

Component Frameworks – A Case Study

Herbert Praehofer, Johannes Sametinger, Alois Stritzinger
Johannes Kepler University, Linz, Austria

hp@cast.uni-linz.ac.at, sametinger@acm.org, stritzinger@swe.uni-linz.ac.at

Abstract

This paper reports on an effort to use both the system theoretic DEVS (discrete event simulation) formalism and the JavaBeans component model as a basis for a component-based discrete event simulation framework. 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.

Component frameworks are dedicated and focused architectures with a set of policies for mechanisms at the component level. In this paper we describe the component framework we have developed for discrete event simulations. Simulation components are based on this framework and can be composed for the creation of various simulation scenarios.

1. Introduction

We have developed a set of JavaBeans components for the creation of discrete event simulations. The goal was to investigate how discrete event simulation applications can profit from an up-to-date component technology. The idea was to create a component framework for discrete event simulation, a set of basic simulation components together with visualization and animation components that can be arranged and connected on a worksheet. The modeling approach is based on the DEVS, a system theoretic formalism for discrete event modeling that provides a theoretic framework for modular, hierarchical modeling [5, 6].

Component frameworks are dedicated and focused architectures with a set of policies for mechanisms at the component level [3]. Component frameworks have similarities with application frameworks. They provide a framework for components rather than objects but are not necessarily used for the creation of entire applications. A component framework for discrete event simulation has to provide mechanisms for simulation control and item flow. Simulation is controlled by events that activate components. In order to be part of the simulation, components have to be registered for events at certain points of time. Additionally, components have to process, provide and receive items, thus allowing items to flow from component to component and being processed by these components. In order to enable item flow, all components have to adhere to a mechanism defined by the framework.

In Section 2 we describe the component-based simulation methodology. The component framework is described in Sections 3. Design considerations of the framework are described in Sections 4. Conclusions are drawn in Section 5.

2. Component-based simulation methodology

A component-based modeling and programming framework for simulation applications has to enable developers to interactively pick components from libraries and place them onto a worksheet. Such components comprise simulation components as well as visualization, animation and statistics components. As a next step, connecting these components has to be accomplished, such that signals and data can be exchanged among them. Additionally, convenient interactive customization of parameters has to be supported. The component system has to be simply executed and used as a simulation application or as a more complex component in other simulation applications.

We envision a component-based simulation methodology which provides component libraries for different purposes, different users, and different applications, see [1, 2]. Components advocate and support a plug and play framework. It should be possible to develop simulation systems with less effort by mainly selecting, extending, customizing, and connecting components. It should be less effort for developers to realize specific components or specific simulation systems and environments. The component provider is faced with the challenge of designing components in a way so that they can be reused in a wide range of applications.

3. Component framework

Components for discrete event simulation need to have a common basis for basic simulation principles, e.g., a common simulation time and a common mechanism for event scheduling. Additionally, they need to have a common understanding of how items are constituted and how item flow is realized. These aspects have to be defined in the component framework. Any components being based on this framework may be combined for construction of simulation scenarios. The component framework for discrete event simulations consists of the simulation kernel and a set of interfaces for component coupling. They will be described subsequently.

3.1. Simulation kernel

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

For discrete event simulation, the kernel includes support for

- event scheduling and event handling,
- models, model containers and hierarchies of models,
- state variables and property change mechanism, and
- utility services for data collection and analysis.

In order to participate in a simulation scenario, a component has to register itself to the simulation kernel, see Fig. 1. Additionally, it can register itself for activation at a certain point of time. A simulation scenario is built by composing simulation components that have to communicate with the simulation kernel, as well as components for visualization, animation and statistics. Components that are not directly involved in the simulation, e.g., visualization components, do not have to be aware of the simulation kernel. Therefore, arbitrary components may be included for animation, for visualization, and for statistical evaluations.

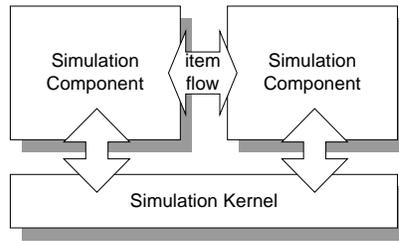


Figure 1. Simulation kernel

The simulation kernel drives the simulation by processing events that have been registered by components. The effect of an event depends on the component that is activated thereupon. For example, a component may generate or process items. The simulation kernel does not have any influence on what events trigger. Additionally, the kernel does not care about items. Components have to take measures to handle items and their flow through the system, i.e., to receive items from other components and to provide items to other components. In order to enable item flow, all components have to adhere to the same mechanism, which is also defined in the framework, and described in the next section.

