On the Specification of Components - the JavaBeans Example

Maritta Heisel, Thomas Santen, Jeanine Souquières

To cite this version:

Maritta Heisel, Thomas Santen, Jeanine Souquières. On the Specification of Components - the JavaBeans Example. [Intern report] A02-R-025 || heisel02a, 2002, 20 p. inria-00107634

HAL Id: inria-00107634

https://inria.hal.science/inria-00107634v1

Submitted on 19 Oct 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Specification of Components – the JavaBeans Example

Maritta Heisel[1], Thomas Santen[2], and Jeanine Souquières[3]

[1] Institut für Praktische Informatik und Medieninformatik
Technische Universität Ilmenau
Max-Planck-Ring 14, 98693 Ilmenau, Germany
email: maritta.heisel@prakinf.tu-ilmenau.de
[2] Institut für Kommunikations- und Softwaretechnik
Technische Universität Berlin
FR 5-6, Franklinstraße 28/29, D-10587 Berlin, Germany
email: santen@acm.org
[3] LORIA—Université Nancy2
B.P. 239 Bâtiment LORIA
F-54506 Vandœuvre-les-Nancy, France
email: souquier@loria.fr

Abstract. We specify the JavaBean component model and concrete beans using a combination of UML class diagrams, an extension of Object-Z, and life sequence charts. We extend Object-Z by keywords that allow one to concisely describe the interface of a bean by an Object-Z class specification. The component model specification provides specification templates consisting of class diagrams, Object-Z fragments, and life sequence charts that precisely capture the functional behavior of beans in general, including the interaction of beans that cooperate in a system. The new keywords used for specifying concrete beans translate to instances of the component model specification templates, showing that our extension of Object-Z is syntactical sugar only.

1 Introduction

Component-based software engineering [19] is an emerging field of great interest in research and practice. Its goal is to develop software systems not from scratch but by assembling pre-fabricated parts, as is done in other engineering disciplines. These pre-fabricated parts are called components.

Components are independently deployable pieces of software. To use a component, it is desirable to have a description of its properties that abstracts from implementation details but provides sufficient information about the component to decide if it is useful for the purpose at hand and how to combine it with other components without needing to inspect the code.

In this paper, we investigate how such a description of a component might look. Several component models, such as *JavaBeans* [16], *Enterprise Java Beans* [17], *Microsoft COM* [11], and *CORBA* [12] have been proposed. A component model is designed to allow components that are implemented according to the standards set by the model to interoperate.

A specification describing a component ideally consists of two parts: first, a general specification characterizing a component according to a given component model *CM*, i.e. the *component model specification*, and second a concretization describing the specifics of a given component implemented according to the model *CM*, i.e. the *component specification*. The latter should, of course, not rephrase the general specification of *CM* but only refer to it.

For this paper, we concentrate on the functional aspects of components, and we consider a specific kind of component, namely JavaBeans. In Section 2, we briefly introduce the JavaBeans component model. Section 3 shows how we specify concrete beans using an extension of Object-Z [14] by JavaBeans-specific keywords. The semantics of those keywords becomes clear when we specify the JavaBean component model in Section 4. Much of that model consists of *templates*. A JavaBean-specific keyphrase in a bean specification translates to an Object-Z specification fragment and a number of life sequence charts [6] according to the templates described in the component model specification. Section 5 relates the present work to other research on specifying software components in general and JavaBeans in particular. In Section 6, we discuss the approach to specifying components and point out directions of future research.

2 JavaBeans

JavaBeans is a component model originally introduced by Sun in 1996. JavaBeans has an event-based communication model between components, called *beans*: a bean notifies registered listener beans about events its generates, and it registers with other beans to be notified about their events. As we will see in the next section, cooperating beans thus realize three variants of the *observer* design pattern [8].

More specifically, the main aspects of the bean model are [19]:

- A bean can generate and receive arbitrary *events*.
- A bean has a number of *properties*, which are manipulated with specific setter and getter operations.
- Changing a property may generate an event. For a *bound property*, a bean generates a change event whenever the value of that property changes. For a *constrained property*, the bean generates a change event like for a bound property. Additionally, the listeners to that event may *veto* the change, causing the bean to revert the value of that property to the one before the change.
- In addition to the operations implementing the event-based communication between beans, a bean may provide an arbitrary number of ordinary operations in its interface.
- An assembly tool can *introspect* beans and obtain information about their interfaces (events, properties).
- Setting a bean's properties, an assembly tool can *customize* a bean.
- Customized and connected beans are stored *persistently*.

Figure 1 illustrates the interface of two beans and their connection. For the events, an interface is divided into two parts: one to receive events, and one to generate them. When connecting beans, the generating side of an interface can be connected to several

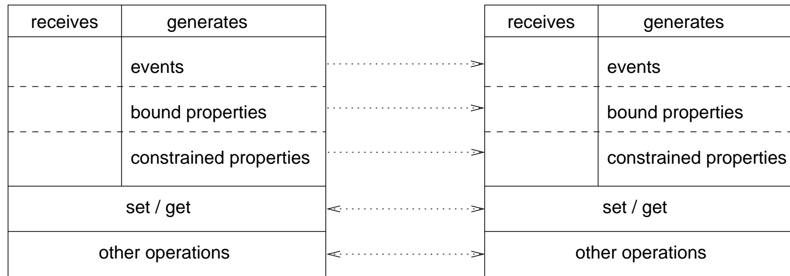


Fig. 1. The Java Bean Interface

receiving sides of other beans that will be notified of events. The set and get operations, and other operations provided by a bean can be called by other beans in an arbitrary fashion.

In the rest of this paper, we specify the functional aspects of the JavaBeans component model, considering arbitrary events, properties, and bound and constrained properties.

3 Specification of a JavaBean

We use the following notations to specify JavaBeans:

- UML class diagrams [13].
  They give an overview of the involved classes, their attributes and operations, and the associations between them.
- Object-Z [14] enriched by some new keywords.
  Object-Z serves to specify the details of a particular bean. We introduce new keywords to obtain concise specifications and to enhance readability. Mapping those new phrases to Object-Z specification fragments, we define the semantics of the keywords.
- Life sequence charts [6].
  As is well known, Z-like languages are not well suited to express certain dynamic properties of the specified systems. To specify the protocols of interaction between beans, we use a slight extension of life sequence charts. Life sequence charts (LSCs) not only have a formal semantics [10], but they also are more expressive than message sequence charts [9]. For example, it is possible to distinguish mandatory and optional behavior.

