

The Vienna Component Framework

Enabling Composition Across Component Models

Johann Oberleitner Thomas Gschwind Mehdi Jazayeri
Technische Universität Wien
Institut für Informationssysteme
Argentinierstraße 8/E1841
A-1040 Wien, Austria
{joe,tom,jazayeri}@infosys.tuwien.ac.at

Abstract

The Vienna Component Framework (VCF) supports the interoperability and composability of components across different component models, a facility that is lacking in existing component models. The VCF presents a unified component model—implemented by a façade component—to the application programmer. The programmer may write new components by composing components from different component models, accessed through the VCF. The model supports common component features, namely, methods, properties, and events. To support a component model within the VCF, a plugin component is needed that provides access to the component model. The paper presents the VCF's design, implementation issues, and evaluation. Performance measurements of VCF implementations of COM, Enterprise JavaBeans, CORBA distributed objects, and JavaBeans show that the overhead of accessing components through the VCF is negligible for distributed components.

1. Introduction

A primary goal of component-based software engineering is to promote the use of reusable and pre-tested components across projects. As in other engineering disciplines, in which components are well established the reuse of existing parts leads to shorter development cycles, higher quality, increased functionality and hence reduced costs.

In recent years component models such as Enterprise JavaBeans, CORBA objects, and COM+ components have emerged that provide standards for component implementation and component interoperability [12]. Additionally, component models provide services and infrastructure to components such as a meta-information facility, naming and trading services, and transaction monitors. Hence, com-

ponent developers can use these predefined services and rely on the vendor of a component model implementation to get this support.

The implementation standards define how a component's external interfaces are accessed. These interfaces are the only way to access a component's functionality, such as its operations or its state, from outside the component, hence enforcing the principle of information hiding.

A component model is an indispensable element in a component-based software technology. However, strict standards are also a limiting factor in a component-based software environment. Although all component models define similar features such as methods, properties and events, a standardized way does not exist for implementing a component for a particular model and for porting it to a different component model. Even worse, the significantly easier problem of using a component that is implemented for one component model from a component of another model is only solved for certain pairs of component models.

In an ideal setting, a developer would be able to use the best components available without having to think about the component model they have been implemented for. Such an example could be a stock ticker application that uses a CORBA component for a stock quote service, a JavaBean component that provides an elaborate charting facility and Microsoft's Internet Explorer, a COM component, for displaying current market news.

To solve these composition problems, we have developed the Vienna Component Framework (VCF), a Java based class framework that allows the access of components across different models and the construction of new components in a platform independent way. Two different factors limit the reuse of component source code: the standards that define how a component has to be constructed and the dependency of the component on services provided only by a certain component model. Our framework abstracts both of

these within interfaces. These interfaces build a meta-model for component models. A particular component model is supported by writing a plug-in that queries the component model for its component's meta-information, builds a representation of the component and all of its features and provides the required functionality to access these features. So far, we have implemented plugins for COM, CORBA, Enterprise JavaBeans and JavaBeans demonstrating the extensibility of the framework.

The remainder of the paper is structured as follows. In Section 2 we discuss today's commonly used component models. On the basis of the commonalities and differences of these component models, we discuss the design and overall architecture of VCF in Section 3. Section 4 explains the steps necessary to add support for a new component model and Section 5 focuses on the construction of new components. The evaluation of our approach is shown in Section 6. Related work is described in Section 7. Finally, we draw some conclusions in Section 8.

2. Component Models

To understand how to abstract the features of component models into a uniform framework, one must first analyze the commonalities and differences among currently available component models. Knowledge of these differences and commonalities will help the reader to understand the design and architecture of VCF presented in Section 3.

2.1. COM+

Microsoft's Component Object Model (COM) [8, 17] is used heavily within Microsoft's operating systems. COM components are declared using Microsoft IDL (MIDL) that supports the description of COM component classes and interfaces. Unlike CORBA, interfaces define only methods. Properties are declared using setter and getter methods having special attributes attached to them. MIDL uses its own type system that is based on the C type system including pointers to interfaces. Components are usually implemented with C++ classes or with another COM-capable language. Recent additions to COM have been server-side facilities for load-balancing and transaction monitoring, better known as COM+. However, the client side programming model has remained the same.

