Toward a Formal Model of Software Components

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

[1] Technische Universität Ilmenau, Germany
email: maritta.heisel@tu-ilmenau.de
[2] Technische Universität Berlin, Germany
email: santen@acm.org
[3] LORIA—Université Nancy2, France
email: souquier@loria.fr

Abstract. We are interested in specifying component models in a way that allows us to analyze the interplay of components in general, and to concisely specify individual components. As a starting point for coming up with a technique of specifying component models, we consider JavaBeans. We capture the JavaBean component model using UML class diagrams, Object-Z, and life sequence charts.

1 Introduction

Component-based software engineering [20] 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 prefabricated parts are called components.

Components are independently deployable pieces of software. Several *component models*, such as *JavaBeans* [17], *Enterprise Java Beans* [18], *Microsoft COM* [12], and *CORBA* [13] have been proposed. A component model is designed to allow components to interoperate that are implemented according to the standards set by the model. Building a system from components means selecting components that adhere to a particular component model and composing them in a way that is suitable to achieve the desired system behavior.

To ensure the interoperability of components, a component model must address the following aspects: the *syntactic conventions* for building component interfaces (often called the "interface specification"); the *dynamic behavior* describing allowed and forbidden flows of events between connected components; and the *semantics* of operations of a component, e.g. registering a call-back procedure with a component has an effect on the state of the component, although that may not be immediately visible.

An analysis of a component model must demonstrate that the interplay of components fulfills the expectations described in the component model in any case, i.e. that the component model is consistent (components can interoperate), and that it is complete (all possible behaviors are covered).

For an individual component, a concise way of instantiating the component model (saying that it *is* a component) is needed. Additionally, a specification describing the specifics of the individual component is necessary to analyze whether a specific way of composing individual components actually achieves a desired system behavior.

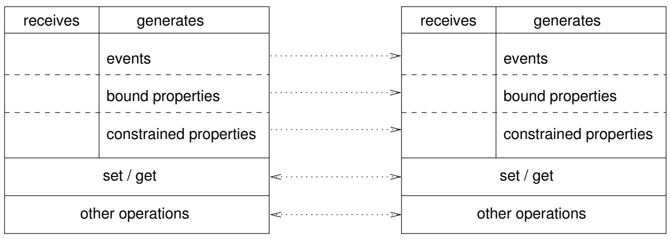

Fig. 1. The JavaBean Interface

The general aim of our research is to come up with a technique of specifying the different *semantic* aspects of a component model, capturing the dynamic behavior and the effects of operations in sufficient detail for analyzing consistency and completeness of the model. A component model specification should be easily instantiable to yield a concise, easily extensible specification of concrete components. We use a combination of UML class diagrams [14], an extension of sequence diagrams called life sequence charts (LSC) [6], and the formal specification language Object-Z [15] to capture the different aspects of a component model. In the present work, we focus on the JavaBean component model [17]. JavaBeans are a well-established technology. The JavaBean component model is reasonably simple to allow us to illustrate our general approach to specifying component models without obscuring the presentation with the technical detail of a more elaborate component model.

2 Introduction to JavaBeans

JavaBeans is a component model originally introduced by Sun in 1996. It has an event-based communication model between components, which are called *Beans*: a Bean notifies registered listener Beans about the events that it generates, and it registers with other Beans to be notified about their events. As we will see in Section 3, cooperating Beans thus realize three variants of the *observer* design pattern [8].

More specifically, the main aspects of the Bean model include [20]:

– 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.

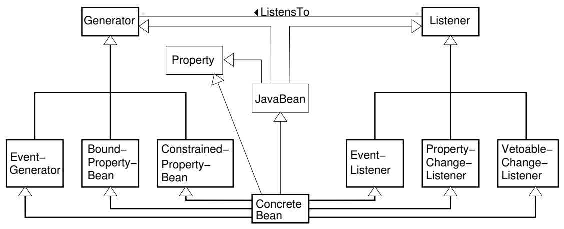

Fig. 2. Overview of the JavaBean component model

Figure 1 illustrates the interfaces of two Beans and their connections. 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 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.

