Concepts and Architecture of a Simulation Framework Based on 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

Abstract

We report on a combination of system theoretic simulation modeling methodology with the JavaBeans component model as a basis for a component-based simulation framework. While system theory formalisms can serve as formal, mathematical foundations for modular, hierarchical modeling and simulation, the JavaBeans component model provides the appropriate implementation base. The result of the synergism is a powerful component-based simulation framework. In this paper we present the basic concepts and overall architecture of our JavaBeans modeling and simulation framework. We review the underlying system modeling formalisms for simulation modeling, sketch the layered architecture of the framework, and show elementary simulation programming and interface-based, hierarchical coupling of simulation components in more detail. Finally, we show the current state of implementation and demonstrate how simulation model can be developed using standard bean builder tools.

Keywords: simulation, simulation components, state space models, Java, component programming

1 Introduction

Component-based software development [16], [18] is a new programming paradigm that emphasizes software construction from prefabricated building blocks that adhere to a standardized component model. JavaBeans [19] is the component model for Java [5].

Component frameworks are dedicated and focused architectures with a set of policies for mechanisms at the component level [20]. Component frameworks share similarities with application frameworks. They provide a framework for components rather than objects but are not necessarily used for the creation of entire applications. Component frameworks typically implement generalized common behavior and define (abstract) interfaces that have to be implemented (or extended) to utilize the common behavior.

This paper poses two theses:

1. Modeling and simulation technology can profit from a component-based software methodology

2. Modular, hierarchical modeling fostered by the systems method provides a methodology for building reusable component architectures.

Our modeling approach is based on system theoretic formalisms [27], most notably the DEVS formalism [24], [25]. The DEVS formalism is a formal, system theoretic formalism for discrete event modeling and provides a methodology for modular, hierarchical modeling. It has been extended to support multiformalism modeling [15].

In the course of the *SimBeans* project we have developed a set of JavaBeans components for the creation of simulation systems. The goal of the project was twofold. First, component models in general and the JavaBeans component model in particular were to be evaluated in a specific application domain. Second, we wanted to investigate whether simulation applications can profit from recent developments in software 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.

1.1 The Vision of a Component-Based Simulation Methodology

We envision a component-based simulation methodology that provides component libraries for different purposes, at different levels, for different users, and for different applications. The main objective is *reusability*; i.e., simulation systems can be built with less effort mainly by selecting, extending, customizing, and assembling components from libraries. Such components comprise simulation units (model components) as well as components for output, visualization, animation and statistical analysis. 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. Convenient interactive configuration, coupling, and customization of component parameters must be supported.

Hence different types of users use different libraries in different ways:

- The *application engineer* uses a set of predefined, ready-to-use components to model the application entities, customize the components by setting various parameters in convenient user interface dialogs, interactively connect the components, and run experiments with them. The application engineer does not want to do any programming in Java and must be supported in quickly setting up different system configurations and experiments. This type of user needs a predefined set of components for modeling the application-specific entities, couplings and control mechanisms. Besides the components, the application engineer needs an easy-to-use tool for setting up system configurations and for running experiments.
- The *simulation programmer* has to realize the components that are needed in the application context by the application engineer. This user will rely mainly on a library of elementary components for simulation modeling as well as for output, visualization/animation and output analysis. However, the simulation programmer will also need to program in Java in order to implement parts for which no components can be found. This engineer should heavily use program development tools and interactive builder tools in order to set up and test individual simulation components and entire simulation systems.
- The *simulation expert* has to realize elementary simulation components for different application domains. This requires a good understanding of the underlying modeling and simulation concepts, the JavaBeans component technology, and the application domain. The simulation expert 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. This user will profit most from the component technology. It should be less effort 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. This expert is faced with the challenge of designing components 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.

1.2 Main Ideas

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:

- 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 is provided from which 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 [21].
- Interfaces and interface-based classification of model components are used to define generic templates for coupled models that define the components' interfaces and coupling structure but not the components themselves. At design time, these generic components can be configured by instantiating model components that adhere to the interface requirements.
- Simulation systems are primarily built in a bottom-up way by hierarchical composition and coupling of model components.
- The component library can easily be extended to meet special needs.
- By means of elementary simulation component libraries and a framework program for simulation and experimentation environments, special-purpose simulation environments for particular application domains can be realized.

Figure 1 illustrates how simulation systems are assembled from components. According to these ideas, we distinguish the following ways of assembling components:

- *Configuration*
  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 model components in a hierarchical bottom-up way.
- *Attachment*
  Attach components for simulation output, statistical computation, visualization and animation to state variables (properties) of models.

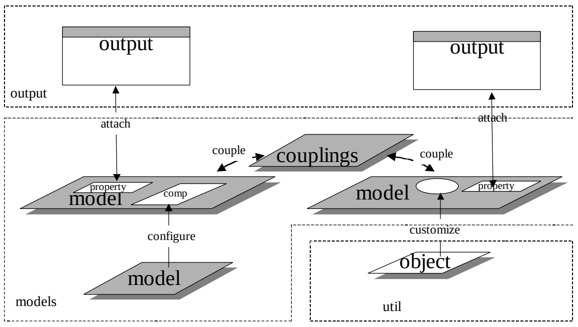


Fig. 1 Assembling simulation systems from components

We argue that such a vision of a component-based simulation methodology is made feasible by reliance on system modeling concepts that support modular, hierarchical modeling and on the JavaBeans component technology. In the following we briefly review system modeling and the JavaBeans component technology in this sense.

2 Background

2.1 Modular Hierarchical System Modeling Reviewed

Modular, hierarchical system modeling [24], [25], [11] is an approach to complex dynamic system modeling where modular building blocks, i.e., system components with a well defined interface, 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. Modularity allows for setting up bases of reusable building blocks that can be plugged into a system through their well defined input and output interfaces. System modeling means interface-based object composition.

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

State Space Representation of Simulation Models

In modular hierarchical system modeling, the elementary building blocks are modeled using system formalism which allow a “state space representation” of simulation models [3], [27]. Different types of formalism are available for different forms of model specification: classical *differential equation specified systems* (DESS) serve for continuous model specification, the DEVS formalism represents the system theoretical basis for discrete event modeling and the DEV&DESS formalism [27], [15] as a combination of both serves as a basis for combined discrete/continuous multiformalism modeling.

A state space model is a modular unit. It comprises input and output interfaces through which all the interactions with the environment occur. The interior of the model is represented by state variables. Dynamic behavior definition occurs by state transition and output functions which adopt different forms in the different types of formalisms. In DESS modeling state behavior is specified using a *derivative function* to specify continuous state behavior while in DEVS state behavior specification is event-like.

In DEVS modeling and its extensions [12], we distinguish the following types of events:

- *External events* are caused by external event inputs at the input interface.
- *Conditional events* depend on event conditions and are caused by value changes of discrete variables.
- *State events* depend on event conditions and are caused by continuous changes of continuous states and inputs.
- *Time events* are scheduled and occur when the simulation time is forwarded to the time of the event.

Visual Representation of State Space Models

Visual representations are a preferable form for model specification and documentation. While block diagrams [3], as employed in SIMULINK [9] and others, are advantageous visual representations of continuous models, we use state transition diagrams in the state chart form [6], [14], [2] for representing event behavior. Our model implementations as discussed in Section 4.1 are direct translations of these visual representations.

In the DEVS formalism and its multiformalism extensions, atomic model specification is organized around various *phases* that denote *abstract* system states. Actually the different phases of a model define a partitioning of the state space into a set of mutually exclusive blocks, where the different blocks identify qualitatively different system behaviors [14]. In combined modeling, the phases can be used to associate different continuous behaviors. The state space phase partitioning and the dynamic behavior specification organized around phases can serve as a basis for a graphical state diagram model representation. The phases and phase transitions are naturally represented by a state transition diagram.

In the state transition graph, the nodes depict the phases and the edges the event transitions. According to the different types of events in event models as discussed above, we distinguish different types of transition edges that are labeled in different ways:

- *External event edges* are labeled by the triggering external event input.
- *Conditional event edges* and *state event edges* are labeled by the triggering event condition, usually in squared brackets.
- *Time events edges* are labeled by the scheduled time advance of the transition.

