Mastering D-SOL: A Java based suite for simulation.

Peter Jacobs, Alexander Verbraeck and Stijn-Pieter van Houten

Delft University of Technology
Faculty of Technology, Policy and Management
Systems Engineering Group
Jaffalaan 5, 2528 BX Delft, the Netherlands

August 30, 2006

Contents

1 Introduction

1.1 Who should read this tutorial?

This tutorial explains and demonstrates the fundamentals of creating simulation models with DSOL. It provides a straightforward, no-nonsense explanation of the technology, Java classes, interfaces, simulation models, output statistics, etc.

Although this tutorial aims to provide a primer, it's not written for dummies. Readers are expected to have an understanding of simulation and Java. Before reading this tutorial, you should be fluent in the Java programming language and have some practical understanding of simulation. If you are unfamiliar with the Java programming language, we recommend you to read *The Java(TM) Programming Language* [Arnold et al., 2000] or *The Java(TM) Tutorial* [Campione et al., 2000]. If you are completely unfamiliar with the concept of simulation we recommend you to study the *Handbook of simulation* [Banks, 1998] or *Simulation modeling and analysis* [Law and Kelton, 2000].

1.2 Copyright

Copyright (c) 2006 by Delft University of Technology, the Netherlands. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any forms by any means without the prior written permission of one of the authors.

1.3 Organization

We have organized this tutorial as follows:

- Section 1, *Introduction*. This is the section you are currently reading. It provides an overview of the tutorial.

- Section 2, *Overview of DSOL*. This section introduces the set of services that make up DSOL. In this section you will learn what the different DSOL services stand for, what they do and how they relate. This section ends with our first lines of code: an *'order generating customer'* example.

- Section 3, *Theory on simulation*. This section introduces simulation, state-time relations and its formalism dependent implementation.

- Section 4, *Basic simulation modeling*. This section introduces the steps involved in creating a simulation model in DSOL. It furthermore examines the coding examples which can be found at `http://www.simulation.tudelft.nl`.

- Section 5, *Conclusions and further reading*. This section concludes the tutorial with an overview of further reading material and a clear invitation

to get involved in the DSOL community. It also addresses a number of topics for future versions of this tutorial.

1.4   Software and Versions

In this tutorial the following software is used:

- DSOL, as of version 2.0.0. DSOL can be downloaded from sourceforge at `http://sourceforge.net/projects/dsol`. Appendix A describes how to download the source code of DSOL.

- Java, as of version 1.5.x. Be sure that DSOL is not used with prior versions of the Java programming language.

- We highly recommend using an integrated development environment for the construction of simulation models. This tutorial refers to the eclipse development environment (version 3.1.x) which can be downloaded from `http://www.eclipse.org`

1.5   Comments and Questions

DSOL is developed with a strong focus on providing high quality open source software. For this reason we are very interested in your experiences with it. Please provide us with as much feedback as possible on this tutorial, the code, the examples, the site, etc. Please address your comments and questions to:

> DSOL @ Systems Engineering / A. Verbraeck
> Delft University of Technology
> Jaffalaan 5
> 2628 BX, Delft, the Netherlands
> tel: +31(0)152781136
> fax: +31(0)152783429
> email: a.verbraeck@tudelft.nl

We furthermore invite you to join the mailing-lists at sourceforge. Subscriptions are available at `http://www.simulation.tudelft.nl`.

2   Welcome to DSOL

The promise of DSOL is to provide a set of Java-based services for simulation modeling with a clear focus on: distributed and service based model specification, web-enabled animation and output, multi-formalism modeling, and interoperability with external information systems. Before we start with our first model, we briefly introduce our view on simulation and explain why we see the need for yet another simulation environment.

2.1   What is Simulation?

In the Systems Engineering Group of Delft University of Technology, simulation is considered to be more than a tool. Simulation is seen as a method of inquiry which supports decision makers in the generation and evaluation of scenarios to deal with ill-structured problems [Sol, 1982]. Examples of ill-structured problems are infrastructure planning, supply chain coordination, coordination of business processes, etc. [Keen and Sol, 2003].

Simulation as a method of inquiry embodies a sequence of steps [Banks, 1998]. First of all a problem situation is conceptualized. Conceptualizing starts with the definition of a system under investigation. The system is the selected set of objects we choose to take into account for further analysis. Conceptualization results in several models of which the object-oriented class diagrams and process oriented IDEF-0 diagrams are typical examples.

The second step is to specify the system under consideration. In order to provide future what-if scenarios we specify the (potential) state changes of the system as a function of time. These state changes can either be continuous or discrete which leads to either continuous or discrete simulation models.

The third step is to design an experiment for our specified simulation model. In this experiment, we will need to specify the different scenarios (treatments) for the experiment and provide all run-control parameters (run length, number of replications, etc.).

After selecting an appropriate simulator, we execute the experiment and collect all the statistics. In order to both present the outcome of a simulation model, and to support validation and verification, animation is considered as a valuable asset. DSOL supports distributed 2D, text-based animation, and, on platforms where the Java3D library is available 3D animation as well.

2.2   Why DSOL?

The hypothesis underlying the development of DSOL is that the web-enabled era has provided us with the tools and techniques to specify the concepts of simulation in a set of loosely-coupled, web-enabled services. An underlying notion on the quality of most current simulation tools is that they lack:

- *usefulness* with respect to their embedded knowledge of the system under investigation. It is too difficult to link simulation tools to underlying transactional information systems.

- *usability* with respect to coordination of distributed conceptualization and specification of models representing the system under investigation.

- *usage* with respect to their integration and use within (portalled) organizational management information systems.

Besides the theoretical requirements stated in section 3.1 on page 18, this has lead to the following requirements for DSOL:

- N stakeholders should be supported over the internet (distributed or parallel usage).

- N model formalisms should be supported (not constrained to just one formalism).

- N experts should be simultaneously be supported in the specification of models (distributed or parallel development).

- Simulation services should be equipped to be linked to other web-enabled services (reporting, databases) (simulation in a distributed setting as a service);

- N processors should be accessible for the execution of experiments (distributed or parallel model execution).

2.3   Services not included in DSOL

For now, DSOL supports all activities and phases in this method of inquiry but conceptualization. DSOL therefore does not contain any modeling services or tools. You will have to stick to the whiteboard, Visio[1], or Rational Rose[2]. Some plans are currently made to start new research on this particular activity though. The future will point out to what level we will extend DSOL with a multi-actor conceptual modeling environment.

2.4   Anatomy of DSOL

The following services constitute the DSOL suite for simulation:

- *event*: this service provides a distributed asynchronous event mechanism. It provides a set of interfaces and classes for listeners and producers of events. This service forms the basis for all other services.

---

[1]Visio is a registered trademark of Microsoft corporation
[2]Rational Rose is a registered trademark of IBM corporation

- *logger*: this service extends Java's logging mechanism by providing a set of logging classes facilitating package based logging for the DSOL framework.

- *naming*: this service links DSOL to the JNDI framework. The Java Naming and Directory Interface (JNDI) provides Java technology-enabled applications with a unified interface to multiple naming and directory services. In these naming and directory services both model objects, and statistical output can be stored and shared.

- *jstats*: this service provides a set of continuous and discrete distribution functions and links DSOL to both external mathematical libraries and chart libraries.

- *dsol*: the core service providing a set of interfaces and classes for simulation. In the dsol service you will find various simulators, model formalisms, etc.

- *dsol-xml*: this service parses experiment definitions from xml into their Java representation. This service enables users to define experiments in xml. Schema validation is used to check all xml files.

- *dsol-gui*: a graphical user interface to be used with the DSOL suite for simulation.

In order to develop a suite of modular services, much attention is given to the dependencies between the different services. This is emphasized by the names of the different services. Services that are used by the simulation core, do not depend on simulation and therefore do not have the dsol prefix in their name. Services which are add-ons to *dsol* start with the prefix *dsol*. The dependencies are illustrated in table 1.

	event	logger	naming	jstats	dsol	dsol-xml	dsol-gui
event							
logger	●						
naming	●	●					
jstats	●	●					
dsol	●	●	●	●			
dsol-xml	●	●	●	●	●		
dsol-gui	●	●	●	●	●	●	

Table 1: dependencies between the services in DSOL

2.5 A first DSOL model

In this subsection we follow the steps of writing a simple discrete event model, and thereby introduce some of the fundamental concepts and facilities of DSOL. The task is to provide a customer who generates orders. This we consider as an initial step in the world of supply chain simulation.

2.5.1 Coding the model

The first step is to conceptually understand what to code: 3 different classes and an experiment. The 3 classes are the *Customer* class, the *Order* class and the *Model* class.

```
 1 /*
 2  * @(#) Order.java Dec 4, 2003 Copyright (c) 2002-2006 Delft University of
...
10  * The Order class as presented in section 2.5 in the DSOL tutorial.
...
22 public class Order
23 {
24     /** the product of an order */
25     private String product = null;
26
27     /** the amount of product to order */
28     private double amount = Double.NaN;
29
...
36     public Order(final String product, final double amount)
37     {
38         super();
39         this.product = product;
40         this.amount = amount;
41     }
42
...
47     public String toString()
48     {
49         return "Order[" + this.product + ";" + this.amount + "]";
50     }
51 }
```

As illustrated, there is nothing special about the *Order* class. An order is constructed with two arguments: a product and an amount (line 36).

```
 1 /*
 2  * @(#) Customer.java Dec 1, 2003
...
 7 package nl.tudelft.simulation.dsol.tutorial.section25;
...
27 public class Customer
28 {
29     /** the simulator we can schedule on */
30     private DEVSSimulatorInterface simulator = null;
31
...
37     public Customer(final DEVSSimulatorInterface simulator)
38     {
39         super();
40         this.simulator = simulator;
41         this.generateOrder();
42     }
...
47     private void generateOrder()
48     {
49         try
50         {
51             Order order = new Order("Television", 2.0);
52             System.out.println("ordered " + order + " @ time="
53                     + this.simulator.getSimulatorTime());
54
55             // Now we schedule the next action at time = time + 2.0
56             SimEventInterface simEvent = new SimEvent(this.simulator
57                     .getSimulatorTime() + 2.0, this, this, "generateOrder",
58                     null);
59             this.simulator.scheduleEvent(simEvent);
60         } catch (Exception exception)
61         {
62             Logger.warning(this, "generateOrder", exception);
63         }
64     }
65 }
```

In this example, we enable the construction of a *Customer* with a reference to the simulator it can schedule its behavior on (line 37). Because we are developing a discrete event model, the simulator must implement the *DEVS-SimulatorInterface*[3] which is defined in the *nl.tudelft.simulation.dsol.simulators* package (line 11) and deployed in the *dsol.jar*.

Whenever the *generateOrder* method is invoked on the Customer, the first step is to create the actual Order (line 41). The next steps include the creation and the scheduling of a simulation event. This simulation event takes care that

---

[3]DEVS stands for Discrete Event System Specification [Zeigler et al., 2000]

the *generateOrder* method (line 47) is called every 2 time units.

  Lines 56-58 create a *SimEvent* which implements a *SimEventInterface*. Both the interface and the class belong to the *nl.tudelft.simulation.dsol.formalisms.devs* package as part of the *dsol* services. A *SimEvent* schedules the invocation of a method on a target with arguments. The first argument in the constructor of the *SimEvent* refers to the time at which the invocation should occur (line 57), i.e. the current time plus two time units. The second argument (line 57) refers to the source object which creates the *SimEvent*. The third argument refers to the target on which to invoke the method named as the fourth argument. Line 58 finally refers to the arguments with which the method must be invoked. In this case, the call actually contains an instruction to execute the method *this.generateOrder* 2 time units after the current simulation time.

  Line 59 schedules the created simEvent on the simulator. The simulator is now able to sort and execute the event based on its scheduled execution time. Suppose the current time on the clock of the simulator is 10, then the scheduleEvent method instructs the simulator to call *this.generateOrder* at time 12 on the simulator's clock. Line 60 catches an exception: the *Exception*. The *Exception* may occur whenever the network link to the simulator fails (remember that distributed or parallel execution is one of the prime properties of DSOL) or whenever:

- a *SimEvent* is scheduled in the past. DSOL does not provide any roll-back functionality which would be needed to schedule behavior in the past.

- the method referred to as argument of the constructor on line 44 does not exist.

- this method exists but is not visible or accessible by the source named on line 42.