Meta-information is provided by type libraries. These libraries can be constructed from the IDL with Microsoft's IDL compiler. To access a component dynamically the so-called `IDispatch` interface can be used. This interface provides a method to create an invocation dynamically.

2.2. CORBA

The Common Object Request Broker Architecture (CORBA) [20, 13] has been defined by the OMG to provide an object infrastructure for interoperability among different hardware and software products. A CORBA object is declared by writing an IDL file that contains the interface definition of the object. This interface takes the definitions of an object's operations and its attributes. The IDL file is compiled by an IDL compiler that generates client stubs and server skeletons for a given language.

The IDL has its own type system which is loosely based on C++. This type system is mapped onto the type system of a given programming language as defined by the corresponding language binding. In case of the Java programming language primitive types are converted to Java types and CORBA object types are converted to client stubs.

CORBA's communication model is based on object invocation where objects may reside locally or remotely. Each request to a CORBA object is processed by the client stub which forwards the request to an Object Request Broker (ORB) that is located on the host of the client. This ORB uses a communication channel to communicate with the ORB on the host of the object's server process. This second ORB forwards the invocation to an appropriate method in the server process.

The OMG has predefined many different services that can be used to enhance the functionality of a CORBA object at development time. A client can use the name service to look up an existing object. Other important services are responsible for transactions, persistence, notification and security. Meta-information for an object is available through an Interface Repository (IR) that provides programmatic access to information about objects. The interface repository usually obtains this information from the IDL files. This information together with Dynamic Interface Invocation (DII) can be used to construct and make dynamic calls at run-time.

2.3. Enterprise JavaBeans

The Enterprise JavaBeans (EJB) [5] component model is an essential part of Sun's J2EE environment and uses the type system of Java. EJBs are components that reside within a container on an application server. The implementation of an EJB consists of Java classes that are deployed in the container. Clients use an enterprise bean's home and remote interface to invoke its methods. The home interface defines methods to create or to look up component instances. The remote interface provides access to a given instance. To interact with an EJB component, the client first obtains a reference to the bean's home interface which the client can use to create a new component instance of the bean or to

look up an existing one. Both of these operations return a reference that implements the remote interface.

Enterprise JavaBeans can implement different concepts. Entity beans model business concepts that are represented in database tables. Session beans model a workflow and thus implement a particular task [18]. Usually, they are stateless and have no properties. Message-driven beans are similar to session beans but work in message-oriented middleware settings. They are not considered in this paper.

An EJB application server provides distributed services to their components such as a persistence service, a transaction service, and a security service. These services can be used in a programmatic way and in a descriptive way. Hence, it is not necessary to prepare a component to define transactions and security facilities at compile time since these features can be added by providing a descriptor file when a component is deployed within the application server.

Unlike other component models, EJBs do not support events. Additionally, there is no standardized means to find out at runtime if a component uses a particular service provided by the EJB application server.

2.4. JavaBeans

JavaBeans [11] is a simple component model that relies on the Java programming language. Unlike the other component models presented so far, it only supports components executed locally within the client's virtual machine. A JavaBean is a Java class that has a default constructor and supports the Java serialization mechanism.

JavaBeans support methods, properties and events. These can be defined using the following naming conventions [11]. Publicly accessible methods that have a `get` or a `set` prefix are considered to model property access. The name of the property is deduced from the method's name. A similar approach has been followed for events. All methods that do not fall into properties or eventsets are just methods of the JavaBean. These syntax guidelines, however, can be overridden providing a `BeanInfo` class that specifies which properties, events and methods are accessible from clients.

Since a JavaBean is just a Java class the component model uses Java's type system. An instance of a component is a normal Java object and hence clients access these instances like any other Java object. To query a JavaBean for its meta-information, however, introspection should be used instead of Java's reflection mechanism. Introspection automatically derives the available properties and events on the basis of the above naming conventions and makes use of a `BeanInfo` class if available.

2.5. Comparisons

In the last sections we have shown the characteristics of widely used component models. Of particular interest are the type systems of the different component models. JavaBeans and Enterprise JavaBeans rely on Java's type system, CORBA and COM, however, use their own type systems. CORBA object servers and clients use the type system defined by the OMG's IDL. CORBA's Java language binding provides a mapping to Java types. Unfortunately, a language binding for Sun's Java Virtual Machine does not exist for COM.

