DISCRETE EVENT SIMULATION USING THE JAVABEANS COMPONENT MODEL

Herbert Praehofer, Johannes Sametinger, Alois Stritzinger
Department of Systems Theory and Information Engineering
C. Doppler Laboratory for Software Engineering
Johannes Kepler University, A-4040 Linz / Austria

e-mail: hp@cast.uni-linz.ac.at, [sametinger | stritzinger]@swe.uni-linz.ac.at

KEYWORDS

JavaBeans, discrete event simulation, simulation components, simulation beans, DEVS formalism

ABSTRACT

This paper reports on an effort to use both the system theoretic DEVS formalism and the JavaBeans component model as a basis for a component-based discrete event simulation framework. While the DEVS formalism can serve as a formal, mathematical base for modular, hierarchical discrete event modeling and simulation, the JavaBeans component model provides the appropriate implementation base. The result of the synergism of DEVS and JavaBeans is a powerful component-based simulation framework together with a set of flexible bean components for building simulation systems.

In this paper we try to give answers to several questions like whether the JavaBeans component model is suitable for the creation of such a component set and whether the available bean environments are powerful and flexible enough to serve as simulation tools. We will describe the DEVS formalism, the JavaBeans component model, the architecture of the SimBeans framework and component set, the problems we have encountered in developing SimBeans, problems we had with various builder tools, and experiences with the JavaBeans component model.

1. INTRODUCTION

In the project *SimBeans* we have developed a set of JavaBeans components for the creation of discrete event simulations. The goal of the project was twofold. First, component models, in general, and the JavaBeans component model, in particular, should be evaluated in a specific application. Second, it should be investigated, whether discrete event simulation applications can profit from an up-to-date component technology.

The idea was to create a set of basic simulation components together with visualization and animation components that can be arranged and connected on a worksheet. Comfortable visual composition of simulation scenarios together with visualizations and animations should be possible with these components.

In the spirit of component technology, simulation components should be designed to be reusable in different contexts, customizable for a wide range of different applications, and extensible for particular unforeseen requirements. We have tried to accomplish these objectives by pursuing the following goals in component design:

- Defining abstract component functions in the form of Java interface specifications.
- Providing modular interfaces for components to make them independent of the environment in which they should operate.
- Making a strong separation between components and coupling of components.
- Composing complex components from primitive ones and coupling them together hierarchically.

Our modeling approach is based on the DEVS formalism (Zeigler 84, 90). The DEVS formalism is a formal, system theoretic formalism for discrete event modeling and provides a theoretic framework for modular, hierarchical modeling (Zeigler et al 98).

In this paper we try to give answers to several questions in this context. For example:

- Is the JavaBeans component model suitable for the creation of a component set for simulation?
- Are the available bean environments powerful and flexible enough to serve as simulation tools, or is it necessary to create a dedicated tool?
- Is the resulting simulation tool more flexible and more general than comparable tools that have been implemented with different technology?

We describe

- the enabling concepts in particular, the DEVS formalism and the JavaBeans component model
- the architecture of the framework and component set we have developed,
- the problems we have encountered in developing it,
- problems we had with various commercially available tools, and
- our experiences with the JavaBeans component model.

2. DISCRETE EVENT SIMULATION WITH JAVABEANS

2.1 Vision of a Component-Based Simulation Methodology

A component-based modeling and programming framework for simulation applications should enable developers to interactively pick components from libraries and place them onto a worksheet. Such components comprise simulation units (model components) as well as components for output, visualization, animation and statistics calculations. As a next step, wiring among these components has to be accomplished somehow, such that signals and data can be exchanged among the components. Additionally, convenient interactive customization of component parameters has to be supported by the framework. Such a component system should simply be executed and used as a simulation application or be usable as a more complex component in other simulation applications. In case complex components are created by assembling, it is necessary to precisely define the new component's interface.

We envision a component-based simulation methodology which provides component libraries for different purposes, different users, and different applications. However, the main objective is *reusability*, i.e., simulation systems can be built with less effort by mainly selecting, extending, customizing, and coupling together components from libraries. Thereby, we see different type of users which use different libraries in different ways:

- The *application engineer* uses a set of predefined, ready-to-use components modeling his application entities, customizing the components by setting various parameters in convenient user interface dialogs, interactively connecting the components together, and running experiences with them. He does not want to do any programming in Java. He must be supported in quickly setting up different system configurations and experiments. This type of user needs a predefined set of components for modeling his application-specific entities, couplings, and control mechanisms. Besides the components, the application engineer needs an easy-to-use tool for setting up his system configurations and for doing his experiments.

