BUILDING REUSABLE SIMULATION COMPONENTS

Herbert Praehofer[1], Johannes Sametinger[2], Alois Stritzinger[2]
[1] Department for Systems Theory and Information Technology
[2] Department for Software Engineering
Johannes Kepler University, A-4040 Linz / Austria
e-mail: hp@cast.uni-linz.ac.at

ABSTRACT

In the SimBeans project we apply a component-based software engineering approach to the development of simulation systems and frameworks. Libraries of reusable simulation components have been realized on the basis of the component model JavaBeans for various simulation application domains. The main objective thereby has been to enhance model reusability. Although the system modeling approach provides an underlying foundation for building reusable models, there exist no application-specific patterns and architectures, which define how such libraries of reusable components should be built. In this paper we present an attempt in this direction. We discuss general guidelines for the development of reusable simulation components and present component architectures for various application domains. The architectures have been designed with these guidelines in mind.

KEYWORDS

simulation components, reusable components, JavaBeans, software reusability, state space modeling

INTRODUCTION

Component-based software engineering strives for improving the software engineering process by providing reusable components. A component is a software object which has well defined interfaces, which can be customized to various needs, and which is usable in different contexts.

In the SimBeans project [Praehofer et al. 99a, 99b; Praehofer and Schoeppl 2000] we apply a component-based software engineering approach to the development of simulation systems. Libraries of reusable simulation components have been realized for various simulation application domains and on the basis of the component model JavaBeans [Sun 97]. Such components comprise simulation units (model components) as well as components for output, visualization, animation and output analysis.

Simulation modeling is based on system modeling concepts as defined in [Zeigler et al. 99]. The modular hierarchical modeling approach is seen as a foundation for building reusable components. It provides concepts for building models as modular units with a well-defined interface and for hierarchical coupling. The component model JavaBeans has been proven to provide a suitable implementation platform to realize modular hierarchical simulation models [Praehofer et al. 99a, 99b]. The systems approach provides modeling concepts for the building of reusable components. However, it does not define concretely how model interfaces and collaborations should be designed. We have made the experience that well-designed model interfaces and coupling structures defined thereupon are essential for the development of reusable component libraries.

In this paper we take the ideas from design patterns [Gamma et al. 95, Buschmann et al. 96], component frameworks [Szyperski 98], and role modeling [Reenskaug 97] to discuss component architectures for simulation modeling. First, we present general guidelines for simulation modeling. In particular, we propose separation of flow and effort components similar to Bond-graph modeling and usage of model containers as important principles. Then, we show several examples of component architectures that are designed according to these guidelines.

SYSTEM MODELING REVIEWED

Modular, hierarchical system modeling 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 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 explicit input and output interfaces in the form of method calls that they accept as well as events that they generate. Component-based programming primarily means interface-based component composition. Components can be constructed hierarchically using finer-grained components.

Our approach for building reusable simulation components is strongly influenced by the work of [Elmquist et al 98] on object-oriented, non-causal modeling of physical systems and Bond-graph modeling [Paytner 61, Cellier 91]. In these approaches models are built based on two types of variables – called *flow* and *effort* or *across* and *through* depending on the approach taken – which can be chosen arbitrarily but must multiply to *power*. Typically, a flow is a variable representing some flowing magnitude, e.g., current, while an effort typically represents a status, e.g., voltage. In such models Kirchoff's laws apply. Thus, in a node all the flows must add to 0 and the efforts must be equal. Modeling of components is done by giving the dependencies between these variables in equation (non-algorithmic) form.

The advantage of object-oriented, non-causal modeling is often seen to lie in the non-causal description of simulation systems, which allows the definition of components without having to foresee how these component will be used. In our approach for building reusable components we do not pursue a non-causal description of simulation components, but rather our goal is to realize ready-to-use model components. We adopt and generalize the way how component interfaces are specified and component coupling is done in non-causal modeling and we regard that as the key for building reusable components. According to these ideas, we propose the pursuit of general guidelines for building reusable simulation components as presented in this paper.