In order to enable communication between the simulation kernel and components, two interfaces have been designed, interface DEVS and interface Resource, see Fig. 2. DEVS defines the interface to the simulation kernel, whereas Resource defines a minimal interface to components. The simulation kernel defines a routine to register components. This is necessary to control a simulation run and, for example, allows the kernel to reset all components before a new simulation run is started. The simulation kernel defines another routine to register events, i.e., components are registered to be triggered at a certain simulation time. These events are triggered by calling the routine trigger when the specific simulation time has arrived.

3.2. Simulation components

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

- *resource components*, which are active or passive and may be occupied by items,
- *items*, which flow through a system of resource components, and
- *glue components*, which control item flow among resource components.

A simulation system, therefore, is viewed as consisting of several resource components, where items are placed and processed, and a coupling structure which realizes the flow of the items from one resource component to the next. Glue components decide which items can flow from which resource components to the next based on requirements of items and space availability of resource components. This is a general, abstract view which fits to all types of discrete event simulation. Systems differ in what type of resource components are used, the types of items used, the glue structure, 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.

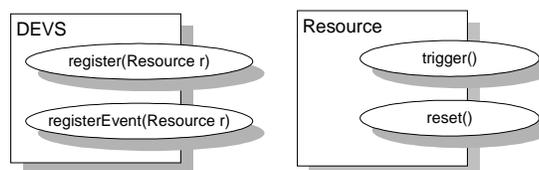


Figure 2. Interfaces DEVS and Resource

3.3. Resource components

Resource components form the key elements from which simulation systems are built. They are specific for the application domain, i.e., there is a library of resource components for different types of simulation, e.g., discrete event simulation (as discussed here). Resource components are crucial for the success of the component-based simulation framework. It is the challenge to foresee a wide range of applications in the domain and provide a set of easy-to-use and easy-to-extend components.

Resource components of discrete event simulations include what we call generators, sinks, processors, queues, places, and delays. Generators make new items available after some production time. Processors can process one single item at a time. Upon receipt of an item they get occupied and immediately start processing. After some processing time, they signal that an item is available and wait that it is accessed by another component. Delay components are never occupied and can always receive items. They delay items for some time and then make them available for access again. Place components can take one item. They signal that they need an item when there is no one on the place. When they receive an item they signal that they are occupied and that they have an item available for access. Other components behave in analogous way.

A resource component can be active, i.e., it can do some processing on an item, or it can be 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 component will process received items, which will take some time, and afterwards want to get rid of them. Also components may have space available, i.e., they may need or be able to take further items.

Two interface definitions are crucial to the realization of item flow. These are the *Receiver* interface for components which may receive items and the *Provider* interface for components which may provide items. 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 define methods to provide/receive an item and to inspect/test an item. They also signal an event when they have/need an item. Additionally, they signal an event whenever they actually provide/receive an item, see Fig. 3. For example, a generator, i.e., a component generating items, only implements the provider interface and makes new items available. More complex components are built by either coupling together components in a hierarchical way or by implementing them in Java using elementary simulation functions. By implementing the provider/receiver interfaces, they can be used in bigger coupled models according the same coupling concepts.

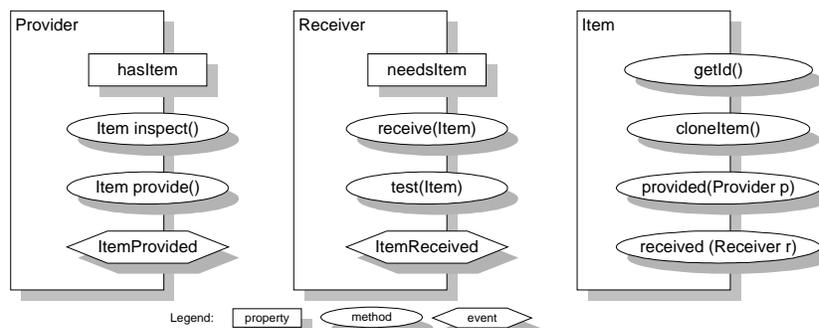


Figure 3. Provider, Receiver and Item interfaces

3.4. Items

Items are also described by an interface. They have a unique ID and provide several methods that may be called when events occur, e.g., when the item has been received or provided, i.e., when it has been moved on from a resource component, see Fig. 3. Components can implement the provider/receiver interfaces in different ways. They can also implement the item interface and, thus, be used as flowing items. Therefore, components are not of a particular type but rather they play the role of a particular type. They may also play various roles of different types. For example, a transportation component may on the one hand serve as a container, i.e., be a resource for some items (a resource component), but on the other hand it may flow as an item through the transportation system itself.