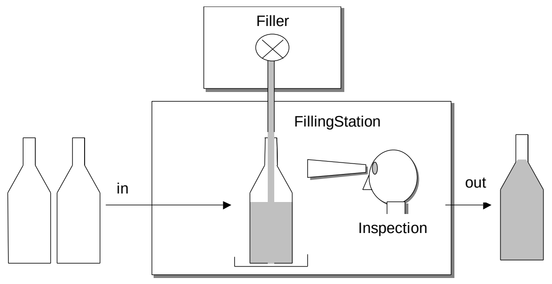


Fig. 2 Bottle filling station

As an illustrative example we adopt a model of a bottle filling station as shown in Fig. 2 (the model has been selected since it is appropriate for showing the different modeling features). Bottles enter a filling station and are filled with liquid. The station uses a filler which is also used by others and therefore represents a scarce resource. After filling up, the bottles are inspected and then released/ejected. The bottles' content is modeled by a continuous state variable whose derivative is given by the outflow of the filler. The operation of the filling station is discrete and modeled by various events.

Figure 3 shows a state chart representation of the bottle filling station behavior. The filling phase is expanded to show the underlying block diagram model of the continuous bottle process. In the initial phase idle, an input event *input(bottle)* will bring the model into phase *wait*, where it has to wait for the filler resource. The conditional event transition models the transition to the filling phase upon availability of the resource. The next transition is a state event transition that models the event when the bottle is full. The last event is time scheduled; it models the output of the bottle and the transition to phase *idle* after inspection, which lasts 1 time unit.

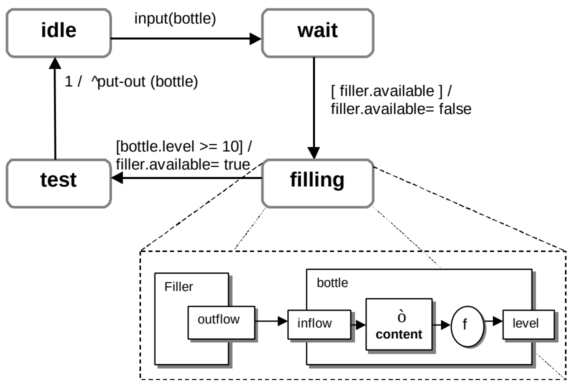


Fig. 3 State transition diagram of a bottle filling station

The block diagram models the bottle filling process. The derivative of *content* is defined by the current *inflow*, which in turn is directly connected to the *outflow* from the filler. The *level* output variable is computed by a function *f* from the content.

Coupled Models

Complex coupled models are specified by connecting the output and input interfaces of model components as shown in Figure 4. Components provide their interfaces, and coupled models rely on these interfaces to define

- how components are coupled and
- how their own input and output interfaces are realized based on the components' interfaces.

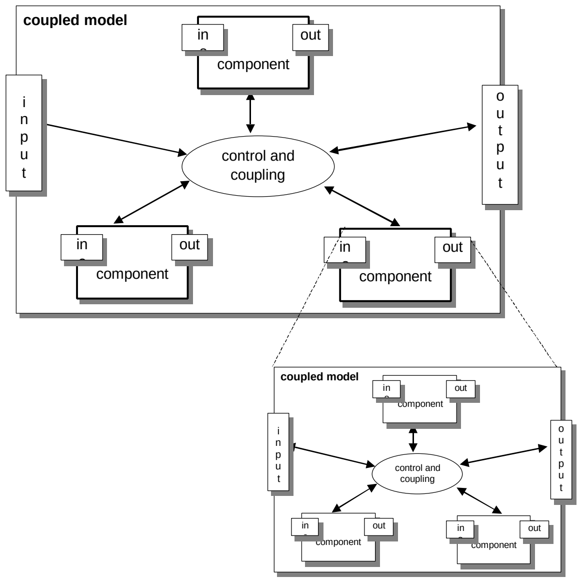


Fig. 4 Hierarchical coupling of components

In contrast to classical system modeling, our coupling schemes are not restricted to pure port and variable connections. For utmost flexibility we allow the following:

- Coupled components employ arbitrary coupling schemes, e.g., direct coupling, broadcast coupling, cellular coupling, etc.
- For variability of component structure and coupling [26], [22], [1], the coupled model changes its structure dynamically.
- The coupled model specifies its own dynamic behavior.

We regard the modular hierarchical composition of components as our approach to accomplishing model reusability. We strive to design the model interfaces carefully so that they are independent of the environment in which they are embedded. In Sections 4.2 and 4.3 we further elaborate on this idea by discussing the implementation of coupled models in Java and the design of interfaces for resources and discrete item flow.

2.2 JavaBeans Component Model

JavaBeans is the component model of Java [19]. A JavaBean is a reusable software component that can interactively be modified and combined with other components. Tools that support component assembling range from simple layouting tools to complex component-based, visual programming environments. To realize components based on JavaBeans has the advantage that it defines a common and generally accepted standard (*design patterns*) for component analysis and handling. Components based on JavaBeans can be used in any development tool supporting the JavaBeans standard.

JavaBeans components are defined and implemented by Java classes that have to adhere to certain conventions. For example, bean classes need to have a public null-constructor, so that they can be instatiated in any context. Bean instances have also to be serializable so that they can be made persistent. A bean's features can be accessed via an introspection mechanism that provides (meta-) information about method, property and event sets for the environment. The two most important concepts for our simulation model implementation are *events* and *bound properties.* We review these in more detail below.

Events

JavaBeans provides a standard event model to allow event firing and event communication between components. A particular *event type* is realized by providing an event object (class `<`*`EventType`*`>-Event`), which must be a specialization of the standard class `EventObject`, and an interface for event listeners (interface `<`*`EventType`*`>EventListener)` with specifications of event methods. Any component firing the event must implement methods to add and remove event listeners (methods `add<`*`Event-Type`*`>EventListener` and `remove<`*`EventType`*`> EventListener`). Upon firing an event, the component calls the listeners registered for the event with the respective event method.

Let us illustrate events by an event type implementation, `bottleEvent`, which will be used in Section 4.1 to realize distribution of bottles (objects of type `Bottle`) between stations in our implementation of the bottle filling station. REFFORMATVERBINDENFigure 5 shows the `bottleEvent` realization. `BottleEvent` is the event object as a specialization of `EventObject`. `BottleEventListener` is an interface with event method `bottleInput`. Any component signaling `bottleEvents` has to realize methods `addBottle-EventListener` and `removeBottleEventList-ener` to register event listeners. Firing a bottle output event will distribute a `Bottle` object in a `Bottle-Event` object to the registered `BottleEventListen-ers`.

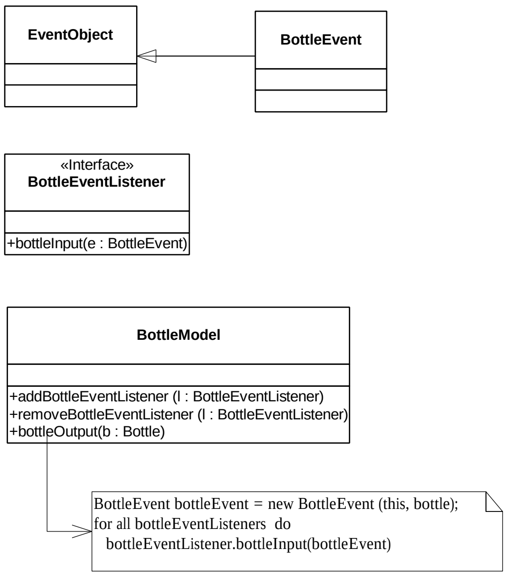

EventObject
BottleEvent
«Interface»
BottleEventListener
+bottleInput(e : BottleEvent)
BottleModel
+addBottleEventListener (l : BottleEventListener)
+removeBottleEventListener (l : BottleEventListener)
+bottleOutput(b : Bottle)
BottleEvent bottleEvent = new BottleEvent (this, bottle);
for all bottleEventListeners do
bottleEventListener.bottleInput(bottleEvent)

Fig. SEQARABISCH5 Bottle event

Properties

Properties are named attributes of a bean component that allow controlling the appearance and behavior of the component. Properties provide state information of components. Properties can be read and set at building (design) time and of course at run time via *getter* and *setter* methods. These methods have to obey the naming convention `get<`*`Property`*`>` and `set<`*`Property`*`>`. A builder tool typically provides a property sheet to visualize the current property states and a set of property editors to change property values.