A bean called *JavaBean* is specified as shown by the following schematic class specification in (extended) Object-Z.

$__$ *JavaBean* $__$

$\upharpoonright$ (...exported operations...)

generates event x of *XEvent*

receives event y of *YEvent*

receives property change of s

receives vetoable change of t

attr : ...   – – private attribute

properties

p : *PType*

bound properties

q : *QType*

constrained properties

r : *RType*

invariant

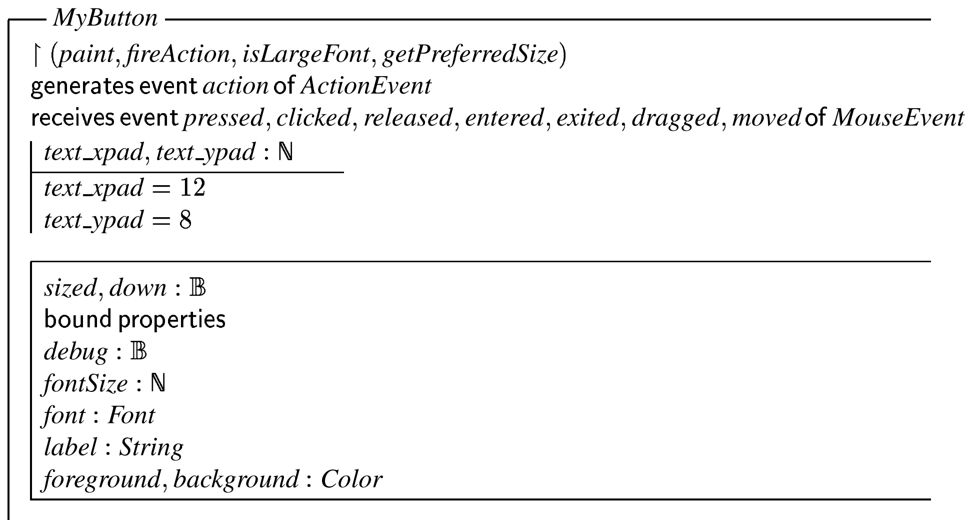
The export list of the class specification contains all the exported operations of the bean. These include the operations to receive events, setter and getter operations of properties, and other operations the bean supplies to its environment.

The following phrases are extensions of Object-Z indicating which events *JavaBean* generates that are not related to property changes, which ones it receives, and which property change events of bound or constrained properties it can receive from other beans.

We partition the attributes of *JavaBean*, which are declared in the state schema of the class, into four variants: the declarations in the state schema preceeding the keyword properties are the private attributes of the bean. We do not admit public attributes that are not properties, because public attributes without setter and getter operations are difficult to handle uniformly when combining components, and because properties serve the same purpose as public attributes do.

The attributes following the keyword properties are the properties, i.e. the publicly accessible attributes of the bean. Qualifying an attribute as a property means providing setter and getter operations for that attribute. Properties are further subdivided into bound (keyword bound properties) and constrained properties (keyword constrained properties). For bound and constrained properties, the bean must provide operations to register listeners and to notify listeners of changes of the properties' values.

The following Object-Z class *MyButton* is the specification of a simple bean that implements a text button for a graphical user interface [18]. The button has a text field of a certain size which displays the *label* of the button. Using the mouse pointer, which is implemented as another bean, the button can receive a number of events signaling the status of the mouse pointer and the mouse button. An instance of *MyButton* generates an event called *action* if the mouse button is pressed while the mouse pointer is on the button.



The specification of *MyButton* declares the generated and received events *action*, *pressed*, *clicked*, etc. The classes *ActionEvent* and *MouseEvent* show what information those events carry: an *action* event identifies the source of the event, which is an instance of *MyButton*. A *MouseEvent* carries the coordinates of the mouse pointer.

The text field of a button has a fixed size, which is determined by the two constants *text_xpad* and *text_ypad*. The two private attributes *sized* and *down* are used internally in *MyButton*. All properties of *MyButton* are bound. Therefore, whenever a property, e.g. *label*, changes its value, the button notifies all beans that have registered to property changes of the button.

4 Specification of the JavaBean Component Model

In the previous section, we saw how the specification of a concrete bean such as *MyButton* looks in an extension of Object-Z. The present section gives a semantics to the new keywords used there to give a concise description of the properties of a bean and the events generated and received by a bean. We define that semantics partly by specifying the JavaBean component model in general. Much of the component model, however, cannot be captured by a general specification. The specification formalisms we use can only prototypically describe the behavior restrictions associated with events, for example, for a given event. Therefore, much of the following specifications are specification

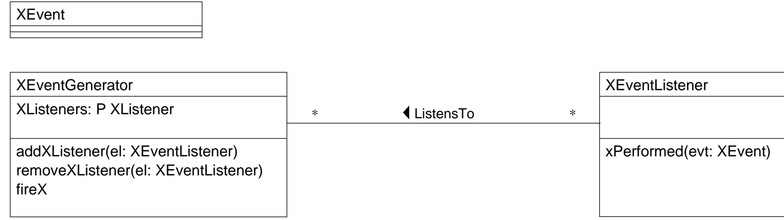


Fig. 2. Event Class Diagram

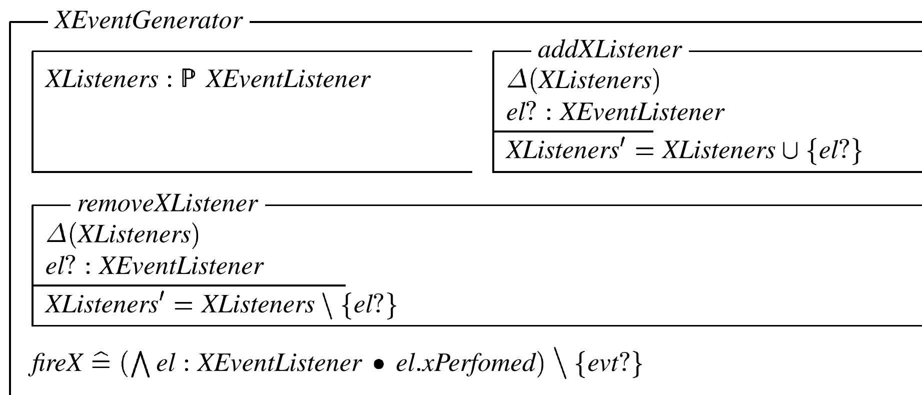
templates rather than concrete specifications. Those templates show how phrases such as "generates event *action* of *ActionEvent*" can be translated to class diagrams, LSCs, and Object-Z specification fragments. In the following sections, we show how each of the bean-specific declarations we introduced in the preceeding section translates to a specification fragment.

