THE DISCRETE EVENT SIMULATION FRAMEWORK DESMO-J: REVIEW, COMPARISION TO OTHER FRAMEWORKS AND LATEST DEVELOPMENT

Johannes Göbel, Philip Joschko, Arne Koors, Bernd Page
Department of Informatics
University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
E-mail: {goebel, joschko, koors, page}@informatik.uni-hamburg.de

KEYWORDS
Discrete Event Simulation, Simulation Software, Simulation Framework, Open Source, Java, .NET, Visualization.

ABSTRACT

This review paper focusses on DESMO-J, a comprehensive and stable Java-based open-source simulation library. DESMO-J is recommended in numerous academic publications for implementing discrete event simulation models for various applications. The library was integrated into several commercial software products. DESMO-J's functional range and usability is continuously improved by the Department of Informatics of the University of Hamburg (Germany). The paper summarizes DESMO-J's core functionality and important design decisions. It also compares DESMO-J to other discrete event simulation frameworks. Furthermore, latest developments and new opportunities are addressed in more detail. These include a) improvements relating to the quality and applicability of the software itself, e.g. a port to .NET, b) optional extension packages like visualization libraries and c) new components facilitating a more powerful and flexible simulation logic, like adaption to real time or a compact representation of production chains and similar queuing systems. Finally, the paper exemplarily describes how to apply DESMO-J to harbor logistics and business process modeling, thus providing insights into DESMO-J practice.

INTRODUCTION

A practitioner aiming to conduct a simulation study has the choice between two types of simulation software to base his or her model on:

- *Integrated simulation development environments*, typically commercial software, often support the simulation study as whole, including data collection, model design, experimentation and evaluation. Model design often is done by *assembling* ready-to-use components on drag and drop basis in a graphical user interface. Plant Simulation (Siemens PLM Software, www.plm.automation.siemens.com) or FlexSim (FlexSim Software, www.flexsim.com) are well-known examples.
- *Simulation libraries* have a narrower focus, typically concentrating on model implementation and experimentation. They require models are *coded* in a general-purpose or special programming language, sacrificing comfort for flexibility. Though their modeling capabilities are often similar to commercial development environments, most of such libraries are open source software: apart from being available for free, advantages include source code analysis, debugging, modification, and the permission to re-distribute extended versions according to the relevant license. Examples include DESMO-J (Page 2013) and others, as compared later in this paper.

The purpose of this paper is easing the difficulty of this choice by clarifying the state of the art for the second type of software: This paper presents DESMO-J as example of a modern open source library for Java-based discrete event simulation. The subsequent section describes DESMO-J's functional range and important design decisions and provides a comparison to other open source simulation libraries. For examples of how DESMO-J is applied in different real-world scenarios consider the next section, namely extensions for harbor logistics and business process modeling. The following section can be understood as an update to previous publications like Page and Neufeld (2003) and Göbel, Krzesinski and Page (2009), as the latest extensions to DESMO-J are described, namely real time capability, 2D and 3D visualization, a .NET port, efficient simulation of processes based on coroutines and continuations, generic components for production chains and other queuing systems, recording and logging of simulation objects, as well as advanced simulation dynamics analysis on basis of quantitative finance risk metrics. This permits the conclusion in the final section that libraries like DESMO-J should be recognized as alternatives to commercial development environments.

DESMO-J

DESMO-J (Discrete-Event Simulation and Modelling in Java) is a comprehensive framework for developing discrete event simulation models, see Banks et al. (2010) or Page and Kreutzer (2005), in the object-

Proceedings 27th European Conference on Modelling and Simulation ©ECMS Webjørn Rekdalsbakken, Robin T. Bye, Houxiang Zhang (Editors)
ISBN: 978-0-9564944-6-7 / ISBN: 978-0-9564944-7-4 (CD)

oriented programming language Java. The first subsection discusses the reasons for choosing Java, followed by a short primer about modeling in DESMO-J. Afterwards, the most important design decisions and a comparison to other simulation libraries are addressed.

Implementation Language

Simulation modeling in DESMO-J actually means implementing models in *Java*. Particularly, the model structure including properties and behavior of all components has to be coded in appropriate Java classes. In contrast, the simulation infrastructure, e.g. simulation clock, event list, random number distributions and experiment conduction including reporting is readily available.

In comparison to the other main approach of providing simulation functionality to a user, namely graphically assembling models on "drag and drop" basis, simulation programming may be less intuitive to learn (especially for beginners) and slower to apply; this at least holds for standard cases like production lines which are covered by the building blocks included in GUI-based modeling environments like Plant Simulation or FlexSim. However, the most important advantage of simulation programming based on a library like DESMO-J is flexibility, as any model logic can be described in a general purpose programming language like Java; no constrains are imposed by a restricted simulator API or a product-specific script language. Therefore, DESMO-J is particularly well-suited for complex models for which graphical modeling cannot be done adequately and efficiently.

Furthermore, choosing Java as simulation programming language means addressing a large community of programmers. It ensures all features of a modern object-oriented language are available. Java's pervasiveness is unmatched: so-called Java virtual machines are provided for almost every modern operating system. Java programs can not only be executed on desktops, but also as Web Service or as Applets on web sites. Just-in-time compilation including optimization of Hot Spot execution and garbage collection has helped to achieve run time performance similar to languages that compile sources to binary code.

DESMO-J in a Nutshell

We refer to DESMO-J as a simulation *framework* as it provides a coherent software architecture of components exhibiting a well-defined cooperative behavior designed to effectively and conveniently serve the task of model building and experiment conduction: as much implementation effort as possible is removed from the user. Wherever feasible, DESMO-J makes available so-called *black-box* components, which are classes that are ready-to-use. Such classes are parameterized by the user; usually, their code needs not be touched. Figure 1 shows the most important classes from DESMO-J's core functionality. Black-box components include the `Experiment` class responsible for conducting discrete event simulation runs. Following the *façade* design pattern, `Experiment` hides the infrastructure it requires, like the scheduler and the event list it operates on, the simulation clock, and the generation of experiment reports. Additional black-box components offer generic model components like queues with finite or infinite space, random number generators based on a variety of random number distributions and different means of collecting statistical data. Important functionality of the statistics classes includes counting, uniform or time-weighted aggregation of samples, determining confidence intervals and generating histograms. The last-mentioned black-box components are subclasses of `Reportable`, automatically generating statistical data available in the experiment report.