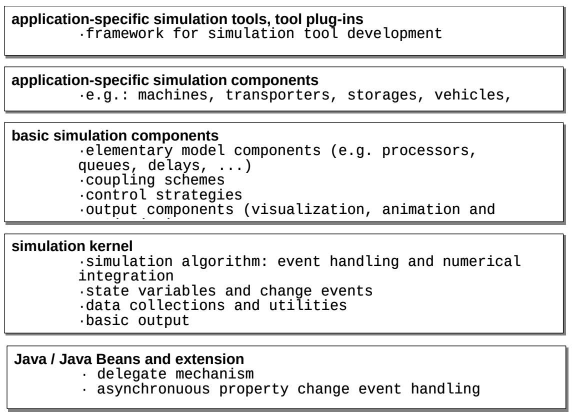
Bound properties are properties that use the event concept to signal state changes to its environment. For this purpose, class `PropertyChangeEvent` and interface `PropertyChangeListener` are defined. A bean implementing a bound property has to define add and remove methods to allow registration of `PropertyChangeListeners` (the naming convention is `add<`*`PropertyName`*`>ChangeListener` and `remove<`*`PropertyName`*`>ChangeListener`). Any other component interested in changes of the property can be added and will be informed whenever the value changes.

The bound property change mechanism is import in our implementation concepts of the simulation model. As we show in Section 4.1, an extension thereof (`VariableChangeListener`) is used to realize state dependencies between model components.

3 Architecture of SimBeans

3.1 Layered Architecture

In accordance with the different types of users envisioned in Section 1.1, we identify various layers in the *SimBeans* framework; see Figure 6.


application-specific simulation tools, tool plug-ins
·framework for simulation tool development
application-specific simulation components
·e.g.: machines, transporters, storages, vehicles,
basic simulation components
·elementary model components (e.g. processors, queues, delays, ...)
·coupling schemes
·control strategies
·output components (visualization, animation and
simulation kernel
·simulation algorithm: event handling and numerical integration
·state variables and change events
·data collections and utilities
·basic output
Java / Java Beans and extension
· delegate mechanism
· asynchronous property change event handling

SEQARABISCHFig. 6 Layers of SimBeans framework

Layer 1: Java/JavaBeans Extensions for Simulation

The lowest layer is the Java programming language and the JavaBeans component model. Besides minor additions to the basic infrastructure, we introduced the following to meet the special needs of discrete event simulation:

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

Asynchronous Event Delivery Mechanism:

The JavaBeans 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*) which calls listeners' event methods in the sequence the listeners have registered. However, this simple approach becomes problematic when feedback occurs in event handling, i.e., when the property triggering an event is changed in a listener method. The effect is that the next event listener may receive a property change with the old value but the value has been changed in the meantime. We have experienced that this mechanism results in obscure behavior in discrete event simulations; such behavior is impossible to track and handle.

For this reason we implemented an asynchronous property change mechanism (called `Variable-ChangeEvent`; see below). A change event is not executed directly but registered at the simulator. The simulator processes events in FIFO order and thus guarantees that an event is fully executed before the next change event is launched.

Delegate Mechanism:

Beans are usually coupled by means of adapter objects that listen to one or more event sources and provide the code for handling the events. The event handling code is adapter-specific and easily leads to an opulent proliferation of adapter classes. The situation becomes even worse when many specific event/listener types are used. In order to reduce the number of adapter classes and avoid trivial adapters, we provide a generic adapter class *Delegate,* which simply delegates the concrete event handling to any target object. *Delegates* hold a target object and a target method (object) and implement an *EventListener* interface. The behavior of delegates is simple: "on event handling, invoke target method on target object and pass the event source as argument." The constructor of the *Delegate* expects two arguments: the target (*Object*) and the target method name (*String*). During construction of a delegate object, the specified method is searched for in the target class; next the constructor checks whether this method expects an event as a single argument (this can be done using Java's reflective features) and assigns the method object to the *targetMethod* variable. Our delegate class resembles Microsoft's J++ delegate concept without extending Java, but instead using reflective method lookup and invocation. Naturally, delegate construction with dynamic method lookup is less efficient. Fortunately, delegate construction does not happen frequently (typically before listener registration). During event handling the "installed" method is invoked, which is as efficient as a regular method invocation.

Layer 2: Simulation Kernel

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

This layer includes support for the following:

- models, model containers and hierarchies of models
- event scheduling, event sets and event handling
- numerical integration for continuous simulation
- discrete and continuous variables and variable coupling
- utility services for random number generation
- utility services for simulation data collection and analysis
- elementary output and visualization

Layer 3: Elementary Simulation Components

This is the central layer containing the elementary simulation components. It is specific to the problem domain; for example, there is one library for discrete process simulation (discussed in more detail below) and one for hydrodynamic systems. The layer contains components for the following:

- elementary model units
- classical control algorithms
- component coupling schemes
- utility services
- output, output analysis as well as visualization components

This layer is crucial for the success of the component-based simulation framework. It is a challenge for the simulation expert to foresee a wide range of applications in the domain and to provide a set of easy-to-use and easy-to-extend components. Modeling is based on system theoretic formalisms. We describe the modeling approach in more detail in Section 4.

Layer 4: Application-Specific Simulation Components

In layer 4 application-specific components have to be provided. In contrast to components of layer 3, these components are to model concrete real-world entities that are meaningful for the application engineer. They already have customized visualization and user interfaces using the representations usual in the application domain. They are assembled mainly by using and adapting elementary components from layer 3.

Layer 5: Application-Specific Simulation Systems and Environments

At the top of the hierarchy of layers there are application-specific simulation systems and environments. They are built up from the lower layer components as stand-alone systems. Like components of layer 4, the simulation systems and environments of layer 5 are specific to 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 (*plug-ins*), and simulation environments, which allow building and testing different configurations by assembling components from a library.

We regard the design of a simulation tool framework as a special challenge. Such a tool framework should allow building customized simulation tools in an easy way. We strive for a framework that will allow the application engineer together with the simulation expert to specify the following:

- the kind of model components used in the application domain
- the different ways how these components can be customized by utility components
- the different types of coupling schemes available
- a set of controller components and means to define such
- the possibilities for output and output analysis

By means of these features, the tools should be able to support the application domain. We can achieve these objectives by heavily relying on the following concepts:

- the modeling approach and component libraries as discussed above
- an object-oriented model representation in UML notation
- meta-representations of simulation components and systems.

3.2 Modules

In each layer, different *modules* realize different problem domains. Modules become more and more specific in the higher layers; see Figure 7. While at the lowest kernel layer we distinguish modules for implementations of discrete event, continuous and combined simulation mechanisms, at the layer of basic simulation components, differentiation occurs based on the elementary behavioral aspects. For example, the task of the discrete items module is to provide elementary components for items and resources. The moving objects module should allow the realization of entities that move in space and on predefined paths; these entities rely heavily on the discrete event simulation part. By contrast, the liquid flow module is based mainly on continuous and combined simulation. Its objective is to provide elementary building blocks for modeling liquid in vats and liquid flow in pipes.

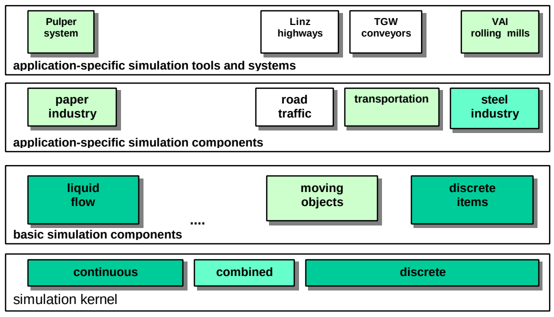


Fig. 7 Modules of the SimBeans Framework

At the next higher application-specific component layer, modules are created for particular application domains. For example, the paper industry module provides building blocks for paper mill plants and is based mainly on the liquid flow module. At the highest layer, there are modules that realize particular simulation systems and tools, for example, a simulation system for simulating the pulper in a paper mill, or a simulation tool for rolling mills of the steel equipment manufacturer.

The design of the elementary simulation components is crucial. In the following we describe these components in more detail.

4 Simulation Model Implementation

In this section we present the translation of the system modeling concepts discussed in Section 2.1 into Java source code. This translation is strongly based on the JavaBeans component model. We first show how atomic model behavior is realized by basing on a set of components and various event models from the simulation kernel layer. In Section 4.2 we discuss realization of interface-based coupling of simulation models, and we illustrate the ideas with an example component architecture in Section 4.3.