4.1 Events

The event mechanism of the JavaBean component model is an instance of the *observer* design pattern [8].

For a generated event x of class *XEvent* (declared by the phrase generates event x of *XEvent*), a bean takes the role of an *XEventGenerator* as described in Fig. 2. Hence, it has a (private) attribute *XListeners*, which contains the set of listener beans that have registered to the event x. The methods *addXListener* and *removeXListener* accomplish registering and un-registering listeners to that event. Usually, the bean generates the event x internally. The operation *fireX* additionally allows other beans to request that an event x be generated.

The following specification fragment shows how the class *XEventGenerator* in Fig. 2 translates to Object-Z.



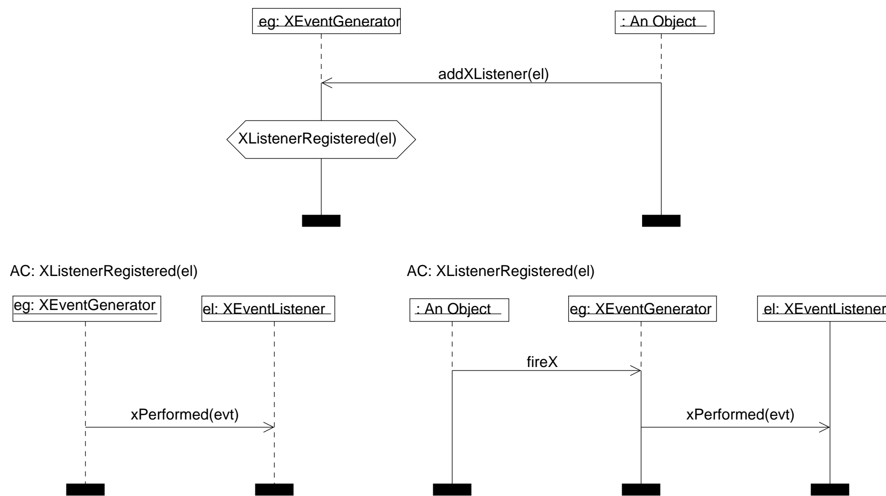
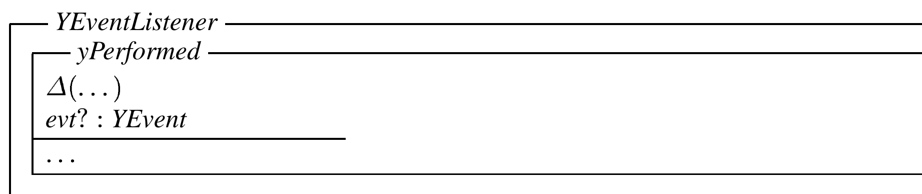


Fig. 3. Registering an Event Listener and Processing an Event

The schemas *addXListener* and *removeXListener* are complete (and hence can be generated automatically), whereas further detail can be added to the definition of *fireX* in an refinement, e.g. the exact definition of the event *evt* and possible modifications of the bean's state.

On the listener side of Fig. 2, each *XEventListener* must supply an operation *xPerformed*. The *XEventGenerator* calls *xPerformed* to signal the event x to a listener (c.f. the specification of *fireX*). Hence, each phrase "receives event y of *YEvent*" corresponds to the following specification fragment. The details of how to react to an event y are specific to the concrete listener bean and cannot be prescribed by the component model.



The LSCs shown in Fig. 3 describe the interaction between event generators and listeners. Any object can register a listener *el* with a generator *eg* by calling $eg.addXListener(el)$. Only registered listeners will be notified about an event x as the two LSCs at the bottom of the figure show: both have an activation condition $AC : XListenerRegistered(el)$, which is established by the LSC at the top of the figure.

The LSC at the left-hand side of the figure shows the usual case that the generator *eg* (internally) decides that an event x occurs and, calling *xPerformed*, notifies the listener

el about that event. At the right-hand side, the figure shows the case where a different object requests *eg* to generate *x*, which, as a consequence, notifies the listener *el*.

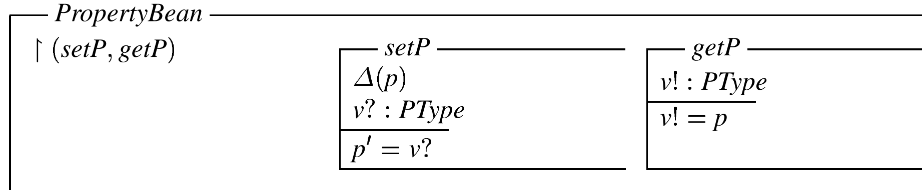
In LCSs, dashed *instance axes* (or *lifelines*) mean that the corresponding behavior is optional whereas solid instance axes indicate that the corresponding behavior is mandatory. For example, in the LSC at the top of Fig. 3, it is not required that the operation *addXListener* ever be called. If it is called, however, then it must establish the condition $XListenerRegistered(el)$.

Note that neither the Object-Z nor the LSC specification show that *all* registered listeners are notified when *eg* generates an event internally. The predicate of *fireX* in Object-Z requires this behavior in case of an external triggering of the event. Because any operation of *XEventGenerator* could trigger *x* internally, there is no way in Object-Z of (explicitly) requiring that behavior for internally generated events.

4.2 Properties

For simple properties that are not bound or constrained properties, the JavaBean component model just requires setter and getter operations which provide a standard interface to change and access the value of a public attribute. Therefore, each property $p : PType$ listed in the state schema of a bean under properties induces specifications of the operations *getP* and *setP*. As was illustrated in Section 3, these operations are not explicitly shown in the specification of a concrete bean.

Setters and getters for attributes that are properties are implicitly included in the export list – the attributes themselves are *not* part of the export list, because they are accessed by the setter and getter operations only.



Since no complex behavior or interaction between beans is required for simple properties, there are no corresponding class and sequence diagrams in our component model specification.

4.3 Bound Properties

As for general events, a bean generating change events for its bound properties and the beans receiving those events communicate according to an instance of the observer pattern. The class diagram in Fig. 4 shows the situation. Because changes of all properties of a bean can be handled by one instance of the observer pattern, we have more information in the diagram in Fig. 4 than in the one for events in Fig. 2. In the following, we first describe the listener side of the pattern, then the generator side, and finally the interaction between the two.