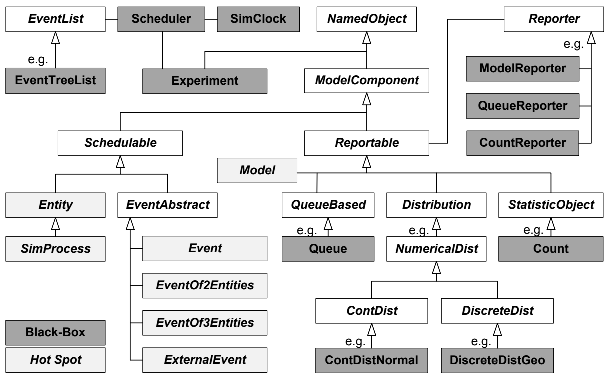

Figure 1: Important DESMO-J Classes

With this set of black-box components at hand, the modeler is able to focus on implementing the logic of the system to simulate by using additional objects referred to as `Schedulable`s. Such `Schedulable` objects can be put onto the event list; they are typical examples of DESMO-J's white-box components or *hot spots*: the unknown structure and behavior of a user's model require more flexibility than parameterized black-boxes can provide. Consequently, hot spots are abstract Java classes whose methods have to be completed by the user.

To implement a model, the DESMO-J user may choose between the event-oriented and the process-oriented view:

The *event-oriented perspective*, also referred to as "bird's eye view", requires the user to describe the model behavior in terms of event routines which are assumed executed as an atomic transaction without interruption and without simulated time passing. Model dynamics arise from sequentially executing events. Entities are represented by classes inheriting from `Entity`. The events implemented by the user have to be derived from one of the four subclasses of `EventAbstract`; which subclass to base a modeled event on depends on the number of entities associated to the event, namely one (`Event`), two (`EventOfTwoEntities`), three (`EventOfThree Entities`) or none (`ExternalEvent`). For example, a service end event of an item leaving a machine in a production line typically is modeled as an `EventOfTwoEntities`, as two entities are affected: one item proceeding to the next machine and one machine processing the next item, if available. In contrast, an event referring to no specific entity, but to the system as whole, e.g. a power failure, could be implemented as `ExternalEvent`. An event's behavior is defined in its `eventRoutine()` method; typically, event routines include entities being created and destroyed, entities entering or leaving queues, statistical data collectors being updated and further events scheduled or cancelled.

In contrast, *process modeling* describes model logic in terms of processes that persist as simulation time passes. Model dynamics arise from process interaction and transfer of program control among each other. For each process, the user has to subclass `SimProcess`, providing a *life cycle* containing the behavior of the process over time, yielding a "worm's eye view" of the model. In their user-defined `lifeCycle()` methods, processes may create other processes (which are special entities), modify queues or update statistic objects. Furthermore, simulation processes are able to wait for a certain period of simulated time ("hold") or for an indeterminate period ("passivate") until activation by another process ("activate"). A process may interrupt another process on hold, causing the interrupted process to resume its life cycle execution at a time instant prior to its original schedule.

With true *coroutines* not being available in Java, process execution internally is based on event scheduling: each process runs in its own Java thread; process threads are suspendable and are resumed by events implicitly scheduled when processes are held or activated/interrupted. Note that the section describing latest developments presents an alternative approach of implementing processes which is less resource-consuming.

DESMO-J does not enforce an exclusive decision for either event or process modeling; the user is free to combine both modeling styles in a single model (e.g. an event activating a process which in turn schedules another event), so that the modeling perspective best suited for each specific aspect of a model can be applied.

Comparison to other Tools

When conducting a simulation study using a Java-based simulation framework like DESMO-J, the model developer may choose out of a number of different tools. An of course non-exhaustive list of open source discrete event simulation libraries in Java includes

- DESMO-J (Page 2013),
- DSOL (Verbraeck 2009),
- J-Sim (Kačer 2006),
- JavaDEMOS (Computer Science Group 2009),
- JSL (Rossetti 2013),
- PtolemyII (Lee 2011),
- SimKit (Buss 2012) and
- SSJ (L'Ecuyer 2012).

For URLs of these libraries including API documentation and examples see the corresponding Reference entries. Table 1 compares some key features of DESMO-J and its competitors.

Table 1: DESMO-J compared to other Java Discrete Event Simulation libraries

Package name	Events / Processes	Visualization	Random distributions	Tutorial / Examples	License **	Commercial use	Last version
DESMO-J	E/P	2D/3D	25	Yes	ASL2.0	Yes	2013
DSOL	E/P	2D	21	Yes	Special	–	2009
J-Sim	E/P	–	5	Yes	AFL2.1	–	2006
J'DEMOS	E/P	–	15	Yes	Special	–	2009
JSL	E	–	9	Yes	GPL	–	2013
PtolemyII	E/P*	2D	23	Yes	Special	Yes	2011
SimKit	E	2D	25	Yes	LGPL	Yes	2012
SSJ	E/P	–	64	Yes	GPL	–	2012

* Processes based on an Actor approach
** ASL = Apache Software License, AFL = Academic Free License, GPL = GNU General Public License, LGPL = GNU Lesser General Public License

These key features include support of event and process modeling and availability of 2D or 3D visualization of model behavior; different means of 2D visualization are available, e.g. schematic representation of the spatial model structure similar to Plant Simulation, where entities are drawn using icons, or important statistics as displayable in DESMO-J, or dynamically annotated event graphs as in DSOL. The number of random distributions has not much intrinsic value on its own, as generators for additional distributions can be implemented quickly. Nevertheless, it is included in the table as an exemplary indicator for each framework's extent, which is difficult to measure. E.g. number of classes or download size might be misleading measurements, as they depend on various design decisions, e.g. few monolithic or many specialized classes, data collection separated from the source generation of the data or not, functionality partially delegated to sub-libraries… Furthermore, the table addresses the availability of tutorials or example models, the licenses under which the libraries are available to the public and whether commercial use is permitted. For commercial application, we particularly require permission for usage in closed-source, proprietary software and inclusion in releases of such software without endorsement from the libraries' authors. The table concludes with the date of the most recent version as of February 2013.