All component models provide dynamic invocation. Although the APIs differ considerably, they allow construction of requests dynamically at runtime. Metadata provides information about components and can be used to query the features of component models. The implementation of this metadata interface and the specifics of the information differ among the component models. All of the component models presented above, however, provide enough information to build an internal representation of a component's features.

3. Design

The purpose of the Vienna Component Framework (VCF) is to provide an API that allows for the use of multiple different component models in a uniform way. This enables the use of components implemented for different component models from within a single project without having to deal with the internals of the different component models. This increases the range of components available to software developers.

For the implementation of the VCF, we have chosen to use the Java programming language. The principles of this framework, however, apply equally well to implementations in other object-oriented programming languages.

3.1. Architecture

VCF supports several component models. Each component model is represented by a *plugin*. Support for a new component model can be added by implementing a new component model plugin. Each component model plugin has to provide the functionality to access the features of the corresponding component model. This includes the features for controlling the lifecycle of a component's instance, for making that instance persistent, and for accessing its state and operations.

Component model plugins are not used directly but instead are accessed through a façade component as shown in Figure 1. This allows the integration of new functionality that can be applied to any component model in the façade

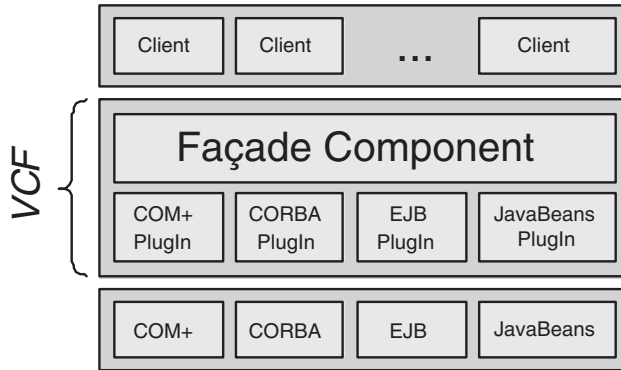


Figure 1. Architecture of the VCF

class without changing the plugins' syntactic or semantic structure [10]. Each instance of a particular component is hosted in an instance of the corresponding plugin class.

3.2. Metadata Interface

One key feature of today's component models is that they provide metadata that allows clients to identify the features that a component provides during run-time. Features identify the different means to interact with a component such as a property that can be changed or an event that may be triggered by the component. Features may statically apply to all instances of a component such as the lifecycle feature, or they may apply to individual instances of a component such as a property.

Since different component models use different features, a component model plugin only has to provide support for those features provided by the underlying component model. VCF provides support for the most commonly used features. A component model plugin, however, may provide support for additional features in case a component model requires a feature not provided natively by VCF.

Currently, VCF provides support for the following features:

Lifecycle provides methods to create and to explicitly destroy instances of a component.

Persistence allows a component instance to be stored on and retrieved from persistent storage.

Method gives access to the methods provided by a component.

Property allows the manipulation of a component's state.

Event allows other components to react to events generated by this component.

All features that are provided by a plugin are returned in a feature container returned by the the plugin's `getFeatures` method. This feature container provides standardized means to add and remove features, and allows queries for a particular feature. These queries range from retrieving all features of an instance to fine-grained queries like searching for all methods that have a particular return type and whose names match a regular expression.

3.3. Component Access

Components may be accessed through the metadata interface or through component stubs created by VCF. The metadata interface allows a component to be queried for its features similar to Java's reflection API. The classes representing the individual features provide methods to access the functionality provided by a feature implementation. Using a component's metadata interface is necessary to interact with components that are loaded during run-time. For instance, this allows application builders to discover, instantiate, and manipulate arbitrary software components. A sample class using VCF's metadata interface is shown in Figure 2.

```
public class Test {
    protected Component anInstance;

    void createComponent (
        ComponentDescription desc) {
        anInstance=Factory.create (desc);
    }
    void accessComponent () {
        IFeatureContainer fc=
            anInstance.getFeatures (
                new PropertyQualifier ("text"));
        IProperty property=fc.firstElement ();
        property.setValue ("new_Text");
    }
    String retrieveValue () {
        IFeatureContainer fc=
            anInstance.getFeatures (
                new PropertyQualifier ("text"));
        IProperty property=fc.firstElement ();
        return property.getValue ();
    }
}
```