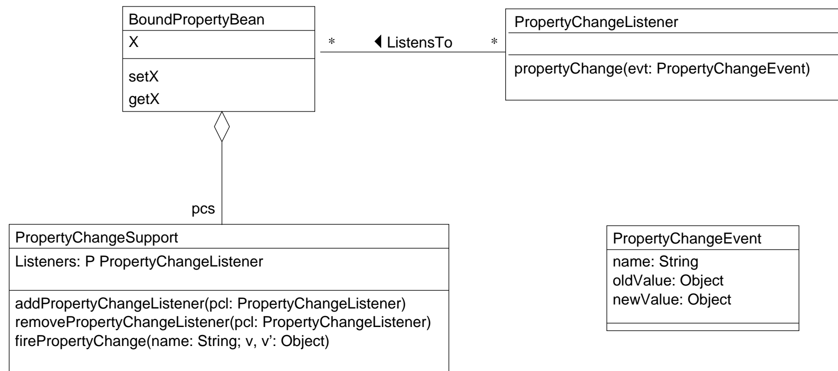
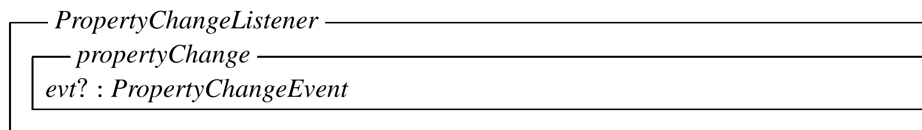
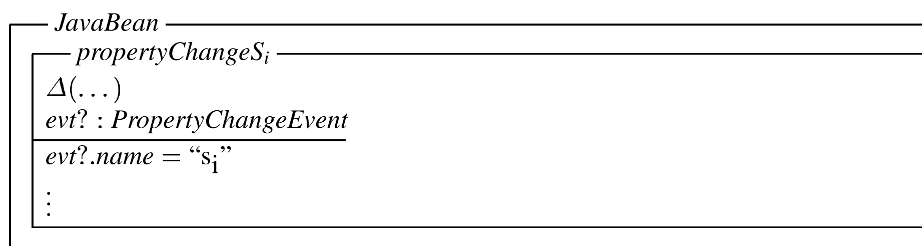


Fig. 4. Property Class Diagram

Property Change Listeners A *PropertyChangeEvent* contains the name of the changed property, its old and its new value (c.f. Fig. 4). A listener to property changes inherits from the class *PropertyChangeListener*.



The operation *propertyChange* is an abstract operation that cannot be specifi ed further in class *PropertyChangeListener*. Descendants of *PropertyChangeListener* refi ne *propertyChange*. The idea for this refi nement is that *propertyChange* must realize a case distinction over the (names of) bound properties the listener bean is interested in. Therefore, each phrase "receives property change of s_i" in the specifi cation of the listener bean corresponds to the following specifi cation fragment:



The ellipses in *propertyChangeS* specify the effect of receiving a property change event for the property s.

Now let s_1 through s_n be all properties mentioned in phrases "receives property change of s_i" in the listener bean. The operation *propertyChange* chooses between the property change

operations for those phrases. That choice is deterministic because the precondition of $propertyChangeS_i$ implies $evt?.name =$ "s$_i$", and the names "s$_i$" are pairwise distinct.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{JavaBean}} \\
\upharpoonright (propertyChange) \\
PropertyChangeListener \\
propertyChange \mathrel{\widehat=} propertyChangeS_1 \mathbin{[]} \ldots \mathbin{[]} propertyChangeS_n
\end{array}
$$

Property Change Generators The class *PropertyChangeSupport* as provided by the `java.beans` package specifies much of the behavior of the generator side of property change events. A bean with bound properties aggregates an instance of that class, c.f. Fig. 4.

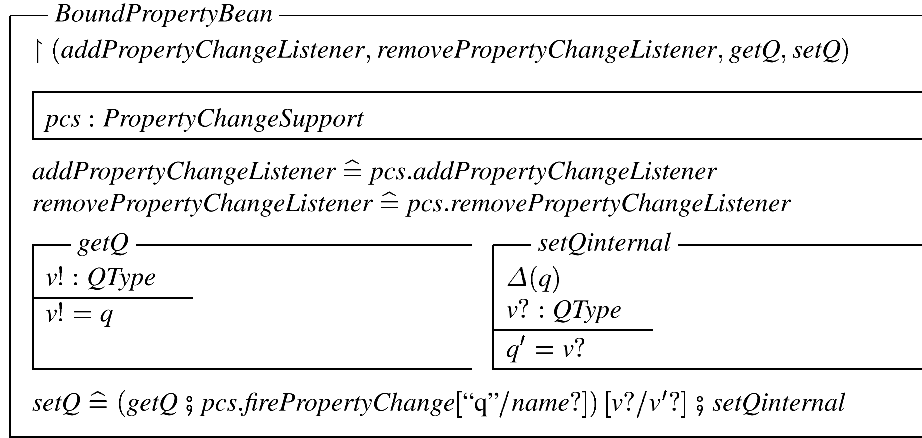
The state of a *PropertyChangeSupport* consists of the set of registered listeners for property changes, *PCListeners*. As for events, there are operations to register and unregister property change listeners.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{PropertyChangeSupport}} \\
\upharpoonright (addPropertyChangeListener, removePropertyChangeListener, firePropertyChange) \\
\quad PCListeners : \mathbb{P}\ PropertyChangeListener \\
\quad \textit{addPropertyChangeListener} \\
\quad \Delta(PCListeners) \\
\quad pcl? : PropertyChangeListener \\
\quad PCListeners' = PCListeners \cup \{pcl?\} \\
\quad \textit{removePropertyChangeListener} \\
\quad \Delta(PCListeners) \\
\quad pcl? : PropertyChangeListener \\
\quad PCListeners' = PCListeners \setminus \{pcl?\} \\
\quad \textit{mkPCE} \\
\quad name? : String \\
\quad v?, v'? : Object \\
\quad evt! : PropertyChangeEvent \\
\quad evt!.name = name? \\
\quad evt!.oldValue = v? \\
\quad evt!.newValue = v'? \\
firePropertyChange \mathrel{\widehat=} mkPCE \mathbin{⨟} (\bigwedge pcl : PCListeners \bullet pcl.propertyChange)
\end{array}
$$

The operation *firePropertyChange* is responsible for notifying all registered listeners of a property change event $evt?$, which is a parameter of the operation *propertyChange* of a *PropertyChangeListener*. We specify that effect by a conjunction of invocations of *propertyChange* on all members of *PCListeners*. The (private) operation *mkPCE* assembles the parameters $name?$, $v?$, and $v'?$ to make up the property change event. This is a way of constructing complex parameters in Object-Z.