4.1 Implementing Atomic Models

Recall state space representation of simulation models from Section 2.1. A simulation model in state representation is a modular building block with an input and output interface, internal state variables, and a behavior specification in the form of derivative functions for continuous models and discrete events for discrete models. In the following we show how state space models can be implemented. The implementation of state space models is low level and meant to serve as a basis for higher coupled model concepts as discussed in Section 2.1, but also for future higher model description languages like state transition diagrams or rule-based formalisms.

We show atomic model implementation by presenting the implementation of the bottle filling model introduced in Section 2.1.

Defining Models and Variables

Models in our environment are subclasses of `BasicModel`. Models first of all have to define the static structure, that is, its subcomponents, if any, its variables, and their coupling.

States as well as input, output, and auxiliary variables in simulation models are implemented by subclasses of `Variables` (like `BooleanVariable`, `IntVariable`, `DoubleVariable` and `ObjectVariable)` that encapsulate values of different type. Read and write access has to occur through the methods `getValue` and `setValue`. The reason for using these wrapper objects to store state information is – in the sense of Proxies [4] – to allow additional functionality for state variables. Thus `Variables` objects provide elementary implementation of change event mechanism and `ContVariable` contains the fundamental functionality for numerical integration.

Figure 8 shows the definition of variables in `FillingStation`, `Bottle` and `Filler` classes. `Bottle` defines the input variable `inflow`, the output variable `level`, and the continuous-state variable `content`. The `Filler` is a resource and shows its availability through `BooleanVariable available`. Additionally, it provides its liquid output in the variable `outflow`. `FillingStation` models the discrete operations and has the discrete variables `phase` for its phase and `bottle` to hold the current bottle to be filled.

```
public class Bottle extends BasicModel {
public ContVariable content =
            new ContVariable(Variable.VARIABLE,0);
  public DoubleVariable inflow =
            new DoubleVariable (Variable.INPUT);
  public DoubleVariable level =
            new DoubleVariable (Variable.OUTPUT);
  ...

public class Filler extends BasicModel {
  public BooleanVariable available =
            new BooleanVariable (Variable.VARIABLE,  true);
  public DoubleVariable outflow =
            new DoubleVariable (Variable.OUTPUT);
  ...

public class FillingStation  extends BasicModel
      implements VariableChangeListener, BottleEventListener
{
  public StringVariable phase =
            new StringVariable (Variable.VARIABLE, "idle");
  public BottleVariable bottle =
            new BottleVariable (Variable.VARIABLE, null);
  ...
```

Fig. SEQARABISCH8 Filling station model implementation: Class definition

Continuous Model Implementation

Continuous model implementation is accomplished by translating the block diagram representation as discussed in Section 2.1 to a set of `Variables`. `ContVariable` is the class for realizing a continuous state variable over which integration occurs. The derivative for a state variable is defined by using `setDerivative` with the derivative variable as parameter. Direct connections between variables is established using function `connect`. Other more complex dependencies can be implemented in the model method `equation`, which is called during integration to set variables values.

```
// Bottle class continued
public Bottle () {
    inflow.setValue(0.0);
    content.setDerivative(inflow);
}
public void equation() {
    level.setValue (levelFromContent (content.getValue());
}
protected double levelFromContent (double content) { ... }
```

Fig. 9SEQARABISCH Bottle model implementation: continuous model `equation`

Figure 9 continues the implementation of the continuous model `Bottle`. In the constructor the derivative of `content` is set to be equal to the variable `inflow`, and method `equation` computes the current `level` from `content` using the auxiliary method `levelFromContent`. `Inflow` is set to 0.0 initially but will be `connected` to `outflow` of `Filler` when bottle filling (see Fig. 11).

External Events

Recall that we identified four different types of events in discrete models: external events, conditional events, time events and state events. The simulation kernel layer provides elementary interfaces and components for dealing with the different types of events.

Input events are realized with standard JavaBeans event methods (see `BottleEvent` in Section 2.2). Simulation model components fire particular events (event outputs) to which other components react accordingly. Listener interfaces with event methods are specified for model classes with input events.

Figure 10 shows the implementation of the input event `bottleInput` in `FillingStation`. This `FillingStation` is registered as an event listener of the previous bottle station. Upon input of a new bottle in a `BottleEvent` object (method `bottleInput`), the bottle is stored in the variable `bottle`, the model transits to phase `wait`, and the model sees whether the filler is available and can start filling. Additionally, a `fullGuard` state event object is set to monitor the fill level of the bottle (see below).

```
// FillingStation continued: input event
previousStation.addBottleEventListener (this);

public void bottleInput (BottleEvent bottleEvent) {
    if (phase.getValue() == "idle") {
      bottle.setValue (bottleEvent.getBottle());
      fullGuard.watch (bottle.level, 10);
      phase.setValue("wait");
      startFilling();
    }
}
```

Fig. 10 Model implementation `FillingStation`: input event

Conditional Events

To realize conditional events, variable change events have been realized as an extension of the standard bound-property change mechanism of JavaBeans. Variable change events extend bound-property change mechanisms by introducing asynchronous event handling (see Section 2.2). In discrete model realization, variable change events are used to realize state dependency of model components, for example, the dependency of a model from the availability of some resource. Here variable change mechanism can be seen as an efficient version of the activity scanning modeling approach [23] in other simulation systems.

Variable change events are realized by the event object `VariableChangeEvent`, which is a specialization of `PropertyChangeEvent`, and by the interface `VariableChangeListener` as a specialization of `PropertyChangeListener`. `Variable` classes as discussed above implement the variable change event handling in that they allow adding and removing listeners and fire variable change events in the writer method `setValue`.

```
// FillingStation continued: conditional event
  Filler filler;
  filler.available.addVariableChangeListener (this);
}
public void variableChange () {
    startFilling ();
}
public void startFilling () {
    if (phase.getValue() == "wait" &&
        filler.available.getValue()) {
      filler.available.setValue(false);
      filler.outflow.connect(bottle.inflow);
      phase.setValue ("filling");
    }
}
```

Fig. 11 Model implementation `FillingStation`: conditional event *filler available*

Figure 11 shows the use of the variable change mechanism for implementing the conditional event of the model `FillingStation`. The transition from phase `wait` to phase `filling` is triggered by the availability of the resource `filler`. The `FillingStation` adds itself as a listener for changes of `BooleanVariable available` of its `Filler`. Upon occurrence of a variable change, first method `variableChange` is called, which in turn calls method `startFilling`. After the phase and the availability of `filler` has been tested, the `filler` is seized by setting `available` to `false`, the `outflow` of `filler` is connected to the `inflow` of the `bottle`, and the `phase` is set to `filling`.

State Events

Time events and state events as well as events originating from user inputs are handled by the simulation event list implemenation and are based on the `simEvent` event type. Figure 12 shows the classes and interfaces for `simEvents`. As the direct descendant of `EventObject`, `SimEvent` is the general class for simulation events and specializes into `TimeEvent`, `StateEvent` and `InputEvent`. `SimEventListener` and its specializations `TimeEventListener`, `StateEventListener`, and `InputEventListener` are the respective event listener interfaces.

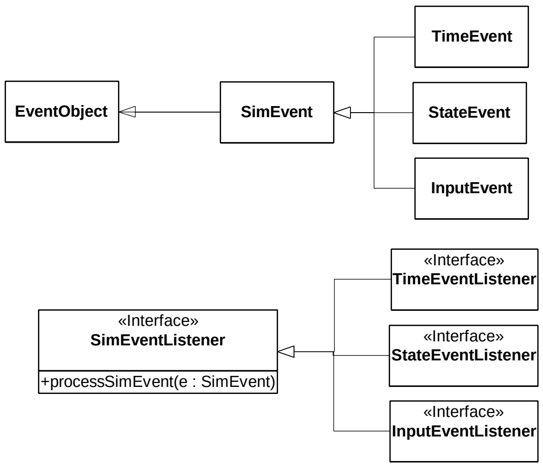


Fig. 12 SimEvents

These different types of events are triggered by different components. For example, different interface components are available that trigger `inputEvents` to insert user input values into the simulation model for visual interactive simulation. `Timer` is a component that triggers time events and functions similar to a clock. It can be set by calling `activateIn(deltaT)` or `activateAt(t)` and will trigger a `timeEvent` when the simulation time has come.

`StateGuard` implementations are employed to signal state events. They are set to watch particular continuous variables and then trigger whenever the continuous variable passes this threshold.