- The *simulation programmer* has to realize the components which are needed in the application context by the application engineer. He will mainly rely on a library of elementary components for simulation modeling as well as for output and visualization/animation. But he will also need to program in Java in order to implement parts for which no components can be found. He should heavily use program development tools and interactive builder tools in order to set up and test single simulation components and entire simulation systems.
- The *simulation expert* has to realize elementary simulation components for different application domains. He needs a good understanding of the underlying modeling and simulation concepts, the JavaBeans technology, and the application domain. He will mainly do programming in Java using the underlying simulation concepts and, eventually, components from other application domains.

We see the *simulation programmer* as the main client for a component-based simulation methodology. He will profit most from the component technology. It should be less effort for him to realize application-specific components or application-specific simulation systems and environments, which can then be used by application engineers. The simulation expert is responsible for providing the elementary components for the simulation programmer. He is faced with the challenge of designing components in a way so that they can be reused by the simulation programmer in a wide range of applications. The application engineer, however, will profit from a component-based simulation technology that facilitates the realization of customized simulation systems and tools with application-specific component libraries and user interfaces.

We argue that such a vision of a component-based simulation methodology is feasible by relying on the DEVS formalism, which supports modular, hierarchical modeling, and the JavaBeans component technology. In the following we shortly review the DEVS formalism and JavaBeans in this sense.

2.2 DEVS Formalism

The base of the component library for discrete event simulation is the DEVS formalism as a system theoretic formalism for modular hierarchical simulation modeling. Modular, hierarchical system modeling (Zeigler 84, Zeigler 90, Pichler, Schwärtzel 92) is an approach to complex dynamic system modeling where modular building blocks, i.e., system components with a well defined interface in the form of input and output ports, are coupled in a hierarchical manner to form complex systems. In system modeling, we distinguish between atomic and coupled models. While an atomic model specifies its internal structure in terms of its set of states and state transition functions, a coupled model's internal structure is specified by its components and its coupling scheme, i.e., how ports are connected. System modeling means building complex systems by interface based object composition. Modularity allows for setting up bases of reusable building blocks which can be plugged into a system through their well defined input and output interface.

Modular hierarchical system modeling concepts have been applied in several domains, most notably in hardware design and communication engineering. Zeigler (Zeigler 76, Zeigler 90) introduced modular hierarchical modeling for discrete event simulation. The DEVS-formalism is an application and computer implementation independent formalism for discrete event modeling and parallels the differential equation specified and finite state automaton formalism. Zeigler (Zeigler 90) compares modular hierarchical system modeling and the object oriented paradigm. While both share various concepts they also show major differences. System models as well as objects use the concept of internal states. However, objects are not dynamic systems in the sense of system theory which show dynamic, concurrent behavior specified over a time base. Nor are objects modular units in the above sense. In particular, objects lack an explicit output interface. The output interface is only given implicitly by the method calls to other objects. Therefore, an object has to have knowledge of objects that it communicates with. In contrast to that, a system model only defines an output port, the communication with other components is defined independently on the higher coupled system level. Additionally, object oriented systems are not hierarchical in the above sense.

Component based software engineering represents a step towards a modular system modeling approach. Components define an explicit input and output interface in the form of method calls they accept and events they generate. Component based programming should primarily mean interface based component composition. Components can be constructed hierarchically using smaller components.

2.3 JavaBeans Component Model

JavaBeans is the component model of Java and has been introduced in 1997 (Sridharan 97, Sun 97). A JavaBean is a reusable software component that can interactively be modified and composed with other components. Tools that support component assembling range from simple layouting tools to complex component-based, visual programming environments. JavaBeans components support the following concepts:

- *properties*
  Properties are named attributes that can be read and written by calling appropriate methods. Naming conventions have to be obeyed for the definition of methods. Two methods, get<*Property*> and set<*Property*> have to be provided for reading and writing property values. So-called *bound properties* allow *PropertyChangeListeners* to watch properties for value changes. PropertyChangeListeners are added to properties by the method *addPropertyChangeListener*. PropertyChangeListeners become informed of changes in the property by a call of their *propertyChange* method.
- *events*
  Events are used for event communication among bean components. With events, a bean can notify other beans that something interesting has happened. For that purpose event listeners are registered to event sources. An event source has to provide two methods that allow the registration of listeners, i.e., add<*Event*>Listener and remove<*Event*>Listener to register components that are interested in receiving <*Event*> events, where <*Event*> is the name of the event.
- *introspection*
  Information about properties, events, and operations of JavaBeans components can be provided by means of naming conventions in classes, see examples above. Tools for JavaBeans components have access to variables and methods of Java classes and, thus, recognize such conventions. Instead of adhering to naming conventions a programmer can also choose to develop a BeanInfo class, where information about properties, events, etc. is provided explicitly.
- *persistence*
  In order to keep modified properties, a component has to be able to serialize its state onto an external storage medium. A built-in serialization mechanism simplifies the support of persistence.
- *customization*
  Modifying simple properties is not always sufficient for convenient customization of complex components. Additional support may be necessary to allow the modification of complex properties. Specific property editors and customizers may be provided for ease of customization.

JavaBean components can be arbitrary Java classes conforming to a few (non-critical) requirements. The component interface is being defined by events, properties, and methods. The interface can be defined by adherence to naming conventions or by the definition of a separate BeanInfo class which provides all the information about a component. A JavaBean can be a simple Java class, but it can also consist of many classes and persistent objects, which are typically stored in a Java Archive File (JAR).

3. CONCEPTS AND ARCHITECTURE OF SIMBEANS

3.1 Main Ideas

In the *SimBeans* simulation framework, the realization of simulation systems is based on the following ideas and concepts:

- A set of elementary, yet powerful building blocks is provided for simulation modeling and simulation output, for statistical evaluation, and for visualization.
- A library of utility and support objects are provided. By them, model components can be customized in their functionality in order to meet particular requirements.

- Model interfaces are defined which specify where and how model components can be used. The interface specifications are put into a classification hierarchy to define compatibility between model components (Thomas 94).
- Interfaces and interface-based classification of model components are used to define generic templates for coupled models which define the components' interfaces and coupling structure but not the components itself. At design time, these generic components can be configured by instantiating model components which obey to the interfaces. For example, a *ParallelProcessor* coupled model will specify that it has several parallel components and that all have to obey to the *Processor* interface. At design time we can choose from different processor realizations.
- Simulation systems are primarily built bottom-up by hierarchical composition and coupling of model components.
- The component library can easily be extended to meet special needs.
- Employing the elementary component library, special purpose simulation environments for particular application domains can be realized.

Figure 1 illustrates how simulation systems are put together from components. According to these ideas, we distinguish the following ways for component composition:

- *selection*
  Select concrete model components in a coupled model for which only the interfaces are specified in a top-down manner.
- *customization*
  Customize simulation components to meet different requirements by using utility components, e.g., random distribution functions, control strategies.
- *coupling*
  Couple models in a hierarchical bottom-up way.
- *attachment*
  Attach components for simulation output, statistical computation, visualization and animation to state variables (properties) of models.

3.2 Architecture

SimBeans is a set of model components together with components for visualizations and animations, that can be used to build discrete event simulation systems. All these components are realized as JavaBeans. In correspondence with the different types of users envisioned (see above) we identify the following layers in the *SimBeans* framework:

JavaBeans Extensions for Simulation

The lowest layer is the Java programming language and the JavaBeans component model. The Beans specification does not define the semantics of invocation order and synchronization of multiple event handlers listening to the same event. A default implementation exists for synchronous event handling (class *PropertyChangeSupport*). The coupling of beans is usually done by means of adapter objects that have to be defined by an individual adapter class leading to an opulent proliferation of classes. With the help of Java's reflection features, we implemented a class *Delegate*, that can deliver events to any target object.

Besides minor additions to the basic infrastructure we introduced

- synchronous and asynchronous event delivery mechanisms and a
- *delegate* mechanism for flexible event coupling

to meet the special needs of discrete event simulation.

Simulation Kernel

The simulation kernel layer provides the simulation infrastructure and implementation concepts for the simulation components. This layer is specific for different types of simulations, e.g., there is a infrastructure for discrete event simulation, for continuous simulation and for combined simulation.

