at an instant.

Time and state concepts are related through a further set of definitions; an *event* is a change in the state of an object. A *process* is the succession of states of an object over a span.

Where this set of definitions uniquely describes the static structure of a simulation model, there are different approaches to describe the dynamic structure of a simulation model. These approaches are referred to as *world view*, *formalism* or *modeling construct*. For discrete event simulation, three classical formalisms are known: *event scheduling*, *activity scanning* and *process interaction* (Fishman 1973). In addition (Balci 1988) presents a unique strategy for the implementation of each particular formalism.

The core notion underlying this paper is that all three formalisms have a certain value with respect to providing guidance to a modeler in the conceptualization and specification of the dynamic structure of a system. Since the choice for a particular formalism depends both on the system under investigation and on the view of the modeler, most simulation languages support more than one formalism.

Throughout the last decade many have attempted to implement the *process interaction* formalism in Java. These initiatives mostly resulted in multi-threaded simulation languages in which a `Process` extends a `Thread`. These threads are then sequentially suspended and resumed. The article "Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?" (Sun Microsystems 1999) implicitly ended most of these deadlock prone initiatives.

As revealed by the title, this paper introduces a *single threaded* implementation of the process interaction formalisms in Java. This paper is organized as follows: section 2 starts with an introduction of formalisms based on the concept of *locality*. Section 3 provides an introduction in the event scheduling formalism and section 4 provides a formal definition of the process interaction formalism. In this section the downside of current multi-threaded implementations is discussed. Section section 5 introduces several potential approaches to a single threaded implementation of this formalism.

Section 6 presents the *interpreter* as the preferred implementation of the process interaction formalism in Java. Section7 presents a benchmark of the interpreter; section 8 finishes with conclusions and recommendations for further research.

2 FORMALISMS: MODELS FOR DYNAMIC STRUCTURE DESCRIPTION

In discrete event simulation, three classical formalisms exist to describe the dynamic structure of a system under investigation. These formalisms are meta-models of dynamic structure representation and as such are models too (Vangheluwe 2002). The meta-model of these formalisms is the selected set of state and time definitions introduced in the first section.

The differences between these formalisms is based on the concept of *locality* (Weinberg 1971). Locality refers to the degree to which all relevant parts of a model are found in the same place. The three classical formalisms for discrete event simulation are:

- *Event Scheduling*: the event scheduling formalism provides *locality of time*: a modeler defines events at which discontinuous state transitions occur. Since object oriented programming languages provide methods to effectuate these state transitions, event scheduling results in *scheduled method invocation* (Jacobs et. al. 2002). An event can cause (by means of scheduling) other events to occur. As described by (Balci 1988) the strategy for the event scheduling world view is to repeatedly select the earliest scheduled event, to advance the system time to the execution time of that event and to invoke the method specified by the event.
- *Activity Scanning*: the activity scanning formalism provides *locality of state*. Besides the events described in the event scheduling formalism, a modeler may define *contingent events*, which occur when some condition is met (Nance 1981). The simulation strategy for this formalism differs from the strategy of the event scheduling world view in that the condition associated with contingent events must be evaluated repeatedly. This strategy is generally considered less efficient with respect to computational execution (Balci 1988).
- *Process Interaction*: the process interaction formalism provides *locality of object*: each process in a simulation model specification describes its own *action sequence* (Overstreet 1986). This formalism thus reflects the *autonomy* of an individual process (i.e. life cycle) and the *concurrency* in the execution of distinct processes. In its action sequence, a process must have the ability to *suspend* and *resume* operation.

A simulation language is said to be *event-oriented*, *activity-oriented*, or *process-oriented* whenever it supports simulation models which express their dynamic structure according to the *event scheduling*, *activity scanning* or *process interaction* formalism. The aim of several (Java based) simulation languages is to support multiple formalisms.

3 EVENT SCHEDULING

As introduced in the previous section, event scheduling is accomplished by a strategy of *scheduled method invocation*. As becomes clear in section 5, the approach of this paper is to map (or embed) the process interaction formalism on the event scheduling formalism. For this reason a good understanding of the later formalism is required and a more detailed overview is presented here.

According to the definitions introduced in section 1, *state transitions* occur at *instants*. In an *object oriented* system representation, an object can change the state of another object either by directly changing one of its attribute values or by invoking a method on this object which internally changes one or more of its attribute values.

Most object oriented literature advises modelers to limit the visibility and accessibility of attributes (Eckel 2003). This *design paradigm* (or *pattern*) is referred to as *encapsulation*. Externally invoked state transitions on an object are therefore preferably accomplished by method invocation.