  The next step is to design the model. This is the class which is constructed and invoked at initialization of every replication. The model class must implement the *ModelInterface* which is defined in the *nl.tudelft.simulation.dsol* package and deployed in *dsol.jar*.

```
 1 /*
 2  * @(#) Model.java Dec 1, 2003
...
26 public class Model implements ModelInterface
27 {
...
31     public Model()
32     {
33         super();
34     }
```

```
35

...

40    public void constructModel(final SimulatorInterface simulator)
41      {
42          DEVSSimulatorInterface devsSimulator = (DEVSSimulatorInterface) simulator;
43          new Customer(devsSimulator);
44      }
45  }
```

Line 31 introduces an empty constructor which is a *must* for good quality simulation modeling. In order to understand this, remember that the simulator will reconstruct the model at the initialization of every replication. It is absolutely necessary not to remember model state over multiple replications. Using arguments in the constructor of a model conflicts with this concept. Therefore, it is a *design pattern* not to define arguments for the constructor of a model!
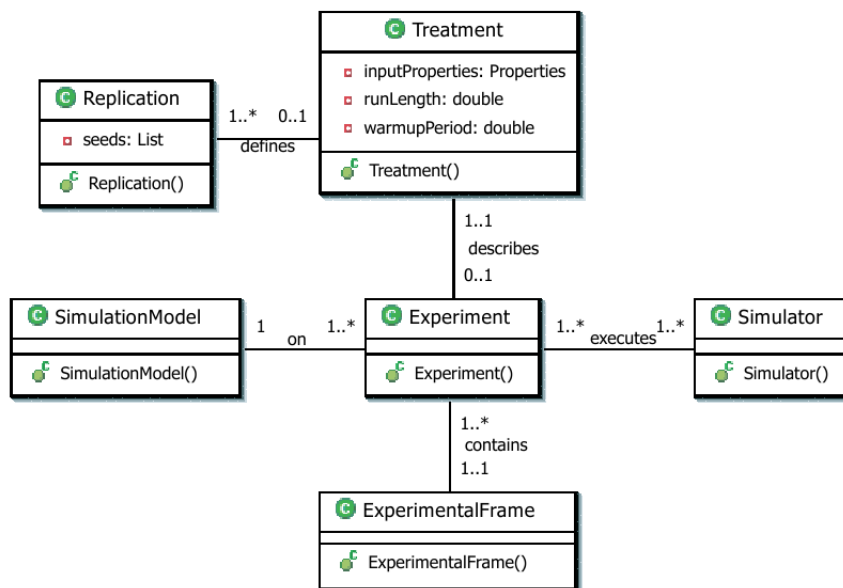
2.5.2   Defining the experiment

Figure 1: A framework for experimentation

The mutual relation between experiments and a model is studied in this section. Our aim in this section is primarily to answer a number of questions such as: What is an experiment? What are the constraints imposed by an

---

experiment on model design?

In line with [Law and Kelton, 2000] we argue that these questions are important because often a great deal of time and money is spent on model development, but little effort is made to analyze the output of an experiment appropriately [Law and Kelton, 2000].

To prevent misuse of tools and techniques, we pay a lot of attention to the concept of an experimental frame. This concept, presented in figure 1 follows [Öeren and Zeigler, 1979, Sol, 1982] in their specification of an experiment, a treatment, a run control and a replication.

According to [Sol, 1982] a simulation model is transformed into an executable simulation model system by including provisions to expose it to a treatment by which it is placed in an experimental frame.

The following concepts define a framework for experimentation:

- an experimental frame, is defined as a set of possible experiments.

- an experiment is a set of treatments for the same simulator and model.

- a treatment consists, according to [Öeren and Zeigler, 1979, Sol, 1982], of input data, initialization conditions, and run control conditions, i.e. the runlength and the warmup-period [Law and Kelton, 2000].

- a replication is one run out of a collection of runs under the same treatment, except for initialization conditions that provide statistical independence, i.e. the seeds for the pseudo-random number generators.

As a result of the loose coupling design pattern, interfaces are extensively used between the DSOL services. For example, the *SimulationModel* in figure 1 implements the *ModelInterface* (see the *nl.tudelft.simulation.dsol* package) and the *Simulator* implements the *SimulatorInterface* (see the *nl.tudelft.simulation.dsol.simulators* package.

Where the model is coded in Java, the experiment is defined in xml. The easiest way to understand the experiment definition is to look at a very simple xml representation of an experiment.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <dsol:experimentalFrame
03 xmlns:dsol="http://www.simulation.tudelft.nl"
04 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
05 <experiment>
06 <model>
07     <model-class>nl.tudelft.simulation.dsol.tutorial.section25.Model*
08 </model-class>
09         <class-path>
10             <jar-file>/tmp/tutorial.jar*</jar-file>
11         </class-path>
```

```
12    </model>
13 <simulator-class>nl.tudelft.simulation.dsol.simulators.DEVSSimulator
14 </simulator-class>
15 <treatment>
16    <startTime>2004-09-01T00:00:00</startTime>
17    <timeUnit>MINUTE</timeUnit>
18    <warmupPeriod unit="MINUTE">0</warmupPeriod>
19    <runLength unit="MINUTE">10</runLength>
20    <replication description="replication 0">
21          <stream name="default" seed="1"/>
22    </replication>
23    <replication description="replication 1">
24          <stream name="default" seed="2"/>
25    </replication>
26    <replication description="replication 2">
27          <stream name="default" seed="3"/>
28    </replication>
29 </treatment>
30 </experiment>
31 </dsol:experimentalFrame>
```

* These are (operating system) specific settings and may change depending on the location of your jar file or the integrated development environment (IDE, e.g. Eclipse) you use.

Line 1 starts with an xml-specic header. Lines 2-31 introduce the actual experimental frame. The experimental frame contains a specic set of experiments. Line 5 starts with the first experiment. Lines 6-12 define the model. The definition of the model includes the model-class representing the model and a class-path containing all the jar-files for the model. The class-path element is optional (see 2.5.3 on page 14).

Lines 13-14 define the simulator to be used in this experiment, e.g. the DEVSSimulator, the DESSSimulator or the Animator. Lines 15-29 define the treatment defined for this experiment. A treatment has a startTime (line 16), a timeUnit (line 17), a warmupperiod and a run length (lines 19). The startTime is defined according to the ISO 8601 Numeric representation of Dates and Time standard[4] .

The treament furthermore defines a set of replications to run. Replications are identical copies of a treatment with respect to all parameters except the seed of the pseudo random number generators used [Law and Kelton, 2000]. A replication is thus defined by its description and seed by which the pseudo random number generator is initialized.

---

[4]http://www.iso.ch/iso/en/prods-services/popstds/datesandtime.html

2.5.3 Executing the experiment

The final hurdle is to execute the experiment. There are two approaches to execute the experiment: with the standard DSOL Graphical User Interface or with a custom made execution class. Though the first approach is more simple, the custom approach has the advantage of reducing overhead and therefore to improve execution speed. Both approaches are discussed here.

Executing within the DSOL-GUI application

The experiment.xml file created in the previous section can directly be opened and executed from within the dsol application (we get to the how of starting a dsol application in a moment). When the experiment is successfully opened a screen as shown in figure 2 should be visible.
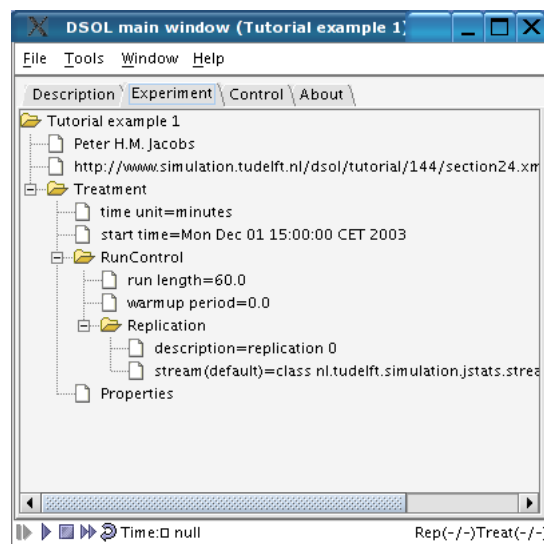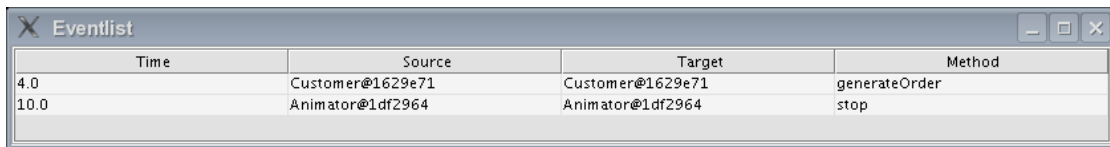

Figure 2: The loaded experiment

In order to make the above simulation model accessible to the dsol-gui two approaches can be followed :

- to add the directory containing the above binary Java class files to the *CLASSPATH* when starting dsol-gui. This results in starting dsol by invoking something like: "*java -classpath .;C:/development/tutorial -jar dsol-gui.jar*"

- to zip the entire model in a jar and to add it to the xml-based experiment. To zip your model into a Java Archive Resource is done with a statement similar to "*jar -cvf C:/tmp/tutorial.jar C:/development/tutorial/*"*. For the remainder of this section, we assume this approach is chosen.

The experiment can now be started with the *'play button'* on the bottom of figure 2. The simulation time is in time units of the current treatment (i.e. minutes) and is shown on the right of the control panel. After starting the experiment, it becomes possible to schedule a pause moment. This is done in the *Tools-menu* of the application. Figure 3 on page 15 shows the eventlist of the simulator executing this experiment at simulation time t=2.1.

Time	Source	Target	Method
4.0	Customer@1629e71	Customer@1629e71	generateOrder
10.0	Animator@1df2964	Animator@1df2964	stop

Figure 3: The event-list at t=2.1

Executing with a custom launcher

The second option is to start the experiment with a custom launcher. In this tutorial we create a custom launcher called *ConsoleRunner*.

```
 1  /*
 2   * @(#) ConsoleRunner.java Dec 1, 2003 Copyright (c) 2002-2005 Delft University
...
 7  package nl.tudelft.simulation.dsol.tutorial.section25;
...
29  public final class ConsoleRunner
30  {
31
...
35      private ConsoleRunner()
36      {
37          // unreachable code
38          super();
39      }
40
...
46      public static void main(final String[] args)
47      {
48          if (args.length != 1)
49          {
50              System.out.println("Usage : java nl.tudelft.simulation.dsol."
51                      + "tutorial.section25.ConsoleRunner [experiment-url]");
```