For discrete event simulation, this layer includes support for

- event scheduling, event sets and event handling,
- models, model containers and hierarchies of models,

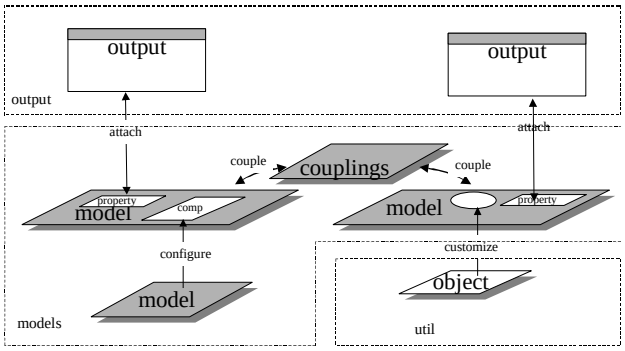

Figure 1: *Putting together simulation systems from components*

- state variables and property change mechanism ,
- utility services for simulation data collection and analysis, and for
- elementary output and visualization.

Elementary Simulation Components

This is the main layer containing the elementary simulation components from which simulation systems are built. It is specific for the application domain, for example, there is one library for classical discrete process simulation (discussed in more detail below). The layer contains components for

- elementary model units,
- classical control schemes,
- component coupling,
- utility services, and
- domain specific, elementary analysis, output, and visualization.

This layer is crucial for the success of the component-based simulation framework. It is the challenge for the simulation expert to foresee a wide range of applications in the domain and provide a set of easy-to-use and easy-to-extend components.

Application-Specific Simulation Components

Using the elementary simulation components, application specific components should be built which realize specific real world elements. They have a customized visualization and user interface which is meaningful for the application engineer.

Application-Specific Simulation Systems and Environments

At the top of the hierarchy of layers there are application-specific simulation systems or environments. They are built up from the lower layer components as stand-alone programs. They are specific for the application at hand and provide a user interface in the context of the application engineer. We distinguish between simulation systems, which have a single simulation model and are often part of a bigger application, and simulation environments, which allow to build and test different configurations by composing components from a library.

We observe that the design of the elementary simulation components is crucial. In the following we describe the design of elementary simulation components for classical discrete process simulation in more detail.

3.3 Component Library for Discrete Process Simulation

The library of elementary simulation components for discrete process simulation has been designed to facilitate utmost reusability and extensibility. Recall from above that we regard modular interfaces, separation of component and coupling, hierarchical component coupling as key design principles to achieve reusability.

For discrete process simulation we have identified the following principal types of elements:

- *resources*, which are active or passive and can be occupied by items,

- *items*, which flow through the system from resource to resource and occupy them,
- *couplings*, which realize the item flow, and
- *control* of the item flow.

A simulation system, therefore, is viewed as consisting of several resources, where items are placed and processed, and a coupling structure which realizes the flow of the items from one resource component to the next. The control part then decides which items can flow from which resources to the next based on requirements of items and availability of resources. This is a general, abstract view which fits to all types of discrete event simulation. The systems then differ in what type of resources are used, the types of items used, the structure of coupling, and in particular, who is in control and how is the control of the item flow.

The components were designed according to those principal types. Components are not of a particular type but rather they *play the role* of a particular type. A component also can serve different roles and, therefore, belong to different types. For example a transportation component may on the one hand serve as a container, that is a resource for some items, but on the other hand it may flow as item through the transportation system.

The Java interface concept corresponds to the roles components can play. A component which wants to play a particular role has to implement the corresponding interface. The first step towards a reusable component library, therefore, was the design of the interfaces for the different element types. In the following we show the interface design for the resources and the coupling schemes.

Resource Interfaces

Recall that a resource can be *active*, i.e., it can do some processing on an item, or *passive*, i.e., it can only passively store items. In any case, their elementary functions are to receive items, hold them, and provide them to other components. While a passive storage component will only store received items and provide them for access, an active server component will process received items, which will take some time, and afterwards want to get rid of them. Also resource components may have space available, i.e., they may need or be able to take a further item.