SIMULATION PROGRAMMING IN THE SIMBEANS SIMULATION FRAMEWORK

In this section, we will review the implementation of simulation models. First, the component model JavaBeans will briefly be discussed as the underlying implementation base, then, basic coupling concepts for simulation components will be presented.

Component Model JavaBeans

JavaBeans is the component model based on Java [Sun 97]. A JavaBean is a reusable software component that can interactively be modi-

fied and coupled with other components. JavaBeans defines a common and generally accepted standard for component analysis and handling in interactive development environments (IDE). Programming, events and properties play an essential role in JavaBeans.

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<EventType>EventListener` and `remove<EventType> EventListener`). Upon firing an event, the component calls the listeners registered for the event with the respective event method.

Properties are named attributes of a bean component that allow controlling the appearance and behavior of the component. Properties provide state information of components. They can be read and set at building (design) time as well as 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 their environment. For this purpose, class `PropertyChangeEvent` and interface `PropertyChangeListener` have been 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 component interested in changes of the property can be added and will be informed whenever the value changes.

Implementing simulation components

In [Praehofer et al. 99] and [Praehofer and Schoeppl 2000] realization of simulation components has been presented in detail. In summary, the SimBeans simulation framework allows for:

Simulation variables: Simulation variables are provided as objects to store dynamically changing model states. They extend the bound property mechanism to allow other components to listen to their value changes.

Simulation events: An event model has been realized to deal with simulation events on basis of the JavaBeans event model. This event model allows the definition and handling of time scheduled events (`TimeEvent`), input event which come from a human user in visual interactive simulations (`InputEvent`), and state events that are triggered from the continuous changing model variables (`StateEvent`).

Continuous behavior: `ContVariables` and `ContinuousEquation` components allows the definition of differential equation specified systems in similar (but more flexible) form as usual block diagram descriptions.

Variable structure models: The simulation framework allows dynamic structure changes. This means that sub-model and model elements like variables, continuous equation components, event objects, and couplings can be added and removed during a simulation run. A structure event propagation takes care that the simulator can cope with the structure changes.

In the approach of building reusable simulation components, coupling and communication of simulation components is of particular importance. Coupling and communication can occur with the following elementary means:

Object references and method calls: One component can store another component reference and can then call a method directly.

Event bindings: A component may publish an event and allow event listeners to register for the event. The component will fire the event at a particular time and the registered listener will be notified.

Bound properties: A component will publish some of its properties (usually implemented as Variables). Listeners dependent on changes of the property can be registered for this bound property and will be notified of any changes.

Variable connections: A component may publish some of its inner variables as output variables. Other components will publish some of the variables to be input variables and defined from outside. Output variables of one component can directly be connected to input variables of others.

Equations: Instead of connecting component variables directly, coupling at the variable level can also occur by means of equations. This means that an input variable of a component is defined by a functional dependency based on the values of other variables.

GUIDELINES FOR BUILDING REUSABLE COMPONENT ARCHITECTURES

General guidelines for building reusable simulation components are as follows:

Separation of concerns*:* Components should be identified with a clear separation of concern in mind. Typically, components should be distinguished for realizing (1) physical objects themselves (effort states, see below), (2) the coupling structure (flows, see below), (3) control schemes, (4) containers of physical objects modeling the environmental conditions (see below), (5) output, visualization, animation, and output analysis.

Modular interfaces: Interface definitions should clearly characterize how a component can be used and how it can communicate with other parts of the system. The component itself should make as less assumptions about its environment as possible (use model containers, see below).

Hierarchical composition: Components should be organized hierarchically. More complex components should be built by using primitive modular components and by defining how they work together by realizing the respective coupling structures.

Separation between effort and flow components: Similar to the distinction made in bond-graph modeling between 1-junctions and 0-junctions, we propose a similar distinction to be made for our model components. However, as we do not deal with classical physical modeling exclusively, we extend and generalize these ideas. In our view, a clear distinction should be made between components "storing states" and components "realizing flows". While *effort components* should be seen to provide state information and are influenced by flows, *flow components* realize the flows between components which are computed based on the state information of the connected effort components. This view applies for physical system modeling, but also for discrete item flow or pulp flow in paper mill models.