```
52              System.exit(0);
53          }
54          try
55          {
56              //We are ready to start
57              Logger.setLogLevel(Level.WARNING);
58
59              // First we resolve the experiment and parse it
60              URL experimentalframeURL = URLResource.getResource(args[0]);
61              ExperimentalFrame experimentalFrame = ExperimentParser
62                      .parseExperimentalFrame(experimentalframeURL);
63
64               experimentalFrame.start();
65          } catch (Exception exception)
66          {
67              exception.printStackTrace();
68          }
69      }
70 }
```

The *ConsoleRunner* is a static final class with one *main* method. This main method expects one argument: the url of the experimental frame. The next step is to parse the experimental frame inputstream to an actual experimental frame (line 61-62). Line 64 starts the experiment. Before we continue with some more realistic simulation models, section 3 first introduces further insight in the boundaries of DSOL with respect to the field of modeling and simulation.

3 Theory on modeling & simulation

In this section, we introduce the scope and boundaries of DSOL. In order to do this, we introduce a theoretical view on modeling and simulation.

According to Shannon [Shannon, 1975] *simulation* is the process of designing a model of a real system and conducting experiments with the model for the specific purpose of experimentation. The model is expressed in a language. According to Sol, this language merely reflects a chosen vehicle of communication [Sol, 1982]. Computer simulation is therefore an application domain of programming languages. Based on this, Öeren et. al.[Öeren and Zeigler, 1979], and Sol [Sol, 1982] introduced the following concepts:

- a model, or model system is a selected set of objects and relations describing a system under investigation.

- an experiment, or experimental frame, defines the conditions under which an experiment must take place. The experiment contains the treatments (or scenarios) to be executed.

- a simulator is a computational device for generating behavior of the model, based on the parameters of the experiment.

Based on the method of execution, Nance introduced the following taxonomy of simulation [Nance, 1993]:

- *discrete event* simulation in which the model specifies discrete state changes as a function of time. Time changes may either be continuous or discrete.

- *continuous simulation* in which the model specifies continuous state changes as a function of time. Time changes may again be either continuous or discrete.

- *Monte-Carlo* simulation which uses models of uncertainty where representation of time is unnecessary. It is a method by which an inherently non-probabilistic problem is solved by a stochastic process.

Based on this taxonomy, several related forms of simulation are defined. Law & Kelton [Law and Kelton, 2000] introduce *combined simulation* as a form in which a model contains both discrete and continuous components. Shantikumar [Shanthikumar and Sargent, 1984] introduced *hybrid simulation* as a form in which a model contains discrete event and analytical components. The Department of Defense standards [DoD, 1995] speak of *virtual, live and constructed simulation* whenever real-time, human or hardware components are involved in the specification of a model.

This taxonomy results in the first insight in the boundaries of DSOL. DSOL aims to support all of the above but *Monte-Carlo*. DSOL inherently aims to provide decision support for time-dependent ill-structured problems.

3.1 Basic requirements for a simulation language

In order to prove the validity of DSOL, we must understand its requirements. As will be pointed out, its requirements form an extension to the requirements given by Nance [Nance, 1993] and Sol [Sol, 1982] for any simulation language:

- to provide pseudo random numbers. A good overview of pseudo random numbers can be found in [L'Ecuyer, 1997].

- to provide a set of statistical distributions. More information on these distributions can be found in [Law and Kelton, 2000].

- to provide time-flow mechanisms to represent an explicit representation of time. More information on time-flow mechanisms can be found in [Balci, 1988].

- to provide an experimental frame to represent the experiment(s).

- to provide statistical output analysis. More information on the statistical output of simulation can be found in [Banks, 1998, Law and Kelton, 2000].

In simulation a clear distinction is made between *real time* and *simulation time*. The concept *real time* is used to refer to the wall-clock time. It represents the execution time of the experiment. The *simulation time* is an attribute of the simulator and is initialized at t=0.00. Simulation time can be stopped, incremented either continuously or in discrete steps [Balci, 1988]. Nance [Nance, 1981] formalized the time-state relations as follows:

- an *instant* is a value of simulation time at which the value of at least one attribute of an object can be altered.

- the *state* of an object is the enumeration of all attribute values of an object at a particular instant.

- an *interval* is the duration between successive instants and a *span* is the contiguous succession of one or more intervals.

- an *activity* is the state change of an object over an interval.

- an *event* is a change in object state, occurring at an instant, that initiates an activity precluded prior to that instant.

- a process is the succession of state changes of an object over a *span*.

3.2   Modeling perspectives

In order to construct a model a *Weltanschauung* must implicitly be established to permit the construction of a simulation language [Lackner, 1962]. This *Weltanschauung* expresses the meta-model of the language and is also referred to as *formalism*, *modeling construct* or *world-view* [Balci, 1988]. Literature has identified three basic world-views for discrete event simulation [Fishman, 1973, Overstreet and Nance, 1986]:

- *event scheduling* which provides a locality of time: each event in a model specification describes related actions that may all occur in a single instant.

- *activity scanning* which provides a locality of state: each activity in a model specification describes all actions that must occur due to the model assuming particular state.

- *process interaction* provides a locality of object: each process in a model specification describes the entire action sequence of a particular object.

DSOL primarily supports the *event scheduling* formalism [Jacobs et al., 2002]. *Activity scanning* is not supported because of its inefficient implications to execution. *Process interaction* is supported as well. More information on the danger of specifying a *process interaction* formalism in Java can be found here [Lang et al., 2003].

Vangheluwe and Lara [Vangheluwe and de Lara, 2002] published in 2002 a formalism transformation graph (figure 4). The arrows in this graph denote a behavior-preserving homomorphic relationship using transformations between formalisms. Further development will show to what extent DSOL can implement these relationships and provide the full set of modeling formalisms.

Besides the *event scheduling* formalism for discrete event scheduling, DSOL supports continuous differential equations, and *DEV&DESS* formalisms. After this insight into the scope of DSOL, it is now time to see some more realistic examples.
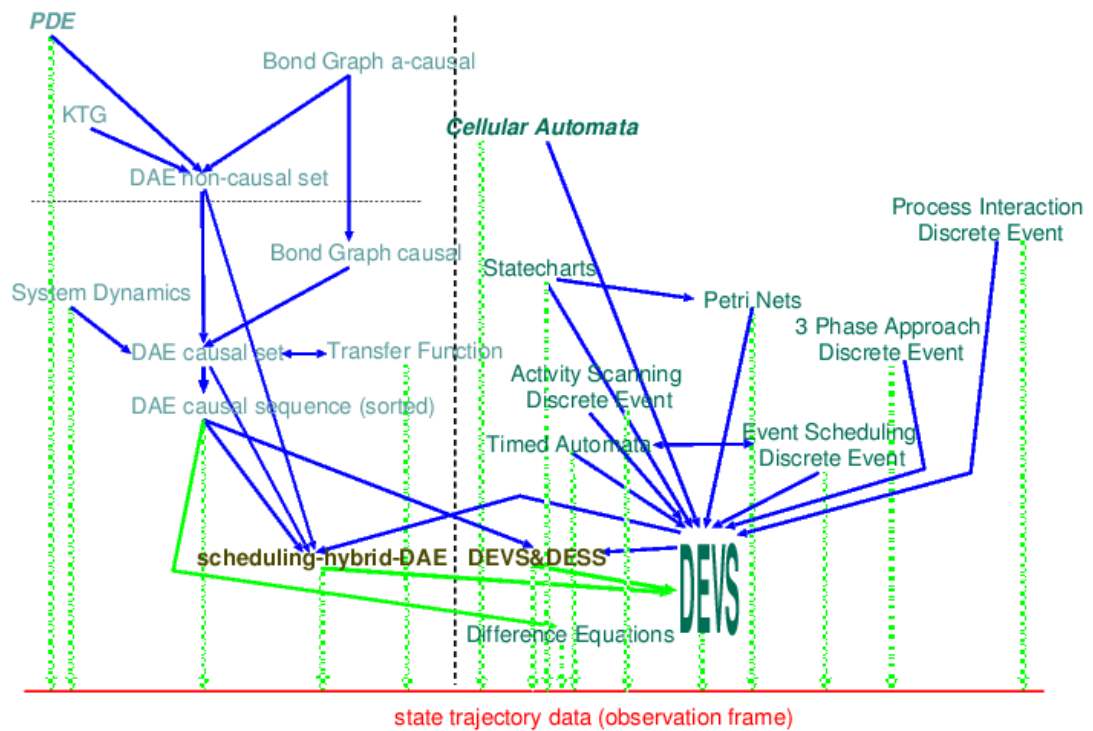
Figure 4: Formalism transformation graph [Vangheluwe and de Lara, 2002]

4 Basic Simulation Modeling

This section introduces a number of basic simulation models. The examples are directly executable from the help-menu in the dsol graphical user interface. By providing these examples we aim to extend your knowledge with basic concepts of simulation in DSOL: statistics, world-views, and animation.

4.1 The single server queuing system

This section reveals the DSOL implementation of the single server queuing system. We consider a system consisting of a single server which receives customers arriving independently and identically distributed (IDD). A customer who arrives and finds the server idle is being serviced immediately. A customer who finds the server busy enters a single queue. Upon completing a service for a customer, the server checks the queue and (if any) services the next customer in a first-in, first-out (FIFO) manner.

The simulation begins in an empty-and-idle state. We simulate until a predefined fixed number of customers n have entered the system and completed their service. To measure the performance of this system, we focus on a number of output variables. First of all we focus on the expected delay $d(n)$ of a customer in the queue. From a system perspective we furthermore focus on the number of customers in queue $q(n)$. The final output variable we consider is the expected utilization of the server $u(n)$. This is the proportion of the time the server was in its busy state. Since the simulation is dependent on random variable observations for both the inter-arrival time and the service time, the output variables $d(n)$, $q(n)$ and $u(n)$ will be random and, therefore, expected to be variable.

4.1.1 Conceptual model

The first activity we describe is the formal conceptualization of the above introduction. As described in [Fishman, 1973] a number of world views can be used to express our conceptual model. In this example, the event-based world-view of the previous section is used. Figure 5 shows a flowchart of the single server queuing system. This figure conceptualizes the flow oriented perception of the customer waiting for service. Besides the actual flowchart three classes of the DSOL flow library are introduced: *Seize*, *Delay*, and *Release*. This *Flow* library represents a modeling world-view specified on top of the event-based world-view. The classes belong to the *nl.tudelft.simulation.dsol.formalisms.flow* package and are deployed in *dsol.jar* library file.

4.1.2 Specification

The next activity in the construction of the model is its specification in the DSOL framework. For this particular example the values of table 4.1.2 are
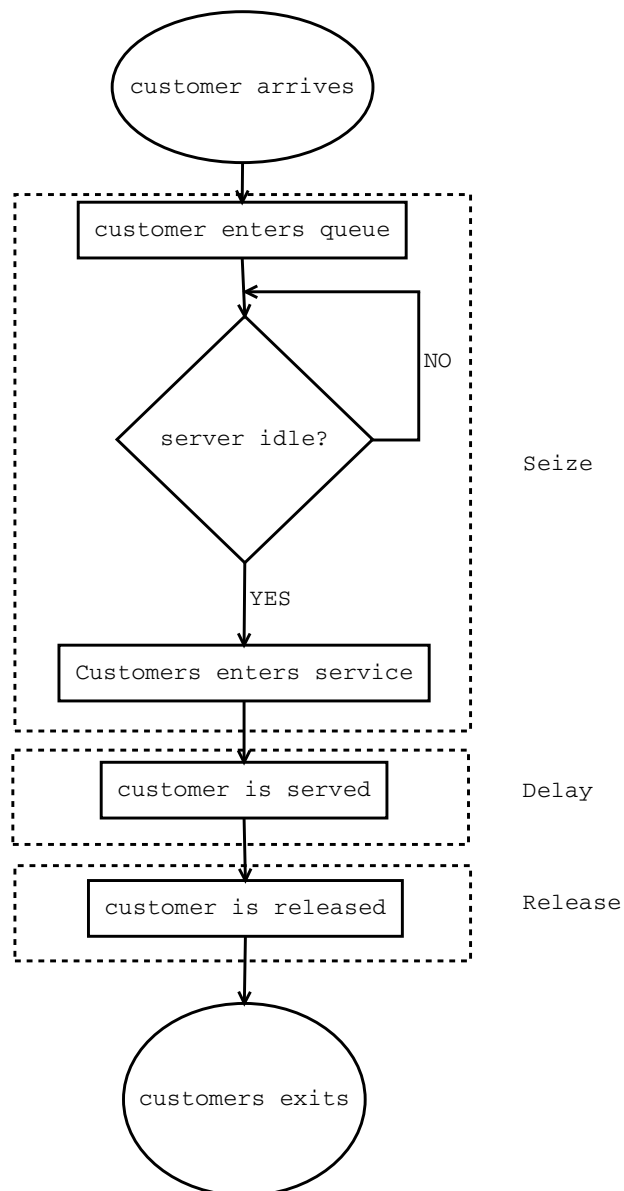
Figure 5: Flowchart of the single server queuing system

used. More information on the distributions used in this example can be found in [Law and Kelton, 2000]. In order to create an M/M/1 Queue with the above defined statistics, only one class is defined: *MM1Queue*:

description	value
simulator.runLength	Double.MAXVALUE
randomizer.seed	555
generator.startTime	0
generator.interarrivalTime	EXPO(1.0)
generator.batchSize	1
generator.maxCreated (n)	1000
server.resourceCapacity	1.0
server.serviceTime	EXPO(0.5)

Table 2: Specification values for the single server queuing system

```java
 1 /*
 2  * @(#) MM1Queue.java Sep 21, 2003 Copyright (c) 2002-2005 Delft University of
...
 7 package nl.tudelft.simulation.dsol.tutorial.section41;
 8
...
44 public class MM1Queue implements ModelInterface
45 {
...
49     public MM1Queue()
50     {
51         super();
52     }
53
...
58     public void constructModel(final SimulatorInterface simulator)
59             throws SimRuntimeException, RemoteException
60     {
61         DEVSSimulatorInterface devsSimulator = (DEVSSimulatorInterface) simulator;
62
63         StreamInterface defaultStream = devsSimulator.getReplication()
64                 .getStream("default");
65
66         // The Generator
67         Generator generator = new Generator(devsSimulator, Object.class, null);
68         generator.setInterval(new DistExponential(defaultStream, 1.0));
```