In contrast with simulation models, software systems only have an *implicit* notion of time. The only time used in software systems is the *real* time, which is also referred to as *wall-clock* time. A good example of this implicit notion of time is the time-out used in most IO-connections.

As presented in section 1, simulation models use an *explicit* notion of time. Whenever a state change on an object is scheduled on an instant of the system time, direct method invocation cannot occur: the invocation must be scheduled.

In order to illustrate this difference, we consider a customer-retailer relation in which a customer orders a specific product. The `Order` class is defined with two attributes : `customer`, and `product`.

The direct invocation of the publicly accessible `receiveOrder` method on an instance of the `Retailer` class by an instance of the `Customer` class might look as follows:

```
public class Customer {
  private void createOrder()
  {
    Order order = new Order(this,"TV");
    retailer.receiveOrder(order);
  }
}
```

The strategy of *scheduling method invocation* used in simulation is required whenever we assume that our customer *waits* for a particular duration before actually placing the order. Our pseudo-code might look like:

```
public class Customer {
 private void createOrder()
 {
  Order order = new Order(this,"TV");

  //the system time for ordering
  double orderTime =
        simulator.getTime()+ 2.0;

  //the ordering process is scheduled
  Event event = new Event(orderTime,
     retailer, receiveOrder, {order});
  simulator.schedule(event);
 }
}
```

The event scheduled on the simulator is defined as a tuple $< \tau, p, o, m, A >$. In this definition $\tau : \tau \in R^{+}_{0,\infty}$ is the execution time of the event, and $p : p \in [Priority.MIN, Priority.MAX]$ represents the priority of an event. In the above code p is implicitly set to $Priority.DEFAULT$. The object on which the method m must be invoked with the set (array) of arguments in A is described by o.

4 PROCESS INTERACTION

This section discusses the process interaction formalism. In order to discuss this formalism in a language independent fashion, two modeling frameworks are used: Zeigler's DEVS framework (Zeigler et. al 2000) and the unified modeling language (UML) (Booch et. al 1999). Where the DEVS framework assists in understanding what we will refer to as the *control state*, the UML framework draws the requirements for further specification.

An autonomous system with a discrete event structure is presented in the DEVS formalism as

$$< X, \Omega, Y, \delta_{int}, \delta_{ext}, \lambda, \tau > \quad (1)$$

In equation (1) X is the set of input values of a model. Ω defines the initial state of the model, and Y is the set of output values to a model. $\delta_{int} : s \rightarrow s$ represents the *internal transition function*, $\delta_{ext} : s * X \rightarrow s$ represents the external transition function. In these definitions s represents a state of the model at an instance and $\lambda : s \rightarrow Y$ is the output function of the model. The *time advance function* is represented by $\tau : R^{+}_{0,\infty}$. A very illustrative and intuitive interpretation of this formal representation is given by (Cota et al. 1992): For a given state s, $\tau(s)$ is the amount of time the system will remain in state s until a next internal event occurs. The sequential state then changes from s to $\delta_{int}(s)$. If in between $\tau(s)$ an external event occurs, the system changes from s to $\delta_{ext}(s)$.

In the terminology of the DEVS framework, a formalism is an approach to structure the transition functions δ_{int} and δ_{ext} and time advance function τ. (Zeigler et. al 2000) provides mathematical formalizations of these functions for all three classical formalisms.

In the process interaction formalism, a modeler defines processes. The formal distinction between a process and an object is the fact that a process has a *control state* attribute. In its control state a process stores its reactivation point in its sequence of activities. The requirements for a `Process` class become:

Process
-controlState : ControlState
#hold(duration : double) : void #process() : void +resume() : void #suspend() : void

- The `Process` class is *abstract*. Processes as such cannot be instantiated. Classes extending `Process` are required to implement the abstract `process` method and to specify the actual sequence of activities.
- The `suspend`, `hold`, and `process` methods have *protected* and therefore limited visibility. They cannot be invoked publicly. It is important to understand that unlimited visibility would conflict with the *locality on object* and thus with a required *encapsulation* of the process.
- The `resume` method is *public*, which delineated unlimited visibility. This is required since an object cannot `resume` itself in a suspended state.

In Java it is far from trivial to access the control state of an object. Until today the approach chosen by all process-oriented simulation languages implemented in Java was to circumvent its access by implementing the `Process` class on top of a Java `Thread`.