Two general interface definitions are crucial to the realization of the resource components. These are the *Receiver* for any component which may receive items and the *Provider* for a component which may provide items. Simulation then works by distributing items between providers and receivers. Receivers may be occupied and not be able to receive further items. Providers signal the availability of items and provide access to them. The interfaces for providers and receivers have the following features:

Receiver

- `receiveItem(Item)`
  a method for handing over an item to the component
- `boolean needsItem`
  a bound property to signal that the component can receive an item (the component may also *need* an item)
- `testItem(Item)`
  a predicate used to see if an item is appropriate to be received by the component
- `itemReceived`
  an event which is triggered at the time when an item is actually received

Provider

- `Item provideItem()`
  a method allowing other components to access available items
- `boolean hasItem`
  a bound property to signal that an item is available in the component
- `Item inspectItem()`
  an access method to get the next item for preliminary inspection only
- `itemProvided`
  an event which is triggered at the time when an item is actually provided

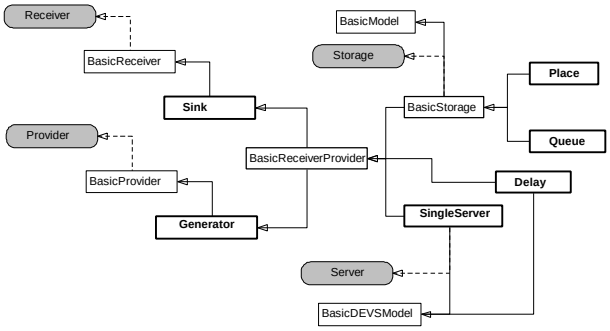

Figure 2: *Class hierarchy for elementary model bean (interfaces are gray, model beans are bold)*

The interfaces *Provider* and *Receiver* are not coupled directly, but rather additional components are used. We discuss how coupling should be done after presenting some elementary resource components.

Resource Components

A set of elementary building blocks are realized on base of the receiver and provider interfaces and implement them in different ways. Building blocks include *Generator*, *Sink*, *Processor*, *Queue*, *Place*, and *Delay*. Figure 2 shows the class hierarchy of the resources implementation.

The components implement the *Provider/Receiver* interfaces in different ways. All of them also implement the Item interface to be able to be used as flowing items. The *Generator* only implements the *Provider* interface and makes a new item available after some interarrival time. The *SingleServer* can process one single item at a time. It signals that it needs a new item when the last one has been moved on. Upon receipt of an item it gets occupied and immediatly starts processing. After some processing time, it signals that an item is available and waits that it is accessed from another component. A *Delay* component is never occupied and can always receive items. It delays them for some time and then makes them available for access. A *Place* is a passive component which can take one item. It signals that is needs an item and has no item when there is no item on the place. When it receives an item it signals that it is occupied and does not need any further item and that it has an item available for access. The other components are implemented in analogous way.

Other more complex components should be built by either coupling together components in a hierarchical way (see below) or by implementing them in Java using elementary simulation functions. By implementing the *Provider/Receiver* interface, they can be used in bigger coupled models according the the same coupling concepts.

Coupling and Control: Realization of Item Flow

More complex components and simulation (sub-)systems are built up from the basic components. The general idea of coupling components is that the containing model listens to property changes (*hasItem* and *needsItem*) of its subordinates and reacts to those by distributing items between its subordinate providers and receivers based on its individual control scheme. Figure 3 shows a hierarchical coupling of provider and receiver components. The coupled model listens to changes in the property values of its subordinate and reacts according to its control scheme by distributing items from providers to receivers.

We use the event and bound property change concepts to realize event coupling and communication in discrete event models. Models which rely on states of other models are registered as listeners of the other model's state property and are, thus, informed whenever a change in state happens. For example, a processor needing a particular tool registers itself as a listener of the *needsItem* property of the tool pool. As soon as a tool gets available, it is informed and can access it. In a similar way, events can be used to distribute objects between model components in an eventistic way.

Control and coupling can be arbitrarily complex, ranging from simplest linear forwarder to a transportation system built up by a complex coupled

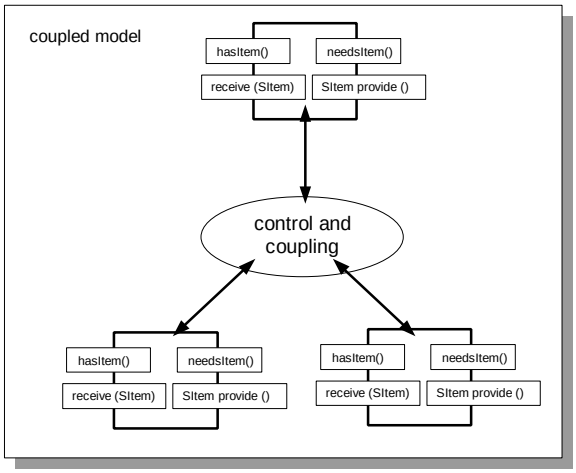

Figure 3: *Hierarchical coupling of provider and receiver components*

model by itself. Let us depict the range of couplings by considering different examples.