Figure 13 shows the implementation of the state event that is triggered using a `StateGuard fullGuard` when the bottle is full. The `fullGuard` was set in method `startFilling` (Fig. 10) to watch for the variable `level` to reach 10. Below an anonymous

`StateEventListener` is created to listen to this event and to call method `handleFull`. In method `handleFull` the phase is changed to `test`, the variable `outflow` of `filler` and variable `inflow` of `bottle` are disconnected, the `fullGuard` is passivated, and an `endOfTestTimer` timer object is scheduled to trigger in 1 time unit (see below).

```
// FillingStation continued: state event
StateGuard fullGuard = new StateGuard (level, 10);
fullGuard.addStateEventListener (new StateEventListener () {
        public void processSimEvent (SimEvent e) {
                                      this.handleFull(e);
}});

public void handleFull (StateEvent e) {
    if (phase.getValue()=="filling" && level.getValue()>=10) {
      phase.setValue ("test");
      filler.outflow.deconnect(bottle.inflow);
      bottle.inflow.setValue(0.0);
      filler.available.setValue(true);
      fullGuard.passivate ();
      endOfTestTimer.activateIn (1.0);
    }
}
```

Fig. 13 Model implementation `FillingStation`: state event *full*

Time Events

Figure 14 shows the implementation of the transition of the time event from phase `test` to `idle`; this event was scheduled in the full event. A `Timer endOfTestTimer` and an anonymous `TimeEventListener` are created. The `TimeEventListener` listens to the time events and calls `endOfTest`, whereupon firing the output event `bottleOutput` ejects the bottle, the bottle is set to `null`, and the `idle` phase is entered. Finally the implementation of `bottleEvent` is shown.

4.2 *Realizing Interface-Based Coupling of Simulation Components*

While atomic model implementation as presented above is low level in nature, we provide concepts and implementations for interface-based coupling of simulation components that we regard as higher level. In the following we discuss how this can be accomplished in Java/JavaBeans.

Recall from Section 2.1 that modular hierarchical modeling means building autonomous (modular) units and coupling them hierarchically employing different coupling schemes. The strict separation of component interfaces and coupling structure is our most fundamental design principle.

```
// FillingStation continued: time event
Timer endTestTimer = new Timer ("endTest");
endTestTimer.addTimeEventListener(new TimeEventListener(){
public void processTimeEvent (TimeEvent e) {
      this.endTest(e); }});

public void endTest (TimeEvent e) {
      bottleOutput (bottle.getValue());
      bottle.setValue (null);
      phase.setValue ("idle");
}
Vector bottleEventListeners = new Vector();
public void addBottleEventListener(BottleEventListener l){
      bottleEventListeners.addElement (l); }
public void removeBottleEventListener(BottleEventListener l) {
      bottleEventListeners.removeElement (l); }
protected void bottleOutput (Bottle b) {
    //call input of all bottleEventListeners
    ...};
```

Fig. 14 Model implementation `FillingStation`: time event `endTest`

We use Java interface definitions as the basis for coupling and hierarchical composition. Generic interfaces are defined for various simulation modeling problems to specify access to simulation components. According to the modeling features for continuous and discrete event modeling, interfaces can contain the following:

- for continuous couplings: access to input and output variables which can be connected directly calling `connect`
- for conditional events: add and remove methods to register variable change listeners
- for event outputs: add and remove methods to register various event listeners
- for input events: input event methods

These interface definitions are implemented by simulation modeling components and are used for coupling components but also for adding control strategies and attaching visualization, animation, and output analysis components.

Our methods to identify components and coupling schemes and definition of interfaces follow the ideas of bond-graph modeling [10], [3], which distinguishes *flow* and *effort* variables as well as *0-* and *1-junctions.* In bond-graphs a flow is a variable typically representing some flowing quantity, while an effort typically represents a status (although the selection is arbitrary). A 1-junction represents an effort reservoir and is associated with Kirchoff's voltage law; a 0-junction represents a linkage of effort reservoirs and is associated with Kirchoff's current law.

Our component technology interprets this rather abstract way of thinking clearly (see [7] for a similar approach). We distinguish between *reservoirs*, which store states, and *coupling* components, which model flows. Reservoirs are the model components that have interfaces for providing the state information of their effort states, e.g., the pressure in a hydrodynamic system, and that define their flows in and out, e.g., the amount of liquid flow in and out. On the other side, coupling components implement the flow between reservoirs, e.g., a pipe system with liquid flow based on the reservoirs' state information. In the next section we present the design of components for discrete item processing and distribution and show that this approach is also appropriate for discrete modeling. A similar component library has been designed for pulp processing and flow in paper mills [17].

4.3 Component Library for Discrete Item Processing

In discrete item 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 implement the item flow
- *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 that implements the flow of the items from one resource component to the next. The control part then decides which items can flow from the current resource 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 process 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 the control of the item flow is accomplished.

In the view of flows, efforts, effort reservoirs, and flow components, our modeling elements are classified as follows:

- Items are the flows.
- Reservoirs are the resources that can receive items, hold items, process items and provide items.
- States of resources signaling whether items are needed and items can be provided are the efforts.
- Coupling structures together with decisions on item distributions are the flow components.

Accordingly we defined the following elementary core interfaces and classes for our discrete process component library.

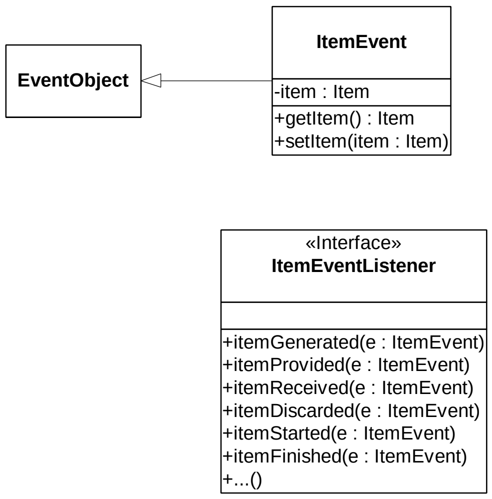

EventObject
ItemEvent
-item : Item
+getItem() : Item
+setItem(item : Item)
«Interface»
ItemEventListener
+itemGenerated(e : ItemEvent)
+itemProvided(e : ItemEvent)
+itemReceived(e : ItemEvent)
+itemDiscarded(e : ItemEvent)
+itemStarted(e : ItemEvent)
+itemFinished(e : ItemEvent)
+...()

Fig. 15 ItemEvents

Core Interface and Class Definitions

`Item`

The `Item` interface is a general interface for the items that can flow through the system. It defines general methods for operating with items, like obtaining/assigning a unique ID for the item and reacting to events that are triggered by resources on various operations on items (see below).

`ItemEvent and ItemEventListener`

We defined a general event type `ItemEvent` according to the JavaBeans event model for any event that might happen to be an `Item`. The resource components will signal various events to give other components a chance to listen and react to them.

Figure 15 shows the `ItemEvent` event type implementation. `ItemEvent` is the event object and refers to the item. `ItemEventListener` is an interface and specifies a set of event methods, like `itemReceived`, `itemProvided` and `itemStarted`, for reacting to various events that involve items.

Provider

Together with the Receiver, the Provider interface is fundamental for the realization of model components. These two components represent the basis for coupling. The Provider defines the output interface for an item reservoir to provide an item. It defines properties to provide its state, i.e., whether an item is available (bound property hasItem), methods to provide access to the item (retrieveItem), a method to allow inspection of the next item that can be provided (inspectItem), as well as an event interface to signal retrieval of an item (itemProvided event).

- boolean hasItem is a bound property to signal that an item is available in the component.
- Item retrieveItem() is a method allowing other components to access an available item.
- Item inspectItem() is an access method to retrieve the next item for preliminary inspection only.
- itemProvided is triggered when an item is actually retrieved.

Receiver

The complement of the Provider is the Receiver. The Receiver defines the input interface for an item reservoir. It defines properties to provide its aggregate state, i.e., whether an item can be received (bound property needsItem), methods to receive an item (receiveItem), a method to test whether an item can be retrieved (testItem), as well as an event to signal receipt of an item (itemReceived event).

- boolean needsItem, a bound property, signals that the component can receive an item.
- receiveItem(Item) is a method for handing over an item.
- testItem(Item), a predicate, is used to determine whether an item is appropriate to be received by the component.
- itemReceived is an item event that is triggered when an item is actually received.