3.5. Glue components

The provider and receiver interfaces are not coupled directly, but rather additional glue components are used. The general idea of glue components is that they listen to property changes (`hasItem` and `needsItem`) of resource components and react to those by distributing items between providers and receivers based on an individual control scheme.

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 item registers itself as a listener of the `needsItem` property of the appropriate item provider. Everything flowing through the system is regarded as being an item, e.g., tools needed to process other items (work pieces). As soon as an item gets available, the processor is informed and can access the item. Control and coupling can be arbitrarily complex, ranging from simplest linear forwarder to a transportation system built up by a complex coupled model by itself. Fig. 4 shows a glue component coupling one provider and two receivers. Thick arrows from the glue component to the resource components designate method calls, thin arrows from resource components to the glue component designate the flow of events.

Examples of prefabricated glue components are what we call *forwarder*, *receiverDecisionPoint* and *providerDecisionPoint*. 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 to the forwarder is the *receiverDecisionPoint*. It 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, i.e., 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 the mentioned glue components 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 robot has direct access to the places in a 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 cell components and generates transport commands to the robot. The robot then realizes item flow by accessing items from the specified provider and by placing them on the specified receiver.

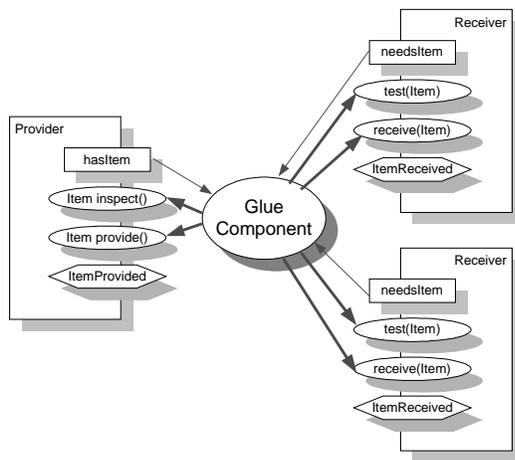


Figure 4. Glue component

3.6. Framework architecture

Fig. 5 depicts how the JavaBeans component model, the simulation component framework, simulation components and regular JavaBeans components interrelate. JavaBeans builds the foundation enabling any JavaBeans components to be included in simulation scenarios. The simulation kernel is built on top of the JavaBeans component model and contains the central event mechanism. All simulation components have to be built on top of the simulation kernel. In addition, they have to adhere to the framework's interfaces in order to exchange items. Item distribution is separated from item processing by providing resource components and glue components, thus providing more flexibility in building simulation scenarios.

4. Design considerations

We have implemented the component framework in Java and the components as JavaBeans [4]. JavaBeans provides an attractive platform-independent component model, making our components platform-independent and allowing us to easily integrate components for visualization, for animation, and for statistical evaluations from other sources. JavaBeans offers simple mechanisms for component coupling, i.e., events and the property change mechanism. The Java interface concept corresponds to the roles of components. A component playing a particular role has to implement the corresponding interface.

In this section we describe design considerations of the component framework. There were two central design decisions. First, in the simulation kernel we had to decide issues based on the event mechanism. Second, item flow among components can be realized in many different ways. Picking a suitable solution is interesting and offers insights in the way components can be connected.

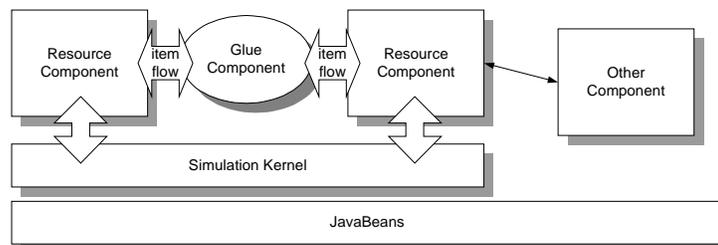


Figure 5. Simulation component framework

4.1. Event handling in the simulation kernel

The simulation kernel needs to control simulation time and has to trigger events based on event registrations of resource components. The simulation kernel is based on 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 is given for synchronous event handling (class `PropertyChangeSupport`). However, this mechanism is not appropriate for simulation events.

In a simulation scenario it is typical that a resource component signals, for instance, the availability of a new item. The first listener being notified about such an event reacts by requesting the item and processing it. Other registered listeners get informed about the event when actions of the first listener have already taken place. For example, a component intended to trace item flow will get notified about the event too late because through the actions of another component the information provided by the event is outdated.