```
69            generator.setStartTime(new DistConstant(defaultStream, 0.0));
70            generator.setBatchSize(new DistDiscreteConstant(defaultStream, 1));
71            generator.setMaxNumber(1000);
72
73            // The queue, the resource and the release
74            Resource resource = new Resource(devsSimulator, 1.0);
75
76            // created a resource
77            StationInterface queue = new Seize(devsSimulator, resource);
78            StationInterface release = new Release(devsSimulator, resource, 1.0);
79
80            // The server
81            DistContinuous serviceTime = new DistExponential(defaultStream, 0.5);
82            StationInterface server = new Delay(devsSimulator, serviceTime);
83
84            // The flow
85            generator.setDestination(queue);
86            queue.setDestination(server);
87            server.setDestination(release);
88
89            // Statistics
90            Tally dN = new Tally("d(n)", devsSimulator, queue, Seize.DELAY_TIME);
91            Tally qN = new Tally("q(n)", devsSimulator, queue,
92                    Seize.QUEUE_LENGTH_EVENT);
93            Utilization uN = new Utilization("u(n)", devsSimulator, server);
94
95            // Charts
96            new BoxAndWhiskerChart(devsSimulator, "d(n) chart").add(dN);
97            new BoxAndWhiskerChart(devsSimulator, "q(n) chart").add(qN);
98            new BoxAndWhiskerChart(devsSimulator, "u(n) chart").add(uN);
99        }
100 }
```

Line 44 introduces the *MM1Queue* class and states that this class imple-
ments the *ModelInterface*. According to the design pattern introduced in section
2.5.1 on page 11, the constructor of this class is empty. Line 58 introduces the
required *constructModel* method.

Lines 67-68 points out how to use pseudo random number streams in a
model. This is done by requesting the stream from the currently active replica-
tion. Streams are all known by their name, which is specified in the experiment.

Line 66-71 introduce a customer generator. This generator is the first flow
object used in this example. As described in the javadoc, a generator regularly in-
vokes the constructor of a class. In this particular example, the *java.lang.Object*
class is invoked. By using this class, we emphasize that there are no require-

ments for classes to be used in DSOL simulation modeling and that simulation modeling is all about reduction. Since there is no need to define any intelligence in the customer, there is no need to create a *Customer* class.

Lines 68-71 initialize the generator with the appropriate distributions. These distributions are part of the *nl.tudelft.simulation.jstats.distribution* package and deployed in the *jstats.jar* library. In later examples we will see how to define the parameters for these distributions in the experiment definition.

Line 74 creates a resource with a capacity of 1.0. Line 76-82 create the queue, server and release objects of the M/M/1/Queue. Lines 85-87 create the flow of the model; they link the different stations.

Lines 89-93 create the statistics. Line 90 defines a tally which is interested in *Seize.DELAY_TIME* events fired by the queue. A tally computes and represents the min, max, mean, variance and standard deviation values of the events it receives. Line 93 introduces an *Utilization* object computing and representing the utilization of a station. In this particular example, the utilization of the server is computed. Lines 96-98 create three independent *BoxAndWhiskerCharts*.

Since the model is constrained by a maximum number of customers the runLength of this experiment can be infinite. In the experiment a stream with the name *default* is defined with seed value *555*. The output of the *M/M/1 Queue* is illustrated in figure 6.

Figure 6 presents a typical statistics outputscreen of DSOL. On the left side an overview is given of all experiments executed and stored. In these experiments a number of statistical objects are generated and displayed as leafs of the tree. The rigth element of figure 6 illustrates a spreadsheet alike panel. One drags and drop's the statistical elements from the left side to this panel.

4.2   Inventory challenge

The second example introduced in this section is again taken from Law & Kelton [Law and Kelton, 2000]. In this example we introduce a retailer who sells a single product and would like to know how many items he should have in inventory for each of the next *n* weeks. Both the conceptual model and its specification in DSOL represent a realistic supply chain model.

4.2.1   Introduction

Customers demand the product according to the following demand function *d*.

$$d = EXPO(0.1week) * BS \tag{1}$$

Orders in the above equation are ordered every *EXPO(0.1)* week. *BS* represents the batchsize to be ordered. Its value is:
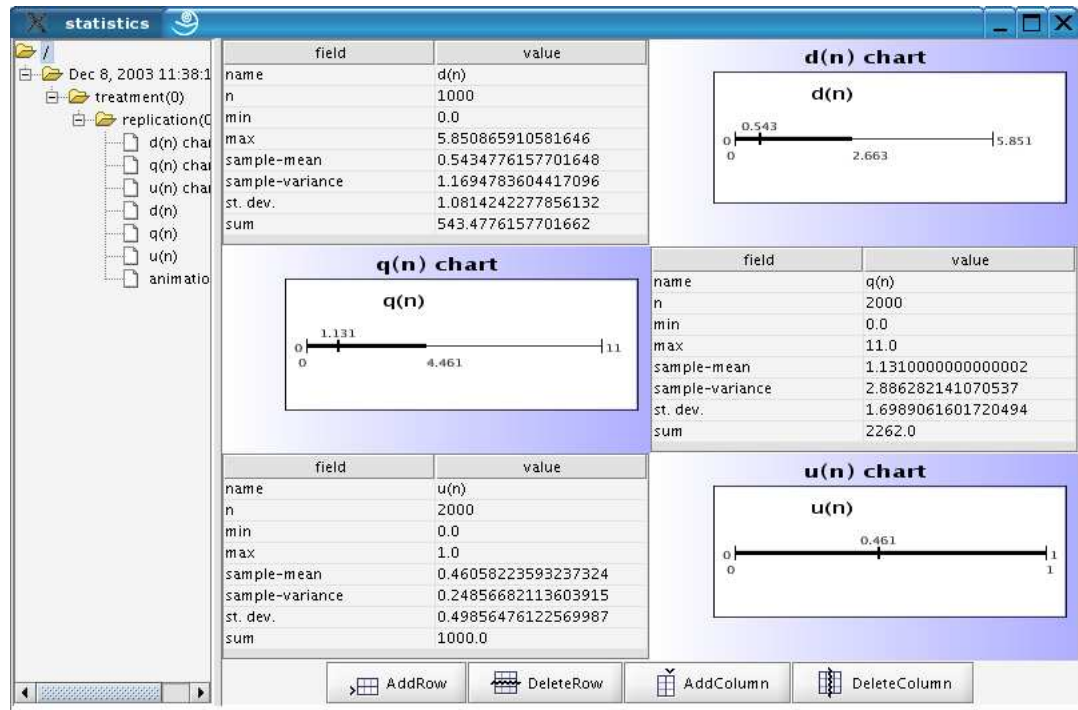
Figure 6: Result of M/M/1 Queue

$$BS = \begin{cases} 1 & \text{probability} = \frac{1}{6} \\ 2 & \text{probability} = \frac{1}{3} \\ 3 & \text{probability} = \frac{1}{3} \\ 4 & \text{probability} = \frac{1}{6} \end{cases} \tag{2}$$

At the beginning of each week, the retailer reviews the inventory level and decides how many items to order from its supplier. The costs of ordering OC consists of setup costs SC and marginal costs IC per ordered product q. The exact ordering cost function is:

$$OC = SC + IC * q, \text{SC=32} \wedge \text{IC=3} \tag{3}$$

The *lead time* or *delivery time* of an order is a random variable that is distributed uniformly between 0.5 and 1 week. The retailer uses a stationary (s,S) policy to decide how much to order q, i.e.

$$q = \begin{cases} S - I & \text{if I} \leq \text{s} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Equation 4 explicits that when the actual inventory I decreases below a threshold s the quantity q to be ordered will increase the inventory to an upper bound of S. When the actual demand occurs, it is satisfied immediately if the inventory level is at least as large as the demand. If the demand exceeds the inventory level, the excess of demand over supply is back-logged and satisfied by future deliveries. When ordered products are delivered, they are first used to satisfy outstanding back-logged orders. The remainder is added to the inventory.

Besides the ordering costs, the retailer has to face the costs of holding HC and the shortage costs, or backlog costs BC. In this specific example we specify HC=1 per product item per week and BC=5 per product item per week. We furthermore assume that the initial inventory is 60 at t=0 and no order is outstanding. We simulate the model for 120 weeks. The retailer is interested in the total costs of its operation and wants to get insight in its demand and supply processes. The retailer is furthermore interested in the total costs of operation when he changes the threshold and upper bound of his stationary ordering policy.

4.2.2 Conceptualization

The first step in the conceptualization activity is to determine the simulation events and their relations. Three events are identified: the autonomous and self-invoking demand event, the autonomous and self invoking inventory evaluation event and the (sometimes) resulting order event.

Though it is possible to define the model in several conceptual meta-models (Event Graphs, flow charts, sequence diagrams), this section uses flowcharts. The reason is that flow charts are easy to understand.

Figure 7-a illustrates the business logic triggered by an *order arrived event*. First the previous acquired backlog is served as much as possible. The rest is added to the inventory. Whenever a *demand arrived event* (figure 7-b) is triggered the demand is served as much as possible. If the current inventory does not allow completion of the order, the rest is stored as backlog. On evaluation (figure 7-c) the simulation computes the weekly inventory and backlog costs after which it decides whether to schedule the *order arrived* event. Evaluation schedules a successive *evaluation event*.

In contrast with the previous examples, the conceptualization presented in figure 7 is not sufficient to start the specification in DSOL. A more detailed conceptual model is presented in the class diagram of figure 8.

In this class diagram, the *SellerInterface* and *BuyerInterface* define the functional behavior of a seller and a buyer. A buyer invokes the *order* method on the *SellerInterface*; upon completion the seller invokes the *receiveProduct* method on the buyer. In this example a customer implementing the *BuyerInterface* generates the demand as specified by BS of equation 2. A warehouse implementing the *SellerInterface* supplies requested the product to the retailer. The retailer implements both the *SellerInterface* and the *BuyerInterface* and forms
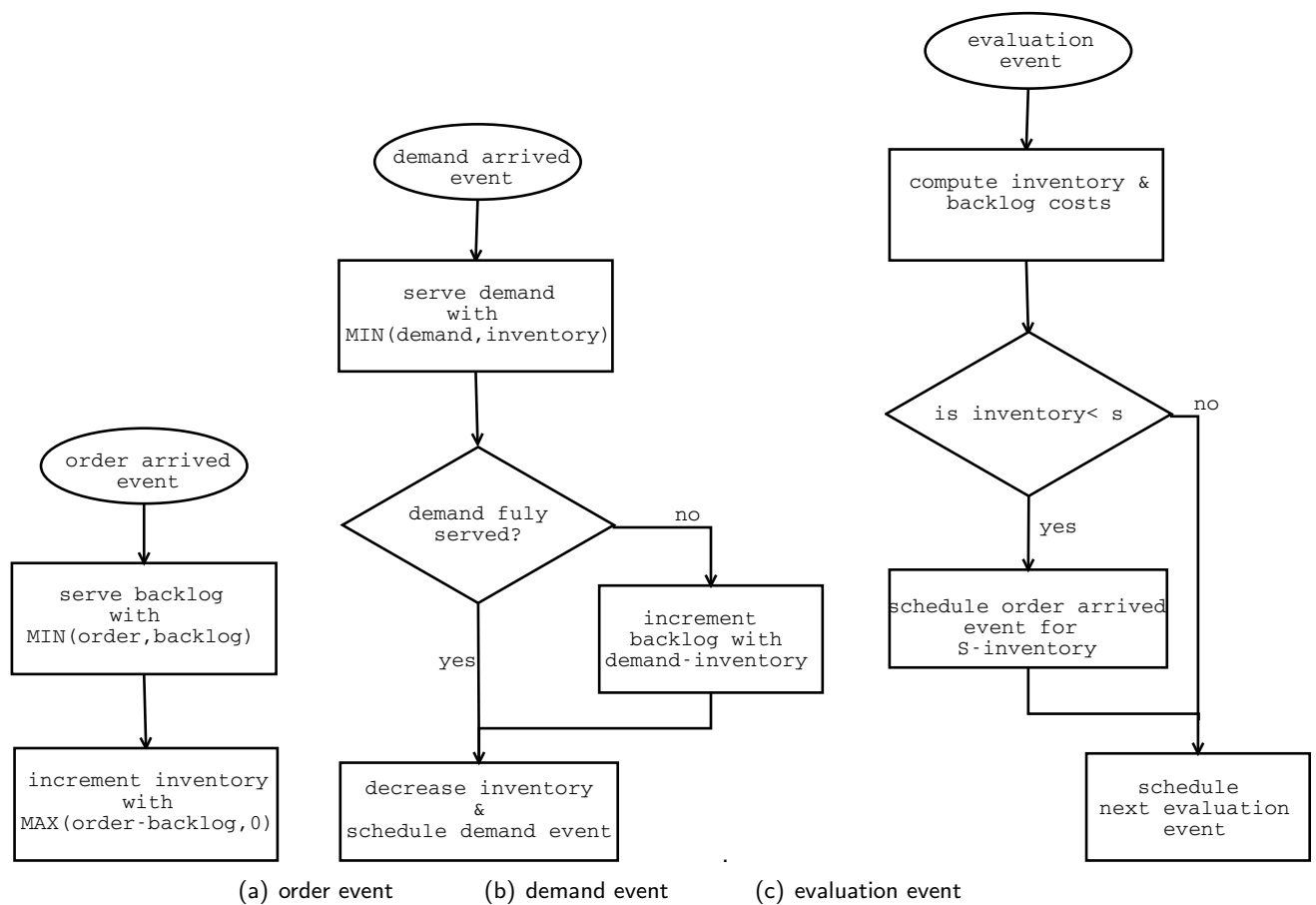
(a) order event   (b) demand event   (c) evaluation event

Figure 7: Flowcharts of inventory challenge

the center of our investigation.

4.2.3   Specification

Seller and Buyer Interface

The first step is to design the two interfaces: the *BuyerInterface* and *SellerInterface*. They are implemented as follows:

```
1 package nl.tudelft.simulation.dsol.tutorial.section42;
2
3 public interface BuyerInterface
4 {
...
```
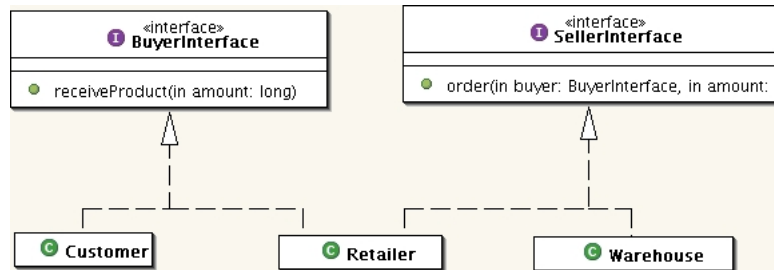
Figure 8: Class diagram of inventory problem

```
 9   void receiveProduct(final long amount);
10 }
```

In order to enforce a modular design, the *BuyerInterface* is used as argument of the *order* method of the *SellerInterface* (line 10). This is a typical example of what is called *design by contract* or *service oriented design*.

```
1 package nl.tudelft.simulation.dsol.tutorial.section42;
2
3 public interface SellerInterface
4 {
...
10   void order(final BuyerInterface buyer, final long amount);
11 }
```

Ordering policies

The last step before the actual specification of the customer, the retailer, and the warehouse is the specification of the ordering policy. Similar to the selling- and buying interfaces, an interface design is chosen to enforce modularity. An *OrderingPolicy* interface and *StationaryPolicy* class implementing this interface are deployed in the *nl.tudelft.simulation.dsol.tutorial.section42.policies* package.

```
1 package nl.tudelft.simulation.dsol.tutorial.section42.policies;
2
3 public interface OrderingPolicy
4 {
...
```

```
10    long computeAmountToOrder(final long inventory);
11 }
```

The *StationaryPolicy* implements the *OrderingPolicy* and implements the ordering policy as specified by equation 4 on page 26. Line 11 introduces the *lowerBound* and the *upperBound*. Lines 36-44 implement the required behavior and compute the amount of product to order.

Lines 40-51 define the constructor of the *StationaryInterface*. In this constructor the values for the *lowerBound* and the *upperBound* are read from the treatment. This is the first example in which we use treatment-dependent variables. This enables us to design several treatments (or scenarios).

```
 1 /*
 2  * @(#) StationaryPolicy.java Dec 8, 2003 Copyright (c) 2002-2005 Delft
...
 7 package nl.tudelft.simulation.dsol.tutorial.section42.policies;
...
26 public class StationaryPolicy implements OrderingPolicy
27 {
28     /** the lower bound of the policy */
29     private long lowerBound;
30
31     /** the upper bound of the policy */
32     private long upperBound;
...
40     public StationaryPolicy(final SimulatorInterface simulator)
41             throws RemoteException
42     {
43         super();
44         Properties properties = simulator.getReplication().getTreatment()
45                 .getProperties();
46
47         this.lowerBound = new Long(properties.getProperty("policy.lowerBound"))
48                 .longValue();
49         this.upperBound = new Long(properties.getProperty("policy.upperBound"))
50                 .longValue();
51     }
52
...
57     public long computeAmountToOrder(final long inventory)
58     {
59         if (inventory <= this.lowerBound)
60         {
61             return this.upperBound - inventory;
```

```
62           }
63           return 0;
64       }
65 }
```

Warehouse

The *Warehouse* class implements a *SellerInterface* and represents the supply-chain actor selling the product to the retailer.

