

Improving Composition Support with lightweight Metadata-based extensions of Component Models

Johann Oberleitner and Michael Fischer

Vienna University of Technology, Vienna A-1040, Austria,

joe@infosys.tuwien.ac.at,

WWW home page: <http://www.infosys.tuwien.ac.at/Staff/joe/index.html>

Abstract. Software systems that rely on the component paradigm build new components by assembling existing prefabricated components. Most currently available IDEs support graphical components such as .NET controls or JavaBeans for building GUI applications. Even though all those IDEs support arrangement and layout of those desktop components, composition support is rather limited. None of the most important composition environments support built-in validation of composition for .NET components or JavaBeans no further than type checking. Enhanced component models and composition environments on the other hand are not widely adopted and require incompatible components that may not be reused in most IDEs.

Our approach avoids these problems with lightweight extensions of existing component models with metadata attributes. We use metadata attributes to assign additional information to components. Subsequently, we enhance the built-in composition facilities of the component model and the composition environment to exploit those metadata attributes. As we show the metadata attributes may be used to support required interfaces, constraint checks for method invocation or if all participants in a component collaboration satisfy a certain protocol. Since all the attributes are optional, components that use the attributes still operate in default environments.

1 Introduction

Prefabricated software components are known to improve the quality of software construction and reduction of the development costs [1]. Component models [2] define the structure, components adhere to and how they can interoperate. Instead of implementing every functionality from scratch new features are built by assembling existing components.

Different component models have been introduced for different purposes, ranging from desktop component models to distributed component models. Most development environments focus primarily on desktop component models that are intended to be used in the development of client applications with graphical user interfaces. Hence, most today's utilized platforms, such as Java and

.NET include simple component models for building desktop applications such as JavaBeans [3] or components for .NET [4]. Components that adhere to these component models usually represent graphical widgets.

These components may be either be used programmatically or may be assembled in composition environments to form composite components. Almost all IDEs targeted for GUI development provide graphical composition environments that support one of these component models.

In graphical composition environments developers may create component instances and put them in composite components, visualized in graphical design windows, browse and configure component instance properties such as fonts or colors, and create event listeners. These environments have in common that they support the import of arbitrary third-party components that adhere to one component model.

Inventors of component models wanted a fast adoption of those models, hence the requirements on components are rather small. The only requirement imposed on a Java class to become a JavaBeans component is to provide a default constructor. Similarly, .NET introduces a component model primarily targeted for GUI components. The single requirement for a .NET class to be used in standard .NET composition environments is that this class inherits from `System.ComponentModel` class.

Unfortunately, these simple component models and composition environments only support simple instantiation and creation features. For instance, in Visual Studio .NET 2003 it is possible to configure the property value of a component with the instance of any other component created in the composition environment. However, there are no standardized checks if such compositions are valid or required. Furthermore, there are no generic ways to use proxies or adaptors for such compositions.

New composition environments and component models have been introduced that provide validity checks at design time or the generation of adapters [5]. The flexibility of this introduction comes with the cost that these composition environments are not seamlessly integrated in the IDEs, making the adoption of those component models difficult if not impossible. Furthermore, these component models require rather large runtime environments.

We focus on this problem and introduce lightweight extensions of the existing desktop component models. Primarily, we rely on two different functionalities. First, we define metadata to describe additional information such as validity requirements or if component properties are required to be set to let the component work. Components are enriched by this metadata. Since, the use of this metadata can be ignored by a composition environment all components that use this metadata can still be used in composition environments unaware of it. Second, we use the extension mechanisms provided by standard composition environments to build extensions for the composition environment to enforce the composition conditions described by the metadata. These extension mechanisms are defined by the component models but optional to use by components and composition environments.

We have designed and implemented our approach for the desktop component model of the .NET framework and Microsoft Visual Studio .NET 2003 as the appropriate composition environment. We show three examples for extending composition capabilities:

- introduction of required interfaces,
- use of OCL constraints [6] for methods and classes, and
- use of protocols for verifying component collaborations.

Although our implementation focuses on .NET the metadata annotation feature of JDK 1.5 allows that large parts of the approach can be ported for Java and JavaBeans, too.

The structure of this paper is as follows. Section 2 explains the metadata facilities of .NET, and how the .NET component model can be supported by graphical composition environments. In section 3 metadata attributes for enhancing composition facilities of components are introduced. Section 4 further extends these facilities with support for composition environments. We discuss related work in section 5 and our future plans in section 6. We draw our conclusions in section 7.