3 The JavaBean Component Model

In order to adequately specify all relevant aspects of the JavaBean component model, we use several complementary formalisms: UML class diagrams [14] describe the static structure of the component model. Because of their graphical nature, class diagrams provide a readily accessible overview of the component model. Object-Z [15] serves to specify the detailed semantics of the classes contained in the class diagram and their features. In particular, it states properties such as class invariants, and it specifies the effect of operations. As the information shown in the class diagrams is also contained in the Object-Z specification, the class diagrams are not strictly necessary. However, they are useful to provide an overview of the component model. Life sequence charts (LSCs) [6] specify the behavioral aspects of the component model. Such aspects cannot be expressed in languages like Object-Z in a satisfactory way. We use LSCs instead of message sequence charts [10] or UML sequence diagrams, because, first, they have a formal semantics [11], and second, they are more expressive than message sequence charts or UML sequence diagrams. In particular, we need to distinguish between optional and mandatory behavior, and we need to use activation conditions and forbidden messages.

3.1 Top-level Model

We now present a formal specification of the JavaBean component model. Figure 2 shows the top-level class diagram of the component model. In general, a JavaBean comprises the functionalities of generators, listeners and properties (c.f. Figure 1). The class *JavaBean* in the center of Figure 2 illustrates this fact. It specializes the three abstract classes *Generator*, *Listener*, and *Property*, and thus serves as a focus relating those three classes.

Corresponding to the three variants of events (simple events, change events for bound properties, and for constrained properties), there are three specializations on the generator and on the listener sides of the class diagram. A concrete component *ConcreteBean* will specialize those classes rather than the abstract classes *Generator* and *Listener*. Therefore the class *JavaBean* and its associated generalization / specialization- relations are not strictly necessary in the component model.

A Bean can take the role of a generator and of a listener at the same time. This results in multiple inheritance in two ways: first, a concrete Bean can inherit from the descendants of the class *Generator* as well as from the descendants of the class *Listener*. Second, it may have several bound properties or listen to several events, for example. Hence, multiple inheritance – with a suitable renaming – from the same class will occur.

For the specification of the component model, we are not interested in the question whether the programming language in which the components are implemented (in our case Java) supports multiple inheritance. We rather aim at clearly describing the essentials of the component model.

3.2 Constrained Properties

In the following, we present the part of the component model describing constrained properties in more detail. The specification of events, simple properties, and bound properties can be found in [9].

Class Diagram Figure 3 shows the class diagram for JavaBeans with constrained properties. Each Bean having constrained properties incorporates an object *vcs* of class *VetoableChangeSupport*. That object is responsible for registering and de-registering listeners, and for notifying the listeners of an intended change of a constrained property. A *ConstrainedPropertyBean* also incorporates a number of objects of class *ConstrainedProperty*. Each *ConstrainedProperty* has a name, a value, and getter and setter methods to query and change that value. The object *vcs* is responsible for notifying registered listeners of an intended change of a constrained property. For that purpose, the object *vcs* maintains references to the registered listeners, which are objects of class *VetoableChangeListener*. Each *VetoableChangeListener* incorporates objects of the class *ObservedProperty*. This class provides an operation *VetoableChange* to process change events. Processing a change event may amount to vetoing the proposed change.

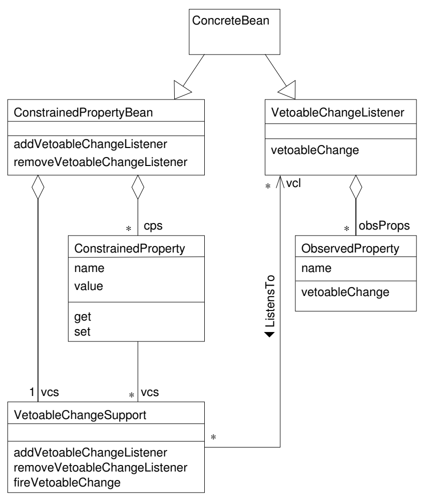

Fig. 3. Constrained properties

As in Figure 2, we indicate in Figure 3 that a concrete Bean can multiply inherit from both classes *ConstrainedPropertyBean* and *VetoableChangeListener*.

Property Specification The following Object-Z specification gives a precise meaning to all classes, attributes, and methods mentioned before.