Component container: A model container should be provided where the model components "live in". Such a model container should represent the environmental conditions (physical constraints) which apply. For any component which is added to live within the container, the conditions are enforced. A model container will often work with structural changes to realize the varying relations between model components. Environmental conditions thereby may be very diverse. Examples of model containers are a 2-dimensional space for moving objects which has to enforce the rule that no two objects can occupy the same space at the same time, a gravity space which has to model that an object is accelerated into the direction of the gravity when it is not supported, or radio communication that is a container for model components which communicate over a broadcast radio communication channel.

The first two guidelines are well-known and appreciated and, therefore, will not be discussed further. In the following section we will show the application of the other guidelines, especially separation of effort and flow components and component containers, by three different component architectures.

COMPONENT ARCHITECTURES

In this section we show several component architectures for diverse modeling domains which have been designed according to the guidelines above. Component libraries have been realized on basis of these architectures in the SimBeans simulation framework. The first is in the continuous domain and has been designed for realizing training simulator for paper mill operators as presented in [Praehofer and Schoeppl 2000]. It shows how the idea of effort and flow can be generalized. The second is a general architecture for building classical discrete simulations, i.e., processing and flow of items. Here again we distinguish between effort components that can accept and deliver items and flow components that realize distribution of items. The third application domain is for simulations of 2-dimensional motions of objects. In particular, it shows the usage of a component container – the space – to care for the contained objects in motion.

Pulp flow in paper processing systems

This component architecture has been created for building paper mill training simulators. It mainly represents a continuous simulation domain. Elementary for modeling paper production is the representation of pulp, which is basically a mixture of water and fiber. This has been implemented by a structure `Pulp` with two variables `total` and `fiber` representing the total amount of pulp and the amount of fiber contained.

The architecture for this domain has been defined according to the ideas of effort and flow variables and components. While the effort variables give the information upon the type and availability as well as the capability to take pulp, the flow variables define the amount and type of pulp actually transported. Therefore, effort components typically are the tanks which store pulp and flow components are the pipes, pumps, etc. which transport pulp.

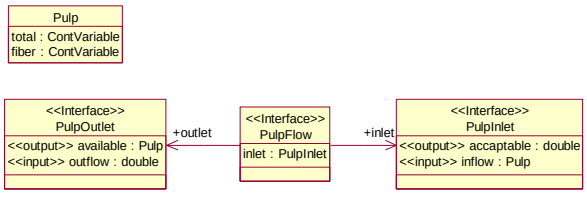


Fig. 1 Pulp outflow and inflow

Figure 1 shows the architecture by means of an UML class diagram. `PupOutlet` defines the interface of an effort component which gives the information about the availability of pulp (`<<output>> available`) and receives from the flow component the information how much pulp currently flows out (`<<input>> outflow`). On the other side, the `PulpInlet` gives information of how much pulp this effort component can currently take (`<<output>> accaptable`) and receives from the flow component the information of how much and what type of pulp currently flows in (`<<input>> outflow`). The task of the flow component (`PulpFlow`) is to compute the current flow based on the `available` and `accaptable` information by its coupled `PulpOutlet` and `PulpInlet` (which can be of arbitrary complexity, in particular, it can employ arbitrary complex control schemes to determine the flow.)

Discrete item processing

The *discrete item* component library is a general simulation package for processing and distribution of discrete items, as classical in manufacturing, job shop, transportation and others. In this domain the following principal types of elements have been identified: (1) items, which flow through the system from resource to resource and occupy them; (2) resources, which are active or passive and can be occupied by items; (3) transportation systems, which implement the item flow; (4) 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, effort components are the resources that can receive items, hold items, process items and provide items, effort states of resources signaling whether items are needed and items can be provided are the efforts, and transportation systems and/or realization of coupling structures together with decisions on item distributions represent the flow components.