A concrete bean *BoundPropertyBean* with bound properties aggregates an instance *pcs* of *PropertyChangeSupport*. It promotes the operations of *pcs* for registering listeners to its own interface.

Additionally, each bound property $q : QType$ listed in the state schema of *BoundPropertyBean* following the keyword bound properties induces two operation specifications *getQ* and *setQ*: the former is similar to the setter operation for simple properties; at the end of its execution, the latter must additionally notify all registered listeners about the change of q it accomplishes.



Technically, the invocation of *getQ* in the definition of *setQ* provides the old value of q as a parameter to *firePropertyChange* (note the parentheses). The following renaming of $v'?$ to $v?$ identifies that parameter of *firePropertyChange* with the input parameter $v?$ of *setQinternal* – and thus with the input parameter of *setQ*.

Dynamics of Property Changes Figures 5 and 6 show the set-up of property change listeners in a bound property bean *bpb*. Upon its creation, expressed by the activation condition *initialstate*, the bean must create an instance *pcs* of *PropertyChangeSupport*. After the bean has created *pcs* (activation condition *PcsCreated*(*bpb*)), it is ready to register property change listeners.

Once a listener *pcl* has been registered, it will receive appropriate property change events whenever the operation *setQ* for a bound property q is invoked. Figure 7 clarifies that *setQ* calls *pcs.firePropertyChange* at the *end* of its execution. This cannot be specified in Object-Z but is captured in the the sequence chart in Fig. 7, which is an extension of the LSC notation showing the durations of operation executions. In the chart, however, it is not obvious that the listener *pcl* prototypically stands for all members of *bpb.pcs.PCListeners*.

4.4 Constrained Properties

As for bound properties, listeners to constrained property changes are notified whenever a constrained property r changes its value (because *setR* is invoked). Therefore, the

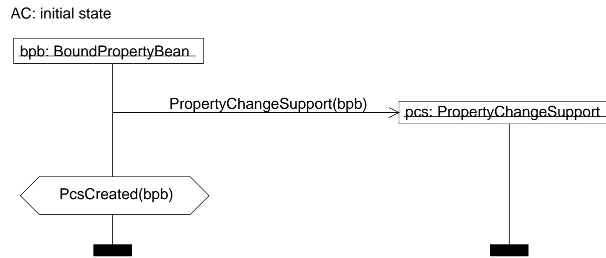


Fig. 5. Creating a PropertyChangeSupport

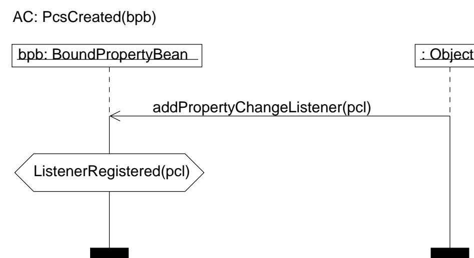


Fig. 6. Registering a Property Change Listener

class diagram for constrained properties in Fig. 8 is very similar to the one for bound properties in Fig. 4.

Unlike for bound properties, a listener may *veto* a change to a constrained property, raising an exception of type *PropertyVetoException*. This difference between bound and constrained properties becomes most obvious in the LSCs describing the interaction between a *ConstrainedPropertyBean* and a *VetoableChangeListener*. Therefore, we describe the LSCs for constrained properties before explaining their Object-Z speci fi cation templates.

Figure 9 describes the interaction in the case that *no* listener vetoes the change of *r*, i.e. no property veto exception is "thrown". The expression "forbidden msgs: throw(PropertyVetoException)" states this condition on the sequence of messages in Fig. 9. The possibility to state such negative conditions is one more means of expression where life sequence charts extend message sequence charts.

In the case that no veto occurs, the interaction between *cpb* and its listeners is much the same as the one for bound properties in Fig. 7. A minor difference is, however, that the invocation of *fireVetoableChange* need not occur as the fi nal action of *setR* but can take place at any point of the execution of *setR*.

Figure 10 shows the more complex behavior in the case that one of the listeners vetoes the change of a constrained property *r*. In that situation, we need to consider two prototypical listeners, *vcl* and *ovcl*. An invocation of *setR* causes a call to *fireVetoableChange* with appropriate parameters. All listeners, i.e. *vcl* and *ovcl* are no-

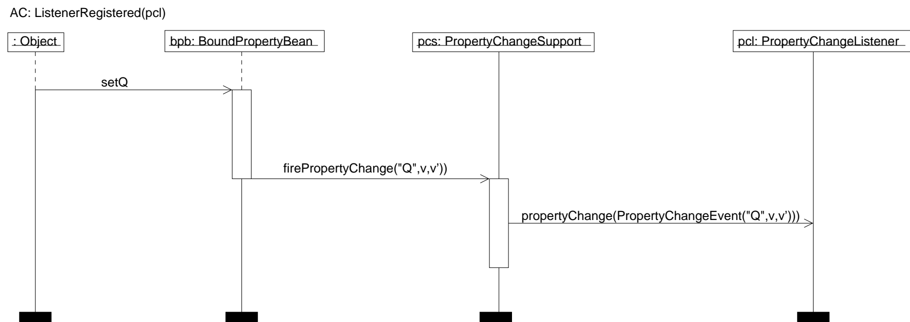


Fig. 7. Processing a Property Change

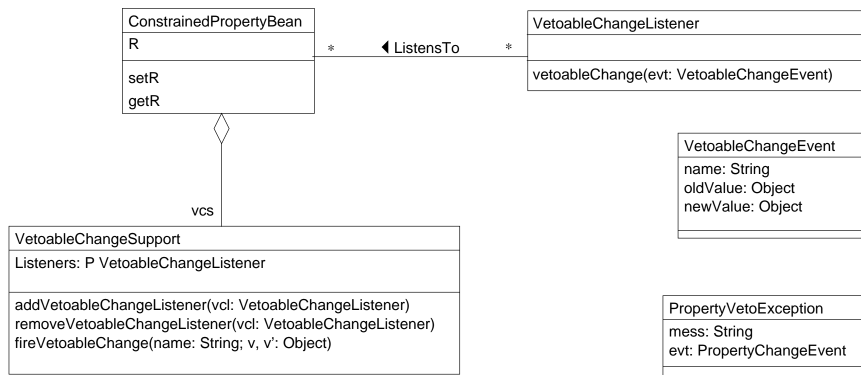


Fig. 8. Constrained Property Class Diagram

tifi ed of the proposed change of r's value from v to v'. Vetoing the change, *vcl* throws a *PropertyVetoException*. As a consequence, all listeners must be notifi ed of the veto. A way to do so [7] is to notify all listeners of a change of r's value *back* from v' to v. This may also be the only way to notify listeners of a veto, because the JavaBean component model does not enforce explicit confi rmations of (vetoable) changes. Therefore, a listener must assume that a change is not vetoed unless it receives a *vetoableChange* message reverting the value of a property back to its previous one.