Observe that the combination of features DESMO-J offers is unmatched among the other Java-based simulation frameworks: as already explained, DESMO-J allows event and process modeling and offers both 2D and 3D visualization of the model behavior as will be described below. The DESMO-J website (Page 2013) contains an extensive tutorial not only describing how a container logistics example model is implemented using either events or processes and how experiments are conducted. A variety of advanced topics is also ad-

dressed, e.g. different data collectors and higher modeling features like conditional waiting or implicit process synchronization.
Another unique feature is the availability of a companion book: *The Java Simulation Handbook* (Page and Kreutzer 2005), available as printed version and as eBook, covers discrete event simulation fundamentals and simulation modeling based on UML and DESMO-J as well as simulation statistics, model validation and verification, multi-agent simulation, simulation optimization, simulation projects in practice and various other topics.

EXTENDING DESMO-J AND APPLICATION SCENARIOS

General Expandability

Every time a model is implemented with DESMO-J by deriving entities and processes/events from DESMO-J classes, a kind of 'domain-specific extension' is written. Classes designed with the intention of general reusability within diverse models are called DESMO-J *extensions*. These might be general, more technical extensions like multi-agent-based simulation entities (Knaak, Kruse and Page 2006) or domain-specific extensions, containing reusable entities for easier composition of models in that particular domain, see next sections or Joschko, Page and Wohlgemuth (2009).
Furthermore, DESMO-J can be integrated into other software products, such as extensive modeling suites with own graphical user front-end and model editors, allowing modeling without writing Java code. Due to the flexibility of ASL 2.0 under which DESMO-J is licensed, it is possible to implement individual solutions without using an open-source license – an important issue in non-public, commercial projects.
Since expandability is a very important aspect in using DESMO-J, we sum up some domain-specific solutions in the following sections.

Harbor and Container Terminals

Simulation is an established method for optimizing strategies and resource allocation in logistic contexts. Since Hamburg accommodates one of the ten largest container ports worldwide, we had the opportunity to gain substantial experience in simulating container terminals in a number of cooperation projects. We presented our first DESMO-J harbor extension in Page and Neufeld (2003). This class library extension is still available in DESMO-J, offering three types of objects: dynamic, mobile, temporary objects like ships, trucks and trains; dynamic, mobile, permanent objects like cranes and van carriers; and stationary, permanent objects like holding areas, gates, jetties and yards (Joschko, Brandt and Page 2009).
Worldwide, many other working groups use DESMO-J in logistic investigations in harbor context, see e.g. Asperen et al. (2004) and Henesey, Aslam and Khurum (2006).
The traditional aim in executing logistic simulation experiments is to compare different handling strategies in order to determine terminal layout or optimize usage of transport vehicles (Bornhöft, Page and Schütt 2010). These strategic simulation approaches take place in the design and implementation phase of container terminals. In the operation phases of container terminals, tactical simulation can be used to support decision-making in resource allocation, finding good storage positions and accepting orders.
A completely different approach in this phase is to use simulation for integration tests on terminal operating systems. Together with the Hamburger Hafen und Logistik AG (HHLA), we implemented a DESMO-J extension for a broad range of applications in the context of container terminals, called COCoS, see Brandt (2008) or Joschko, Brandt and Page (2009). Entities in COCoS (van carriers, quay cranes etc.) are assembled from different exchangeable layers and sub-components that manipulate model state by scheduling DESMO-J events. The granularity depends on the level of detail needed for the object of investigation. Whereas in logistic experiments an abstract, stochastic representation of transport device behavior is needed, a high level of detail is required when connecting the model to a real terminal operating system. The exact kinematic characteristics of transport devices have to be mapped. A TCP-based communication layer enables message exchange between the simulation model and the container terminal operating system. Last but not least, deceleration of simulation adjusts the model to real time (see next chapter). A graphical user interface comprises visualization of the model's state and buttons permitting user interaction with the job list or a device. Fulfilling these conditions with DESMO-J and COCoS, a simulated terminal system can be controlled by a real terminal operating system. In this way, a "terminal operating system can be tested with help of a terminal model" (Joschko, Brandt and Page 2009).

Business Process Modeling

In Business Process Analysis, a graphical modeling notation (BPMN, EPC, UML, Petri-Nets etc.) is used to visualize production processes and information flows. Apart from other purposes, such a graphical representation facilitates communicating existing procedures and discussing improvements. Augmenting such methodology with simulation capabilities enables empirically founded comparison of alternatives, e.g. resource allocation or strategy optimization. Regardless of the chosen modeling notation, a business process model can be transformed into a simulation model if it is enhanced with simulation properties. Particularly, stochastic parameters affect the duration of activities and the interarrival time of events. Once again, resource allocation is one of the most interesting issues. The total cost of activities, the number of concurrently running processes, the duration of (sub-)processes, the length of waiting queues and the occurring frequency of specified events are also relevant performance indicators.

Several commercial business process modeling tools use DESMO-J as simulation engine in order to support such analysis. To our knowledge, DESMO-J is a part of *Tibco Business Studio*, *Borland Together*, *eClarus Business Process Modeler for SOA Architects* and *Intellivate IYOPRO*, the latter being our favorite in user friendliness. This list may be incomplete as not all DESMO-J software integrators necessarily get in touch with us.