Figure 2. Changing a Property through a Component's Metadata

Using these interfaces has two disadvantages: it is tedious to use for programmers and it imposes a performance overhead since several lines of code are required to access

```

public void printBook(String isbn) {
    // obtain a reference to an instance of the
    // COM library component
    COMLibrary library=new COMLibrary (isbn);

    String [] authors=library.getAuthors ();
    String title=library.getTitle ();
}

```

Figure 3. Sample Code - Interface Access

a feature. Even worse, the access is not statically typed which can lead to programming errors which are hard to find. Hence, to overcome these deficiencies we have provided a facility that generates stub classes providing direct access to the component. A sample method using such a stub is shown in Figure 3.

Although some component models already provide Java interfaces to access their components' features, only methods are accessed in a uniform way. Events and other complicated features are accessed syntactically and semantically in different ways. Our framework maps these different APIs to a single one. If a Java interface exists for a particular component model, the stub classes use these methods to avoid the performance overhead of the metadata interface.

3.4. Typing

Like all of the widely used component models, VCF uses strongly typed components. Different component models, however, use different type systems. Hence, to enable interoperability between these component models, VCF converts between the different type systems used by the different component models.

Type conversion can be effected by simulating the different data types used by the individual component models. While this approach ensures a minimal loss of information, it imposes a huge performance overhead since every operation on a primitive type would have to be simulated by a user-defined type. Additionally, it would require a conversion routine for every type of a component model to the corresponding types of all of the other component models, or otherwise the loss of information would not be minimal.

Due to these problems, we have chosen to convert every type to and from its Java counterpart. This limits the number of conversion routines to two times the number of component models and allows us to encapsulate the conversion routines within the plugin that supports a given component model. Most primitive types such as number values, booleans or character strings can be converted easily across different type systems. Arrays and records that contain no operations can be converted to Java arrays and Java classes respectively. However, we encountered several problems.

Java does not support unsigned data types. Hence, the use of such unsigned data types in COM or CORBA could lead to a loss of precision when converted to the corresponding signed data type in Java such as converting an unsigned long to a Java long. If such data types are used the full value range is used rarely. However, we made the conversion routines exchangeable at component instantiation time to let VCF users decide which conversion routines shall be used by the plugin.

Another problem is that Java does not support to pass method parameters by reference. Some component models, however, such as COM and CORBA support outgoing parameters and require such parameters. Although Java's call-by-value semantics does not prohibit the modification of object parameters some Java classes do not permit such a modification. To solve this problem we create holder classes automatically for the corresponding Java classes. A similar approach has been taken by CORBA and the Java language binding [20].

Of particular interest with typing issues are types that transport a component's instance pointer or reference. We can distinguish two different occurrences of this. When a reference to an instance is returned to a component's client it has to be converted into a VCF component reference. This is the responsibility of the component model's plugin. Each plugin examines the reference stored in the plugin's component model format, e.g. a COM pointer. If a corresponding VCF component exists already a reference to this component is returned. Otherwise a new VCF component is created, the reference to the native component instance is stored and this instance is analyzed by the plugin.

Passing a component reference to another component is more difficult. When the passed component reference is built for the same component model as the target component the problem can be reduced to extracting the component's reference from the plugin. Other cases are more problematic. In general it is necessary to use a proxy [10] component for the target component model that delegates calls back to the passed component. Currently we have built these proxy only for JavaBeans and EJB. A proxy for COM would need similar mechanisms to the connection point mechanism described in Section 4.4. Similarly a CORBA proxy can be realized by using CORBA's Dynamic Skeleton Interface (DSI) [20].

4. Implementing a Component-Model Plugin

This section discusses the implementation choices for a component model plugin and those that we have made. The description is intended to help the reader understand the work involved in implementing a new component model plugin.

A plugin for a component model consists of one main

plugin class that implements the `IComponentPlugIn` interface. This class creates the features responsible for lifecycle control and persistence in its constructor. All features that are provided by the plugin are returned in a feature container by the the plugin's `getFeatures` method. This feature container provides standardized means to add and remove features, and allows queries for a particular feature. These queries range from retrieving all features of an instance to fine-grained queries like searching for all methods that have a particular return type and whose names match a regular expression.

