# 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
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.

Our approach addresses these problems with lightweight extensions of existing component models with metadata attributes. 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 satisify a certain protocol.

## 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.

In graphical composition environments developers may create component instances and put them in composite components, visualized in graphical design

© Springer-Verlag Berlin Heidelberg 2005

windows, browse and configure component instance properties such as fonts or colors, and create event listeners.

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 JavaBean 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 the `Component` class of the `System.ComponentModel` namespace.

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 a component's 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 can optionally be used 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 to Java and JavaBeans, too.

The structure of this paper is as follows. Section 2 discusses existing techniques we build on. 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.

### 2.1 Metadata Facilities

Metadata attributes are used to attach additional static information to programming entities such as classes, fields, or methods. The compiler stores the attributes in the executable files and DLLs. The runtime environment provides read access to the attributes with reflection mechanisms.

For space reasons we cannot discuss in detail the .NET metadata facility. Instead we refer to Figure 1 that shows the assignment of metadata attributes to methods with edged braces and on an intuitive understanding of the reader. A complete discussion of .NET metadata can be found in [7] and our webpage [8].

### 2.2 Desktop Component Models

Instances of desktop components are instantiated in graphical design environments and 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.

### 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. Configuration property-sheets are created dynamically based on component's reflection features.

## 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

Most component models define a notion of *provided interfaces*. Furthermore, some component models introduce a notion of *required interfaces*.

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. An example of the required attribute can be seen in Figure 1. Furthermore, for property arrays and collections we support a minimum and a maximum number of instances that must be set for the property.

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.

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 (see Figure ref:RequiredUsage.

### 3.2 OCL Constraints

The application of pre- and postcondition in programming is widely accepted. Unfortunately, only few programming languages such as Eiffel [9] 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 constraint expressions since it can be parsed easily, can be learned quite fast, and is simple to understand. Figure 2 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 attributes. The OCL constraints are provided as string parameters for the attribute classes. Unfortunately, is is not possible to execute the attribute constructor at compile time to verify if the expression is a syntactic valid OCL expression.

At the bottom of figure 2 we show how OCL constraints can be verified by calling a static method of the `OCLCheck` class we have implemented.

```
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. 1.** Required attribute

```
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. 2.** 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 [10,11]. 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 3 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(IReader))]
public interface IProvider
{
  [Transition("Closed", "Open")] void Open();

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

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

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

**Fig. 3.** Protocol specification

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

The .NET component model defines some metadata attributes for layouting and arrangement of .NET widgets and components. Some of these attributes are used in conjunction with the .NET propertysheet 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 propertysheet in Visual Studio .NETs designer and allows modification of those properties. When a developer selects a cell in the propertysheet the environment detects if an editor attribute is attached to the datatype or the property.

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.

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 with a multiplicity larger than 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. We have implemented a code serializer that generates an appropriate source code fragment for initializing the required component compositions. The code serializer generates the appropriate initialization statements for arrays and collections and inserts it into the predefined code generation stream provided by .NET.

### 4.2 Verification Adaptors

For automatic evaluation of constraints described with the attributes defined above we generate adaptors that include verification code. These adaptors include checks if required properties are bound, if method arguments satisify preconditions or method results satisfy postconditions, and if a protocol sequence is still satisifed. 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 adaptor initialization code instead of field assignments. Figure 4 shows an example for such a code fragment. 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
// this.RequiredProperty = required1;

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

**Fig. 4.** Adaptor Initialization

The adaptors are generated on demand. Here we use another .NET feature for dynamically creating or loading assemblies.

## 5 Related Work

The use of metadata beyond type information is frequently used within some component models. Enterprise JavaBeans [12] rely on metadata stored in deployment 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. 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.

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 [9] with its support for Design-by-Contract [13]. For Java different approaches implement pre- and postconditions such as JML [14] 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 .NET attributes for constraints has already been described before in [15].

## 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 layouting 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 containers to draw graphical representations for the compositions of components.

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 composition environments such as the Component Workbench [5] or Eclipse requires more effort.

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 and traverses recursively all child components and checks if all constraints are fulfilled.

## 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 refering to small helper classes that implement the validation semantics.

## 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. Richter, J.: Applied Microsoft .NET Framework Programming. Microsoft Press (2002)
8. Johann, O.: Webpage: .NET Metadata Facilities (2005) http://www.infosys.tuwien.ac.at/Staff/joe/dotnet-metadata.html.
9. Meyer, B.: Object Oriented Software Construction. Prentice Hall (1997)
10. 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
11. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. **19** (1997) 292–333
12. DeMichiel, L.G., Yal cinalp, L.Ü., Krishnan, S.: Enterprise JavaBeans Specification, Version 2.0. Sun Microsystems. (2001) Proposed Final Draft 2.
13. Meyer, B.: Applying Design by Contract. IEEE Computer **25** (1992) 40–51
14. 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
15. Sjörgen, A. In: A Method for Support for Design By Contract on the .NET platform. Artech House Publishers (2002) 12–20