In cooperation with Intellivate GmbH, the developer of IYOPRO, our working group has developed a DESMO-J extension for simulating business processes notated in Business Process Model and Notation 2.0 (BPMN). Since IYOPRO is a Silverlight web application, we used a .NET port of DESMO-J (see below) in order to implement a BPMN extension for DESMO-J. This software now contains a special BPMN-process derived from DESMO-J's `SimProcess` class as well as implementations of most BPMN flow elements, like activities, several event types and sequence flows. Furthermore, it includes message flows, pools, swim lanes and data-objects derived from DESMO-J's `Entity` class. Integrated into the graphical model editor of IYOPRO, model parameters can be set using a property editor. Additionally, process variables for data-objects and expressions for splitting gateways can be declared. Therefore, the choice of path can depend on the state of a process instance. The simulation report is enhanced with pie charts and histograms, linked to the corresponding model elements. See Joschko et al (2012) for more details about business process simulation, enhancing models for simulation purposes and deploying simulation experiments using BPMN 2.0 and IYOPRO.

LATEST DEVELOPMENT

Leaving the application level, we now describe some recent features of DESMO-J itself, thus giving an insight into the library's continuous development process.

Real-time Capability

The handling of simulation time has been completely re-engineered as of DESMO-J version 2.2.0, see Klückmann (2009): typically, a simulation experiment is executed as fast as possible; simulation time advance depends on CPU speed only. Special cases, however, may require intentionally decelerating an experiment: examples include concurrent animation or real-word systems in which the behavior of some components is emulated by a simulator. DESMO-J now offers the feature to link simulation time advance to real time, subject to a user-defined time lapse factor. If this factor is set to 1, the simulation experiment will execute synchronously to real time.

Re-engineering time handling also introduced some minor improvements: for the modeler's convenience, references to time can alternatively be based on time instants or durations, thanks to parameter overloading: for example, either an absolute point in time (`TimeInstant`, "hold until") or a duration (`TimeSpan`, "hold for") can be passed to a process' `hold()` method. Additional improvements include the support of `java.util.Calendar` and `java.util.Date` for reading and writing time statements, multiple time zones in a single model, a class for shift schedules, as well as time-weighted data collection (`Accumulate`) being switched on and off, e.g. in order to ignore a night's downtime.

2D Visualization

Modern simulation tools support model state and behavior visualization; reasons include communication with model users and decision makers as well as detection of erroneous model logic and – though no replacement for a statistically well-founded analysis – basic means of evaluation, e.g. identification of potential bottlenecks. DESMO-J supports two different means of visualization, presented in this and the next subsection.

The 2D visualization component – a contribution of Prof. Dr. Christian Müller and his research group at the Technical University of Applied Sciences Wildau, Germany – provides a means of schematically representing the model logic on a 2D plane: every entity (including processes, compare Figure 1) can be shown in the visualization, after an icon and location in terms of x/y-coordinates have been assigned. Visualization supports uniform entity motion from an origin to a destination during a certain time span on a pre-defined path. Entities can also be shown inside a waiting area of queues while enqueued. Furthermore, data collectors can be included in the visualization, featuring their current or last values as well as mean and standard deviation values. Figure 2 shows an example screenshot from a bungee tower model; see the DESMO-J webpage (Page 2013) for a Java Applet version of this animation running directly inside the web browser.

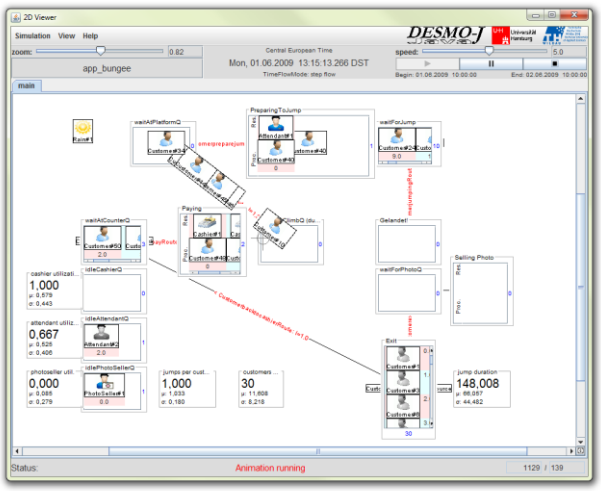

Figure 2: Screenshot of the 2D Visualization of a bungee tower model

To create a 2D visualization, a developer just has to replace the DESMO-J core classes like `Model`, `Entity`, `SimProcess`, `Queue` or `Count` with their appropriate subclasses `ModelAnimation`, `EntityAnimation`, SimProcessAnimation, QueueAnimation or CountAnimation from the 2D visualization package. Constructors and methods are identical to the core classes except for additional means for defining a position or exchanging an entity's icon reflecting a state modification, e.g. job completion or change of order priority.

As 2D visualization works *offline*, a simulation experiment has to finish before visualization starts: rather than providing a "live" view of the experiment, opposed to the 3D framework described below, a simulation run generates an XML-script describing all updates to appear in the visualization, e.g. entity motion or data collector values adjusted. After completion of a simulation run, a viewer is launched in which the script can be played back. Basic features include zooming and adjusting animation speed (time lapse, stepwise execution). Additional examples, a 2D visualization tutorial and documentation are available at a dedicated web site (Müller 2011).

3D Framework

Alternatively, we provide a 3D framework which includes several libraries for three-dimensional modeling and visualizing. First, there is a DESMO-J extension which provides a basic spatial concept, see Sun (2010): the interfaces SpatialObject and MovableSpatialObject enhance the DESMO-J classes Entity, SimProcess or Queue with coordinates, orientation and movement behavior. The class SpatialData encapsulates coordinates and orientation in a 4 × 4 transformation matrix, thus movements are represented as matrix multiplications. The environment's layout contains navigation points and routes between them. It can be defined in an XML file.