```
 1 /*
 2  * @(#) Warehouse.java Dec 8, 2003 Copyright (c) 2002-2005 Delft University of
...
 7 package nl.tudelft.simulation.dsol.tutorial.section42;
...
30 public class Warehouse implements SellerInterface
31 {
32     /** simulator. the simulator to schedule on */
33     private DEVSSimulatorInterface simulator = null;
34
35     /** the delivery or leadTime */
36     private DistContinuous leadTime = null;
37
...
44     public Warehouse(final DEVSSimulatorInterface simulator)
45             throws RemoteException
46     {
47         super();
48         this.simulator = simulator;
49
50         StreamInterface stream = this.simulator.getReplication().getStream(
51             "default");
52         this.leadTime = new DistUniform(stream, 0.5, 1.0);
53     }
54
...
60     public void order(final BuyerInterface buyer, final long amount)
61     {
62         try
63         {
64             this.simulator.scheduleEvent(new SimEvent(this.simulator
65                     .getSimulatorTime()
66                         + this.leadTime.draw(), this, buyer, "receiveProduct",
```

```
67                    new Long[] { new Long(amount) }));
68          } catch (Exception exception)
69          {
70             Logger.warning(this, "order", exception);
71          }
72      }
73 }
```

In the *Warehouse* class, a leadTime is defined as a uniform distribution
(line 52). This distribution is constructed with a stream named *"default"*. The
second part of interest is the block between lines 60-72. They embody the
scheduled invocation of the *receiveProduct* method on the buyer who requested
the order.

Customer

```
 1 package nl.tudelft.simulation.dsol.tutorial.section42;
...
13
14 public class Customer implements BuyerInterface
15 {
...
37   public Customer(
38     final DEVSSimulatorInterface simulator,
39     final SellerInterface retailer)
40     throws RemoteException
41   {
42     super();
43     this.simulator = simulator;
44     this.retailer = retailer;
45
46     StreamInterface stream =
47       this.simulator.getReplication().getStream("default");
48     this.intervalTime = new DistExponential(stream, 0.1);
49     this.orderBatchSize =
50       new DistCustom(
51         stream,
52         new DistCustom.Entry[] {
53           new DistCustom.Entry(1, 1.0 / 6.0),
54           new DistCustom.Entry(2, 1.0 / 3.0),
55           new DistCustom.Entry(3, 1.0 / 3.0),
56           new DistCustom.Entry(4, 1.0 / 6.0)});
57     this.createOrder();
58   }
```

```
59

...
64    public void receiveProduct(final long amount)
65    {
66      Logger.finest(this, "receiveProduct", "received " + amount);
67    }
68

...
72    private void createOrder()
73    {
74      this.retailer.order(this, this.orderBatchSize.draw());
75      try
76      {
77        this.simulator.scheduleEvent(
78          new SimEvent(
79            this.simulator.getSimulatorTime()
80              + this.intervalTime.draw(),
81            this,
82            this,
83            "createOrder",
84            null));
85      } catch (Exception exception)
86      {
87        Logger.warning(this, "createOrder", exception);
88      }
89    }
90  }
```

First of all the above code only describes the *Customer* class partly to ensure a readable tutorial. This will be the case for all further examples. In the *Customer* class two distributions are created: a continuous inter-order-time (line 48) and a discrete batchsize distribution (lines 50-56). These distributions are used by the customer to create an order (line 74) and to schedule the next invocation of the *"createOrder"* method (line 80).

The *Customer* class implements the *BuyerInterface* and must therefore provide a public *receiveProduct* method. Its implementation is to log received products[5].

Retailer

```
1 package nl.tudelft.simulation.dsol.tutorial.section42;
2
```

---

[5]Loggers are accessible in the windows menu of the dsol-gui.

```
..
14 public class Retailer
15   extends EventProducer
16   implements BuyerInterface, SellerInterface
17 {
18   public static final EventType TOTAL_ORDERING_COST_EVENT =
19     new EventType("TOTAL_ORDERING_COST_EVENT");
20
21   public static final EventType INVENTORY_LEVEL_EVENT =
22     new EventType("INVENTORY_LEVEL_EVENT");
23
24   public static final EventType BACKLOG_LEVEL =
25     new EventType("BACKLOG_LEVEL");
26
27   private long inventory = 60L, backLog = 0L;
...
33   private OrderingPolicy orderingPolicy = null;
...
37   public Retailer(
38     final DEVSSimulatorInterface simulator,
39     final SellerInterface warehouse)
40     throws RemoteException
41   {
42     super();
43     this.simulator = simulator;
44     this.warehouse = warehouse;
45     this.orderingPolicy = new StationaryPolicy(simulator);
46
47     Properties properties =
48       this
49         .simulator
50         .getReplication()
51         .getRunControl()
52         .getTreatment()
53         .getProperties();
54     this.backlogCosts =
55       new Double(properties.getProperty("retailer.costs.setup"))
56         .doubleValue();
57     this.holdingCosts =
58       new Double(properties.getProperty("retailer.costs.holding"))
59         .doubleValue();
60     this.marginalCosts =
61       new Double(properties.getProperty("retailer.costs.marginal"))
62         .doubleValue();
```

```
63      this.setupCosts =
64        new Double(properties.getProperty("retailer.costs.setup"))
65          .doubleValue();
66      this.reviewInventory();
67    }
68
69    public void receiveProduct(final long amount)
70    {
71      long served = this.backLog - Math.max(0, this.backLog - amount);
72      this.backLog = Math.max(0, this.backLog - amount);
73      this.inventory = this.inventory + Math.max(0, amount - served);
74      try
75      {
76        this.fireEvent(
77          INVENTORY_LEVEL_EVENT,
78          this.inventory,
79          this.simulator.getSimulatorTime());
80        this.fireEvent(
81          BACKLOG_LEVEL,
82          this.backLog,
83          this.simulator.getSimulatorTime());
84      } catch (RemoteException exception)
85      {
86        Logger.warning(this, "receiveProduct", exception);
87      }
88    }
89
90    private void reviewInventory()
91    {
92      double costs =
93        this.holdingCosts * this.inventory
94          + this.backlogCosts * this.backLog;
95      long amount = this.orderingPolicy.computeAmountToOrder(this.inventory);
96      if (amount > 0)
97      {
98        costs = costs + this.setupCosts + amount * this.marginalCosts;
99        this.fireEvent(TOTAL_ORDERING_COST_EVENT, costs);
100       this.warehouse.order(this, amount);
101     }
102     try
103     {
104       this.simulator.scheduleEvent(
105         new SimEvent(
106           this.simulator.getSimulatorTime() + 1.0,
```

```
107              this,
108              this,
109              "reviewInventory",
110              null));
111      } catch (Exception exception)
112      {
113        Logger.warning(this, "reviewInventory", exception);
114      }
115    }
116
117    public void order(final BuyerInterface buyer, final long amount)
118    {
119      long actualOrderSize = Math.min(amount, this.inventory);
120      this.inventory = this.inventory - actualOrderSize;
121      if (actualOrderSize < amount)
122      {
123        this.backLog = this.backLog + (amount - actualOrderSize);
124      }
125      try
126      {
127        this.fireEvent(
128          INVENTORY_LEVEL_EVENT,
129          this.inventory,
130          this.simulator.getSimulatorTime());
131        this.fireEvent(
132          BACKLOG_LEVEL,
133          this.backLog,
134          this.simulator.getSimulatorTime());
135      } catch (RemoteException exception)
136      {
137        Logger.warning(this, "receiveProduct", exception);
138      }
139      buyer.receiveProduct(actualOrderSize);
140    }
141 }
```

The center of investigation is the *Retailer*. Lines 14-16 introduce the *Retailer* as a class implementing and therefore providing both the *SellerInterface* and the *BuyerInterface*. It furthermore introduces the *Retailer* as an *EventProducer*.

Lines 18-25 introduce 3 constant static flags defining the eventTypes of the *Retailer*. Listeners can subscribe their interest to events of one of these types. Lines 76-83,99,127-134 show that the *Retailer* indeed fires event with corresponding eventTypes.

Line 27 instantiates the inventory and backlog with their initial values. Line

33 introduces the orderingPolicy of the *OrderingPolicy* interface. Line 45 sets the value of this policy to an instance of a *StationaryPolicy*. Lines 47-65 read the values for the marginal, setup, holding, and backlog costs from the treatment.

Lines 69-88 define the behavior of receiving goods from the warehouse. Backlog is served as much and the rest is added to the inventory. Lines 90-115 implement the inventory check. Line 101 starts to compute the inventory costs for this week. Then the policy is checked to determine the amount of product to order (line 95). If product must be ordered, the ordering costs are computed (line 98) and the order is made (line 100).

Lines 117-141 implement the selling behavior of the *Retailer*. If an amount is ordered, as much as possible is delivered. The rest is back-logged. Inventory and backlog values are fired to subscribed listeners.

Model & Experiment

The final step is to code the model and the experiment. In this particular example, the *constructModel* method of the *Model* merely constructs our three actors. The code is therefore not introduced in this tutorial. The experiment is more sophisticated than previous experiments though. In this example, we will define multiple treatments representing multiple scenarios. In these treatments the parameters of the model are given as properties of the treatment.

```xml
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <dsol:experimentalFrame xmlns:dsol="http://www.simulation.tudelft.nl" xmlns:xsi='
 3   <experiment>
 4       <model>
 5           <model-class>nl.tudelft.simulation.dsol.tutorial.section42.Model
 6           </model-class>
 7       <class-path>
 8           <jar-file>/tmp/tutorial.jar</jar-file>
 9       </class-path>
10     </model>
11       <simulator-class>nl.tudelft.simulation.dsol.simulators.DEVSSimulator
12       </simulator-class>
13     <treatment>
14       <startTime>2003-12-01T14:30:00</startTime>
15       <timeUnit>WEEK</timeUnit>
16       <warmupPeriod unit="WEEK">0</warmupPeriod>
17          <runLength unit="WEEK">120</runLength>
18       <properties>
19           <!-- The cost properties -->
20           <property key="retailer.costs.backlog" value="1"/>
21           <property key="retailer.costs.holding" value="1"/>
22           <property key="retailer.costs.marginal" value="3"/>
```

```
23              <property key="retailer.costs.setup" value="30"/>
24              <!-- The ordering policy properties -->
25              <property key="policy.lowerBound" value="8"/>
26              <property key="policy.upperBound" value="80"/>
27          </properties>
28              <replication description="replication 0">
29                  <stream name="default" seed="555"/>
30              </replication>
31              <replication description="replication 1">
32                  <stream name="default" seed="100"/>
33              </replication>
34              <replication description="replication 2">
35                  <stream name="default" seed="29"/>
36              </replication>
37      </treatment>
38      </experiment>
39      <experiment>
40          <model>
41              <model-class>nl.tudelft.simulation.dsol.tutorial.section42.Model
42              </model-class>
43              <class-path>
44              <jar-file>/tmp/tutorial.jar</jar-file>
45              </class-path>
46          </model>
47          <simulator-class>nl.tudelft.simulation.dsol.simulators.DEVSSimulator
48          </simulator-class>
49      <treatment>
50          <startTime>2003-12-01T14:30:00</startTime>
51          <timeUnit>WEEK</timeUnit>
52          <warmupPeriod unit="WEEK">0</warmupPeriod>
53              <runLength unit="WEEK">120</runLength>
54          <properties>
55              <!-- The cost properties -->
56              <property key="retailer.costs.backlog" value="1"/>
57              <property key="retailer.costs.holding" value="1"/>
58              <property key="retailer.costs.marginal" value="3"/>
59              <property key="retailer.costs.setup" value="30"/>
60              <!-- The ordering policy properties -->
61              <property key="policy.lowerBound" value="30"/>
62              <property key="policy.upperBound" value="55"/>
62          </properties>
64              <replication description="replication 0">
65                  <stream name="default" seed="555"/>
66              </replication>
```