Since a Java thread wraps an *operating system* thread, and operating system threads provide methods to suspend and resume, it is a fairly easy task to specify a multi-threaded process interaction formalism in Java.

Besides the advantages of easiness and apparent correctness, this approach has some very strong disadvantages:

- Sun Microsystem published "Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?" (Sun Microsystems 1999). In this article Sun Microsystems argues that invoking these methods leads to deadlock prone code. With respect to re-producable experiments it must further be understood that the sequence of execution of multi-threaded processes depends on the operating system scheduler. This makes the approach platform, and context dependent.
- Operating system threads are expensive with respect to CPU resources required for their instantiation and the amount of allocated memory. An operating system thread typically keeps track of open files, file and user permissions, management information, etc. A personal computer can therefore only handle between 1000-5000 concurrent threads. Models specified in process-oriented simulation languages are therefore limited to an equal amount of concurrent processes.
- Since a Java thread only wraps an underlying operating system thread, Java threads are not *serializable*. Processes extending a Java thread can therefore not be streamed over a network (distributed simulation) nor stored to file (model persistency).

Based on these disadvantages, it became our goal to explore possible approaches of single threaded implementations.

5 SINGLE THREADED IMPLEMENTATION OF PROCESS INTERACTION

This section explores single-threaded approaches to implement the process interaction formalism in Java. In the terminology of the DEVS framework, the goal of this section is to discuss possible approaches to implement the `Process` class with the transition functions and time advantage function corresponding to the *event scheduling* formalism! (Vangheluwe 2002) and (Zeigler 2000) would refer to *embedding* the process interaction formalism in the event scheduling formalism.

5.1 Method Splitting

The first approach discussed here is called *method splitting*. This approach is based on the idea of rewriting the model in an event-scheduling formalism by splitting the process method. The approach is illustrated by the customer-retailer example of section 3. In the process interaction formalism, the process of our customer might look like:

```
protected void process() {
  Order order = new Order(this,"TV");
  this.hold(2.0);
  retailer.receiveOrder(order);
}
```

In order to embed the process interaction into the event scheduling formalism, the code is splitted around all `suspend` and `hold` statements. The above invocation of `hold(2.0)` is removed and a new method is inserted. The event-scheduling code becomes:

```
protected void process() {
  Order order = new Order(this,"TV");
Event event = new Event(2.0,"processA",order);
  simulator.schedule(event);
}

private void processA(Order order) {
  retailer.receiveOrder(order);
}
```

We conclude that the strategy of this approach is to split around `hold` and `suspend` methods, and to schedule the second part of a splitted method on every `hold`. In order to transfer the local variables used in the original method, they become *parameters* for these newly created methods.

Method splitting can occur in three modes. Firstly a modeler can rewrite a model by hand. This is a time consuming task, which probably results in specifying a model in the event scheduling formalism in the first place. The second option is to pre-compile a model by a pre-compiler which translates a process interaction model into an event scheduling model. The later is then compiled into byte-code. The last option is to translate an already compiled model process interaction model into an event scheduling one. This is called *byte code engineering*.

The second and third options require a unique translating scheme. For the following reasons, the creation of such a scheme is far from trivial:

- A first problem arises whenever the process does not directly invoke `suspend` or `hold` methods, but invokes another method which invokes either `suspend` or `hold`. Splitting this method does not work since the return given by the first part of the splitted method does not reach the simulator.

The result is received by the `process` method which incorrectly assumes successful completion and continues its next operation.

- Recursive methods, or methods invoked through reflection are very difficult, if not potentially impossible to split.

The result of these disadvantages is to conclude that though *method splitting* works for easy situations, it results in constraints on the set of language constructs available to the modeler. Another conclusion is that un-allowed language constructs are not likely to be noticed by general purpose compilers.

5.2 Stack swapping

In order to understand the concept of *stack swapping*, this paper introduces the Java virtual machine specifications of method invocation.

Whenever Java source code is compiled, a unique `.class` file is generated reflecting the compiled byte code for the particular class. Though the format of this file may seem unreadable, it consists of integers, shorts, utf-8 characters, etc.

A `.class` starts with several constants defining the name of the class, its superclass, and the set of interfaces it implements. Then the *constantpool* is specified. The constants in this pool are used by the fields and methods described in the `.class` file and merely serve the purpose of preventing unnecessary bytes in the file (a pointer to the 3^{rd} constant in a constantpool of 255 positions occupies only 1 byte, where duplicating a complete string will most certainly not).