It is quite typical that event handlers react upon notification and modify the state of the event source, e.g., receiving and processing an item. In this situation problems arise, because conceptually parallel event handling is simply serialized such that each component performs event handling actions before the next component is notified about the event. The solution for this problem is the use of asynchronous event handling. Events must not be processed immediately, but have to be collected in a global event queue. When a component triggers an event, event objects for all registered listeners are queued, then event handling takes place in a first-in-first-out order whereby new events may be appended to the queue. As a consequence, events have to know their listeners, such that they can be delivered when being taken out of the queue.

Synchronous events can only be used for components that do not have any influence on the state of resource components. For example, whenever an item is received by a component, this should be visualized by corresponding components immediately, even when the item is put forward. Asynchronous events are important when several listeners are registered with the same provider. The first listener receiving the event would get the item before the second one has a chance of even getting the information that there is an item available, i.e., the second listener would get the information when the item is not available any longer. Summarizing, we use synchronous events for animation and visualization components and asynchronous events for glue components.

4.2. Item flow

Item flow among components is of utmost importance for efficient simulation applications. Any component that is intended to be used within a simulation has to adhere to the component framework in order to participate in the item flow, i.e., in order to be coupled with other components and to provide and receive items. Subsequently, we will describe different solutions and discuss their advantages and drawbacks.

4.2.1. Property changes: Property changes are simple and can be used for the connection of any JavaBeans component. Thus, it is possible to integrate visualization components with simulation components, even though visualization components are not based on the simulation component framework. The concept of bound properties is based on events. Whenever the property of a component is modified, an event is triggered in order to notify listeners. This mechanism can be used to inform an item receiver that an item provider has an item available, see Fig. 6a.

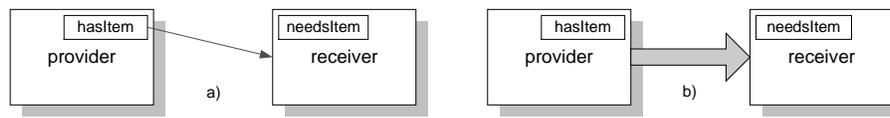


Figure 6. Component coupling a) with bound properties b) with interfaces

In the example of Fig. 6a an event is sent to the receiver component whenever the boolean property `hasItem` of the provider component is modified. This connection can simply be established by the following Java code, which is usually generated by a building tool:

```
provider.addPropertyChangeListener("hasItem",receiver);
```

The receiver can check the value of the `hasItem` property of the provider and, if available, receive the item. However, it turns out that communication between components becomes clumsy and unnecessarily complex, because after the property change has occurred, the provider and the receiver have to identify themselves and to negotiate about item flow. This is necessary because several receivers can be registered at the same provider. They all get notified when an item is available, but only one receiver may actually receive the item.

The concept of property changes is not powerful enough to realize sophisticated component coupling, but for simple data exchange among components it is a possible solution. The main disadvantage of this type of item flow is that the simple mechanism of bound properties is burdened with complex communication and because it is not possible to control item flow. For example, it is not feasible to move items to receivers based on a specific strategy. Another disadvantage of using bound properties is the fact that there is no static type check, i.e., it can only be detected at runtime whether components are listening to the properties they are interested in.

4.2.2. Abstract interfaces: For intensive cooperation abstract interfaces are a useful means to define operations. Such interfaces make cooperation more efficient, but all cooperating components have to implement the specific interfaces, see Fig. 6b. In the example of Fig. 6b the provider knows the identity of the receiver and can use various operations defined in the receiver's interface in order to move an item forward. Such a connection can again be established by a simple Java statement:

```
provider.setItemReceiver(receiver);
```

Using interfaces provides flexibility, but there is still a major drawback. First, components have to know each other. Second, the provider (or the receiver) has to make decisions on where items should be delivered. Thus, the provider not only has to know all the receivers, it also has to have additional information in order to implement a distribution strategy, e.g., to randomly or linearly distribute items. This means, that the same resource component has to be available in different forms, implementing different distribution strategies. Additionally, all resource components should provide all distribution strategies, leading to a vast number of similar components.