In additional to these interfaces for item flow, several further interfaces define what components can do with items. For example, the Processor is an interface for components doing processing on items, Transporter is an interface for components transporting items, and Storage is an interface for components passively storing items. These interfaces are implemented by different types of components. For illustrative purposes we present the Processor interface.

Processor

- boolean idle is a bound property to signal that the component is not processing items currently.
- boolean busy, a bound property, signals that the component is processing items currently.
- itemStarted is an item event that is triggered when the component starts to process an item.
- itemFinished is an item event that is triggered when the component has finished the processing of a particular item.

Resource Components

A set of elementary building blocks is realized on the basis of the Receiver and Provider and other interfaces. Building blocks include Generator, Sink, SingleServer, MultipleServer, Queue, Place and Delay. Figure 16 shows the class hierarchy of some elementary resource implementations.

The components implement the interfaces in different ways. The Generator only implements the Provider interface and makes a new item available after some interarrival time. The SingleServer can process one item at a time; it signals that it needs a new item when the previous one has been moved on. Upon receipt of an item it is marked as occupied and immediately starts processing. After some processing time, it signals that an item is available and waits until it is retrieved from outside. Then it becomes free to receive the next item. The MultipleServer can process a number of items (its capacity) concurrently before being marked as occupied. A Delay component is never occupied and can always receive items; it delays items for some time and then makes them available for access. A Place is a passive component that can hold one item. 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 analogously.

More complex components can be built by either coupling 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 larger coupled models according the same coupling concepts.

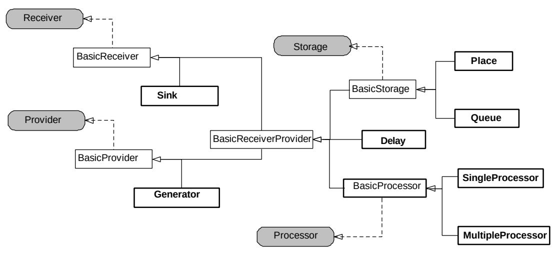


Fig. 16 Class hierarchy for elementary model bean (interfaces are gray; model beans are bold)

Coupling and Control: Realization of Item Flow

The interfaces `Provider` and `Receiver` are not coupled directly; instead, additional components corresponding to the flow components are used. We use the variable change event concepts to realize event coupling and communication in discrete event models. The general idea of coupling components is that the coupling components listen for changes of variables `hasItem` and `needsItem` of its connectors and react to these by distributing items among its connector providers and receivers based on their individual control schemes. Control and coupling can be arbitrarily complex, ranging from simple linear connections to a transportation system comprising a complex coupled model. Let us depict the range of couplings by considering some examples.

A `Connection` (Fig. 17(a)) realizes a direct flow of items from a `Provider` to the next `Receiver`. It listens to the `hasItem` variable of the `Provider` and the `needsItem` variable of the `Receiver` and, when both are true, takes the item from the `Provider` by calling `retrieveItem` and hands it over to the `Re-ceiver` by calling `putItem`. No control decision is needed here.

The `ReceiverDecisionPoint` (Fig. 17(b)) is used to couple a single provider with a set of receivers. The selection of the receiver of the next available item is based on a control strategy, a `ReceiverSe-lection`, which is a strategy component [4] selecting from a set of receivers. Components implementing different control strategies are possible, for example, selecting at random, based on given percentages, the receiver waiting longest, based on an next item operation, etc. In the same way a `ProviderDe-cisionPoint` (Fig. 17(c)) couples a set of providers with a receiver. With `Connection`, `Provider-DecisionPoint` and `ReceiverDecisionPoint`, coupled systems can be built that are typical of *Flow Shop* models.

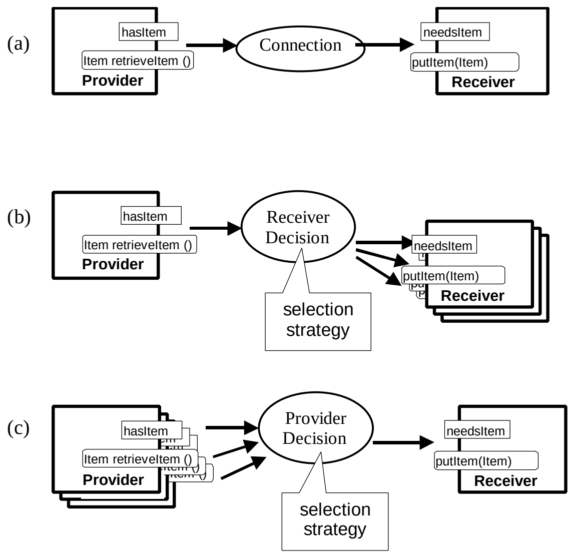


Fig. SEQARABISCH17 Connection, ReceiverDecisionPoint and ProviderDecisionPoint

A different coupling scheme should be used when modeling a robotic manufacturing cell. Here the item flow and control scheme is much more complex. A robot has direct access to the places in the cell. 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 item flow by accessing the item from the selected provider and by placing it on the selected receiver.

Modeling a manufacturing system with a complex transportation system needs yet 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 that 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 systems are coupled at *loading/unloading* stations. Note that the same type of components can be used in the different layers of the system.

5 Visualization, Animation, and Output Analysis

Visualization, animation, output, and output analysis in our simulation framework are realized by a component-based approach as well. The design objective has been to achieve maximum flexibility. Therefore no output or statistical analysis is fixed with a model component beforehand, but can be attached as requirements dictate. The property and events defined by model components represent the glues to attach any output and analysis component. Visualization, animation, output, and output analysis components serve as pure observers [4] of model states and events.

To illustrate the flexibility of the component-based approach, we present the realization of statistical analysis in some detail. In statistical analysis, two elementary types are important: time-dependent statistics (values per time units) and observational statistics (values per observation). The time-dependent statistics component `VariableStatistics` can be attached to any state variable (property) with numeric values as simple observers (Fig. 18). The variable change events report any state changes to the statistics bean, which can directly compute the statistics.

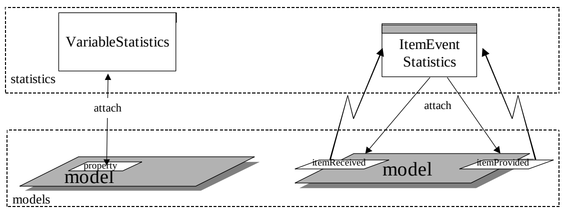


Fig. SEQARABISCH18 Attaching statistics beans.

The computation of time duration statistics per item (e.g., the statistics over the waiting time in a queue or the turnaround time in the system) is accomplished by the `ItemEventStatistics` bean. The `ItemEventStatistics` is based on the `itemEvent` implementation as presented in Section 4.3. It is attached to two item events, a *begin event* and an *end event* as `ItemEventListener`, measures the duration between the two events for the individual items, and computes statistics from the measurements (Fig. 18). Any statistical calculation for item duration in particular system parts can be accomplished easily by attaching an `ItemEventStatistics` to the particular begin and end item events.

6 Current State of implementation

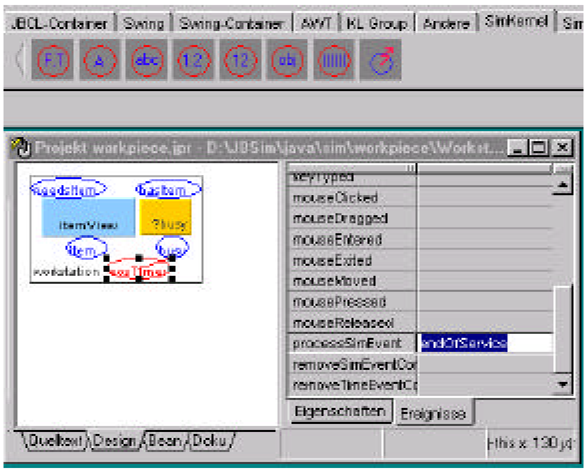
The modeling and simulation concepts discussed above have been realized as a prototype and tested on various examples. The prototype also has been used as an educational tool to teach modeling and simulation for computer scientist students [13].

Currently the focus is still on the lower layers of the architecture, the simulation kernel layer and layer of elementary simulation components. The main objectives of the implementation have been the evaluation of Java in general and the JavaBeans component model in particular as a platform for simulation programming. Also, we have evaluated several Java interactive developments environments (IDE) with JavaBeans builders. We investigated whether these can serve as simulation program development tools. Subsequent sections summarize our experiences.