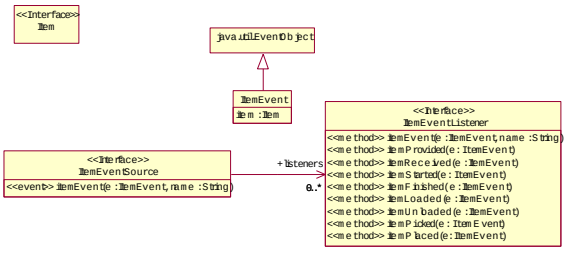


Fig. 2 `ItemEvents`

Core interface and class definitions

The following interfaces and classes are fundamental for the discrete process component library.

`Item`: The `Item` interface is a general interface for 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.

`ItemEvent`: 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. Figure 2 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.

Effort components

Interfaces for effort components are `Receiver` and `Provider`. They define means for observing the effort states of the components and for accessing the flows, i.e., the 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).

`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).

Flow components

Item flow is accomplished by coupling and item transportation components. They realize the connection between `Provider` and `Receiver`s and for that task rely on the `Provider` and `Receiver` interfaces. Item flow can be of any complexity, ranging from elementary direct connections between providers and receivers to complex transportation systems with complex control strategies.

Figure 3 illustrates realization of `Provider` – `Receiver` couplings. The item flow component gets informed of available items by the changes of the `hasItem` variable of the `Providers` and of available space by the `needsItem` variable of the `Receivers`. It can inspect an available item by calling `inspectItem` and test a `Receiver` if it will accept an item by calling `testItem`. Based on this information it has to decide when and what kind of item flow should be established. It accesses an item by calling `retrieveItem` from a selected `Provider` and forwards the item to a selected `Receiver` by calling `putItem`. Additionally it can react to item event `itemProvided` from a `Provider` and `itemReceived` from a `Receiver`.

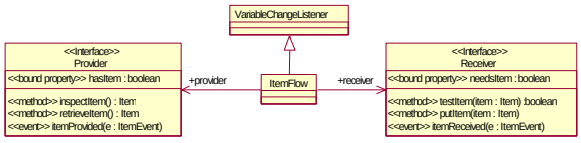


Fig. 3 Item flow components

The simplest item flow components are direct linear connections between `Provider` and `Receiver` components where item flow occurs instantaneously without any delay. With such a direct item flow, coupled systems can be built that are typical of *Flow Shop* models. Depending on whether a 1:1, a 1:*n*, oran *n*:1 connection is desired, different item flow components have to be used.

A `ProviderReceiverConnection` realizes a direct unconditional connection between an provider and a 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 `Receiver` by calling `putItem`. No control decision is needed here.

The `ReceiverDecisionPoint` 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 `ReceiverSelection`, which is a strategy component 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 `ProviderDecisionPoint` couples a set of providers with a receiver.