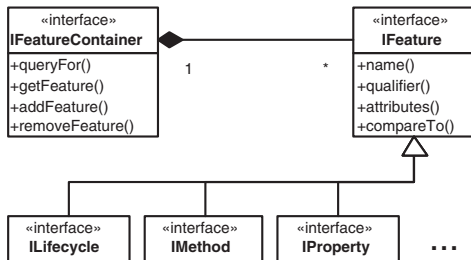


Figure 4. Features

As shown in Figure 4, each type of feature is represented by an interface that extends the `IFeature` interface. Additionally, each feature contains a qualifier that unambiguously distinguishes instances of a feature from each other and may have custom attributes associated with it. A custom attribute can be used to store information about the feature without modifying its functionality. Clients can query for attributes and change their behavior when they detect the existence of such attributes. For instance, GUI tools could hide a feature from the user if its “user level” attribute has the value “expert.”

Unlike CORBA, COM does not provide a language binding for the Java programming language except for Microsoft’s Virtual Machine that is no longer supported. Hence, we access COM using Java’s Native Interface (JNI), Java’s facility to access operating system dependent native code.

4.1. Lifecycle

The lifecycle of a component is controlled with the `ILifecycle` interface (Figure 5). This interface provides methods to create and destroy components. The create method takes a parameter that is provided by clients to specify which component should be instantiated as well as an additional initialization parameter to initialize the component instance.

Once an instance has been created, each lifecycle feature informs the plugin class to analyze the metadata of the

```

interface ILifecycle extends IFeature {
    void create (InitParameter parameter);
    void destroy ();
}

```

Figure 5. Lifecycle feature

newly created component. During this analysis an instance for each feature implemented by the component is created.

The destroy method invokes the appropriate component model specific mechanisms to destroy a component instance, and invalidates the contained instance reference.

4.2. Methods

The `IMethod` interface (Figure 6) provides information about a single method of a component. It provides methods to query for the return type and the parameter types of the method. Parameters have methods to set and retrieve their values. The invoke method takes an array of parameters and invokes the method.

```

interface IMethod extends IFeature {
    Object invoke (IPParameter [] parameters);
    Class getReturnType ();

    Class [] getParameterTypes ();
    IPParameter [] getParameters ();
}

```

Figure 6. Interface for methods

Since all component models support methods, the implementation of `IMethod`’s functionality is straightforward. JavaBeans and Enterprise JavaBeans methods are invoked via Java reflection. CORBA’s Interface Repository provides all necessary information to look up a method’s name, its return type and its parameter types. Invocations are implemented using CORBA’s Dynamic Invocation Interface (DII). COM’s type information provides information about COM methods. To invoke a COM component method we use COM’s `IDispatch` interface. To access this interface we access a C++ DLL that is invoked with Java’s Native Interface (JNI). Since C++ and Java have different binary representations for data types all types are converted by the plugin at runtime.

4.3. Properties

The state of components that is externally visible can be accessed with the `IProperty` interface (Figure 7). The

methods of this interface are responsible for setting and retrieving a property's value. In addition a method for returning a property's type is available. Another method is used to determine if the property is read-only.

```
interface IProperty extends IFeature {
    void setValue (Object value);
    Object getValue ();
    Class getType ();
    boolean readOnly ();
}
```

Figure 7. Interface for properties

In the case of JavaBeans, certain methods are tagged by syntax conventions to be used as property accessors. Although this is not strictly defined for Enterprise JavaBeans we use the same naming conventions for EJBs too. The access provided by CORBA is different. CORBA interfaces provide attributes that can be used to model properties explicitly. We use the information from the interface repository about these attributes to implement the functionality of the IProperty interface. Similar to the implementation of CORBA's method feature the accessor methods are called with DII. COM does not provide attributes as CORBA does. But it uses attributes that mark a method as responsible to get or set a property's value. Hence, we have used these methods to implement properties for COM.

4.4. Eventsets

Eventsets are used to implement callbacks. Component instances send events to their clients leading to the invocation of a method of the client. We followed the design of JavaBeans and used listeners to realize this. As shown in Figure 8 listeners can be added and removed. Additionally, the IEventset interface provides methods that return information about listener methods and the type of the listener class. The listener interface has to be implemented by clients and is called by the component instance when a client has to be notified about the occurrence of an event.

```
interface IEventset extends IFeature {
    IMethod[] listenerMethods ();
    Class getListenerClass ();

    void addEventListener (Listener l);
    void removeEventListener (Listener l);
}
```

Figure 8. Interface for eventsets

The implementation of this feature was more challenging than the implementation of methods or properties. First,

the mapping is not always as straightforward as in the case of methods. Second, since the client component and the server component change their roles when using callbacks it was necessary to provide a listener class for the client that is compatible with the listener interface specified by the component.