2 Background

This section introduces the .NET metadata facility the .NET component model frequently uses and our approach is based on. Furthermore, this .NET component model is introduced and how .NET IDEs such as Microsoft Visual Studio 2003 and Borland C# Builder provide composition environments for components of this model.

2.1 Metadata Facilities

The main task of metadata attributes is to attach additional static information to programming entities such as classes, fields, or methods. Almost all structural programming entities are allowed as targets for attribute assignment. The compiler stores the attributes in the executable files and DLLs. The runtime environment provides read access to the attributes with reflection mechanisms.

The metadata attributes are stored in .NET Bytecode and hence are utilisable from virtually any .NET capable language. However, in this section we will concentrate on the syntax of C# attributes since it is very clear to read. We will also use this convention in the remainder of the paper. Metadata attributes consist of two different parts: the *attribute implementation class* and the *specification/application of an attribute*.

As can be seen in figure 1 an attribute is attached to a programming entity with provision of the name of the attribute class in square brackets directly before the entity. It is also possible to initialize the attribute with positional arguments or named arguments. For the attribute assignment of the `MyMethod2` the integer is a positional argument, and the comment is a named argument. Attributes are

```

public class Testclass
{
    [SomeAttribute]
    public void MyMethod() {}

    [SomeAttribute(5, Comment="Method_Metadata" )]
    public void MyMethod2() {}
}

```

Fig. 1. Attribute specification in C#

compiled and statically attached to programming entities. It cannot be assumed that any initialization data is known at compilation time therefore the types allowed for attribute parameters are restricted to predefined value types such as integers and strings, the .NET **Type** class that represents all types in .NET (similar to **Class** in Java) and single-dimensional arrays of those allowed types.

An attribute implementation class has to inherit from the predefined class **System.Attribute**. A syntax convention of C# states that the attribute's class-name shall end with **Attribute**. When applying the attribute this postfix may be omitted. Each constructor provides a valid sequence for attribute initialization with positional arguments. The positional arguments provided in the attribute attachment must unambiguously be matched by a constructor of the attribute class. Named parameters are realized with non-static public read-write properties or fields¹. To restrict attribute specification to a subset of the available entity types, usage attributes may be assigned to the attribute implementation class. Figure 2 shows examples for attribute classes. In this example the attribute may only be applied to methods because an **AttributeUsage** attribute is applied to the implementation class and specifies that only methods are allowed as targets.

Figure 3 shows how attributes can be queried at run-time. The type of the **Testclass** is retrieved with C#'s **typeof** keyword that returns the type of the class provided as argument. Based on this type reflection information about the method is retrieved and subsequently with this information an array of custom attributes that implement **SomeAttribute** is extracted.

One of the primary differences of .NET compared to Java editions before Java 1.5 is the availability of metadata attributes. Java 1.5 introduces metadata attributes similar to .NET's metadata attributes.

2.2 Desktop Component Models

To allow the instantiation of components in graphical design environments components must support a standardized way for creating new instances. Although it would have been possible to standardize factory methods .NET components

¹ properties are a syntax convenience of C# for automatically creating get and set accessors

```

[AttributeUsage(AttributeTargets.Method)]
public class SomeAttribute: Attribute
{
    private int x;
    private string comment;

    public SomeAttribute() {}
    public SomeAttribute(int x) { this.x = x; }

    public string Comment
    {
        get { return this.comment; }
        set { this.comment = value; }
    }
    public int X { get { return this.x; } }
}

```

Fig. 2. Attribute implementation

```

public void CheckMetadata()
{
    Type t = typeof(Testclass);
    MethodInfo method = t.GetMethod("MyMethod2");
    object[] customAttributes =
        method.GetCustomAttributes(typeof(SomeAttribute), false);

    if (customAttributes != null && customAttributes.Length > 0)
    {
        SomeAttribute attribute = customAttributes[0] as SomeAttribute;
        int x = attribute.X;
        string comment = attribute.Comment;
    }
}

```

Fig. 3. Accessing metadata at runtime

require a default constructor for instantiation in graphical design environments. As a side effect, this allows the instantiation of unconfigured components in these design environments.