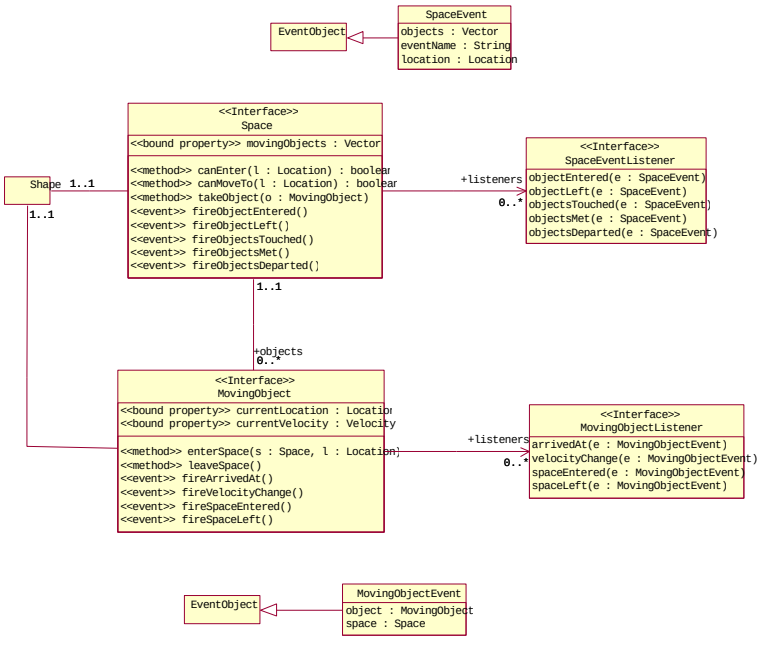


Fig. 4 Moving objects in space component architecture

Moving objects in space

The `movingObject` component architecture serves as a basis for modeling and simulating of objects moving in a 2-dimensional space (movements in a 1-dimensional or 3-dimensional space would be built in an analogous way). This domain demonstrates the usage of the concept of a model container. Actually, a `Space` component represents a 2-dimensional (bounded) space where the objects move around. A Space has several tasks. First, it is responsible for handling the physical con-

straints which apply in a 2-dimensional space, e.g., that two solid objects cannot occupy the same area at the same time. This is done by an interaction protocol between the moving objects and the space component itself. Second, it is responsible for establishing the interactions between various objects which live in the space. Therefore, the space supervises the moving objects and reacts whenever such interactions occur. Third, several spaces might be arranged hierarchically to form a bigger space and the space has the responsibility to forward an object which leaves itself. To accomplish this, spaces can be arranged hierarchically, where the superior space realizes the coupling of its contained spaces.

Based on the desired model precision, space components and their living objects might interact in various forms. A space may represent in detail the terrain of the space and the moving object will then model how it can move based on the terrain of its current location. On the other side, the space might be represented on a more abstract level, e.g., only as a flat area with obstacles. In the following we do not discuss how movements in complex terrain can occur but only represent a basic event protocol between space and moving objects which is intended to serve as a foundation for more detailed models.

To exemplify the usage of this general moving object architecture we briefly discuss two quite diverse applications, moving particle simulation and mobile, cooperating robots.

MovingObjects Component Architecture

The architecture for building simulations of moving objects in space is depicted in Figure 4. It consists of interface specifications `Space`, `MovingObject`, `SpaceEventListener`, `MovingObjectEventListener` and class `MovingObjectEvent`. The MovingObject has properties currentLocation and currentVelocity. It fires events when it arrives at particular locations, when it changes its velocity, and when it enters and leaves a space. It has methods to enter a space and leave a space. The MovingObjectEventListener specifies methods to react to the events.
In similar form, the space fires events when interesting things happen with its components. For example, it will fire an event when an object enters the space, when it leaves the space, when two objects meet within a certain vicinity (see below) or depart, and when objects collide. This event protocol is open so that new events can easily be introduced. The SpaceEventListener defines a interface to react to those events. Note, that this architecture is very general and should serve as a basic pattern to derive special applications

Vicinity detection

A space model which detects whether objects are within a certain distance to each other is immediately derived from the general architecture. It has many applications, e.g., in the many particle system and the moving robots examples below. The `VicinityDetectionSpace` implements `MovingObjectListener` and reacts to `arriveAt` events from its moving objects. Whenever, it detects that two objects come close, it signals itself a `objectsMet` event to registered `SpaceEventListeners` (Fig. 6)

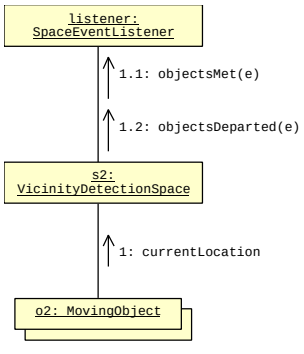


Fig. 5 Collaboration for detection of vicinity of two objects

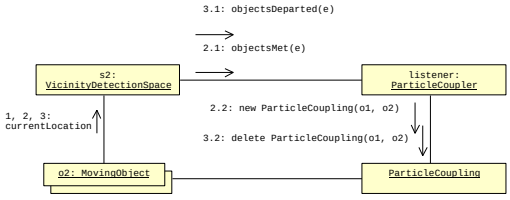


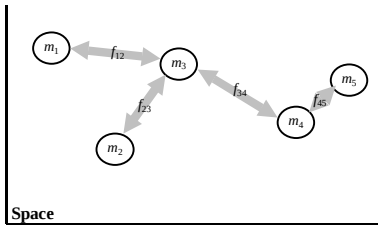
Fig. 6 Moving, rejecting particles in a space