In the case of JavaBeans, the mapping just forwards the calls for registering listeners to the bean instance. Enterprise JavaBeans do not provide events. Although it might be possible to use Java's Message Service (JMS) to simulate events, we do not support this. For CORBA, we use CORBA's event service [23]. An appropriate listener interface is generated and compiled during the first instantiation of a component. COM's analog to an eventset is a connection point [3]. COM components provide connection points that allow clients to subscribe to the events they can emit. Clients may either implement the callback interface or the IDispatch interface [17]. This means that it is not necessary to generate a class that implements the outgoing interface but it suffices to implement the invoke method of the IDispatch interface [3]. This method takes the id of the target method and all parameters to provide the required functionality. From the type information stored in the type library we create Java interfaces for all outgoing interfaces. If not already created and compiled this interface is created at instantiation time of a component. We have implemented an event sink in C++ that forwards the events with JNI to the appropriate listener classes that are implemented in Java.

4.5. Persistence

Persistence allows clients to store a component's state on, and load it from, persistent storage. We have provided the IPersist (Figure 9) interface that provides this functionality. Desktop component models sometimes provide an explicit facility to store an instance data onto persistent storage. Hence, we have encapsulated Java's serialization mechanism and COM's persistence interfaces into IPersist to save and restore instance data from a data storage.

Server-side component persistence that stores instances in relational databases such as EJB persistence is implicit and usually transparent to clients. Hence, for these components, we only store enough information to reconstruct the corresponding instance.

```
interface IPersist extends IFeature {
    void load (Storage s);
    void save (Storage s);
}
```

Figure 9. Interface for persistence

4.6. GUI

To provide a uniform GUI representation of components of different component models we have defined a feature interface that returns a `java.awt.Component` object that displays the component's visual representation. In case of JavaBeans this is the bean instance itself. Enterprise Beans and CORBA objects do not provide a GUI at all, hence the interface is not implemented and will not be found by the feature container. COM provides ActiveX controls that are extensively used within Microsoft's operating systems. Java AWT provides means to get operating system handles of its windows. We use this to create an ActiveX host window inside the AWT window. Finally, we put the ActiveX control inside this host window.

4.7. Summary

To implement a new plugin for a component model, the following steps have to be taken. First, the plugin class has to be implemented. This class creates the feature container where it registers all features that can be used to instantiate a component instance, such as the lifecycle feature or the persistence feature. For each type of feature supported by a component model a class that encapsulates the functionality of the feature has to be implemented. This class should implement the VCF interface that corresponds to the type of feature. If necessary, however, a new type of feature can be added just by defining a new feature interface and a qualifier to identify that feature. Once a component has been created, either by the lifecycle feature or by the persistence feature, its metadata can be analyzed. For each entity found in the metadata the appropriate feature class is instantiated and added to the plugin's feature container.

5. Component Construction

The Vienna Component Framework provides a generic programming model to build new components. This programming model allows the programmer to write regular Java classes to build new components. These Java classes and their source code are used to generate source code for any component model supported by a plugin that implements the code generation feature. Hence, the effort necessary to implement components for Enterprise JavaBeans, or CORBA is reduced considerably. We call the classes implemented for our programming model VCF metaclasses since they are the building blocks for new components.

The programming model we use has been influenced by the naming conventions for JavaBeans. These guidelines provide conventions for developers to define properties and eventsets within the Java programming language but without introducing additional language constructs. We have

extended these conventions to features not covered by the JavaBeans syntax guidelines such as lifecycle and client-side persistence. In particular, the naming conventions for feature methods consist of its feature name and the features qualifier, for instance the lifecycle destroy method would be named `LifecycleDestroy`. Features that are already available in JavaBeans use those naming conventions.