Furthermore, component instances may be configured with property sheets. A property is exposed by a component by accessor methods that allow read and write access to a logical property of the component instance. Components often support emission of events that are delivered to handler methods. In .NET event handlers are exposed from the emitting components with delegates, a type-safe variant of function pointers. The functionality for visualizing a component is supported by inheriting from a particular base class and implementing some drawing methods.

2.3 Component model support for composition environments

The .NET component model is supported by the formular designer included in Microsoft Visual Studio .NET and any other .NET based IDE. These composition environments support instantiation of components and in case of graphical components also positioning and resizing of the components in graphical windows.

Graphical composition environments analyze the components dynamically to construct property-sheets for configuring component instances. Editors are already provided for most predefined data types. However, it is possible to implement additional property editors. In that case the predefined *Editor* attribute has to be attached to a particular property or to the type of the definition that shall be edited.

In .NET configuration of properties of component instances that are put on parent components leads to generation of initialization code of these properties. This code (stored in the method `InitializeComponent`) is called from the constructor of the class that hosts the component instances. This automatically generated code creates the child component instances and sets the properties. The default semantics of the code generation feature supports setting of existing instances or constant values. However, it is possible to attach a *DesignerSerializer* attribute to the parent class. With this attribute a so-called *CodeDomSerializer* can be attached to the class that hosts the components. *CodeDom* is a .NET facility that supports the construction of source code statement sequences independent of the target programming language. Source code statements can be inserted into a syntax tree to generate arbitrary source code. The *CodeDomSerializer* serialization method is called after a property change has been completed in the property sheet. It serializes the code sequence to source code in the appropriate .NET programming language and inserts this code in the initialization method.

3 Attributes for Components and Composition

In this section we introduce three different kinds of attributes that aid the composition process. All three attributes represent assumptions that have to be

satisfied by components to provide a correct application. We support two different approaches for the enforcement of these assumptions. The manual approach relies on manual calls to helper methods for checking if the assumptions are satisfied. The compositional approach introduced in the next section relies on either the composition methods provided by the composition environment or by code injected by the environment to check the assumptions.

3.1 Required Interfaces

In this section we introduce metadata attributes targeted for composition. Most component models define a notion of *provided interfaces* of components. Furthermore, some component models introduce a notion of *required interfaces*. While provided interfaces are a way to access the functionality provided by a component, the required interfaces describe the prerequisites a component has on its environment or other components to accomplish its work.

For marking some properties as required we introduce the *required* attribute implemented by the class `RequiredAttribute`. It can be applied to properties to signal that these properties must be set to let the component work. It is also possible to attach multiple required attributes to require that a property supports all those interfaces. Although it may be possible to introduce a new interface that extends from all those interfaces it cannot be assumed that existing components already support this interface. An example of the required attribute can be seen in figure 4. Furthermore, for property arrays and collections we support a minimum and a maximum number of instances that must be set for the property. For collections an additional semantics is possible that allows setting properties that implement either one interface or another. In that case it is not required that both interfaces are implemented by the same component.

In addition to required interfaces represented as properties we also support events to become marked as required. Instead of components that implement the required interfaces now event listeners have to be set. Since, the parameters for the numbers are also applicable to event listeners, we reuse the same attributes for both, properties and events.

A programmatic check of all required properties is done manually by calling a helper method to detect if all required interfaces are set. The implementation of this checker uses reflection on the component instance, iterates over all properties and verifies those that have a *required* attribute attached to it. The result of the check can be visualized in the composition environment already at design time in overwriting the `OnPaint` method.

3.2 OCL Constraints

The application of pre- and postcondition in programming is widely accepted. Unfortunately, only few programming languages such as Eiffel [7] have built-in support for constraints. We propose two attributes that store the precondition and the postcondition of methods with `PreAttribute` and `PostAttribute`, respectively. We use the Object Constraint Language (OCL) for formulating the

```

public interface IMyInterfaceA {
    public void MethodA ();
}

public interface IMyInterfaceB {
    public void MethodB ();
}

public class Test: Control
{
    private IMyRequiredInterface required;

    [Required(typeof(IMyInterfaceA))]
    [Required(typeof(IMyInterfaceB))]
    public IMyInterfaceA RequiredProperty
    { get { return this.required; }
      set { this.required = value; }
    }

    // paint method checks all required properties
    public override void OnPaint(PaintEventArgs e)
    {
        if (!RequiredHelper.CheckAllRequiredProps(this))
            { /* paint error message */ }
        else { /* normal drawing code */ }
    }
}

```

Fig. 4. Required attribute

constraint expressions since it can be parsed easily, can be learned quite fast, and is simple to understand. Figure 5 shows a component that attaches pre- and postcondition to a withdraw method of an account class.