Moving, rejecting particles

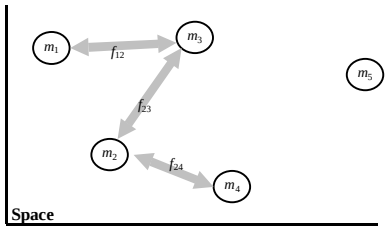
As an application example, modeling of a space with moving and rejecting particles is discussed subsequently. It relies on the moving objects architecture and shows structural changes. A simulation of many moving and rejecting particles is not feasible when all the rejective forces are considered all the time. A more practicable way is to consider the rejective force between two particles only when they are within a certain distance and otherwise ignore them. Such an approach, however, requires that rejective forces are established and resolved dynamically. This can be accomplished by the moving object component architecture with the space component observing the distance between objects and establishing *force coupling components* when they come close.

A `VicinityDetectionSpace` as discussed above is used to check that objects come close or depart again. The `objetsMet` and `objectsDeparted` events are fired by the `VicinityDetectionSpace`. A `ParticleCoupler` (Fig. 7) implements `SpaceEventListener` and establishes and resolves the force couplings.

A force coupling component receives the current position (*x*, *y*) of its two connected particles. Based on that the rejection forces which are input to the particles (with opposite sign) are computed. This is a classical effort and flow architecture with positions being the effort variables, forces being the flow variables, particles being the effort components, and force couplings being the flow components.



(a) no force coupling is estblished betwenn m_4 and m_2



(b) through movements, however, m_4 comes close to m_2, but departs from m_3 and m_5: couplings are resolved to m_3 and m_5 but established with m_2

Fig. 7 Moving, rejecting particles in a space

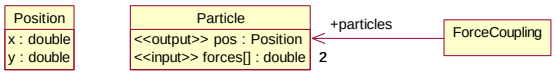


Fig. 8 Force coupling component

Mobile communicating agents

In a similar way a simulation of a family of mobile, communicating robots can be set up (Fig. 9). The space has to check if robots do not collide with obstacles and with other robots. The MobileAgentSpace extends VicinityDetectionSpace and is used to detect if robots collide or come into the radio communication area of other robots and communication channels should be established. Figure 10 shows the collaboration for collision detection and communication channel creation.

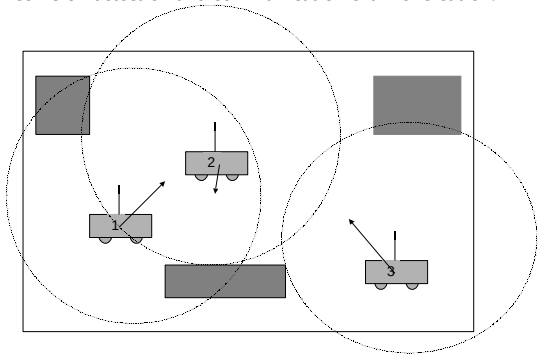


Fig. 9 Moving agents

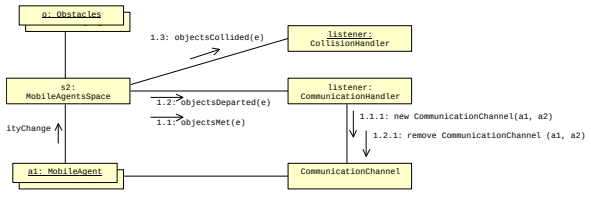


Fig. 10 Mobile agents

SUMMARY AND OUTLOOK

We have discussed guidelines for realizing reusable simulation components. It has been outlined that the design of the component interfaces, the separation of component and coupling structures, and the hierarchical composition and usage of model containers are of principal importance. The applications of the guidelines have been demonstrated by presenting component architectures for various problem domains.