```
67              <replication description="replication 1">
68                  <stream name="default" seed="100"/>
69              </replication>
70              <replication description="replication 2">
71                  <stream name="default" seed="29"/>
72              </replication>
73          </treatment>
74      </experiment>
75  </dsol:experimentalFrame>
```

4.2.4   Statistical output

Figure 9 presents the statistical output of our model. In this figure the inventory and backlog levels are presented in an xy-chart. The left column in this figure represents the first treatment and the right column the second treatment.

4.3   Predator-Prey continuous model

The previous two examples introduces two more or less realistic discrete event models. DSOL nevertheless also supports continuous modeling. A continuous model is introduced in this section.

4.3.1   Introduction

The following example is taken from [Borrelli and Coleman, 1998] to introduce a very commonly discussed continuous problem: the predator-prey population interaction. In the 1920s and 1930s, Vito Volterra and Alfred Lotka[Lotka, 1925, Volterra, 1926] independently reduced Darwin's predator-prey interactions to mathematical models.

This section presents a model of predator and prey where association includes only natural growth or decay and the predator-prey interaction itself. All other relationships are considered to be negligible. We will assume that the prey population grows exponentially in the absence of predation, while the predator population declines exponentially if the prey population is extinct. The predator-prey interaction is modeled by mass action terms proportional to the product of the two populations. The model is named the *Lotka-Volterra* system and specified with the following two equations:

$$x' = (-a + by)x = -ax + bxy \tag{5}$$

$$y' = (c - dx)y = cy - dxy \tag{6}$$

In the above equations, x represents the predator population and u the prey population. All rate constants a,b,c, and d are positive. The linear *-ax* and
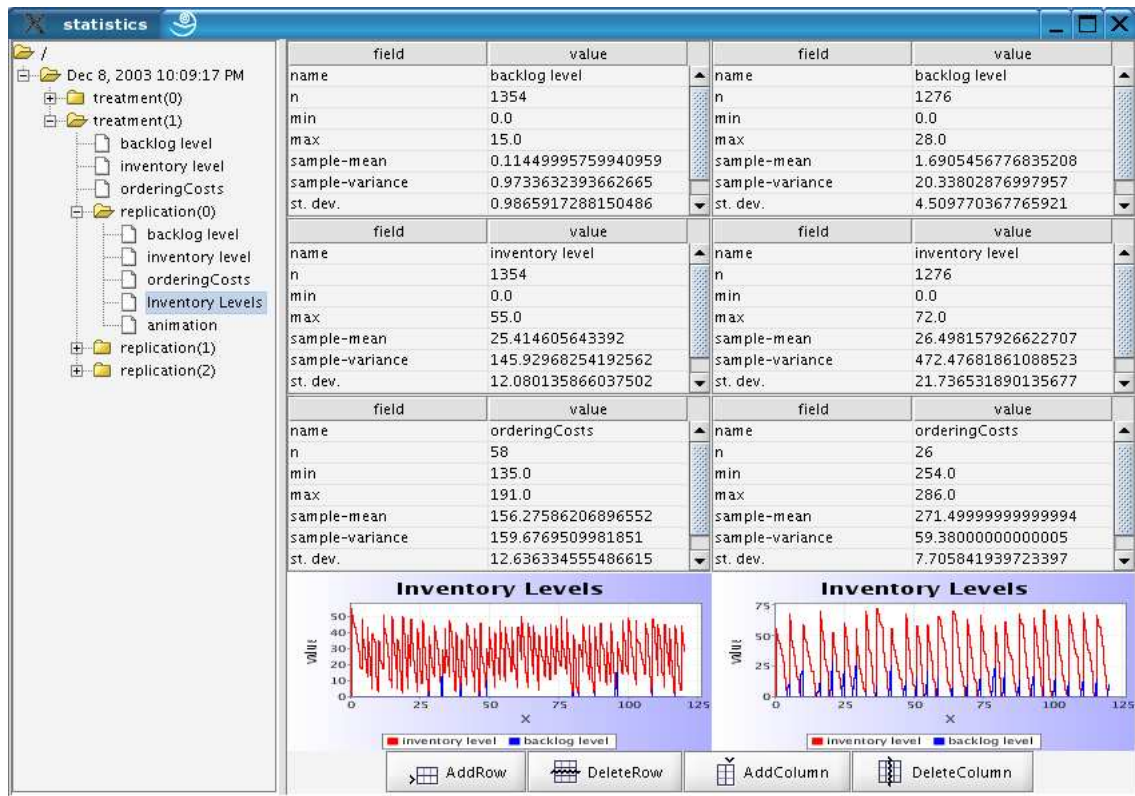
Figure 9: Results of Inventory model

cy terms model the natural decay and growth. The quadratic terms *+bxy* and *-dxy* model the effects of mutual interaction.

4.3.2   Specification

Before we come the actual coding, values for the four parameters and initial states must be specified. Table 4.3.2 represents the Lotka-Volterra parameters to be used. The initial states of the prey and the predator at t=0.0 are 40.0 and 5.0.

parameter	value
a	1
b	0.1
c	1
d	0.2

Figure 10: Specification of *Lotka-Volterra* parameters

Prey

```
 1 package nl.tudelft.simulation.dsol.tutorial.section43;
 2
...
10 public class Prey extends DifferentialEquation
11 {
12   /** the prey to catch */
13   private DifferentialEquationInterface predator = null;
14
15   /** Lotka-Volterra parameters */
16   private double c, d;
17
...
24   public Prey(final DESSSimulatorInterface simulator) throws RemoteException
25   {
26     super(simulator);
27     this.numericalMethod = RUNGE_KUTTA_4;
28
29     Properties properties =
30       simulator.getReplication().getRunControl().getTreatment().getProperties();
31     this.initialize(
32       0.0,
33       new Double(properties.getProperty("prey.initialValue"))
34         .doubleValue());
35     this.c = new Double(properties.getProperty("c")).doubleValue();
36     this.d = new Double(properties.getProperty("d")).doubleValue();
37   }
38
...
43   public double dX(final double time, final double x)
44   {
45     return this.c * x - this.d * x * this.predator.getValue();
46   }
47
...
```

```
53    public void setPredator(final DifferentialEquationInterface predator)
54    {
55      this.predator = predator;
56    }
57 }
```

Line 10 introduces the *Prey* class as a *DifferentialEquation*. This *DifferentialEquation* class is abstract and in order to extend it, the *dX* method must be specified. Lines 43-46 define this required behavior and specify equation 6. Line 27 sets the numerical method to *RUNGE_KUTTA_4*. Lines 31-34 initialize the *Prey* class with the treatment-parameter named *prey.initialValue*.

Predator

```
 1 package nl.tudelft.simulation.dsol.tutorial.section43;
 2
...
10 public class Predator extends DifferentialEquation
11 {
12   /** the prey to catch */
13   private DifferentialEquationInterface prey = null;
14
15   /** Lotka-Volterra parameters */
16   private double a, b;
17
...
24   public Predator(final DESSSimulatorInterface simulator)
25     throws RemoteException
26   {
27     super(simulator);
28     this.numericalMethod = RUNGE_KUTTA_4;
29     Properties properties =
30       simulator.getReplication().getRunControl().getTreatment().getProperties();
31     this.initialize(
32       0.0,
33       new Double(properties.getProperty("predator.initialValue"))
34         .doubleValue());
35     this.a = new Double(properties.getProperty("a")).doubleValue();
36     this.b = new Double(properties.getProperty("b")).doubleValue();
37
38   }
39
...
44   public double dX(final double time, final double x)
```

```
45    {
46      return -this.a * x + this.b * x * this.prey.getValue();
47    }
48
49    /**
...
52    public void setPrey(final DifferentialEquationInterface prey)
53    {
54      this.prey = prey;
55    }
56  }
```

The *Predator* class is alike the *Prey* class. It is a differential equation which specifies equation 5 (line 46).

Model & Experiment

```
 1 package nl.tudelft.simulation.dsol.tutorial.section43;
 2
...
12 public class Life implements ModelInterface
13 {
14
...
18    public Life()
19    {
20      super();
21    }
22
...
27    public void constructModel(final SimulatorInterface simulator)
28      throws RemoteException
29    {
30      DESSSimulatorInterface dessSimulator =
31        (DESSSimulatorInterface) simulator;
32
33      //Prey and Predator definitions
34      Predator predator = new Predator(dessSimulator);
35      Prey prey = new Prey(dessSimulator);
36
37      predator.setPrey(prey);
38      prey.setPredator(predator);
39
40      Persistent preyPopulation =
```

```
41        new Persistent(
42          "prey population",
43          dessSimulator,
44          prey,
45          DifferentialEquationInterface.VALUE_CHANGED_EVENT);
46
47      Persistent predatorPopulation =
48        new Persistent(
49          "predator population",
50          dessSimulator,
51          predator,
52          DifferentialEquationInterface.VALUE_CHANGED_EVENT);
53
54      XYChart chart = new XYChart(dessSimulator, "population");
55      chart.add(preyPopulation);
56      chart.add(predatorPopulation);
57    }
58 }
```

In this particular example, the experiment.xml file is similar to the experiment in the previous inventory example. For this reason, the example is not presented here. The model itself is called *Life*. In the *constructModel* method of *Life* the population of *Predator* and *Prey* are constructed (lines 34,35) Lines 40-52 construct two statistical persistents on both populations. Line 54 constructs a chart representing them.

4.3.3 Statistical output

Figure 11 presents the statistical output of the population. This is a typical curve for a predator-prey cycle. The experiment is executed with a small timeStep (0.01) which ensures accuracy and a smooth curve.

4.4 Animation example

As introduced in the introduction animation is considered as a valuable asset in the validation and presentation of model outcomes. For this reason, DSOL consists of a 2D package for animation. As we will see though, the geographical model state is already three-dimensional which makes a potential 3D animation a matter of using the appropriate libraries.

4.4.1 Introduction

Since we aim just to focus on animation, the example is kept unrealistic simple. The animation example introduced is in this section considers balls moving
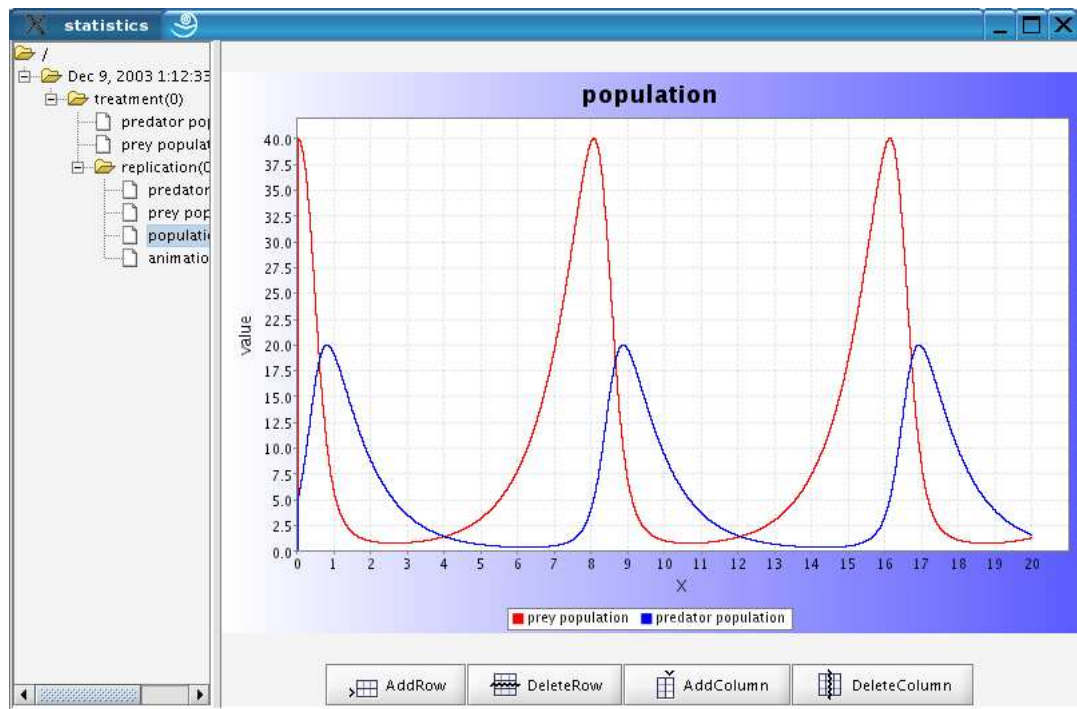
Figure 11: Result of Population model

around. There are two types of balls. One type is modeled in a discrete event formalism and the other in a continuous formalism. What we will show is that animation is loosely coupled and not affected by the underlying model object. We will create one *BallAnimation* class representing both types! First we start with the discrete ball:

4.4.2   Specification

The animation consists of a number of classes. First of all an abstract *Ball* class is defined:

```
1 /*
2  * @(#) Ball.java Mar 3, 2004 Copyright (c) 2002-2005 Delft University of
...
7 package nl.tudelft.simulation.dsol.tutorial.section44;
8
...
```

```
28 public abstract class Ball implements LocatableInterface
29 {
30     /** the number of created balls */
31     private static int number = 0;
32
33     /** the radius of the ball */
34     public static final double RADIUS = 5.0;
35
36     /** the name of the ball */
37     private String name = "";
38
39     /** the origin */
40     protected DirectedPoint origin = new DirectedPoint();
41
42     /** the destination */
43     protected DirectedPoint destination = new DirectedPoint();
44
45     /** the rotation */
46     protected double rotZ = 0.0;
47
...
51     public Ball()
52     {
53         super();
54         this.rotZ = 2 * Math.PI * Math.random();
55         Ball.number++;
56         this.name = "" + Ball.number;
57     }
58
...
62     public Bounds getBounds()
63     {
64         return new BoundingSphere(new Point3d(0, 0, 0), Ball.RADIUS);
65     }
66
...
71     public String toString()
72     {
73         return this.name;
74     }
75 }
```

The most important line of the above code is line perhaps line 28. It intro-
duces the *Ball* as a class implementing the *Locatable* interface. This interface is

---

part of the animation package and expresses the fact that objects implementing
it have a geographic location. It requires in other words the *getLocation()*, and
the *getBounds()* method. Where the *getLocation* is not implemented (Ball is an
abstract class), lines 62-65 implement the *getBounds()* method which defines
the relative volume of the Ball.

In order to move the ball in our model two approaches are followed: a
continuous and a discrete one. The discrete event ball code which displays a
2-dimensional image of a customer now becomes:

```
 1 /*
 2  * @(#) DiscreteBall.java Oct 30, 2003 Copyright (c) 2002-2005 Delft University
...
 8 package nl.tudelft.simulation.dsol.tutorial.section44;
 9
...
28 public class DiscreteBall extends Ball
29 {
30     /** the simulator */
31     private DEVSSimulatorInterface simulator = null;
32
33     /** the start time */
34     private double startTime = Double.NaN;
35
36     /** the stop time */
37     private double stopTime = Double.NaN;
38
39     /** the interpolator */
40     private InterpolationInterface interpolator = null;
41
...
49     public DiscreteBall(final DEVSSimulatorInterface simulator)
50             throws RemoteException, SimRuntimeException
51     {
52         super();
53         this.simulator = simulator;
54         URL image = URLResource.getResource("/nl/tudelft/simulation/dsol/"
55                 + "tutorial/section44/images/customer.jpg");
56         new SingleImageRenderable(this, simulator, image);
57         this.next();
58     }
59
...
66     private void next() throws RemoteException, SimRuntimeException
67     {
```