Then all fields and methods of the class are described by their signature (including, name, (return) type, and possible parameters). The body of a method is specified as a list of sequential *assembly operations*. The Java virtual machine specification (Lindholm 1999) introduces all most 200 of such operations. Whenever a method is invoked, a Java virtual machine simply sequentially executes this list of operations.

In software terms, the invocation of a method is called a *frame* and a *stack* represents a last-in-first-out (LIFO) stack of objects. In order to execute the invocation of a method, a Java virtual machine uses two types of stacks. One is a *frame stack*, exclusive to a thread. The *operand stack* represents a stack for operations and is exclusive to each frame (or method invocation). As will become obvious from the following example, another object used in the execution of a frame is the pool of *localvariables*. This pool holds all local variables of a method (including the method-parameter values).

The execution of byte code can perhaps best be described by a simple example. Consider a class defining two mathematical operations `square` and its more general `pow`.

```
public static double square(double a) {
 return pow(a,2);
}

public static double pow(double a, int b) {
  while(b>1)
  {
   return a*pow(a,b-1);
  }
  return a;
}
```

If `square(4.0)` is invoked, the main thread of execution creates a new *frame* for this invocation and pushes this frame on top of its *frame stack*. Then it starts the execution of this frame resulting in the execution of the following 4 assembly operations:

```
DLOAD_0 //stack.push(localvariable(0))
ICONST_2 //stack.push(constantpool(1))
INVOKESTATIC //invoke
  pow(stack.pop(),stack.pop())
DRETURN //return stack.pop()
```

Based on the source-code presented in this example, it is not difficult to understand that the `INVOKESTATIC` operation invokes the more general `pow(a,2)` method. A new frame is thus created and pushed on top of the previously created frame representing the `square` invocation. The thread first executes this newly pushed frame before it resumes the `DRETURN` operation. The operations of the `pow` frame are:

```
ILOAD_2 //stack.push(localvariable(2))
ICONST_1 //stack.push(constantpool(1))
IF_ICMPLE //if(..<..)
DLOAD_0 //stack.push(localvariable(0))
ILOAD_2 //stack.push(localvariable(2))
ICONST_1 //stack.push(constantpool(1))
ISUB //stack.push(stack.pop()-stack.pop())
INVOKESTATIC //invoke
  pow(stack.pop(),stack.pop())
DLOAD_0 //stack.push(localvariable(0))
DMUL //stack.push(stack.pop()*stack.pop())
DRETURN //return stack.pop()
DLOAD_0 //stack.push(localvariable(0))
DRETURN //return stack.pop()
```

Since the `pow` method is a recursive method, the number of frames created for its invocation is equal to `b-1`. Whenever a frame returns a value, this value is pushed on its *parent* frame. Then the frame is removed from the framestack and the execution of the parent frame resumes.

The approach introduced as stack swapping works as follows: assume a `Process` is paused with a `hold` statement. The simulator thread now:

- pops its framestack up to the point where the `process` method of the `Process` was invoked. Since the `process` method is `void`, no return value is pushed to the parent frame.
- stores this popped part of its framestack as the *control state* of the `Process`.
- stores the index of the last executed operation as *reactivation point* of this control state.
- continues the execution of its framestack which results in executing the next scheduled simulation event. The `suspend` is now successfully accomplished.

Whenever the simulator thread resumes a `Process` it pushes the *control state* of the process on its framestack and resumes its new top frame on the specified *reactivation point*. In contrast with both the *multi threaded* and *method splitting* approaches, *stack swapping* has no constraints; it is therefore the preferred approach for the specification of the *process interaction*. One remaining problem is that the Java programming language does not provide any language constructs to access neither the framestack of a thread, nor its local variables.

6 JAVA INTERPRETER

As described in the previous section, *stack swapping* is the preferred approach to implement the *process interaction* formalism. The only hurdle is now to design a library which provides control over the execution of assembly operations. The Java *interpreter* presented in this paper provides this!

The approach chosen in this paper is to develop an interpretation library. This library can best be seen as a virtual machine implemented in Java. In order to do so, a number of objects were designed and implemented. First of all, the standard `java.util.Stack` was used for both the operand stack and the framestack. Both the `constantpool` and the `localvariables` were specified as ordinary arrays.

A bit more problematic was the specification of the class-file. Where the `Class` class provides meta-information of an `Object` we developed the `ClassDescriptor` as a class which provides the required meta-information of a `Class`. A `ClassDescriptor` holds `MethodDescriptors` which hold `Operations`.