In Object-Z, each class is defined using a "box" that shows the class name at its top, e.g., *ConstrainedPropertyBean*. The symbol "$\upharpoonright$" is used to specify which attributes and operations of a class are publicly visible. Following the export list, the constant attributes of a class are specified (for example, *name* and *vcs* in the class *ConstrainedProperty* below). The unnamed box inside a class box contains the specification of the mutable attributes of the class. These are declared above a horizontal line. Below that horizontal line, the *class invariant* is given. It states integrity constraints that each object of the class must satisfy. The state box of a class definition is followed by the definition of the class operations.

The class *ConstrainedPropertyBean* offers the operations *addVetoableChangeListener* and *removeVetoableChangeListener* to its environment. It contains two private attributes, namely an object *vcs* of class *VetoableChangeSupport* and a set (expressed by the powerset operator $\mathbb{P}$) of objects of the class *ConstrainedProperty*. The class invariant stipulates that the same *vcs* object be used by all objects belonging to the set *cps*. The operations of the class *ConstrainedPropertyBean* are defined to be the promotion of the operations provided by the *vcs* object.

A *ConstrainedProperty* has two constant attributes: the *name* of the property, and a reference *vcs* to the *VetoableChangeSupport* that handles its change messages. The *value* of the property is its mutable private attribute. The operation *get* copies the *value* to its output *v*!. In Object-Z, there is a convention to decorate output variables with an exclamation mark, and input variables with a question mark.

ConstrainedPropertyBean

$\upharpoonright (addVetoableChangeListener, removeVetoableChangeListener)$

$vcs : VetoableChangeSupport$
$cps : \mathbb{P}\ ConstrainedProperty$

$\forall cp : cps \bullet cp.vcs = vcs$

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

The operation *set* is defined as a combination of several auxiliary operations. First, *get* is invoked to provide the current *value* as an input to *vcs.fireVetoableChange*, which also takes the name of the property as an input. The sequential composition operator $\fatsemi$ pipes outputs of its first argument to inputs of its second argument that have the same base name.

A boolean value *veto*! is the output of the operation *vcs.fireVetoableChange*. It indicates whether the change of the property has been vetoed by one of the listeners. Depending on the value of *veto*!, one of the operations *setSuccess* and *setVetoed* is invoked. The operation *setSuccess* has a precondition $\neg\ veto?$, whereas *setVetoed* has a precondition *veto*?. Therefore, the choice operator $setSuccess \ [\!]\ setVetoed$ invokes *setVetoed* in case of a vetoed change, and *setSuccess* otherwise. The operation *setVetoed* has no effect: it does not mention a Δ-list, and therefore it cannot change the state of the bean.

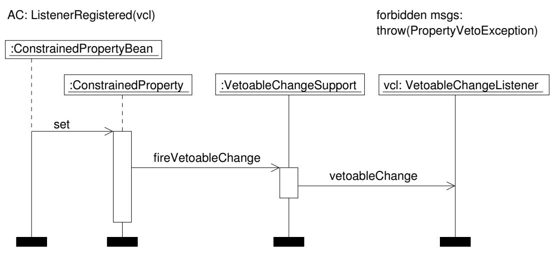

Fig. 4. A Non-Vetoed Property Change

The operation *setSuccess* changes *value* as indicated by $\Delta(value)$. The new $value'$ is determined by the input parameter $v?$.

ConstrainedProperty

$\upharpoonright (name, set, get)$

$name : String$
$vcs : VetoableChangeSuppport$

$value : Object$

get
$v! : Object$
$v! = value$

setSuccess
$\Delta(value)$
$v? : Object$
$veto? : \mathbb{B}$
$\neg\ veto?$
$value' = v?$

setVetoed
$v? : Object$
$veto? : \mathbb{B}$
$veto?$

$$set \mathrel{\widehat=} (get \mathbin{;} vcs.fireVetoableChange[name/name?])\ [v?/v'?] \mathbin{;} (setSuccess \mathbin{[]} setVetoed)$$

Component Collaboration To specify how intended changes of a constrained property are processed, we need life sequence charts that show which objects send which messages in which order. Figure 4 shows how a property change is handled when no veto occurs. The expression "forbidden msgs: throw(PropertyVetoException)" states that no veto may occur in the sequence of messages in Figure 4. The possibility to state such negative conditions is a means of expression not available in message sequence charts.