```
68          StreamInterface stream = this.simulator.getReplication().getStream(
69              "default");
70      this.origin = this.destination;
71      this.rotZ = 2 * Math.PI * Math.random();
72      this.destination = new DirectedPoint(new Point2D.Double(-100
73              + stream.nextInt(0, 200), -100 + stream.nextInt(0, 200)),
74              this.rotZ);
75      this.startTime = this.simulator.getSimulatorTime();
76      this.stopTime = this.startTime
77              + Math.abs(new DistNormal(stream, 9, 1.8).draw());
78      this.interpolator = new LinearInterpolation(this.startTime,
79              this.stopTime, this.origin, this.destination);
80      this.simulator.scheduleEvent(new SimEvent(this.stopTime, this, this,
81              "next", null));
82  }
...
88  public DirectedPoint getLocation() throws RemoteException
89  {
90      if (this.interpolator != null)
91      {
92          return this.interpolator.getLocation(this.simulator
93                  .getSimulatorTime());
94      }
95      return this.origin;
96  }
97 }
```

As becomes apparent from the code, the animation is specied using a *SingleImageRenderable* class. In its construction we provide the URL of the actual image to be displayed. Discrete event balls repeatedly schedule their own next method (line 80). In this method a new destination is assigned.

Finally we have to implement the *getLocation()* method specied by the *Locatable* interface. We return a linear interpolation between the destination and the origin. The simulator time is used to compute the fraction.

The *DirectedPoint* class is part of the animation package and contains a 3-dimensional point (lines 72-74). It furthermore contains 3 attributes expressing its angles (rotX, rotY, rotZ). The continuous ball is specied as follows:

```
1 /*
2  * @(#) ContinuousBall.java May 10, 2004 Copyright (c) 2002-2005 Delft
...
10 import java.rmi.RemoteException;
...
22 public class ContinuousBall extends Ball
```

```
23 {
24
25      /** the positioner */
26      private Positioner positioner = null;
27
28      /** the simulator to use */
29      private DESSSimulatorInterface simulator = null;
30
...
37      public ContinuousBall(final DESSSimulatorInterface simulator)
38              throws RemoteException
39      {
40          super();
41          this.simulator = simulator;
42          this.positioner = new Positioner(simulator);
43          new BallAnimation2D(this, simulator);
44          new BallAnimation3D(this, simulator);
45          try
46          {
47              this.next();
48          } catch (RemoteException exception)
49          {
50              Logger.warning(this, "Ball", exception);
51          }
52      }
...
58      public DirectedPoint getLocation() throws RemoteException
59      {
60          double distance = this.positioner.y(this.simulator.getSimulatorTime())[0];
61          double x = Math.cos(this.rotZ) * distance + this.origin.x;
62          double y = Math.sin(this.rotZ) * distance + this.origin.y;
63          if (Math.abs(x - this.origin.x) > Math.abs(this.destination.x
64                  - this.origin.x)
65                  || Math.abs(y - this.origin.y) > Math.abs(this.destination.y
66                      - this.origin.y))
67          {
68              this.next();
69          }
70          return new DirectedPoint(new Point2D.Double(x, y), this.rotZ);
71      }
...
78      public void next() throws RemoteException
79      {
80          StreamInterface stream = this.simulator.getReplication().getStream(
```

```
81                  "default");
82          this.origin = this.destination;
83          this.positioner.setValue(0);
84          this.destination = new DirectedPoint(-100 + stream.nextInt(0, 200),
85                  -100 + stream.nextInt(0, 200), 0);
86          this.rotZ = (this.destination.y - this.origin.y)
87                  / (this.destination.x - this.origin.x);
88      }
89 }
```

As we see the *getLocation()* method invokes the value function on the positioner which extends a dierential equation. Based on some triangular math calculations the new location is computed based on the y-Value of the positioner. The *next* method is not scheduled but invoked whenever the ball leaves some arbitrary corners. The code for the *Positioner* class is:

```
 1 /*
 2  * @(#) Positioner.java Mar 3, 2004 Copyright (c) 2002-2005 Delft University of
...
 7 package nl.tudelft.simulation.dsol.tutorial.section44;
...
27 public class Positioner extends DifferentialEquation
28 {
...
35      public Positioner(final DESSSimulatorInterface simulator)
36              throws RemoteException
37      {
38          super(simulator);
39          this.initialize(0.0, new double[] { 0.0, 0.0 });
40      }
41
...
47      public void setValue(final double value)
48      {
49          try
50          {
51              super.initialize(this.simulator.getSimulatorTime(), new double[] {
52                      value, 0.0 });
53          } catch (RemoteException exception)
54          {
55              Logger.warning(this, "setValue", exception);
56          }
57      }
58
```

```
...
63      public double[] dy(final double x, final double[] y)
64      {
65          double[] dy = new double[2];
66          dy[0] = y[1]; // v(t) = a(t)
67          dy[1] = 0.5; // a(t) = constant
68          return dy;
69      }
70 }
```

Lines 63-69 clearly show how a second order dierential equation is translated into a set of 2 rst order dierential equations. Continuous balls in our model thus behave according to $y(t) = 0.5^2 + v_0^t + y_0$.
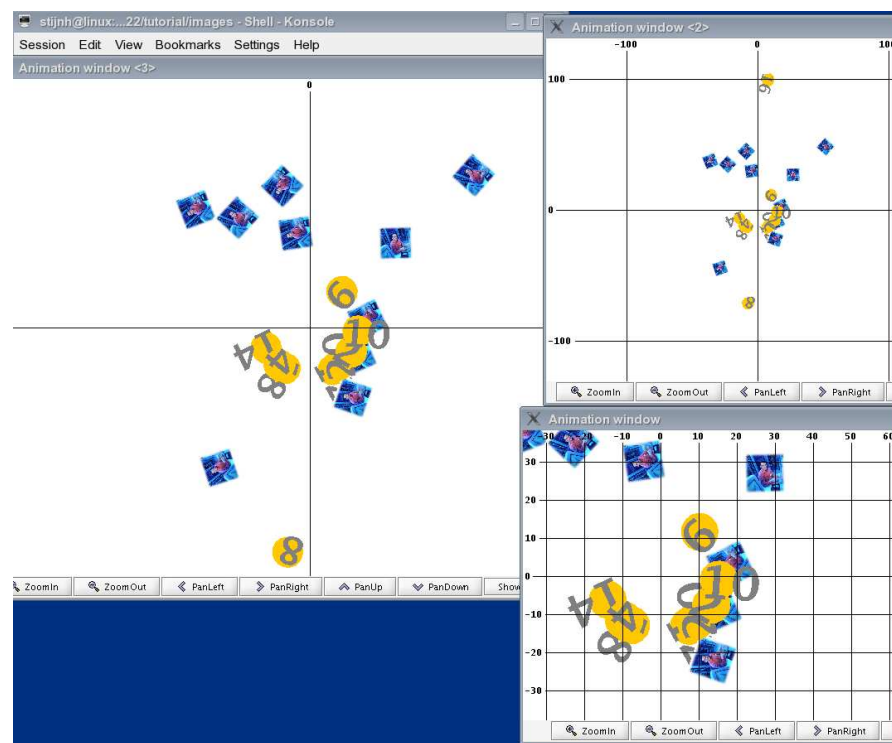

Figure 12: Animation example

Instead of using an image to animate these continuous balls, line 64 of the previously described Ball class presents the BallAnimation class. This class illustrates how to use custom made 2-dimensional animation.

```
 1 /*
 2  * @(#) BallAnimation2D.java Nov 3, 2003 Copyright (c) 2002-2005 Delft
...
 7 package nl.tudelft.simulation.dsol.tutorial.section44;
...
32 public class BallAnimation2D extends Renderable2D
33 {
34     /** the color of the ballAnimation */
35     private Color color = Color.ORANGE;
...
43     public BallAnimation2D(final LocatableInterface source,
44             final SimulatorInterface simulator)
45     {
46         super(source, simulator);
47     }
48

...
55     public void paint(final Graphics2D graphics, final ImageObserver observer)
56     {
57         graphics.setColor(this.color);
58         graphics.fillOval(-(int) Ball.RADIUS, -(int) Ball.RADIUS,
59                 (int) (Ball.RADIUS * 2.0), (int) (Ball.RADIUS * 2.0));
60         graphics.setFont(graphics.getFont().deriveFont(Font.BOLD));
61         graphics.setColor(Color.GRAY);
62         graphics.drawString(this.source.toString(), (int) (Ball.RADIUS * -1.0),
63                 (int) (Ball.RADIUS * 1.0));
64     }
65

...
```

The *BallAnimation2DObject* extends the *Renderable* class which is part
of the animation.D2 package for two-dimensional animation. This *Renderable*
class is abstract and required us to specify the paint method. In this method
we paint on a *TransformedGraphics* object which enables us to draw around
the origin ([x=0.0,y=0.0]). The animation is in runtime automatically scaled
(depending on the zoom level), translated (to the current position), and rotated
(according to theta). Figure 12 illustrates a screen with 3 animation frames of
the balls. This emphasizes the loose structure between model components and
animation.

Roy Chin[6] added 3-dimensional animation to DSOL. Since we added an
instance of the *BallAnimation3D* to our continuous ball on line 44, continu-
ous balls are also animated as illustrated in figure 13 on page 54. Since 3D-

---

[6]http://www.tbm.tudelft.nl/webstaf/royc

animation requires Java3D libraries to separately installed[7] and Java3D requires some knowledge about 3D modeling, the actual code is now considered beyond the scope of this tutorial. Well promise though to address this topic in the future version of this tutorial though.

Again we emphasize that because of the loose structure between animation and model components, DSOL supports concurrent 2D and 3D animation of the same model components.

4.5   Process interaction

Since January 2004, DSOL fully supports the process interaction formalism. 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:

- 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 conict 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.

The Abstract Process class is illustrated in figure 14.

The process interaction example presented in this section comes from the early years of Simula 67. We consider in this examples boats which enter a port. Whenever they enter the port they rst claim 1 jetty. After the jetty is assigned they request for 2 tugs to dock their vessel. Docking takes 2 minutes after which they release the tugs. Now they unload their cargo which takes 14 minutes. In order to leave the port they again request 1 tug for 2 minutes. Then both the tug and the jetty are released.

The port has 2 jetties and 3 tugs and boats arrive at t=0,1,15 minutes. The Port class holds the jetties and the tugs and looks like:

```
1 /*
2  * @(#) Port.java Jan 19, 2004 Copyright (c) 2002-2004 Delft University of
...
```

---

[7]see http://java.sun.com/products/java-media/3D/index.jsp for more information about downloading and install Java3D libraries
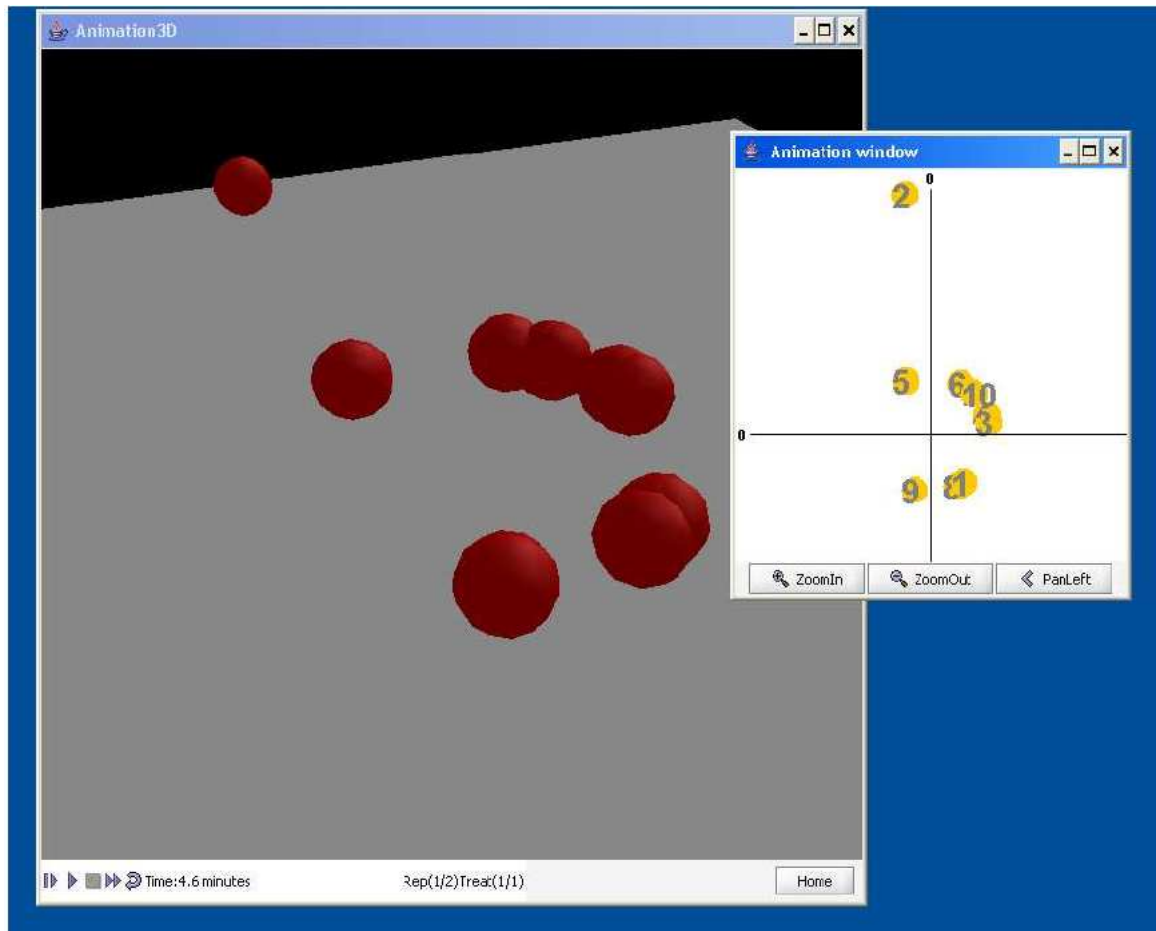
Figure 13: Animation 3D example