The most time consuming, but straightforward task was to specify all 200 assembly operations. An example of such operation is the `DMUL` operation which multiplies two doubles. Its `execute` method is implemented as follows:

```
public void execute(
 final OperandStack stack,
 final Constant[] constantPool,
 final LocalVariable[] localVariables)
 {
  double value2 = ((Number) stack.pop())
                  .doubleValue();
  double value1 = ((Number) stack.pop())
                  .doubleValue();
stack.push(new Double(value1 * value2));
 }
```

The approach for the implementation of the process interaction formalism in the DSOL suite for simulation becomes:

- a `Process` class is specified as introduced in the class diagram in section 4. This class has one attribute called `framestack` which type is `java.util.Stack`.
- the constructor of the `Process` schedules on the simulator the *interpretation* of the `process` method to be invoked at `time=0.0` . The interpretation is thus scheduled!
- Starting at `time=0.0`, the `process` is interpreted sequentially and hierarchically. In order to prevent any overhead, a tight invocation scheme is used: only invocation of pausable methods is interpreted. All other invocations are not interpreted, but directly invoked through reflection.

7 RESULTS

The previous section described the interpretation of Java byte code. In order to see whether this approach indeed solves the scalability, serializability and reproducibility disadvantages without creating an unacceptable loss of performance, a customer delay was specified in both the event scheduling formalism and in the process interaction formalism.

The pseudo-code for the customer implemented in the event scheduling formalism is:

```
 public void process() {
double duration=holdDistribution.draw();
  this.simulator.scheduleEvent(
   new SimEvent(
    this.simulator.getSimulatorTime()
     + duration,
    "resume"));
 }

 private void resume(){}
```

The pseudo-code for the customer implemented in the process interaction formalism is:

```
 public void process() {
double duration=holdDistribution.draw();
 this.hold(duration);
 }
```

The computational performance between the process interaction and event scheduling world-view is illustrated in table 1. The first column of this table illustrates the number of simultaneously created entities at `time=0.0`. The second and third column present the computational execution time in milliseconds of the event scheduling (ES) and process interaction (PI) formalisms. The fourth column presents the relative speed of event scheduling versus process interaction. The benchmark presented was conducted on a Intel

Table 1: Benchmark of Process Interaction

#	DEVS (millisec)	PI (millisec)	PI/DEVS
1	18	358	19.91
10	19	352	18.53
100	30	401	13.39
1000	111	763	6.88
10000	475	3077	6.47
100000	3211	20012	6.23

Pentium III Mobile CPU 1200MHz, 512 Mb ram, SuSE 8.2 (Linux) operating system and Sun's Blackdown-1.4.1-01 JRE. Conducting an equal test with the multi threaded process interaction simulation language `Silk` (Kilgore 2000) resulted in an irrecoverable stack overflow after 1320 processes were instantiated.

As the decreasing `PI/DEVS` factors in table 1 illustrate, a performance loss of around 300 millisecond is due to an initial penalty for parsing the `.class` file and constructing appropriate assembly operations. Based on the `1000`, `10000`, and `100000`, we conclude that the actual interpretation of a process is around 6 times slower than its event scheduling opponent.

8 CONCLUSIONS

This paper introduces a single threaded implementation of the process interaction formalism in Java. The development of this library is considered vital since it replaces the less scalable, deadlock-prone, platform-dependent, and multi-threaded approaches used so far.

The approach presented here extends Java's reflection concept to *class reflection*. Where Java provides language constructs to introspect an object, resolve its class, and indirectly invoke its methods, the approach chosen here is to introspect a class, resolve *method assembly operations* and invoke these operations indirectly.

Suspension and resuming simulation processes is implemented as follows: in the constructor of a `Process` schedules the interpretation of its `process` method at `simulation time=0.0`. Whenever interpretation starts, the list of assembly operations is sequentially executed until new invoke operations occur.

If the target for this new operation is an instance of `Process`, the operation is interpreted. If not, indirect invocation is not required and therefore not used. Java's reflection library is then used to minimize computational overhead.

A further conclusion is that since process interaction is based on a single-thread (i.e. the simulator thread), deadlocks caused by synchronization of objects over processes cannot occur. The penalty of using interpretation is around 300-800 milliseconds per `Class` and around 6 times slower than its event scheduling "opponent".