Second, there is an optional kinematic library for calculating the arrival time of entities, requiring acceleration, deceleration and maximum speed of a MovableSpatialObject being given. While position and orientation are calculated when a movement is finished, the class SpatialMovementManager interpolates speed, position and orientation of objects on demand. Instead of the kinematic calculation, arrival time can also be scheduled conventionally by stochastic distributions.

Third, the visualization framework animates the position, orientation and movement of objects with help of OpenGL and Java 3D. 3D shape files are linked to logical model elements by an XML file, thus the visual appearance of objects is determined. Input and output ports enable 3D modeling of entities like queues. In order to get messages about movement events, the visualization framework signs itself up at the spatial classes. Between start and termination of a movement, the actual position is updated regularly by the SpatialMovementManager. The parallel deployment of spatial concept, kinematic library and visualization framework enables three-dimensional, concurrent animation *during* a simulation run. Figure 3 shows a 3D visualization of a simple logistics model.

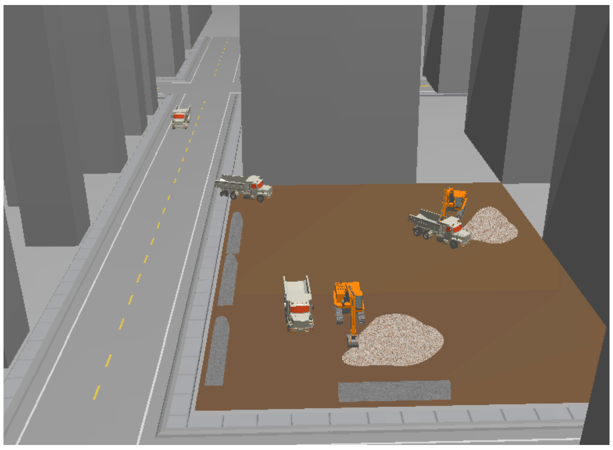

Figure 3: Screenshot of the 3D Visualization of a truck loading model

.NET-Versions of DESMO-J

This far, integration of DESMO-J into existing software suites is limited to Java. A platform similarly widespread is .NET from Microsoft. Like Java, .NET is based on a virtual machine, yet it allows programming in diverse languages, e.g. C# or Visual Basic. Since DESMO-J is constantly improved and enhanced, maintaining two branches of DESMO, one in Java and one in C#, would have been too costly. Nevertheless, a .NET version of DESMO-J always aligned with the maturity level of the Java version was desirable. We successfully explored two approaches of automatically porting DESMO-J into .NET's Intermediate Language code or into C# source code, respectively.

IKVM is an implementation of the Java Virtual Machine for .NET and Mono, see Frijters (2012). It provides an implementation of the Java standard libraries and some tools which enable interoperability of Java and .NET respectively Mono classes. The command 'ikvmc' is able to compile a .NET DLL file out of a Java JAR file. The resulting DLL file has dependencies to several IKVM libraries. These files can be included in a .NET based software application and used as if they were ordinary compiled .NET classes. Since differences between Java and .NET do exist, we examined the feasibility of this solution in practice. We tested whether the behavior of a 'DESMO.NET' library produced by IKVM is identical to that of the original DESMO-J library. Therefore, we compared simulation results of several simulation runs with several models, and could not find any deviations.

As a consequence, we developed a sophisticated ERP simulation model in C#, interfacing with a Microsoft Windows-based ERP system in a .NET environment, while employing broad simulation functionality of DESMO-J converted to .NET by IKVM (Kühnlenz 2011, Schäfer 2011). Though the model was quite complex and extensively used simulation functionality, we did not encounter any problems introduced by the conversion process of IKVM.

Another approach is to directly transform the Java high-level language code into C# code. The syntax of Java and C# is quite similar. However, method calls to classes in the Java core libraries have to be mapped to equivalent method calls in the .NET framework.

Among the tools supporting such transformations, we gathered experience with the open-source software Sharpen (2013). As the transformation process is incomplete, additional work is required. First, converting multi-dimensional arrays and changing the parameter order of class library methods is not fully supported. Second, the tool does not adequately resolve some particular differences between Java and C#. E.g. in C# it is not possible to reference raw types of generic classes, a technique used in DESMO-J.

We developed an Ant script that prepares the Java code before transformation into C# (e.g. removing raw type references), and adjusts the result in order to eliminate remaining errors. In consequence, we are now able to generate C# source code, which is nearly equivalent to our Java-based DESMO-J.

We argue that IKVM offers a fast, easy and reliable way to generate .NET versions out of DESMO-J. If a more lightweight solution is desired that does not require integration of IKVM libraries into the target simulation application, transforming Java sources into C# sources is feasible, with an additional manual effort.

Alternative Process Implementation

The class `SimProcess` (see DESMO-J introduction) internally relies on an instance of `java.lang.Thread` for life cycle execution. This permits halting a simulation process whenever needed, persisting its method pointer and process state, and reactivating it at a later time. A disadvantage of using `java.lang.Thread` is the upper limit of concurrent existing threads in Java – independent of whether they are actually working in parallel or not. For a typical JVM, the maximum number of concurrently existing simulation processes is approximately 2500 plus a few thousand additional processes obtainable by increasing heap space. However, if millions of concurrent processes are needed, the model had to be implemented in an event-oriented manner, until recently.

Now, we present an alternative `SimProcess` implementation that allows simulating huge numbers of concurrent processes in the process-oriented world view (e.g. simulation of telecommunication compatibilities). It is based on the concept of continuations and coroutines, which are not included in the Java standard libraries so far. However, the Apache library JavaFlow fills this gap by providing the concept of continuations. This library is licensed under ASL 2.0. The continuation class permits the implementation of custom coroutines that run mutually exclusively in a single thread. Testing this implementation, we aborted the simulation run manually, after reaching 2.5 million concurrently existing simulation processes.