Here, an unresolved issue of the JavaBean component model becomes obvious: what happens if one of the listeners vetoes the *vetoableChange* back from v to v'? In the following Object-Z specifi cation, we forbid that kind of behavior.

Vetoable Change Listeners Modeling constrained properties in Object-Z, we are faced with the problem of how to model exceptions in that language. In Object-Z, it is impossible to model control flow, and therefore, exceptions are not a language feature. For our purposes, it suffi ces to model the fact that a veto to a change event occurs. We do so by

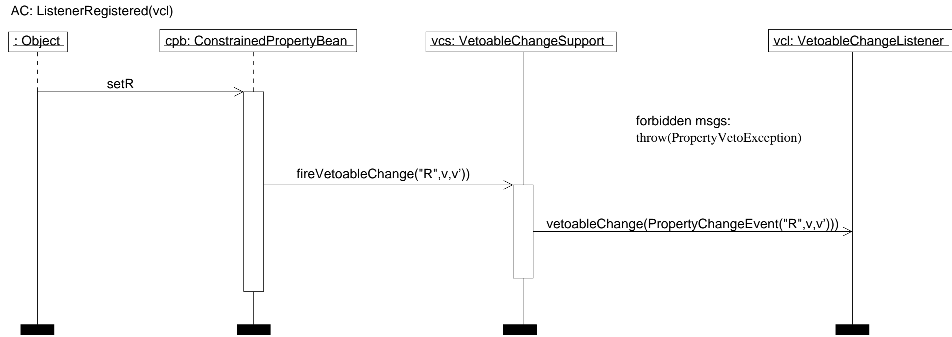


Fig. 9. Processing a Non-Vetoed Property Change

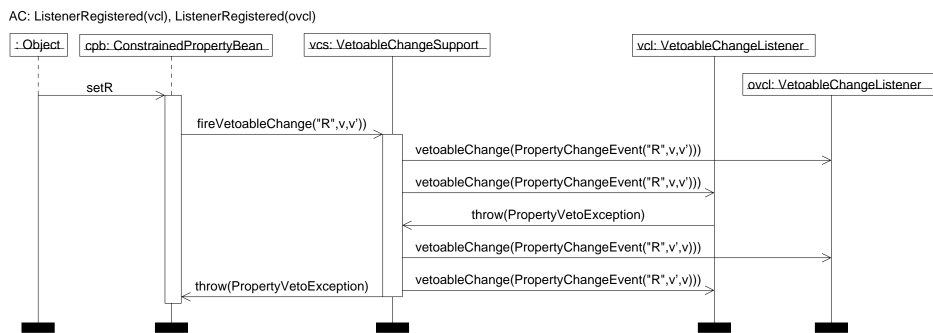
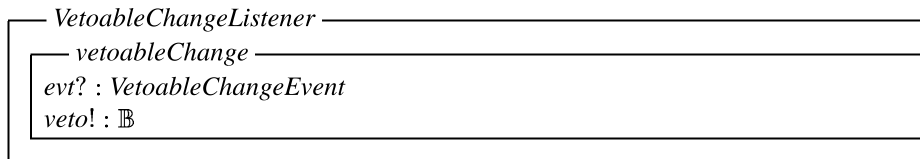
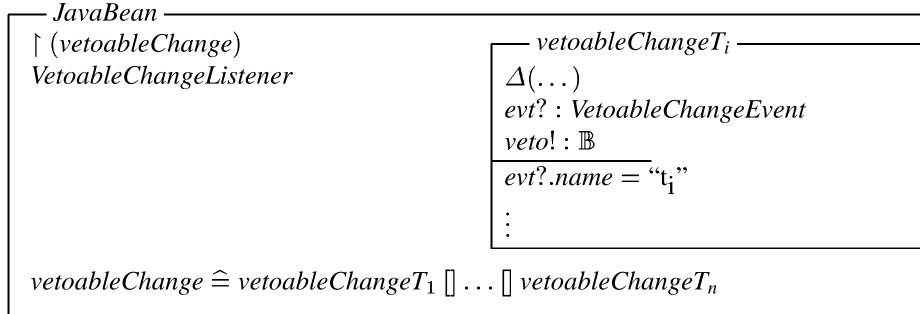


Fig. 10. Processing a Vetoed Property Change

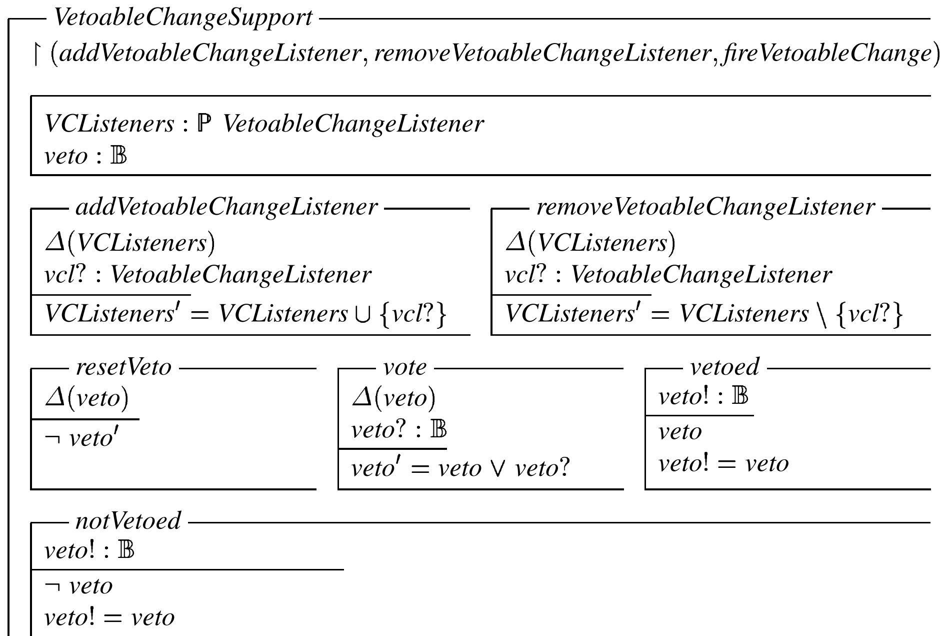
augmenting the operation *vetoableChange*, which a *VetoableChangeListener* provides to accept a *VetoableChangeEvent*, c.f. Fig.8, by a Boolean output parameter *veto*!.