9 OBTAINING THE SOFTWARE

Both the process-interaction formalism and the interpreter are part of the DSOL suite for simulation (see Jacobs et. al 2002, Lang et. al. 2003). DSOL is published under the General Public License. More information on the license can be found at `<http://www.gnu.org/copyleft/gpl.html>`.
The DSOL project description can be found at `<http://www.simulation.tudelft.nl>`
and the software can be downloaded from `<http://sourceforge.net/projects/dsol/>`

AUTHOR BIOGRAPHIES

PETER H.M. JACOBS is a PhD. student at Delft University of Technology. His research focuses on the design of simulation and decision support services for the web-enabled era. His working experience within the iForce Ready Center, Sun Microsystems (Menlo Park, CA), and engineering education at Delft University of Technology founded his interest for this research. His e-mail address is `<p.h.m.jacobs@tbm.tudelft.nl>`.

ALEXANDER VERBRAECK is an associate professor in the Systems Engineering Group of the Faculty of Technology, Policy and Management of Delft University of Technology, and a part-time full professor in supply chain management at the R.H. Smith School of Business of the University of Maryland. He is a specialist in discrete event simulation for real-time control of complex transportation systems and for modeling business systems. His current research focus is on development of open and generic libraries of object oriented simulation building blocks in Java. Contact information: `<a.verbraeck@tbm.tudelft.nl>`.

REFERENCES

Balci O. 1988. The implementation of four conceptual frameworks for simulation modeling in high-level languages. In *Proceedings of the 20th conference on Winter simulation,* ed. M.A. Abrams, P.L. Haigh, J.C. Comfort, 287-295. ACM Press. San Diego, California, United States.

Booch G., J. Rumbaugh , and I. Jacobson. 1999, *The unified modeling language user guide*, Indianapolis, IN:Addison-Wesley.

Cota B.A., R.G. Sargent. 1992. A Modification of the process interaction world view. *ACM Transactions on modeling and computer simulation 2 (2): 109-129.*

Eckel, B. 2003. *Thinking in Java*. 3rd ed. Upper Saddle River, NJ:Prentice Hall

Fishman, G.S. 1973, *Concepts and methods in discrete event digital simulation*, New York: John Wiley and Sons, pp. 22-58, 1973.

Jacobs, P.H.M., N.A. Lang , A. Verbraeck. 2002 *DSOL; A Distributed Java based discrete event simulation architecture*, Proceedings of the 2002 Wintersim conference

Kilgore, R.A. 2000. Silk, Java and object-oriented simulation. In *Proceedings of the 32nd conference on Winter simulation,* ed. P.A. Fishwick, K. Kang, J.A. Joines, R.R. Barton, 246-252. Society for Computer Simulation International. Orlando, Florida, United States.

Kiviat, P.J. 1967. Digital computer simulation: modeling concepts. *RAND Memo RM-5378-PR*. RAND Corporation, Santa Monica, United States.

Kiviat, P.J. 1969. Digital computer simulation: computer programming languages. *RAND Memo RM-5883-PR*. RAND Corporation, Santa Monica, United States.

Lang N.A., P.H.M. Jacobs, A. Verbraeck. 2003. Distributed, open simulation model development with DSOL services. In *Proceedings of the 15th European Simulation Symposium,* ed. A. Verbraeck, V. Hlupic, R. Scoble. 210-218. SCS European Publishing House, Germany.

Lindholm T., F. Yellin, 1999. *The Java(TM) virtual machine specification*. 2nd ed. London, UK:Addison-Wesley.

Nance R.E. 1981. The time and state relationships in simulation modeling, *Communications of the ACM 24 (4): 173-179.*

Sun Microsystems. 1999. Why are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit deprecated? Available online via <http://java.sun.com /j2se/1.4.2/docs/guide/misc/> [Accessed August 3, 2004].

Overstreet C.M., R.E. Nance, 1986. World view based discrete event model simplification. *Modeling and Simulation Methodology in the Artificial Intelligence Era,* ed M.S. Elzas, T.I. Oren, B.P. Zeigler, 165-179. Amsterdam, the Netherlands.

Vangheluwe H., J. de Lara. 2002. Meta-models are models too. In *Proceedings of the 34nd conference on Winter simulation,* ed. J.L. Snowdon, J.M. Charnes, E. Yucesan, C-H. Chen , 128-135. Society for Computer Simulation

Weinberg, G.M. 1971, *The psychology of computer programming*. New York, NY: Van Nostrand Reinhold.

Zeigler B.P., H. Praehofer and T.G. Kim. 2000. *Theory of modeling and simulation. integrating discrete event and continuous complex dynamic systems*. 2d ed. San Diego, CA: Academic Press.