One of the main advantages of strongly basing on the standardized JavaBeans component model is that it is possible to use any JavaBeans builder tool for simulation modeling. Although these tools have proven to be of limited use as general simulation modeling and experimentation environments in the long run (see Section 6.3), it was still possible to do interactive simulation program development already in early phases of our project. In the following section, we show how simulation programming can be done using the JBuilder 2 IDE.

6.1 Simulation Programming with JBuilder

Simulation programming with JBuilder can be done by alternating between interactive component instantiation, customization and source code editing. We briefly show three programming tasks of differing complexity that use different component palettes. According to various tasks, components are grouped into simulation kernel components, model components, output components, and statistic components. The examples emphasize the use of the inspector tool for interactive customization of components.


JBCL-Container | Swing | Swing-Container | AWT | KL Group | Andere | SimKernel | Sim
Projekt workpiece.jpr - D:\JBSim\java\sim\workpiece\Workst...
goodsItem
busyItem
itemView
?busy
item
busy
workstation
eosTimer
Eigenschaften
Ereignisse
Quelltext
Design
Bean
Doku
this x: 130 y:

keyTyped	
mouseClicked	
mouseDragged	
mouseEntered	
mouseExited	
mouseMoved	
mousePressed	
mouseReleased	
processSimEvent	endOfService
removeSimEventCo	
removeTimeEventC	

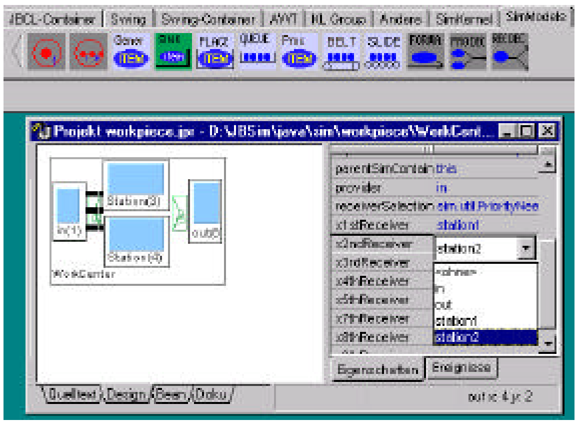
Fig. SEQARABISCH19 Atomic processing model implementation

Implementing Atomic Simulation Models

Figure 19 shows the implementation of an atomic processor model from components of the `SimKernel` palette. Various `Variables` and a `Timer` component are instantiated from the pallete to build up the static structure of the model. A view to visualize the current item being processed and a statistic component to compute statistics over the busy variable are instantiated from the palletes `SimOutput` and `SimStatistics`, respectively. With the `Timer` component `eosTimer` an event method is specified interactively in the inspector tool; this method can then be edited in the source code editor. An anonymous `TimeEventListener` to delegate the event to the method `endOfService` is introduced thereupon automatically by the tool.

Implementing Coupled Simulation Models

Figure 20 shows a two-processor station as a coupling of elementary model components from the palette `SimModels`. Items enter the station at input place `in` and then are forwarded by a `ReceiverDecisionPoint` to one of the processors. After processing, a `ProviderDecisionPoint` places them on an output place `out` for further distribution. Figure 20 also shows how coupling `Providers` and `Receivers` in `ReceiverDecisionPoint` can be done in the inspector. Properties `x1stReceiver`, `x2ndReceiver` etc. are of type `Receiver`. JBuilder allows interactively setting these properties to any component of type `Receiver` that has been instantiated in the worksheet.


JBCL-Container | Swing | Swing-Container | AWT | KL Group | Andere | SimKernel | SimModels
Projekt workpiece.jpr - D:\JBSim\java\sim\workpiece\WorkCent...
in(1)
Station(3)
Station(4)
out(0)
WorkCenter
Eigenschaften
Ereignisse
Quelltext
Design
Bean
Doku
out x: 4 y: 2

parentSimContain	this
provider	in
receiverSelection	sim.util.PriorityNee
x1stReceiver	station1
x2ndReceiver	station2
x3rdReceiver	
x4thReceiver	
x5thReceiver	
x7thReceiver	
x8thReceiver	

```
<ohne>
in
out
station1
station2
```

Fig. SEQARABISCH20 Coupled two-processor station implementation

Implementing Simulation Systems

Figure 21 shows the building of a simple manufacturing system together with visualization and output analysis. Workpieces are transported to workstations by a conveyor system with a ring structure. This system was built mainly interactively by instantiating and customizing components. Workpiece distribution to workstations is accomplished based on an operation sequence for workpieces. Figure 21 shows the specification of control of workpiece distribution in `ReceiverDecisionPoint` based on the next workpiece operation. A control component `NextOperationReceiverSelection` is instantiated (using our own property editor [8]) and the operation for the first receiver is defined to be "schleifen" (grinding). Any workpiece whose next operation is grinding is forwarded to the first receiver.

6.2 *Assessment of JavaBeans*

JavaBeans provides an attractive platform-independent component model. Our experiences with the model have been quite positive. In particular, the JavaBeans event model nicely accommodates the requirements of discrete event simulation. The Java interfaces give a nice implementation concept to realize the idea of interface definitions and interface-based coupling of simulation components.

However, we have encountered a few shortcomings in the JavaBeans model:

- The JavaBeans specification does not contain any information about hierarchical bean structure. Therefore, no access is defined for superordinate or subordinate beans. In our application we regard hierarchical composition of components as essential; hence we missed an explicit representation of the component hierarchy. In JDK version 1.2, however, which we are not using currently, there is corresponding support (`BeanContext` and `BeanContextChild`).
- The property change mechanism propagated by the JavaBeans standard for synchronous event handling fails when feedback occurs. As discussed in the paper, we have extended the standard property change mechanism with variable change events for providing more powerful concepts.
- JavaBeans does not define how code should be generated by bean builder tools. This leads to different approaches in tools and thus prohibits using several builder tools together or switching from one tool to another.

6.3 Assessment of Bean Tools

Relying on the JavaBeans standard allowed us to use existing bean tools for visual interactive simulation programming. We have used and evaluated three tools for that purpose: IBM's VisualAge, Symantec's Visual Cafe, and Inprise's JBuilder. Of these we found JBuilder 2 to serve our purposes best. We found that support for visual interactive programming helps in implementing single simulation systems. However, although we tried hard to adapt our simulation components to the needs of bean tools, we found them unsatisfactory to serve as a simulation environment. Current bean tools have one or more of the following weaknesses:

- They do not support the full JavaBeans standard.
- Their implementation is immature.
- The code they generate for coupling of components is hard to track.
- They do not allow access to their instantiated beans at design time.
- They do not allow access to subcomponents of instantiated beans at design time.
- The possibilities to customize components at design time are limited.

To serve our purposes, we would like bean tools to offer the following additional features:

- access to instantiated beans and their hierarchical structure at design time
- possibilities to define and constrain the types of components allowed
- user definition and use of various concepts for bean coupling
- user definition of the kind of code generated by the tool

One of the points of emphasis for our future work, therefore, will be the realization of a specialized bean tool for simulation model development. This bean tool should overcome the above shortcomings and will be a basis for realizing customized, application-specific modeling and simulation environments as discussed in Section 3.1.

7 Summary and Outlook

In this paper we have investigated and assessed the Java/JavaBeans component model as an underlying technology for the realization of a component-based discrete event simulation methodology. The DEVS formalism and its extensions have been discussed as providing fundamental concepts for a modular hierarchical model that facilitates the design of reusable components.