Descendants of *VetoableChangeListener* refine *vetoableChange* by defining operations *vetoableChangeT* for each phrase “receives vetoable change of *t*” in their bean specification. Similar to the definition of *propertyChange* for bound properties, *vetoableChange* is the choice between all operations $vetoableChangeT_i$.



Vetoable Change Generators Similar to *PropertyChangeSupport*, the class *VetoableChangeSupport* defines the general infrastructure for the generator side of vetoable changes. In addition to the set of listeners *VCListeners*, the state variable *veto* holds the status of veto for an execution of *fireVetoableChange*. Several private operations manipulate *veto*.



$$
\begin{array}{|l}
\hline
\textit{mkVCE} \\
name? : String \\
v?, v'? : Object \\
evt! : VetoableChangeEvent \\
evtRev! : VetoableChangeEvent \\
\hline
evt!.name = name? \land evtRev!.name = name? \\
evt!.oldValue = v? \land evtRev!.oldValue = v'? \\
evt!.newValue = v'? \land evtRev!.oldValue = v? \\
\hline
\end{array}
$$

$$
\begin{array}{l}
fireVetoableChange \mathrel{\widehat{=}} \\
\quad mkVCE \fatsemi resetVeto \fatsemi \\
\quad (\fatsemi\, vcl : VCListeners \bullet vcl.vetoableChange \parallel vote) \fatsemi \\
\qquad (notVetoed \\
\qquad \square \\
\qquad (vetoed \parallel \\
\qquad\; (\fatsemi\, vcl : VCListeners \bullet (vcl.vetoableChange[evtRev?/evt?] \setminus \{veto!\}))))
\end{array}
$$

The defi nition of *fireVetoableChange* reflects the complexity of catching a veto and possibly notifying all listeners of a change back to the old value of a property. Like *mkPCE*, the operation *mkVCE* constructs a vetoable change event $evt!$, which is input to the fi rst invokation of *vetoableChange* on all members of *VCListeners*. It also returns *evtRev*, a change event reverting the value of the property $name?$ back from $v'?$ to $v?$. The invokation of *vote* in parallel with each *vetoableChange* serves to accumulate possible vetos: if one call to *vetoableChange* returns $veto! = true$, then the attribute *veto* becomes true.

The choice $notVetoed \; \square \; (vetoed \ldots)$ processes a possible veto. If *veto* is false, the left branch is taken and the property change succeeds, because *notVetoed* is a no-op with precondition $\neg\ veto$. If *veto* is true, then all listeners are notifi ed of the reverse change event $evtRev!$. In this case, hiding the output $veto!$ of *vetoableChange* prevents a veto to the reverse change from succeeding.

The operations *notVetoed* and *vetoed* both copy *veto* to their output, which becomes an output of *fireVetoableChange*. Thus, *fireVetoableChange* propagates the "exception" to the calling *setR*.

A bean *ConstrainedPropertyBean* with constrained properties aggregates an instance *vcs* of *VetoableChangeSupport* and promotes the operations to register listeners.

Each constrained property $r : RType$ listed in the bean's state schema under constrained properties induces two operation specifi cations *getR* and *setR*. The latter is more complex for a *ConstrainedPropertyBean* than for a *BoundPropertyBean*, because it must handle the case that a listener vetoes the change of r.

$$
\begin{array}{|l}
\hline
\textit{ConstrainedPropertyBean} \\
\upharpoonright (addVetoableChangeListener, removeVetoableChangeListener, getR, setR) \\
\\
\quad vcs : VetoableChangeSupport \\
\hline
\end{array}
$$

$addVetoableChangeListener \mathrel{\widehat{=}} vcs.addVetoableChangeListener$
$removeVetoableChangeListener \mathrel{\widehat{=}} vcs.removeVetoableChangeListener$

getR

$v! : RType$

$v! = r$

setRSuccess

$\Delta(r)$
$v? : RType$
$veto? : \mathbb{B}$

$\neg\ veto?$
$r' = v?$

setRVetoed

$v? : RType$
$veto? : \mathbb{B}$

$veto?$

$$setR \mathrel{\widehat{=}} (getR \fatsemi vcs.fireVetoableChange[\text{“r”}/name?])\ [v?/v'?] \fatsemi (setRsuccess \mathbin{[]} setRVetoed)$$

Depending on the value of *veto*!, which is output by *vcs.fireVetoableChange*, either *setRsuccess* or *setRvetoed* are chosen in the defi ntion of *setR*. Only *setRsuccess* changes the value of the attribute *r*. Thus, a vetoed change does not have an effect on the value of the attribute holding the value of the property *r*.

5 Related Work

Although much has been published about component-based software engineering, the formal specifi cation of components in general and JavaBeans in particular has not yet been undertaken by many researchers.

Cimato [4, 5] proposes an algebraic specifi cation technique for Java objects and components, where the term “component” does not denote an independently deployable piece of software – as in the context of component-based software engineering – but an entity of computation in a software architecture[1]. Cimato’s language *Ljala* (Larch Java interface language) belongs to the Larch family of specifi cation languages. An (architectural) component is specifi ed by an algebraic description of its internal state (corresponding to the state schema in an Object-Z class specifi cation) and an interface specifi cation (corresponding to the operation schemas in an Object-Z class specifi cation).

Consequently, Cimato focuses on architectural issues in his specifi cation of JavaBeans. A bean architecture consists of beans as components and adapters as connectors. A confi guration specifi es how these are connected. Cimato’s formal model of a generic bean defi nes a bean to be a pair, consisting of a component and a set of listeners, that makes use of event queues located at input and output ports, where ports are points of interaction of a component with its environment. Each listener is associated to a specifi c output port of the bean.