Unfortunately, this solution requires byte code re-engineering, not only of the simulation model, but of all classes that may appear on the method stack of a coroutine. Although this can be automated based on e.g. an Ant task, we do not provide this functionality in our standard deployment since compiling becomes unnecessarily complicated for learners and most users. However, all necessary classes, libraries and a build script including byte-code re-engineering are obtainable from our SVN repository for those interested in the alternative simulation process implementation delineated above.

Processing Chains

Model logic frequently consists of repetitive tasks to be executed by multiple model components in a similar way. For example, consider work stations in a typical production line processing items and forwarding them to the next stations. Stations may e.g. differ in processing and setup time distributions. Efficient modeling of such systems may be conducted by providing complex, integrated components specifically designed for the relevant application domain, as e.g. described above for the example of harbor logistics. For application areas, however, in which such solutions do not (yet) exist, a level of abstraction between basic event/process modeling and domain-specific components is desirable, facilitating a compact and redundance-free representation of models containing similar or repetitive tasks.

To address this need, the DESMO-J core contains a set of *higher-level modeling components* since its earliest versions, e.g. finite resource pools or buffers: if a pool or buffer contains fewer resources than requested by a consuming simulation process, the process is implicitly passivated until its demand is met. The user needs not explicitly activate the process at the right instant of time; instead, s/he may proceed in process description, assuming the resource has been acquired successfully.

The *chaining* components (package `desmoj.extensions.chaining`) are higher-level modeling components no longer necessarily requiring an event or process description of the model behavior at all: such components – representing sources, work stations, sinks, mergers and splitters in a queuing system – offer comprehensive means of parameterization (e.g. a workstation: buffer size, number of parallel processors and distributions of setup time, processing time, recovery time, transport time) and, most importantly, they can be chained to each other: e.g. the output of a source is assumed forwarded to a work station. This permits describing basic production or queueing systems with very few lines of code. At the same time, the implementation remains flexible, as more complex work station behavior can be introduced based on subclassing. Flexibility also includes interaction with non-chaining components: all entities whose types do match can be fed into a work station, not only those created by the chaining source. On the other side, the description of what happens with the output of a station is encapsulated as event, which defaults to a forward to the next station; an alternative event provided by the user may for example divert some entities to another station or cause a re-entry to the current station with a certain probability.

Recording and Logging

In various application areas, it may be of interest to examine particular experiment phases in detail, e.g. transition from a warm-up phase to a steady state phase or disruptions of steady state phases.
Although DESMO-J's simulation trace output may be turned on and off at any time during an experiment, the resulting file by default only contains the most important data, like model, time, the acting entity, process or event and the action itself, e.g. scheduling entities or activating/passivating processes, queue manipulation, random number access or statistical updates.
If certain constellations of simulation objects have to be explored in detail, the output trace files are a) too coarse, b) safely accessible only after an experiment has finished and c) not in an easily machine-readable format.
To address these shortcomings, the concept of *recording*s was introduced. A recording contains a sequence of observations of any type of simulation object, e.g. `double` or `long` values, `String`s, entire `Entity` objects or even the whole state of a waiting `Queue`, over consecutive simulation time instants.
A recording may be paused or resumed at any time during a simulation experiment. Its contained sequence of copies of original simulation objects is ready for analysis by further algorithms at any point of simulation time. Recordings are typed, and for clarity we recommend to employ one recording per individual observation variable. Hence, examining the interdependencies of *n* simulation objects leads to *n* recordings. For ease of use, any number of recordings may be linked to a *recorder*. A recorder controls the recordings that registered with it. Pausing and resuming a recorder is passed on to all of its assigned recordings, enabling the experimenter to centrally handle whole groups of recordings. Thus, entire interconnected segments of a model may be recorded, e.g. when observing critical model behavior.
Recordings are created volatilely in memory, with no default mechanism of persistence. As it may be of interest to study recordings after simulation has finished or to visualize recordings during simulation execution, *loggers* may register at recordings. Whenever a recording is updated with an observation, it forwards a copy of the observation together with the current simulation time stamp to all registered loggers. A concrete logger (derived from the abstract class `Logger`) may implement any desirable behavior in order to process the observed data on-line during an experiment run. Applications include – but are not limited to – writing observation objects to files (Text, XML, CSV etc.), storing observations in databases, conducting model-dependent computations on observation streams, representing observation object states in specially tailored online GUIs or simply printing observations on the console, for tracing und debugging purposes.
In all of these scenarios, the simulation framework does not need to contain methods or knowledge of how to process observations in the context of files, databases, GUIs etc., this is left to the registered corresponding loggers. This IOC approach ("inversion of control") has only been used internally in DESMO-J to date, e.g. in reporting and statistics, but so far had not been offered as an interface to arbitrary downstream functionality or external applications.
Now, a comfortable mechanism for easier unidirectional integration with external software components or functional extensions is available.