A *Forwarder* realizes a direct flow of items from a *Provider* to the next *Receiver*. It listens to the *hasItem* property of the *Provider* and the *needsItem* property of the *Receiver* and, when both are true, takes the item from the *Provider* and hands it over to the *Receiver*. No control decision is needed here.

An extension of the *Forwarder* is the *ReceiverDecisionPoint*. It is used to coupled a single provider with a set of receivers. The selection of the receiver of the next item available is based on a control strategy, a *ReceiverSelector*, which is a component selecting from a set of receivers. Components realizing different control strategies can be envisioned, for example, selecting at random, based on given percentages, the receiver waiting longest, etc. In the same way a *ProviderDecisionPoint* couples a set of providers with a receiver. With *Forwarder*, *ProviderDecisionPoint* and *ReceiverDecisionPoint*, coupled systems can be built which are typical for *Flow Shop* models.

A different coupling scheme should be used when modeling a robotized manufacturing cell. Here the item flow and control scheme is much more complex. A roboter has direct access to the places in the cell and the control has to take the whole cell state into account. Nevertheless, we use the same components and coupling principals. The cell controller listens to the *hasItem* and *needsItem* properties of the cell components and generates transport commands to the robot. The robot then realizes the item flow by accessing the item from the given provider and placing it on the given receiver.

Modeling a manufacturing system with a complex transportation system again needs a different item flow mechanism. In such a system, different system and control layers are identified. At the upper layer, the workpiece flow between the machines is controlled by a disposition control system which decides which workpieces are assigned to which machines. On the lower layer the transportation system realizes the workpiece flow. This is again a system consisting of resources and items. Typically, the transportation system can be built as a *Flow Shop* model, where the vehicles are the flowing items and the paths, intersections, parking lots, etc. are the resources. Coupling is linear with decision points at intersections. The machining and transportation system are coupled at *loading/unloading* stations. It should be emphasized that the same type of components can be used in the different layers of the system.

3.4 Visualization and Animation

Model components are used for modeling a simulation scenario. They can be made visible during a simulation run to the user in order to demonstrate the model scheme of the simulation. However, a model component's properties and property changes cannot be made visible to the user. This has to be accomplished by dedicated components for visualization and animation. For example, the state of a container component may be visualized by a simple number, a progress bar, or by a graphical representation of the items in the container. In addition, statistics components may receive events of model components or items, calculate various useful numbers, store the results for later evaluation, and display graphical representations of the results.

It is important for real simulation applications to have powerful visualization, animation, and statistics components available. We have put our primary effort into developing model components in order to test the usefulness of our approach. Therefore, we have currently only basic components available in this category.

4. EXPERIENCES

4.1 Experiences with JavaBeans

JavaBeans provides an attractive platform-independent component model. Our experiences with the model have been quite positive. A few weaknesses have been encountered, however.

- The JavaBeans specification does not contain any information about hierarchically structured beans. Therefore, there is no defined access of super- or subordinate beans. In our application we regard hierarchical composition of component essential and we missed an explicit representation of the component hierarchy. In JDK version 1.2 there will be appropriate interfaces (*BeanContext* and *BeanContextChild*).
- The property change mechanism propagated by the JavaBeans standard only knowing synchronous event handling fails when feedback occurs. Coupling mechanism defined by JavaBeans with event coupling seem to be too weak for complex applications. We foster coupling schemes at a higher level where components are coupled based on their interface specifications (which then are realized by primitive event communication mechanisms).
- There is no restriction for tool developers on how code should be generated and where it should be inserted. This leads to different approaches, which prohibits using several builder tools together or switching from one tool to another.

4.2 Experiences with Bean Tools

The purpose of the SimBeans project has not been to gather experiences with various tools and programming environments. However, during our development process we have encountered significant differences in the support of JavaBeans which we believe is interesting to the reader and potential Bean developer and user.