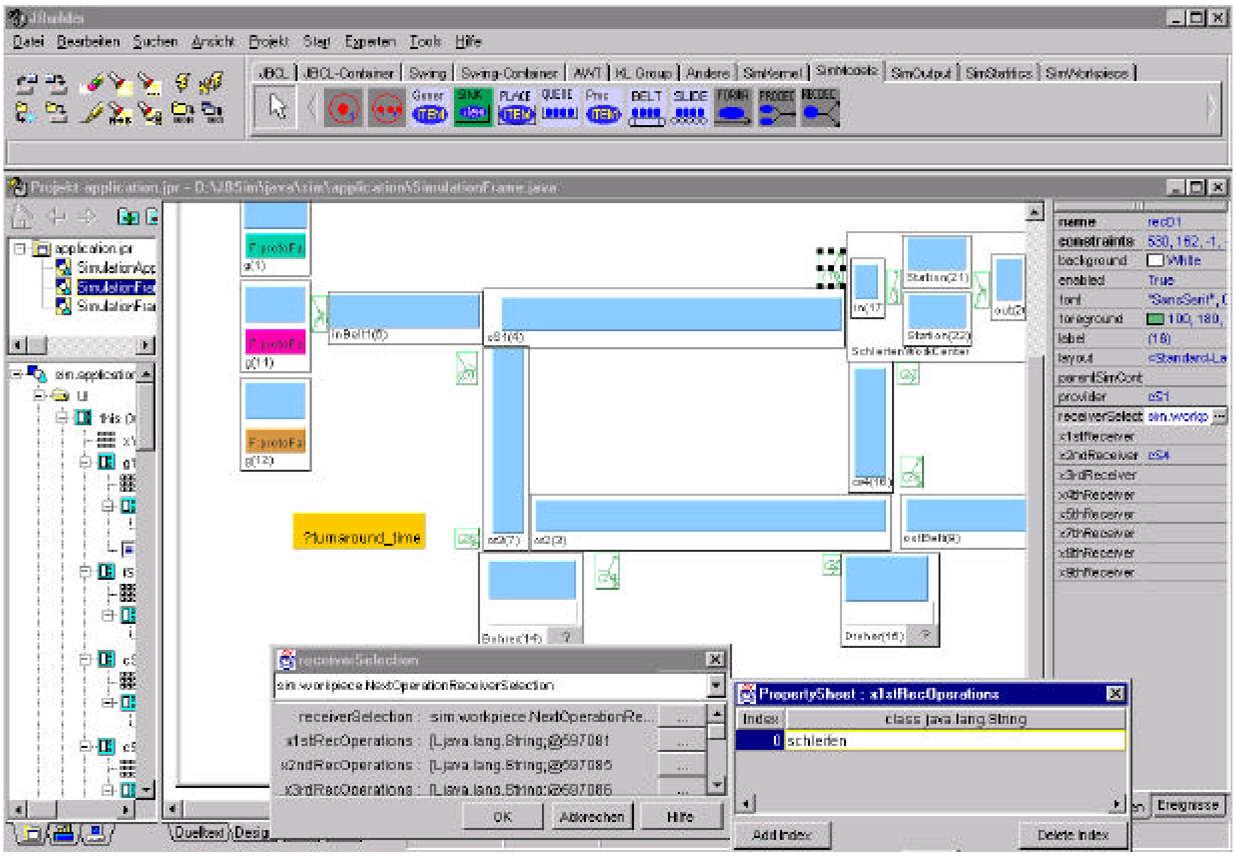
We have presented how a simulation framework can be built based on the JavaBeans component model. We have shown how the event model of JavaBeans nicely accommodates the implementation of discrete event systems. With the design of a component architecture for discrete item processing, we have demonstrated how interface-based, hierarchical coupling of simulation component can lead to reusable components.

In the architecture of the simulation framework we identifiy several layers, from elementary simulation algorithms to elementary simulation components, application-specific simulation components, and on to application-specific simulation systems and environments. All layers comprise reusable components. The architecture is open, allowing for extension and adaptation to specific needs.

Motivated by the good experiences in realizing the prototype, we will continue our work to realize the component framework for modeling and simulation. In that, we see the following points of emphasis for our future work:

- The different layers of the architecture will be elaborated to make them maturer.

- We will implement our own bean tool for simulation model development.
- We will investigate possibilities for higher forms for atomic model specification, like state transition diagrams or rule-based formalisms, and investigate how a translation to our implementation concepts can occur.
- We will investigate how object-oriented design tools (providing UML notation) can support simulation model development.


JBuilder
Datei Bearbeiten Suchen Ansicht Projekt Start Experten Tools Hilfe
JBCL JBCL-Container Swing Swing-Container AWT KL Group Andere SimKernel SimNodes SimOutput SimStatistics SimWorkpieces
Gener SINK PLACE QUEUE Proc BELT SLIDE FORK PRODEC RECDEC
application.jpr
SimulationApp
SimulationFrame
inBelt1(6)
eS1(4)
Station(21)
Station(22)
SchleifenWorkCenter
in(17)
out(2)
?turnaround_time
eS2(7)
eS2(3)
outBelt(8)
Bohrer(14)
Dreher(16)
Quelltext Design

name	recD1
constraints	530, 162, -1, -
background	⬜ White
enabled	True
font	"SansSerif", (
foreground	🟩 100, 180,
label	(18)
layout	<StandardLa
parentSimCont	
provider	eS1
receiverSelect	sim.workp
x1stReceiver	
x2ndReceiver	eS4
x3rdReceiver	
x4thReceiver	
x5thReceiver	
x7thReceiver	
x8thReceiver	
x9thReceiver	

receiverSelection

sim.workpiece.NextOperationReceiverSelection

receiverSelection :	sim.workpiece.NextOperationRe...
x1stRecOperations :	[Ljava.lang.String;@597081
x2ndRecOperations :	[Ljava.lang.String;@597085
x3rdRecOperations :	[Ljava.lang.String;@597086

OK Abbrechen Hilfe

PropertySheet : x1stRecOperations

Index	class java.lang.String
0	schleifen

Add Index Delete Index

Fig. 21 Implementation of a flexible manufacturing system

References

[1] Barros, F.J., Modeling Formalisms for Dynamic Structure Systems, ACM Transaction on Modeling and Simulation, Vol 7, No. 4, 1997, pp. 501- 515.

[2] Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language, Users Guide*, Addison Wesley, 1998.

[3] Cellier, F.E., *Continuous System Modeling*, Springer-Verlag 1991.

[4] Gamma, E., et al, *Desgin Patterns*, Addison Wesley, 1994

[5] Gosling, J. and H. McGilton, *The Java Language Environment – A White Paper*; Sum Microsystems, MV California, 1995.

[6] Harel, D. and M. Politi, *Modeling Reactive Systems with Statecharts*, McGrawHill, 1998.

[7] Helmquist et al 98, ModelicaTM – *A Unified Object-Oriented Language for Physical Systems Modeling; Tutorial and Rationale, Version* 1.1, December 15. 1998, URL: http://www.modelica.org.

[8] Hilpold, T., JavaBeans Property Editors, Technical report, C. Doppler Laboratory for Software Engineering, Johannes Kepler University, Linz, Austria 1999 (in German).

[9] Mathworks Inc., *SIMULINK User's Guide*, 1996.

[10] Payntner, H. M., *Analysis and Design of Engineering Systems*, MIT Press, Cambridge 1961.

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

[12] Praehofer, H., An Environment for DEVS-Based Multiformalism Modeling and Simulation in C++. *Proc of Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*, San Diego, CA, 1996.

[13] Praehofer, H., 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.

[14] Praehofer, H., and D. Pree., Visual Modeling of DEVS-based Systems Based on Higraphs. *Winter Simulation Conference*, 1993, S. 595-603.

[15] Praehofer, H., Systems Theoretic Foundations for Combined Discrete-Continuous System Simulation, PhD Thesis, Johannes Kepler University, Linz, Austria 1991.

[16] Sametinger, J., Software-Engineering with Reusable Components, Springer-Verlag, 1997.

[17] Schöppl, A., *Training simulators for papermill operators*, Diploma thesis, Dept of Systems Theory and Information Engineering, Johannes Kepler University, Linz, Austria 1999 (in German).

[18] Stritzinger, A., *Komponentenbasierte Softwareentwicklung*, Addison-Wesley, 1997.

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

[20] Szyperski, C., Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.

[21] Thomas, C., Interface-based classification of simulation models, *Winter Simulation Conference* 94, Orlando, FL, 1994.

[22] Uhrmacher, A.M. and B.P. Zeigler, Variable Structure Modelling in Object-Oriented Simulation, *International Journal on General Systems*, Vol. 24(4), 359-375, 1996.

[23] Zeigler, B.P., Theory of Modeling and Simulation; Wiley 1976

[24] Zeigler, B.P., Multifacetted Modelling and Discrete Event Simulation. Academic Press, 1984.

[25] Zeigler, B.P., Object Oriented Simulation with Modular, Hierarchical Models. Academic Press, 1990.

[26] Zeigler, B.P. and H. Praehofer, Sytems Theory Challanges in the Simulation of Variable Structure Systems, *EUROCAST* '89; LNCS 410, Springer-Verlag, 1990, pp. 41-50.

[27] Zeigler, B.P., H. Praehofer, and T.G. Kim, *Theory of Modeling and Simulation*, 2nd Edition. Academic Press, 1999 (in preparation).