In addition to pre- and postcondition, the `InvariantAttribute` stores invariant conditions for classes and interfaces. We have faced one problem in using these three attributes. The OCL constraints are provided as string parameters for the attribute classes. Unfortunately, it is not possible to execute the attribute constructor at compile time to verify if the expression is a syntactic valid OCL expression.

When the attribute is accessed for the first time, the OCL expression is parsed and the parse tree is stored for further evaluation. At the bottom of figure 5 we show how OCL constraints can be verified by calling a static method of the `OCLCheck` class we have implemented. The evaluation is done with the visitor pattern. Since OCL allows the use of fields, properties, and methods access to the object's runtime data is implemented using reflection. Parameters are provided as last parameters of the check methods.

```

public class Account
{
    int balance;

    [Pre(" amount >= 0 and self.balance >= amount" )]
    [Post(" self.balance = self.balance@pre - amount" )]
    public void Withdraw(int amount)
    {
        this.balance -= amount;
    }
}

// check code
OCLCheck.PreconditionCheck(this , "Withdraw" , increment );

```

Fig. 5. OCL attribute specification

3.3 Collaboration Protocols

In many scenarios it is not possible to call methods or query and update properties from arbitrary component states. For instance, to operate on a file it must have been opened before. Hence, it is necessary that components interact by following a certain protocol [8, 9]. Protocols define a predefined order in which methods and properties may be accessed, i.e. protocols define state machines for ordering method invocations.

We use various attributes to assign a state machine to an interface. One or multiple *Protocol* attributes declare all protocols an interface participates in.

Besides the name of the protocol it also takes an array of state names of the state machine, and the initial state. Other interfaces that act in this protocol are marked with *Collaborator* attributes that are initialized with the protocol name and the type of the participating interface.

For each method *Transition* attributes are used to declare allowed state transitions associated with the invocations.. Each transition attribute is initialized with the name of the source state and the target state.

Figure 6 shows an example of two interfaces that share the access on a particular resource. These interfaces can be used in the same class or in two different classes. However, the semantics remains the same. Before the reader can access the data the state machine has to be in the open state.

```
[Protocol("Interaction" , new string [] { "Closed" , "Open" } ,
        Initial="Closed" )]
[Collaborator (typeof(I2))]
public interface IProvider
{
    [Transition ("Closed" , "Open" )]
    void Open ();

    [Transition ("Open" , "Closed" )]
    void Close ();
}

public interface I2
[Protocol("Interaction" , new string [] { "Closed" , "Open" } ,
        Initial="Closed" )]
[Collaborator (typeof(I1))]
public interface IReader
{
    [Transition ("Open" , "Open" )]
    object Read ();
}

// check code
StateMachine.Check (this , "Read" );
```

Fig. 6. Protocol specification

Alternatively, it is allowed to define the protocol in an external class and reference it from the protocol attribute with its type. This avoid duplication of state definitions.

We have provided a helper class that verifies if a method invocation starts from the appropriate state. When applying the checks manually this code has to be inserted at the beginning of a method.

4 Tool Support and Automatic Adaptor Generation

This section describes how the composition process can be improved by the attributes defined before.

Without any tool support constraints defined with the attributes described before can only be verified manually with invocation of checking methods and are neither verified nor enforced automatically. However, with the use of adaptors based on these attributes we can automatically enforce verification of those constraints. The .NET component model in connection with design environments such as Visual Studio .NET allow almost seamless use of those attributes.

4.1 Composition Support

As briefly mentioned in section 2 the .NET component model defines some meta-data attributes for layouting and arrangement of .NET widgets and components. Some of these attributes are used in conjunction with the .NET property sheet used by Microsoft's Visual Studio .NET and other IDEs. The *Editor* metadata attribute allows developers to assign user defined editors to classes and interfaces but also to properties. These editors are automatically used by the property sheet in Visual Studio .NETs designer and allows modification of properties. When a developer selects a cell in the property sheet the environment detects if an editor attribute is attached to the datatype or the property. If it finds such an attribute it uses the editor's type stored in the attribute to instantiate a predefined class that handles all editing issues.