VisualAge

VisualAge V1.0 by IBM had promised sophisticated support for JavaBeans due to their support of visual composition. But even the installation of our SimBeans components had been a painful task. Every bean needed a BeanInfo class, which we had not yet available at that time. In particular, modifying beans and reinstalling them had been time consuming. The visual composition tool generated extensive code for each component instantiation and for each link between components. In contrast to other tools the high number of generated methods was amazing. Even though it was possible to include one's own source code lines in most of the generated methods by means of special comments, it was quite difficult to combine visual composition with manual coding. Believing that visual composition is not sufficient in all circumstances, VisualAge did not sufficiently satisfy our needs in the development and use of SimBeans components. Another drawback of the tool was that having more than 100 classes lead to unacceptable response times.

VisualCafe

VisualCafe V2.5 by Symantec offers sufficient support for the use of a component library. However, using the environment is sometimes cumbersome, and several times faulty beans had lead to crashes of VisualCafe. In contrast to VisualAge, VisualCafe generates only a single code fragment in the constructor of the application class where all components are being instantiated and their properties are being initialized. Including code of our own was straightforward and without any further problems. One major drawback of VisualCafe is that serialization of JavaBeans is not being supported. Even though there is a Bean Wizard available, we found it often cumbersome to use by its long sequences of modal dialogs. Also, the wizard cannot be used to create additional information for existing components. The property sheet supports built-in types, a few

special object types, and allows the use of special property editors. But it lacks support for properties which are beans by themselves.

JBuilder

JBuilder V2.0 by Inprise (Borland) promises convenient support for development and use of JavaBeans. We do not have sufficient experiences yet with this tool, but currently it offers the best support for JavaBeans that we have encountered so far. A Bean Wizard is provided which allows the interactive specification of BeanInfo classes. The generated code is highly readable and written into one method. Manual modification and extension of the code are simple and straightforward. Assigning complex properties, i.e., properties which are beans by themselves, is supported elegantly.

We believe that currently the availability of powerful tools that support JavaBeans is satisfactory. However, existing programming environments may be too complex to be used as a simulation tool by application engineers. We are still missing simple JavaBeans builder tools which are flexible enough to configure them as simulation tools.

5. CONCLUSION

Initially, the objectives of the SimBeans project were to evaluate the JavaBeans technology and to investigate JavaBeans as an implementation technology for simulation. We also used a prototypical version of the SimBeans components as an educational vehicle for teaching modeling and simulation (Praehofer et al. 98). Due to the good experiences we made, we decided to continue the project and make the system more mature in order to additionally serve as a simulation framework for real world applications. At the current time we are making a major redesign of the system in order to broaden its applicability. We also plan to extend the framework to allow continuous and combined simulation. This part will be employed in realizing a training simulator for papermill operators.

In this paper we have investigated and assessed the Java programming language and the JavaBeans component model as an underlying technology for the realization of a component-based discrete event simulation methodology. The DEVS formalism provides the fundamental concepts for a modular hierarchical modeling methodology facilitating the design of reusable components. According to our experience JavaBeans together with the DEVS modeling methodology is a superiour combination to enable a simulation technology for developing advanced simulation systems.

6. REFERENCES

F. Pichler and H. Schwärtzel (eds.), *CAST Methods in Modeling*, Springer, 1992.

H. Praehofer , A. Stritzinger, and J. Sametinger, *Using JavaBeans to teach Simulation and using Simulation to teach JavaBeans*, ESM98, 12th European Simulation Multiconference, Manchester, UK, June 16-19, 1998.

C. Thomas, Interface-based classification of simulation models, *WSC 94*, Orlando, FL, 1994.

Sridharan Prashant, *JavaBeans-Developer`s Resource*, Prentice Hall, 1997

Sun Microsystems: JavaBeans 1.01 API Specification. 1997. see http://java.sun.com/Beans/spec.html

B.P. Zeigler, *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, 1984.

B.P. Zeigler, *Object Oriented Simulation with Modular, Hierarchical Models*. Academic Press, 1990.

B.P. Zeigler, H. Praehofer, T.G. Kim, *Theory of Modeling and Simulation, 2nd Edition*. Academic Press, 1998 (in preparation).