To support the construction mechanism, we have implemented a special component plugin that enhances the JavaBeans plugin. Its purpose is to parse the meta-information provided by the metaclasses and to parse their source code files. Each feature instance that is created has also some corresponding source code fragments. Hence, we attach these source code fragments to the features as an attribute. Finally, we have an enhanced representation of a component with its features and with the source code fragments that contain the implementation of the features.

Each component model discussed in this paper has its own programming model. During code generation the enhanced representation is used to construct source code for these programming models. All features are processed iteratively. For each feature the source code fragments stored within the attributes are used to generate the required methods, and fields in the created source files.

Since our metaclasses are based on the JavaBeans syntax conventions, the existing class can already be used as a JavaBean. In case of CORBA, we generate the OMG IDL file, and the CORBA object server implementation as Java classes. Java types have to be converted to appropriate CORBA types. In case of Enterprise Beans we generate the home interface, the remote interface, the bean implementation class and the XML deployment descriptor. It is not possible to access recent Java versions from COM components in a straightforward way. Hence, we generate a basic COM component based on C++. The signatures of the methods in the C++ code and in the MS IDL file are generated from the features in the internal representation of the metaclass. Inside the C++ code we make a lookup if a Java VM exists in the current process. If not, we start a virtual machine with means provided by Java's Native Interface. Inside this virtual machine we instantiate an instance of the metaclass and forward all calls to the COM component to these methods. This delegation code also includes the necessary routines to convert between the COM and the Java type system.

Since our framework can be used inside metaclasses it is easily possible to build composite components that contain other components, possibly of different component models. In particular it is possible to encapsulate Enterprise JavaBeans in COM+ components and vice versa.

6. Evaluation

We have evaluated VCF with respect to its genericity and the performance it offers. As we have presented in Section 4, we have already implemented VCF plugins for the COM, CORBA, EJB, and JavaBeans component models. With the implementation of these plugins, we have demonstrated that VCF is generic enough to support the integration of the most commonly used component models. Additionally, we have started the implementation of plugins for X11 applications, SOAP web services [2], and Gnome Bonobo components. Although the implementation of these plugins has not been completed our current results are promising.

To demonstrate VCF's ability to enable composition across multiple different component models, we have implemented the stock ticker application mentioned in Section 1 which combines CORBA, JavaBeans and COM components. Although Java has some support for CORBA we have been able to remove the code necessary for connecting to a CORBA name server and the code responsible for making a connection to a CORBA event channel [13]. Similarly, the access for COM components was as simple as accessing normal Java classes. Using JavaBeans in VCF does not lead to any reduction in code size. Fortunately it does not increase it either. The uniformity across all component models remained the same for all three component models used.

To test the performance provided by VCF, we have implemented a test component for each of the component models currently supported by VCF. These components provide 4 operations.

ping taking no argument and returning nothing.

pong taking a string as argument and returning the same string.

upper taking a string as argument and returning its uppercase representation.

concat taking five strings as argument and returning the concatenation of them.

Our COM server side component using COM+ was running on a Pentium II/300MHz processor with 128MB memory and Windows 2000 and the CORBA and EJB components were running on a Duron/800MHz processor with 256MB memory and Linux 2.4.7. The JavaBean component as well as our clients were running on a Pentium III/550MHz processor with 256MB memory and Windows 2000. For each operation and component model, we have executed three runs with 10000 executions per run. The average of the three executions is shown in Table 1.

For the COM plugin the overhead is about 20%. One reason for this gap is that the native COM implementation

	COM+	CORBA	EJB	JB
ping (native)	6849	12799	33638	0
ping (VCF)	15031 (6970)	13349	33939	30
pong (native)	17495	20560	38165	100
pong (VCF)	21551 (18006)	21301	38586	150
upper (native)	17595	20289	38536	60
upper (VCF)	21631 (18056)	20980	38605	310
concat (native)	35542	31085	42828	130
concat (VCF)	40508 (38072)	32106	43833	581

Table 1. Measured access times (ms)

was implemented using C++ and hence unlike with VCF, COM's types did not have to be converted to Java. After this test, however, we optimized the COM plugin. The performance values of the optimized version are shown in parenthesis.

For CORBA and EJB the performance difference is only about 5%. Unlike for COM the native tests for CORBA and EJB have been implemented using Java explaining the smaller performance overhead compared to our COM evaluation.