We have implemented such an editor that may be attached to properties that use the required attribute. This editor provides the developers with a list of component instances which components match all *required* interfaces for the particular property.

We enumerate all component instances by using a mechanism of .NET's component model to query the children of components and their parents. In a similar way we could have used reflection for this retrieval. Currently we show only those component instances that have the same parent as the component that requires this interface. After the selection of one of the proposed component instances Visual Studio automatically generates the correct instance assignment in the constructor of the class that hosts the instances. In case of a property multiplicity different from one, either arrays or collections are used. The same editor class is used but it does not provide a combobox but a dialog to select the component instances.

When the editor has finished Visual Studio generates code fragments that reflect the choice the user makes in the editor in the constructor of the component instance owner's class. In our case, it is no longer possible to use the predefined code generation semantics. Hence, a code serializer has to be implemented that generates an appropriate source code fragment for initializing the required component compositions. .NET provides an object model for generating source code in a platform independent way. The code serializer generates the appropriate

initialization statements for arrays and collections and inserts it into the predefined code generation stream provided by .NET. The code serializer is attached to the component implementation class that requires the interface with the *DesignerSerializer* attribute. It is required to be attached to the class instead of the interface because a code serializer of the interface is not aware of any instance fields and it is not possible to generate correct initialization code.

4.2 Verification Adaptors

For the automatic evaluation of constraints we describe with the attributes defined above we use adaptors that include verification code. These adaptors include checks if required properties are bound, if method arguments satisfy preconditions or method results satisfy postconditions, and if a protocol sequence is still satisfied. The adaptors just include code sequences described above.

The verification adaptors are set in the initialization code of the component constructor when an editor has been used. We generate the adaptor initialization code instead of field assignments. Figure 7 shows an example for such a code. Since the *DesignerSerializer* attribute is applied to the whole class and not only to a single property we have the chance to modify multiple initialization statements. However, a serious limitation of our automatic approach is that we cannot change method call statements where the interface used has not been set via properties. Instead of the field assignment the code serializer initializes an adaptor interface and uses the original value as argument.

```
... // code inside InitializeComponents
// old code
// this.RequiredProperty = required1;

// new code
this.RequiredProperty =
    new ConstraintAdaptor_IRequiredInterfaceA (required1);
```

Fig. 7. Adaptor Initialization

The adaptors are generated on demand. Here we use another .NET feature for dynamically creating or loading assemblies. If the assembly that contains the adaptor does not exist it is created dynamically. The disadvantages of this approach are that the adaptor code has to be written in .NET's Intermediate Assembly language and that a new assembly is created for each adaptor.

5 Related Work

The use of metadata beyond type information is frequently used within some component models. Enterprise JavaBeans [10] rely on metadata stored in de-

ployment descriptors to configure components for different installation systems. JavaBeans [3] provides and uses descriptor classes to store additional information about components. For instance, this information can be used to support custom editors similar to .NET's type editors we have used. In .NET metadata attributes are used for instance configuration which we have used and extended. Further attributes are used for remoting and distribution purposes. In particular it is possible to configure security and transaction settings. Another area where metadata attributes are heavily used in .NET are system interoperability. .NET predefines some attributes for importing methods from native DLLs and allows to modify method calling conventions and argument conversion. Common to all predefined metadata attributes in desktop component models is that the attributes are used primarily to ease the configuration process of single components. In contrast to this we use metadata to allow configuration of components that collaborate and support the automatic generation of adaptors that verify constraints or collaborations.

The notion of a required interfaces is well-known for several years [1]. However, component models that support required interfaces are usually not supported by any standard development environment. Our extension can be considered lightweight since it can be used without any modification in all .NET IDEs that support the metadata attributes Microsoft has predefined with .NET. Even, if these attributes are not supported the components are still functional. Validation, however, must be done manually.

The first general purpose object-oriented programming language that supports constraints is Eiffel [7] with its support for Design-by-Contract [11]. For Java different approaches implement pre- and postconditions such as JML [12] or iContract. Since Java did not support metadata these approaches primarily use JavaDoc comments to store the constraints. We expect that some of these approaches will adopt the new metadata notation of JDK 1.5. Using metadata has one crucial advantage. While it is completely optional what happens with metadata stored in comments metadata attributes are part of the compiled code. It is just not possible to compile code enhanced with metadata attributes where the attribute classes are not available. However, attributes share one problem with these approaches: the compiler checks only type correctness but does not execute constructors to verify if arguments are valid. Regardless if such metadata is stored in attributes or in comments its syntax cannot be checked at compile time.