4.2.3. Adapters: A more flexible form of cooperation can be reached by using adapters that listen to bound properties and communicate via interfaces. Fig. 7a demonstrates the use of an anonymous adapter. Such a connection can be established with an anonymous Java class, i.e., it is necessary to manually write Java source code:

```
PropertyChangeListener adapter = new PropertyChangeListener {
    void propertyChange (PropertyChangeEvent e) {
        if (provider.hasItem() && receiver.needsItem()) receiver.receive(provider.provide())
    }
};
provider.addPropertyChangeListener("hasItem",adapter);
receiver.addPropertyChangeListener("needsItem",adapter);
```

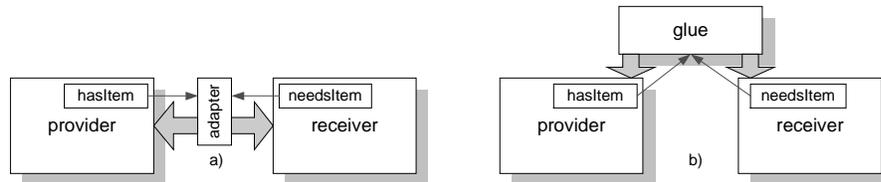


Figure 7. Component coupling a) with anonymous adapter b) with glue component

Adapters provide additional flexibility over the previous solutions, but coupling is rather cumbersome having to implement distribution strategies into anonymous classes. From this point the next step seems obvious, i.e., using components rather than anonymous adapters.

4.2.4. Glue components: Glue components implement various distribution strategies and can simply be placed among arbitrary provider and receiver components, see Fig. 7b). The source code to establish the necessary connections is as follows:

```
provider.addPropertyChangeListener("hasItem", glue);
receiver.addPropertyChangeListener("needsItem", glue);
glue.addProvider(provider);
glue.addReceiver(receiver);
```

Like adapters, glue components register with their provider and receiver components in order to get informed about item availability and item requests. Resource components know nothing about coupling and only send events and property change notifications. (Listeners have to be administered of course.)

4.2.5. Design rationale: Component systems in general and simulation systems in particular may grow to huge systems consisting of thousands of components. It becomes extremely hard if not impossible to understand and maintain such systems, especially when there are arbitrary dependencies among the components. Restricting the dependencies to be only in one direction leads to a layered architecture that is easier to comprehend and maintain. We have paid attention to the fact that communication among components is clearly defined and only one way as much as practical. For example, resource components communicate with their environment only via broadcasting events. Glue components listen to these events and call specific methods of the resources in order to realize item flow, i.e., they have specific knowledge about all the resource components involved in item flow.

It is sometimes impossible to completely restrict communication to one direction. In such cases we use events as dedicated communication vehicles; the number of events has been kept to a minimum with their semantics easy to comprehend and globally known. In other words, we use a layered architecture, i.e., components know nothing about components of layers above them. Whenever it is necessary to communicate signals or data to higher layers, we define appropriate events and listener types. Event sources broadcast events and only know that there may be registered listeners.

The same communication mechanism is used for specialized visualization components that listen to events of resources and may also access specific data of the resources for better visualization. Naturally, resource components are unaware of any visualization mechanisms. Hence, this model-view component architecture is practical not only for a separation of models and views, but also for the realization of layered architectures.

5. Conclusion

We have described a component framework for discrete event simulation that is based on the JavaBeans component model. We regard the framework as being typical for component based software engineering in that it provides an infrastructure that is necessary in a specific domain. Effective coupling of components will only be successful on top of component models that enable simple composition of arbitrary components as well as on top of specific component frameworks that enable more complex and more effective composition of domain-specific components.

So far we have implemented a basic set of resource and glue components as well as some domain specific items. New components can be implemented by extending existing ones. We have also defined abstract classes that contain basic functionality needed for the implementation of resource components like registering with the simulation kernel. We are confident that discrete event simulation applications highly profit from component technology in terms of reusability and flexibility.

We have used the components developed so far for the simulation of simple scenarios only. Due to the good experiences we have made, we have decided to make the system more mature in order to additionally serve as a simulation framework for real world applications. We also plan to extend the framework to allow continuous and combined simulation. The main objective is not only to extend the functionality of the framework and the components, but to gain additional insight into problems and solutions of component based software construction.

Many questions remain unanswered and need more work to be done in the future. For example: Are the design principles of our framework typical for the domain or may they be applied to other domains as well? Are there general mechanisms used in our framework or in frameworks of other domains, that may be included into the component model for better coupling of arbitrary components?

6. References

- [1] 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.
- [2] H. Praehofer, J. Sametinger, A. Stritzinger: "Discrete Event Simulation using the JavaBeans Component Model", WEBSIM99, 1999 International Conference On Web-Based Modelling & Simulation, San Francisco, California, January 17-20 1999.
- [3] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.
- [4] Sun Microsystems: JavaBeans, <http://java.sun.com/beans/>.
- [5] B.P. Zeigler, Multifaceted Modelling and Discrete Event Simulation. Academic Press, 1984.
- [6] B.P. Zeigler, Object Oriented Simulation with Modular, Hierarchical Models. Academic Press, 1990.