```
 7 package nl.tudelft.simulation.dsol.tutorial.section45;
...
25 public class Port
26 {
27     /**
28      * the jetties working in the harbor
29      */
```
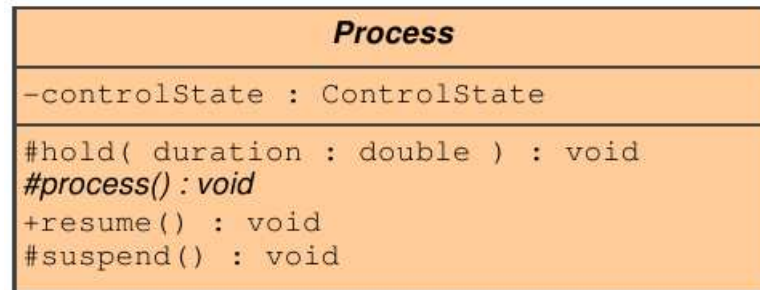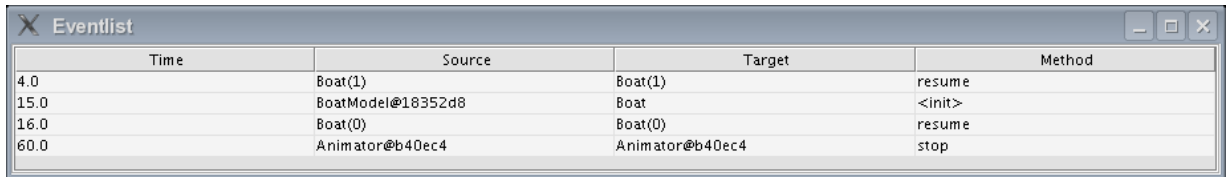
Figure 14: Abstract Process class

```
30      private Resource jetties = null;
31
32      /**
33       * the tugs working in the port
34       */
35      private Resource tugs = null;
...
42      public Port(final DEVSSimulatorInterface simulator)
43      {
44          super();
45          this.jetties = new Resource(simulator, "Jetties", 2.0);
46          this.tugs = new Resource(simulator, "Tugs", 3.0);
47      }
48
...
52      public Resource getJetties()
53      {
54          return this.jetties;
55      }
56
...
60      public Resource getTugs()
61      {
62          return this.tugs;
63      }
64 }
```

The process is expressed in the *Boat* class which extends the abstract *Process* class:

Time	Source	Target	Method
4.0	Boat(1)	Boat(1)	resume
15.0	BoatModel@18352d8	Boat	\<init\>
16.0	Boat(0)	Boat(0)	resume
60.0	Animator@b40ec4	Animator@b40ec4	stop

Figure 15: Event-list of Process interaction

```
 1 /*
 2  * @(#) Boat.java Jan 19, 2004 Copyright (c) 2002-2004 Delft University of
...
 8 package nl.tudelft.simulation.dsol.tutorial.section45;
...
29 public class Boat extends Process implements ResourceRequestorInterface
30 {
31     /** a reference to protect boats from being garbage collection */
32     protected Boat mySelf = null;
33
34     /**
35      * the port to enter
36      */
37     private Port port = null;
38
39     /** boat number */
40     private static int number = 0;
41
42     /** the description of the boat */
43     private String description = "Boat(";
44
45     /**
...
51     public Boat(final DEVSSimulator simulator, final Port port)
52     {
53         super(simulator);
54         this.mySelf = this;
55         this.port = port;
56         this.description = this.description + (Boat.number++) + ") ";
57     }
58
...
```

```
63      public void process()
64      {
65          try
66          {
67              double startTime = this.simulator.getSimulatorTime();
68              // We seize one jetty
69              this.port.getJetties().requestCapacity(1.0, this);
70              this.suspend();
71              // Now we request 2 tugs
72              this.port.getTugs().requestCapacity(2.0, this);
73              this.suspend();
74              // Now we dock which takes 2 minutes
75              this.hold(2.0);
76              // We may now release two tugs
77              this.port.getTugs().releaseCapacity(2.0);
78              // Now we unload
79              this.hold(14);
80              // Now we claim a tug again
81              this.port.getTugs().requestCapacity(1.0, this);
82              this.suspend();
83              System.out.println(this+" am alive @"+super.simulator.getSimulatorTime());
84              // We may leave now
85              this.hold(2.0);
86              System.out.println(this+" am alive @"+super.simulator.getSimulatorTime());
87              // We release both the jetty and the tug
88              this.port.getTugs().releaseCapacity(1.0);
89              System.out.println(this+" am alive @"+super.simulator.getSimulatorTime());
90              this.port.getJetties().releaseCapacity(1.0);
91              System.out.println(this+" am alive @"+super.simulator.getSimulatorTime());
92              System.out.println(this.toString() + "arrived at time=" + startTime
93                      + " and left at time=" + this.simulator.getSimulatorTime()
94                      + ". ProcessTime = "
95                      + (super.simulator.getSimulatorTime() - startTime));
96          } catch (Exception exception)
97          {
98              Logger.severe(this, "process", exception);
99          }
100     }
101
...
108     public void receiveRequestedResource(final double requestedCapacity,
109             final Resource resource)
110     {
111         this.resume();
```

```
112      }
113
...
122 }
```

Lines 63-100 define the process of the boat. It is a sequence of *request*, *hold*, and *release* statements. After requesting a specic capacity, the process is internally suspended. The process is notified of the availability of the resource by the *receiveRequestedResource* method. The process may then *resume*. Figure 15 presents the eventlist of this model. As could be concluded from figure 3 on page 15 the process interaction formalism is under-the-hood translated in discrete event scheduling.

5   Concluding remarks

This section ends the tutorial with a clear focus on the future of DSOL. First of all some remarks on its applicability and license. DSOL is published under the General Public License and will remain open source! We intend to provide a platform for collaborative development of next generation decision support.

In this tutorial a small number of unrealistic simple models are presented. In future publications we will publish better, more realistic and more complex examples. In our opinion the examples of this tutorial form a good basis to get started with DSOL. We are furthermore more then interested in joint projects or publications. Let us know what you have in mind!

Finally some topics which will be described in future documentation: distributing the DSOL services, 3D animation, linking DSOL to external data-sources, input analysis, and domain specific libraries.

About the authors

Peter H.M. Jacobs

Peter Jacobs finished his PhD at Delft University of Technology in 2005. His research focused on the design of 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@tudelft.nl>`.

Alexander Verbraeck

Alexander Verbraeck is a full 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@tudelft.nl>`.

Stijn-Pieter A. van Houten

Stijn-Pieter van Houten is a PhD student at Delft University of Technology since November 2003. He is mainly working in developing software for developing and using business games for supporting operational and tactical decision making in the field of logistics. A representative sample of recent experience with this software includes played business games with more than 200 (Executive) MBA students and demos for the following schools: R.H. Smith School of Business

and Graduate School of Business Administration Zürich. Prior to joining his doctoral program, he built an extensive knowledge in the domain of simulation modeling. In 2002, he worked at University of Genoa Laboratory of Logistics in Savona, Italy, on distributed simulation; from 2003 - 2006, he was actively involved in the supply chain management center of the R.H. Smith School of Business, University of Maryland, US.

A Concurrent Versioning System

If you want to access the sourcecode of DSOL you will have to download its code from sourceforge's concurrent versioning system. Figure 16 illustrates the values as they should be used.
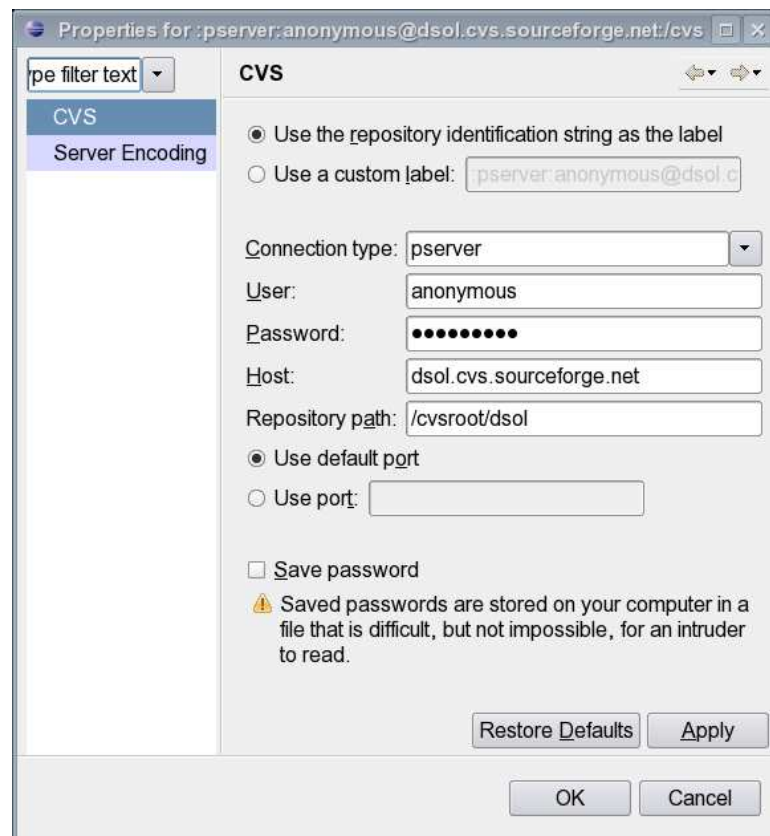

Figure 16: DSOL@Sourceforge.net

References

[Arnold et al., 2000] Arnold, K., Gosling, J., and Holmes, D. (2000). *The Java(TM) Programming Language*. Addison-Wesley, 3rd edition.

[Balci, 1988] 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*, pages 287–295. ACM Press.

[Banks, 1998] Banks, J., editor (1998). *Handbook of Simulation : Principles, Methodology, Advances, Applications, and Practice*. Interscience.

[Borrelli and Coleman, 1998] Borrelli, R. L. and Coleman, C. (1998). *Differential Equations*. John Wiley & Sons.

[Campione et al., 2000] Campione, M., Walrath, K., and Huml, A. (2000). *The Java(TM) Tutorial: A Short Course on the Basics*. Addison-Wesley, third edition.

[DoD, 1995] DoD (1995). Modeling and simulation master plan. https://www.dmso.mil. DoD 5000.59-P.

[Fishman, 1973] Fishman, G. (1973). *Concepts and methods in discrete event digital simulation*. John Wiley and Sons, New York, USA.

[Jacobs et al., 2002] Jacobs, P., Lang, N., and Verbraeck, A. (2002). A distributed java based discrete event simulation architecture. In *Proceedings of the 2002 Winter Simulation Conference*.

[Keen and Sol, 2003] Keen, P. and Sol, H. (2003). *Decision Support Systems for the New Generation*. draft version.

[Lackner, 1962] Lackner, M. (1962). Toward a general simulation capability. In *Proceeding of SJCC*, pages 1–14, San Francisco.

[Lang et al., 2003] Lang, N., Jacobs, P., and Verbraeck, A. (2003). Distributed, open simulation model development with dsol services. In *15th European Simulation Symposium and Exhibition*.

[Law and Kelton, 2000] Law, A. and Kelton, W. (2000). *Simulation modeling and analysis*. Mc Graw Hill, Singapore, third edition.

[L'Ecuyer, 1997] L'Ecuyer, P. (1997). Uniform random number generators: a review. In *Proceedings of the 29th conference on Winter simulation*, pages 127–134. ACM Press.

[Lotka, 1925] Lotka, A. (1925). *Elements of physical biology*. Williams & Wilkins Co., Baltimore, U.S.A.

[Nance, 1981] Nance, R. E. (1981). The time and state relationships in simulation modeling. *Commun. ACM*, 24(4):173–179.

[Nance, 1993] Nance, R. E. (1993). A history of discrete event simulation programming languages. In *The second ACM SIGPLAN conference on History of programming languages*, pages 149–175. ACM Press.

[Öeren and Zeigler, 1979] Öeren, T. and Zeigler, B. (1979). Concepts for advanced simulation methodologies. *Simulation*, 32(3):69–82.

[Overstreet and Nance, 1986] Overstreet, C. and Nance, R. (1986). World view based discrete event model simplification. In Elzas, M. S., Oren, T. I., and Zeigler, B. P., editors, *Simulatwn methodology in the Artificial Intelligence Era*, pages 165–179, Amsterdam, Netherlands.

[Shannon, 1975] Shannon, R. (1975). *Systems Simulation: the art and science*. Prentice Hall.

[Shanthikumar and Sargent, 1984] Shanthikumar, J. G. and Sargent, R. G. (1984). A unifying view of hybrid simulation/ analytic models and modeling. *Operations Research*, 31:1030–1052.

[Sol, 1982] Sol, H. (1982). *Simulation in information systems development*. PhD thesis, Rijksuniversiteit Groningen, Groningen, the Netherlands.

[Vangheluwe and de Lara, 2002] Vangheluwe, H. and de Lara, J. (2002). Metamodels are models too. In *Proceedings of the 2002 Winter Simulation Conference*.

[Volterra, 1926] Volterra, V. (1926). Variazioni e fluttuazioni del numero d'individui in specie animali conviventi. *Memorie della Classe di scienze fisiche, matematiche e naturali*, IV:31–113.

[Zeigler et al., 2000] Zeigler, B., Praehofer, H., and Kim, T. (2000). *Theory of modeling and simulation*. Academic Press, San Diego, CA, 2nd edition edition.