These architectural descriptions do not describe the interaction of beans as we have done in Section 4 using life sequence charts. The only behavioral description is given in a confi guration and states that the events contained in the event queues must lead

[1] Software architectures consist of components and connectors that enable components to interact.

to an invocation of the appropriate operation of the appropriate component. Properties, bound properties, and constrained properties are neither mentioned nor specified.

Whereas Cimato's interface description language is designed so as to resemble the syntax of the target programming language, our goal is to specify components independently of any target programming language. In the JavaBeans case, for example, we do not restrict inheritance in specifications, even if it is restricted in Java. Component specifications should provide all necessary information concerning the component that is needed either to incorporate the component in a system or to implement the component. To support interoperability of different component models, the target programming language should not play a role in the specification.

Brucker and Wolff [2] use UML class diagrams annotated with OCL formulas to support the run-time checking of constraints on Enterprise Java Beans. They do not attempt to specify the component model of Enterprise Java Beans as such, but they exploit the structure of interfaces that the component model imposes on beans to generate specific run-time checks of Java code from OCL constraints that annotate the various parts of the class diagram for a bean. They observe that the constraints on the implementation of an abstract enterprise bean interface should be a data refinement of the constraints on the interface, and they exploit that observation when checking constraints at run-time.

Beugnard [1] and Cariou [3] use UML to describe communication components called mediums. A medium is a means to define communication services needed in distributed applications, offering a specific interface, transportation services and specific services (shared memory, configuration, quality of service). The authors believe that mediums are natural components of an application and that, as for user interface components, they should be developed by specialists and then made available to users to build their applications. A medium is specified following three different views: (i) a collaboration diagram to describe structural aspects. Messages can be added in order to describe operation calls realized to execute a service, (ii) OCL formulas to describe class invariants of the medium, and pre- and postconditions on its services, (iii) state diagrams associated to the medium in order to manage temporal and synchronization constraints.

6 Conclusions

We have shown how a combination of class diagrams, Object-Z, and life sequence charts can serve to specify the component model of JavaBeans and concrete beans as well. To keep the specifications of concrete beans concise, we have augmented Object-Z by JavaBean-specific keywords. Thus, the predicates explicitly mentioned in a bean specification concern only the properties that are specific to that bean but need not rephrase the properties relevant for the component model. Such specifications can serve as a documentation of beans, and they can be used to analyze whether connected beans fit together on the level of semantics [20], not just on the level of syntax as currently available assembly tools such as the Java BeanBox do.

The specification templates of the JavaBean component model described in Section 4 provide a general description of the component infrastructure for JavaBeans, including a description of the communication protocols between beans. The combination of

Object-Z and LSCs is useful to make the specification of all aspects of the interface protocol possible. Such a specification reveals possible omissions and problems in the proposed component communication protocol (e.g. the issue of "vetoed vetos" for constrained properties in Section 4.4).

Future Work. Specification templates, i.e. prototypical specifications with "holes", are not totally satisfactory as a description of a component model. Although the translation from JavaBean-specific keyphrases in extended Object-Z to proper specifications can be automated (except for filling the holes), a formal definition of that mapping – or a parameterized specification that could be instantiated for concrete beans – is desirable. This would make it possible to formally analyze the component model as such, and it could serve as a basis to compare different component models. A higher-order framework such as an embedding of Object-Z in higher-order logic [15] can be a starting point to come up with a specification framework that is powerful enough to capture component models in general.

A long term goal of this research is to find a general understanding of what the characteristics of components are by way of specifying and comparing different component frameworks. This understanding could serve as a basis for unifying component frameworks and allowing components of different frameworks to interoperate.

To reach this goal, it is necessary to investigate other, more complex component models such as EJB and CORBA. It will also be necessary to take the process of composing systems from components into account.

Acknowledgment We thank Jochen Klose for checking the life sequence charts contained in the paper.

References

[1] A. Beugnard. Communication Services as Components for Telecommunication Applications. In *Proc. 14th European Conference on Object-Oriented Programming,ECOOP'2000*. Sophia Antipolis et Cannes (F), 2000. http://www.-info.enst-bretagne.fr/medium.

[2] A. Brucker and B. Wolff. Testing distributed component based systems using UML/OCL. In M. Wirsing, editor, *Workshop on Integrating Diagrammatic and Formal Specification Techniques*, pages 17–23. LMU München, 2001. http://www.pst.informatik.uni-muenchen.de/GI2001/gi-band.pdf.

[3] E. Cariou. Spécification de composants de communication en UML. In *Proc. Objets, Composants, Modèles (OCM'2000)*, 2000. http://www.-info.enst-bretagne.fr/medium.

[4] S. Cimato. *A Methodology for the Specification of Java Components and Architectures*. PhD thesis, University of Bologna, 1999. http://www.cs.unibo.it/~cimato/www/papers/sty.ps.gz.

[5] S. Cimato. Specifying component-based Java applications. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proc. 3rd Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 105–112, 1999.

[6] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. In *Proc. 3rd Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 1999.

[7] D. Flanagan. *Java in a Nutshell*. O'Reilly, 1999.
[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.
[9] ITU-TS, Geneva. *ITU-TS Recommendations Z.120: Message Sequence Chart (MSC)*, 1996.
[10] J. Klose and H. Wittke. An automata based interpretation of live sequence charts. In T. Margaria and Wang Yi, editors, *Proc. TACAS'2001*, LNCS 2031, 2001.
[11] Microsoft Corporation. *The Component Object Model Specification, Version 0.9*, 1995. http://www.microsoft.com/com/resources/comdocs.asp.
[12] The Object Mangagement Group (OMG). *The Common Object Request Broker: Architecture and Specification, Revision 2.2*, February 1998. http://cgi.omg.org/library/corbaiiop.html.
[13] J. Rumbaugh, I. Jacobsen, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1997.
[14] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 1999.
[15] G. Smith, F. Kammüller, and T. Santen. Encoding Object-Z in Isabelle/HOL. In *ZB2002*, LNCS. Springer-Verlag, 2002. to appear.
[16] Sun Microsystems. *JavaBeans Specification, Version 1.01*, 1997. http://java.sun.com/products/javabeans/docs/spec.html.
[17] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0*, 2001. http://java.sun.com/products/ejb/docs.html.
[18] Sun Microsystems. *JavaBeans Tutorial*, 2001. http://developer.java.sun.com/developer/onlineTraining/Beans/beans02.
[19] C. Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1999.
[20] A. M. Zaremski and J. M. Wing. Specification matching of software components. In G. E. Kaiser, editor, *Proc. of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Software Engineering Notes 20(4), pages 6–17. ACM Press, 1995.