Risk Metrics

Until recently, simulation dynamics could only be captured in DESMO-J standard statistics, regarding observation state variables of interest. These statistics typically accomplish counting of (arriving or leaving) entities, tallies or histograms of waiting times, or time weighted accumulation of queue length or resource utilization.
Most standard statistics comprise mean, minimum, maximum and standard deviation values; histograms additionally offer a visual impression of state distribution of observation variables.
Nevertheless, none of these statistics gives an idea of e.g. how fast the state of an observation variable shifts from the median observed state to an extreme state or how typical pathways of fluctuations in steady state phases can be characterized.
In order to give better insight into the potential and risk of model dynamics, the four most accepted risk metrics from the application field of Quantitative Finance have been generalized and transferred to discrete event simulation (Koors and Page 2012). Namely, these metrics are Semi-Variance, Value at Risk, Expected Shortfall and Drawdown.
Semi-Variance measures state deviation, accounting only for positive resp. negative deviation from the mean state. If positive and negative Semi-Variance differ significantly from each other, model dynamics towards higher or lower observation variable states is distributed asymmetrically and should be examined carefully.
Value at Risk was generalized to the metric Delta at Risk (DaR). DaR quantifies the maximum extent of state change expected (i.e. risk, in terms of quantitative finance) with regard to a chosen confidence level α, after a certain time interval, and according to four well-defined reference states of an observation variable (minimum, median, maximum and most frequent observed states). A typical conclusion based on the simulation report could be "Starting with an empty queue and given a confidence level of 99%, the queue length will not exceed *z* entities after 1 hour of simulated wall clock time".
Expected Shortfall was generalized to the metric Conditional Delta at Risk (CDaR). CDaR expresses the expected mean state for the α fraction of cases where DaR is exceeded. A typical finding based on the simulation report could be "If, starting with an empty queue and

given a confidence level of 99%, the queue length exceeds the Delta at Risk value of *z* entities after 1 hour of simulated wall clock time, then an average queue length of $z + c$ entities can be expected". Hence, CDaR is a metric for estimating the extent of state movement in unlikely cases of extreme events (in terms of the choice of α).

In steady state phases, Drawdowns and RunUps describe the magnitude and time structure of interim downward or upward phases in observation variable state, until the median or most frequent state is reached again. This metric and its various self-elaborated derivatives and analysis options give a good insight into distribution, characteristics and individual pathways of both usual and extraordinary model dynamics.

The set of four risk metrics described above depends on the same data basis, and especially Delta at Risk and Conditional Delta at Risk share the same basic time series. As an alternative type of statistics implementation, none of these metrics saves its own internal data, like DESMO-J standard statistics do. Instead, all four metrics refer to commonly shared recordings (see section above) set by the modeller. Thus, a noticeable amount of memory space and processing time is saved by avoiding redundant collection of basic statistic state observations.

All in all, the newly introduced risk metrics facilitate a better assessment of risky or desirable model dynamics than the DESMO-J standard statistics could provide to the experimenter before.

SUMMARY

This paper's aim was to clarify the state of the art in open source simulation libraries by exemplarily presenting functional range and usability of DESMO-J, including a comparison to other Java based simulation libraries. We have pointed out technical improvements like real time capability, recording and logging functionality, which is useful for coupling simulation models to external systems. An alternative process implementation allows concurrent existence of millions of process entities. We have presented two alternatives of automatically generating a "DESMO.NET" out of DESMO-J Java sources. Moreover, enhancements in modeling like processing chains and further analysis functionality like generalized risk metrics have been delineated.

We also have introduced two visualization extensions. While the 2D visualization package can easily be adapted to existing models, the 3D visualization package needs more implementation effort, as 3D shapes for entities are required.

Describing container terminal simulation and business process modeling, we gave two examples how domain-specific simulation applications can build upon DESMO-J. Useful features include implementation of graphical editors, customizing simulation reports as desired and embedding DESMO-J models into external systems.

Despite a variety of new features being introduced, an important design criterion is the backward compatibility of DESMO-J, ensuring models built upon older versions of DESMO-J will also work with the newest version.

Finally, we emphasize that DESMO-J is a powerful, flexible and easily usable simulation framework, recommending it to the reader as a tool to consider for the next simulation study.

ACKNOWLEDGEMENT

We would like to thank Sönke Claasen, Nicolas Denz, Johannes Haan, Tim Janz, Gunnar Kiesel, Felix Klückmann, Sven Kruse, Tim Lechler, Christian Mentz, Ruth Meyer, Christian Müller, Olaf Neidhard, Eugenia Neufeld, Thorsten Planeth, Thomas Schniewind, Fred Sun, Malte Unkrig and Jörg Willig for providing their work to the DESMO-J community.

REFERENCES

Asperen, E. van; R. Dekker; M. Polman; and H. de Swaan Arons. 2004. "Arrival processes in port modeling: insights from a case study". Available at http://publishing.eur.nl/ir/repub/asset/1275. Accessed 2013-02-13.

Banks, J.; J.S. Carson II; B.L. Nelson; and D.M. Nicol. 2010. *Discrete-event system simulation.* Pearson, Upper Saddle River NJ.

Bornhöft, K.; B. Page; and H. Schütt. 2010. "Modelling of innovative Technologies for Container Terminal Yard Stacking Systems using an Object-Oriented Simulation Framework". In *The International Workshop on Applied Modelling and Simulation.* Rio de Janeiro.

Brandt, C. 2008. "Entwurf und Implementierung eines Frameworks zur Entwicklung von Containerterminal-Gesamtmodellen mit DESMO-J". Master's thesis, University of Hamburg, Hamburg, Germany.

Buss, A. 2012. Simulation software *SimKit*. Naval Postgraduate School, Monterey. Available at http://diana.nps.edu/Simkit. Accessed 2013-02-13.

Computer Science Group. 2009. Simulation software *JavaDEMOS*. University of Duisburg-Essen. Available at http://sysmod.icb.uni-due.de/index.php?id=52. Accessed 2013-02-13.

Frijters, J. 2012. Java implementation *IKVM*. Available at http://www.ikvm.net/. Accessed 2013-02-13.

Göbel, J.; A.E. Krzesinski; and B. Page. 2009. "The Discrete Event Simulation Framework DESMO-J and its Application to the Java-based Simulation of Mobile Ad Hoc Networks". In *Proceedings of the 21st European Modeling and Simulation Symposium, Vol. I,* R.M. Aguilar, A.G. Bruzzone; and M.A. Piera (Eds.). La Laguna, Spain (Sep).

Henesey, L.; K. Aslam; and M. Khurum. 2006. "Coordination of Automated Guided Vehicle in a Container Terminal". In *Proceedings of 5th International Conference on Computer Applications and Information Technology in the Maritime Industries.* Oud Poelgeest, Netherlands.