The modeling principles as well as the general architecture are supposed to be an important step towards an component-based modeling and simulation methodology. Our recent work towards this goal follows two main lines: (1) creation of component libraries for different application domains based on the ideas presented, (2) building a interactive development environement (IDE) to support component based modeling and simulation. We briefly discuss the second point in the following.

We found that current Bean builder tools, like Inprise JBuilder, Symantec Visual Cafe or IBM VisualAge, altogether show great weaknesses and still do not really support interactive component assembly. We have identified the following major shortcomings of current component builder: (1) they are not able to deal with hierarchically structured components; (2) they only support a low-level form of coupling (3) they do not allow to influence code generation; (4) they usually work with source code generation and not with components directly.

For that reasons, a new JavaBeans IDE will have to be realized. The tool will differ mainly from others as it will be configurable to particular application domains. It will heavily rely on knowledge on the meta level to guide the user in component assembly and programming. The tool will use and will be configurable with the following knowledge representation schemes on the meta level:

An object oriented model representation in UML notation: The object-oriented model has to purpose to represent the hierarchical composition, component variants, and model aspects in an application domain. In that it resembles the System Entity Structure knowledge representation scheme [Zeigler 84].

Constraints: Additionally to the object model, constraints can be used to represent allowed subranges for attributes, allowed subsets of components, compatibility constraints between components [Rozenblit, Zeigler 88], etc. in an application domain.

Representation of coupling patterns: Ways how components can collaborate can be specified in the form of role models [Sametinger, Keller 2000]. A role model represents a design pattern in that it specifies various participating roles and various forms of collaborations between those roles for the fulfillment of a particular task. In our context, we use role models to define higher, application-specific forms of couplings schemes.

Template components, methods, and scripts: Usually, source code programming cannot be avoided in developing complex simulation programs. It is our objective to minimize programming on the source code level. Template components are provided to serve very specific purposes, e.g., for control of a machine, but can be programmed by the user. A template component implements all the general code and helps the user in implementing the specific functionality. Therefore, template components may support higher-level forms of model specification, like rules, tabular forms, etc., from which source code is generated finally.

REFERENCES

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

Breunese, A.P.J., Top, J.L., Borenink, J.F., Akkermans, J.M., Libraries of Reusable Models: Theory and Application, Simulation 71 (1), 1998, pp. 7-22.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. Pattern-Oriented Software Architecture. Wiley & Sons, 1996

H. Elmquist et al. 1998. ModelicaTM - A Unified Object-Oriented Language for Physical Systems Models, Version 1.1, http://www.modelica.org/documents.shtml

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

Praehofer, H., Sametinger, J., Stritzinger, A.1999a. International Conference on Web-Based Simulation, SF, CA., Jan 1999.

Praehofer, H., Sametinger, J., Stritzinger, A. 1999b. Concepts and Architecture of Simulation Framework Based on the JavaBeans Component Model , Journal of Future Generation Computing Systems, Special Issue on WebbBsed simulation, 1999, (accepted)

Praehofer, H, Schoeppl, A. 200. A Continuous and Combined Simulation platform in Java and its Application in Building Paper Mill Training simulators (this volume).

Reenskaug T, Lehne O A, Working With Objects-Designing Distributed Systems for Reuse; OOPSLA 97 Tutorial

Rozenblit, J.W. and B.P. Zeigler, Design and Modeling Concepts, In International Encyclopedia of Robotics. (Ed. R. Dorf), 308-322, John Wiley and Sons, New York, 1988

Sametinger, J, Keller R. Design Components - Compositional Reuse of Design Expertise, 2000 (to appear)

Sun Microsystems. 1997. *JavaBeans 1.01 API Specification*. Sun Microsystems, Inc., 1997.

Szyperski, C., *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

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

Zeigler, B.P., H. Praehofer, and T.G. Kim, Theory of Modeling and Simulation, 2nd Edition. Academic Press, 1999 (to appear)