Moreover, the chart has an *activation condition*, which means that it is only invoked if the corresponding condition holds. In this case, we must require that the object *vcl* of class *VetoableChangeListener* be registered for the *ConstrainedPropertyBean* whose constrained property is to be changed.

Once a listener *vcl* has been registered, it will receive appropriate property change events whenever the operation *set* for a constrained property is invoked. The object *vcl* prototypically stands for all registered listeners. Here, we see that the Object-Z and the LSC specifications are complementary. In the LSC, it cannot be expressed that the

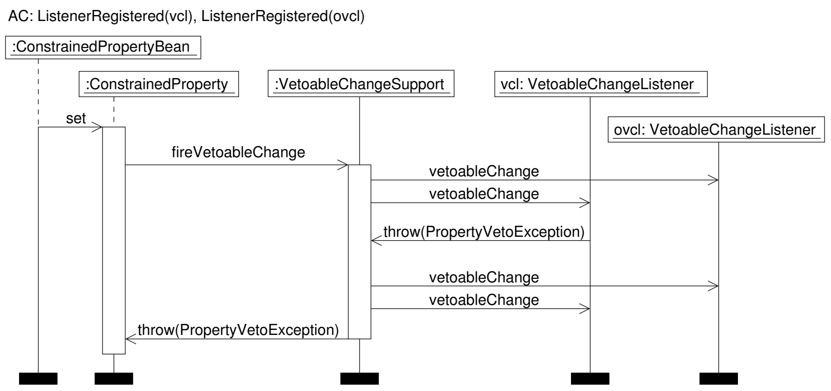
AC: ListenerRegistered(vcl), ListenerRegistered(ovcl)
:ConstrainedPropertyBean
:ConstrainedProperty
:VetoableChangeSupport
vcl: VetoableChangeListener
ovcl: VetoableChangeListener
set
fireVetoableChange
vetoableChange
vetoableChange
throw(PropertyVetoException)
vetoableChange
vetoableChange
throw(PropertyVetoException)

Fig. 5. A Vetoed Property Change

vetoableChange message must be sent to *all* registered listeners. This fact is stated in the Object-Z specification of the class *VetoableChangesupport*, however.

Figure 5 shows the more complex behavior in the case that one of the listeners vetoes the change of a constrained property. In that situation, we need to consider two prototypical listeners, *vcl* and *ovcl*. An invocation of *set* causes a call to *fireVetoableChange*. All listeners, i.e. *vcl* and *ovcl* are notified of the proposed change. Vetoing the change, *vcl* throws a *PropertyVetoException*. As a consequence, all listeners must be notified of the veto. A way to do so [7] is to notify all listeners of a change of the property's value *back* from the new value to the previous one. This may indeed be the only way to notify listeners of a veto, because the JavaBean component model does not enforce explicit confirmations 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.

The class *VetoableChangeSupport* defines the general infrastructure for the generator side of vetoable changes. The class definition expresses crucial information about processing vetoable changes that cannot be expressed in LCSs or are not expressed in the LSC specification for reasons of conciseness and readability. For example, the parameters of all operations are not given in the LSCs, but only in the Object-Z specification.

In addition to the set of listeners *vcl*, the boolean state variable *veto* holds the status of veto for an execution of *fireVetoableChange*. Several private operations manipulate *veto*.

The operations *addVetoableChangeListener* and *removeVetoableChangeListener* just add or remove a new listener to or from the set *vcl*.

The definition 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. The operation *mkVCE* [1] constructs a vetoable change event *evt*!, which is input to the first

[1] The definition of *mkVCE* can be found in [9].

invocation of *vetoableChange* on all members of *vcl*. It also returns *evtRev*!, a change event reverting the value of the property back to its previous value. The invocation of *vote* in parallel with each *vetoableChange* serves to accumulate possible vetoes: if one call to *vetoableChange* returns $veto! = true$, then the attribute *veto* becomes (and remains) true.