Joschko, P.; J. Haan; T. Janz; and B. Page. 2012. "Business Process Simulation with IYOPRO und DESMO-J". In *Proceedings of the International Workshop on Applied Modelling and Simulation,* Bruzzone, Buck, Cayirci, Longo (Eds.). Rome, Italy (Sep).

Joschko, P.; C. Brandt; and B. Page. 2009. "Combining Logistic Container Terminal Simulation and Device Emulation using an Open-Source Java Framework". In *Proceedings of the International Conference on Harbor, Maritime & Multimodal Logistic Modelling and Simula-*

tion, Number c, A.G. Bruzzone, Cunha, Martínez; and Merkuryev (Eds.). La Laguna, Spain.

Joschko, P.; B. Page; and V. Wohlgemuth. 2009. “Combination of job oriented simulation with ecological material flow analysis as integrated analysis tool for business production processes”. In *Proceedings of the 2009 Winter Simulation Conference,* A.G. Bruzzone et al. (Ed.). Austin, Texas (May).

Kačer, J. 2006. Simulation software *J-Sim*. University of West Bohemia. Available at http://www.j-sim.zcu.cz/. Accessed 2013-02-13.

Klückmann, F. 2009. “Realzeitsynchrone Simulation – Begriffe, Anwendungen und exemplarische Umsetzung anhand des Simulationsframework DESMO-J”. Master’s thesis, University of Hamburg, Hamburg, Germany.

Knaak, N.; S. Kruse; and B. Page. 2006. “An agent-based simulation tool for modelling sustainable logistics systems”. In *Proceedings of the iEMSs Third Biennial Meeting: Summit on Environmental Modelling and Software. International Environmental Modelling and Software Society.* Burlington, Vermont.

Koors, A. and B. Page. 2012. “Transfer and Generalisation of Financial Risk Metrics to Discrete Event Simulation”. In *Proceedings of the International Workshop on Applied Modelling and Simulation,* Bruzzone, Buck, Cayirci, Longo (Eds.). Rome, Italy (Sep).

Kühnlenz, C.-M. 2011. “Interaktion von Simulationswerkzeugen mit ERP-Systemen – Konzeption und Realisierung von Interaktionsworkflows am Beispiel von DESMO-J und Infor ERP COM“. Master’s thesis, University of Hamburg, Hamburg, Germany.

L’Ecuyer, P. 2012. Simulation software *SSJ*. Université de Montréal. Available at http://www.iro.umontreal.ca/~simardr/ssj/indexe.html. Accessed 2013-02-13.

Lee, E.A. 2011. Simulation software *PtolemyII*. University of Berkeley. Available at http://ptolemy.berkeley.edu/. Accessed 2013-02-13.

Müller, C. 2011. Additional DESMO-J 2D Animation resources. Technical University of Applied Sciences, Wildau, Germany. Available at http://www.wi-bw.tfh-wildau.de/~cmueller/SimulationAnimation/. Accessed 2013-02-13.

Page, B. and E. Neufeld. 2003. “Extending an object-oriented discrete event simulation framework in Java for harbour logistics”. In *International Workshop on Harbour, Maritime and Multimodal Logistics Modelling and Simulation.* Riga, Latvia.

Page, B. and W. Kreutzer. 2005. *The Java Simulation Handbook – Simulating Discrete Event Systems with UML and Java.* Shaker, Aachen, Germany.

Page, B. 2013. Simulation software *DESMO-J*. University of Hamburg. Available at http://desmo-j.de. Accessed 2013-02-13.

Rossetti, M.D. 2013. Simulation software *JSL*. University of Arkansas. Available at http://www.uark.edu/~rossetti/research/research_interests/simulation/java_simulation_library_jsl/. Accessed 2013-02-13.

Schäfer, F. 2011. “Interaktion von Simulationswerkzeugen mit ERP-Systemen – Konzeption und Realisierung von Datenanalysen sowie technischen Schnittstellen am Beispiel von DESMO-J und Infor ERP COM“. Master’s thesis, University of Hamburg, Hamburg, Germany.

Sharpen. 2013. Eclipse plug-in for multi-platform development. Available at http://community.versant.com/Projects/html/projectspaces/db4o_product_design/sharpen.html. Accessed 2013-02-13.

Sun, F. 2010. “Raumkonzept und 3D-Visualisierung für die ereignis-diskrete Simulationsengine DESMO-J“. Master’s thesis, University of Hamburg, Hamburg, Germany.

Verbraeck, A. 2009. Simulation software *DSOL*. Delft University of Technology. Available at http://sk-3.tbm.tudelft.nl/simulation. Accessed 2013-02-13.

AUTHOR BIOGRAPHIES

JOHANNES GÖBEL holds a master in Information Systems from the University of Hamburg, at whose Modeling & Simulation research group he is scientific assistant and PhD candidate now; his research interests focus on discrete simulation and network optimization.

PHILIP JOSCHKO studied Computer Science at the University of Hamburg. He works as a scientific assistant and PhD candidate within the Modeling & Simulation workgroup of Prof. Dr. Page. Research interests are business process simulation, simulation software development and offshore wind parks. Since 2005 he takes part in improving DESMO-J. He applied DESMO-J in several simulation projects.

ARNE KOORS obtained his master degree in Computer Science from University of Hamburg, Germany. Since then he has been working as a software developer and management consultant in the manufacturing industry, primarily in the field of demand forecasting and planning. Furthermore, he works as a research associate and on his PhD thesis on financial strategy simulation in the Modeling & Simulation research group led by Prof. Dr. Page.

BERND PAGE holds degrees in Applied Computer Science from the Technical University of Berlin, Germany and from Stanford University, USA. As professor for Modeling & Simulation at the University of Hamburg he researches and teaches in Computer Simulation and Environmental Informatics. He is the head of the workgroup which developed DESMO-J and the author of several simulation books.