6 Future Work

We plan to introduce additional metadata attributes and further support for composition environments. The .NET component model supports the implementation of so-called designers, graphical editors for components useable directly in the composition environment's assembly window. When laying out graphical components on the standard containers such as panels or forms provided by .NET no visualization of the compositions is shown. We plan to extend the con-

tainers to draw graphical representations for the compositions of components. In [5, 13] we have built a composition environment that visualizes connections between components with lines. We plan to build a similar mechanism with providing base classes for containers that visualize these lines in design modi of composition environments.

We also plan to port the attributes to Java with JDK 1.5. The attributes and the classes for enforcing the constraints described with the attributes can easily be ported to Java despite the differences of the platforms. However, porting the support for the composition environments requires more effort. The .NET component model provides strong support for .NET composition environments which is not the case for Java or JavaBeans. However, we will investigate if this may be done for one particular composition environment such as the Component Workbench [5] or Eclipse with a JavaBeans widget editor.

Since not everything can be done with checking the validity of component composition attributes at design time we plan to build a simple verifier that takes a root component as input. This verifier will instantiate this root component and will use reflection to iterate over all child instances to check if all constraints defined in the root component's or the child components are fulfilled. Such a verifier can then be included as a post-compilation step in development environments.

Inserting automatic checks only on interfaces has many drawbacks but is a simple way to support the checks without additional tools. With aspect-oriented tools such as [14] we can inject verification code to arbitrary methods and properties. However, although there are some aspect tools for .NET available we are not aware of any tool that supports a declaration of join points with metadata attributes. It is surely worthwhile to extend an existing aspect weaver with join point support for metadata attributes.

7 Conclusions

In this paper we have shown how a simple widely used component model can be extended with metadata attributes specifically introduced for composition. These metadata attributes improve readability and act as additional documentation of components. Furthermore the attributes store additional semantic information beyond the capabilities of the programming languages and component models used. This semantic information may be used to realize simple semantic checks without preventing the use of the components in standardized environments.

We introduced attributes to describe required interfaces that are mandatory to be set before a component instance may be used. We also introduced attributes for describing constraints with OCL. Furthermore, we presented attributes for component collaboration.

The validation of these attributes may be done programmatically in the components or the components' clients referring to small helper classes that implement the validation semantics. Additionally, we show how we have exploited

.NET property editors and code serialization APIs to automatically generate the validation code when binding properties: instead of directly setting property fields in the initialization code we set automatically generated adaptors and mediators that include the verification checks. In the future we plan to add further attributes and evaluate if the approach may be ported to Java.

References

1. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (1998)
2. Heineman, G.T., Councill, W.T., eds.: Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley (2001)
3. Hamilton, G., ed.: JavaBeans. Sun Microsystems, <http://java.sun.com/beans/> (1997)
4. Griffiths, I., Adams, M.: .NET Windows Forms in a Nutshell. O'Reilly (2003)
5. Johann, O., Gschwind, T.: Composing distributed components with the component workbench. In: Proceedings of the 3rd International Workshop on Software Engineering and Middleware 2002 (SEM 2002). (2002)
6. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting your models ready for MDA. Addison-Wesley (2003)
7. Meyer, B.: Object Oriented Software Construction. Prentice Hall (1997)
8. Yellin, D.M., Strom, R.E.: Interfaces, protocols, and the semi-automatic construction of software adaptors. In: OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications, ACM Press (1994) 176–190
9. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. **19** (1997) 292–333
10. DeMichiel, L.G., Yalcinalp, L.Ü., Krishnan, S.: Enterprise JavaBeans Specification, Version 2.0. Sun Microsystems. (2001) Proposed Final Draft 2.
11. Meyer, B.: Applying 'design by contract. IEEE Computer **25** (1992) 40–51
12. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the java modeling language (jml). In: International Conference on Software Engineering Research and Practice (SERP '02), CSREA Press (2002) 322–328
13. Oberleitner, J., Dustdar, S.: Constructing Web Services out of Generic Component Compositions. In: Proceedings of the International Conference on Web Services Europe 2003, Springer (2003) 37–48
14. Rajan, H., Sullivan, K.: Eos: instance-level aspects for integrated system design. In: ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (2003) 297–306