VetoableChangeSupport

$\upharpoonright (addVetoableChangeListener, removeVetoableChangeListener,$
$\quad fireVetoableChange)$

$vcl : \mathbb{P}\ VetoableChangeListener$
$veto : \mathbb{B}$

addVetoableChangeListener
$\Delta(vcl)$
$vcl? : VetoableChangeListener$
$vcl' = vcl \cup \{vcl?\}$

removeVetoableChangeListener
$\Delta(vcl)$
$vcl? : VetoableChangeListener$
$vcl' = vcl \setminus \{vcl?\}$

resetVeto
$\Delta(veto)$
$\neg\ veto'$

vote
$\Delta(veto)$
$veto? : \mathbb{B}$
$veto' = veto \lor veto?$

vetoed
$veto! : \mathbb{B}$
$veto$
$veto! = veto$

notVetoed
$veto! : \mathbb{B}$
$\neg\ veto$
$veto! = veto$

$fireVetoableChange \mathrel{\widehat{=}}$
$mkVCE \mathbin{⨾} resetVeto \mathbin{⨾}$
$(⨾\ vc : vcl \bullet vc.vetoableChange \parallel vote) \mathbin{⨾}$
$(notVetoed$
$[\!]$
$(vetoed \parallel$
$(⨾\ vc : vcl \bullet (vc.vetoableChange[evtRev?/evt?] \setminus \{veto!\}))))$

The choice $notVetoed\ [\!]\ (vetoed \ldots)$ 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 notified of the reverse change event *evtRev*!. In this case, hiding the output *veto*! of *vetoableChange* prevents a veto to the reverse change from succeeding.

Here, an unresolved issue of the JavaBean component model has become obvious: what happens if one of the listeners vetoes the second *vetoableChange* that reverts the first vetoed change? In our Object-Z specification, we have decided to forbid that kind of behavior.

The specifications presented in this section give the reader an impression of how the formal specification of component models might look. The specifications of the other classes shown in Figure 3 can be found in [9].

MyButton

$\upharpoonright (paint, fireAction,$
$\quad isLargeFont, getPreferredSize)$
generates event *action* of *ActionEvent*
receives event *pressed*, *clicked*,
$\quad$ *released*, ... of *MouseEvent*

$text\_xpad, text\_ypad : \mathbb{N}$

$text\_xpad = 12$
$text\_ypad = 8$

$sized, down : \mathbb{B}$
bound properties
$debug : \mathbb{B}$
$font : Font$
$fontSize : \mathbb{N}$
$foreground, background : Color$
constrained properties
$label : String$

MyButton

ConstrainedPropertyBean

...

$label : String$

$\exists\, cp : cps \mid cp.name = \text{“label”}$
$\quad \land\ cp.value = label$

$getLabel \mathrel{\widehat=}$
$\quad [cp : cps \mid cp.name = \text{“label”}] \bullet get$
$setLabel \mathrel{\widehat=}$
$\quad [cp : cps \mid cp.name = \text{“label”}] \bullet set$

Fig. 6. A component specification and its expansion

4 Specification of a Bean

This section illustrates by way of an example how our JavaBean component model can be instantiated to specify individual Beans very concisely, without repeating all the details already stated in the component model specification.

The simple Bean *MyButton* implements a text button for a graphical user interface [19]. 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.

We specify an individual Bean such as *MyButton* in Object-Z augmented by some keywords, as the left-hand side of Fig. 6 shows. Replacement rules define the meaning of those keywords. There is no need to give additional class diagrams or repeat any of the LSCs of the component model. Specializing operations inherited from the component model describes their effect particular to the individual Bean.

The specification of *MyButton* declares the generated and received events *action*, *pressed*, *clicked*, etc. The phrases declaring those events mention the classes *ActionEvent* and *MouseEvent*. These classes – which we do not show here – implement the type of information those events carry: an *ActionEvent* identifies the source of the event, which is an instance of *MyButton*; a *MouseEvent* carries the coordinates of the mouse pointer.

The state box declares the mutable attributes of *MyButton*: *sized* and *down* are private boolean flags; *debug*, the font and color information (*font*, *fontSize*, *foreground*, *background*) are bound properties, and the text *label* displayed by the button is a constrained property.

We exemplify the keyword translation process by considering the expansion of the declaration of *label*, which the right-hand side of Fig. 6 shows, leaving out the parts concerning the other attributes of the Bean.

The value of each property is stored in a mutable attribute. Therefore, the declaration of *label* remains part of the state box. Because *label* is a constrained property, *MyButton* inherits from *ConstrainedPropertyBean*. This provides all the infrastructure discussed in Section 3.2. In particular, *MyButton* has attributes *vcs* and *cps*. An invariant relates the attribute *label* to a member of *cps*, namely one with the name "label". It also relates the *value* of that *ConstrainedProperty* to *label*. Finally, to conform to the naming conventions for JavaBeans, the operations *getLabel* and *setLabel* are defined to be promotions of the *get* and *set* operations of that object.

This example shows how the keywords related to JavaBeans hide formal noise in the Object-Z specification of a Bean. It also shows that we are actually working at the specification level: how an implementation ensures that *label* is realized as a constrained property is of no interest at this level of abstraction; it suffices to *require* that the attribute *label* be related to a constrained property including all the necessary operations.

5 Related Work

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

Cimato [4, 5] proposes an algebraic specification 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 that is connected to other components in a software architecture Consequently, Cimato focuses on architectural issues in his specification of JavaBeans. A Bean architecture consists of Beans as components and adapters as connectors. A configuration specifies how these are connected. These architectural descriptions do not describe the interaction of Beans as we have done in Section 3 using life sequence charts. Properties, bound properties, and constrained properties are not dealt with.

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). They specify components by collaboration diagrams, OCL for class invariants and service specifications, and state diagrams for temporal and synchronization constraints. In contrast to our work, Beugnard and Cariou do not aim at specifying component models in general, but propose a specific component model.

6 Conclusions

A component model is the basis for many applications. Therefore, it deserves a thorough analysis based on a precise description of its semantics. Our approach of specifying component models provides such a precise description. As we have seen for the example of the JavaBean component model, just specifying a component model can make contradictions and omissions in the informal description explicit: the literature [7, 17, 19] does not resolve the problem of "vetoed vetoes". We are probably not the first to detect that problem, but setting up a specification systematically leads one to ask the questions that make problems as this one obvious.

We specify components independently of a 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.

Not referring to specific features of a programming language, specifications of component models support the comparison of different component models – a research that may lead to improved interoperability between different component models.

There inevitably is some "formal noise" in a formal specification. We believe this is acceptable for a component model, because the goal of interoperable components requires a consistent and complete description of the infrastructure that they can build on.

For an individual component, however, the specification highlights the specific services, abstracting from details of the component model by means of specific key-phrases. Specifications of individual components – and the underlying component model – can be the basis for advanced assembly tools that analyze components not only on the level of interface syntax but also on the level the semantics of the services that components provide, and their interaction in a specific system. Such an analysis cannot be provided based on the code alone.

Future Work. We have presented a way of formally describing component models with the motivation of analyzing those models for consistency and completeness. The question how such an analysis is best conducted and what appropriate tool-support for analyzing component model specifications is must still be addressed. An embedding of Object-Z in the logic of a theorem prover [16] can be a starting point to come up with mechanized support for component models analysis.

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.

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 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. M. Heisel, T. Santen, and J. Souquières. On the specification of components – the JavaBeans example. Technical Report A02-R-025, LORIA, Nancy, France, 2002.
10. ITU-TS, Geneva. *ITU-TS Recommendations Z.120: Message Sequence Chart (MSC)*, 1996.
11. 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.
12. Microsoft Corporation. *The Component Object Model Specification, Version 0.9*, 1995. http://www.microsoft.com/com/resources/comdocs.asp.
13. 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.
14. J. Rumbaugh, I. Jacobsen, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1997.
15. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 1999.
16. G. Smith, F. Kammüller, and T. Santen. Encoding Object-Z in Isabelle/HOL. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB2002: Formal Specification and Development in Z and B*, LNCS 2272, pages 82–99. Springer-Verlag, 2002.
17. Sun Microsystems. *JavaBeans Specification, Version 1.01*, 1997. http://java.sun.com/products/javabeans/docs/spec.html.
18. Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0*, 2001. http://java.sun.com/products/ejb/docs.html.
19. Sun Microsystems. *JavaBeans Tutorial*, 2001. http://developer.java.sun.com/developer/onlineTraining/Beans/beans02.
20. C. Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1999.