As shown in Table 1 using the JavaBeans component with VCF imposes a considerable performance penalty. The overhead is much higher compared to the other component models since JavaBeans are executed within the same application as the client and do not incur the overhead of a remote procedure call. Another reason is that the JavaBean plugin still uses Java's reflection API for the generation of the component's stub classes. Hence, it should be possible to eliminate much of this performance overhead by using native Java calls inside the component stubs.

7. Related Work

The design of VCF has its origin in the generic component model of the Component Workbench, a visual component builder developed by the authors [19]. VCF, however, is more powerful from this former model in many aspects such as the component stubs enabling direct access or the component construction mechanism. Hence, we have built a new release of the Component Workbench that is based on VCF. We are going to integrate all concepts of VCF that are not covered by the original model such as component construction into the Component Workbench. A survey of composition environments can be found in [15].

The Eclipse Platform allows the construction of integrated development environments (IDEs) for different application types such as web sites [6, 7], Java or C++ applications. Eclipse does not provide uniform access across different component models such as VCF. However, Eclipse provides a Standard Widget Toolkit (SWT) [6] that combines a platform independent widget library with platform

dependent facilities such as ActiveX controls. We plan to port the Component Workbench to Eclipse in the future.

Interworking specifications support the integration of middleware systems of different kinds [9]. Usually these technologies are restricted to combine two distributed component models such as Interworking between CORBA and COM [9] and are not available for all pairs of component models.

UniFrame [21] is an approach that aims to achieve interoperation of heterogeneous and distributed software components. It provides a metacomponent model that allows the access of various component models, support for the integration and validation of quality of service on an individual component and distributed system level. The UniFrame approach also facilitates the use of generative rules for assembling components out of available choices. One essential part of this approach is the UniFrame Resource Discovery Service (URDS) [24] that provides a solution for the discovery of heterogeneous and distributed software components. URDS has powerful facilities to act as a component trader over the Internet. UniFrame supports Java RMI objects. Efforts have started to integrate other component models [24]. UniFrame, however, does not support the generation of components.

Flexible Packaging [4] defines how a software component's source code has to be structured to defer some decisions about interaction semantics with other components until integration time. Only the essential parts of a component have to be specified early in the design cycle while details can be deferred. Flexible Packaging splits up a component's functionality and its packaging into, respectively, a ware part and a packager part. This separation allows that a component's functionality to be packaged in components that can be used with different architectural styles. Flexible packagers make use of an extension of the C programming language to support the use of typed channels for communication between wares and packagers. This removes the dependencies between packagers and wares and allows the construction of packagers out of description files. While flexible packaging allows the construction of component packages independently of the ware for almost all imaginable component models there is no automatic support for the use of components without writing packaging descriptions. Hence, it is not possible to reuse components where only metadata is available such as VCF's support for component use. On the other hand VCF's component construction is specialized for COTS component models that support common features such as methods and events as described in Section 2.

Jiazzi [16] provides a component infrastructure for Java that enables the construction of large-scale binary components with Java. Jiazzi does not use any core language extensions or language conventions to construct a component.

Instead it uses separate source files that describe the visible structure of classes in packages. Jiazzi supports the use of mixins and the open class pattern. However, it supports only the construction of components that conform to the Java class format. Hence, these components can be used from within other Java-based component models but cannot be used as an Enterprise Bean or a CORBA component in their own right.

ArchJava [1] extends the Java programming language with component and connector constructs. These constructs enhance the Java type system and the ArchJava compiler checks if components can be connected by existence of component ports. However, component type checking works on the compiler level. The ArchJava compiler creates true byte code compatible with normal Java virtual machines. But ArchJava ports can only be used with other ArchJava components. Hence, the use of ArchJava with industrial component models is limited.

Unlike VCF, Jiazzi and ArchJava do not provide uniform access to existing component models nor do they provide facilities to generate component code for other component models. On the other hand, it is possible to build VCF plugins for these environments.

Caboom is a product developed by CalKey Technologies [14] for rapid design and development of component-based enterprise applications. It allows the specification of components in UML with OMG's Model Driven Architecture. Caboom is able to generate ready-to-deploy components and whole multi-tier applications for the J2EE, .NET and COM+ platform. Unlike our framework they generate class code for various programming languages not out of one template class but use UML diagrams created by design tools as starting point for code generation. Caboom does not facilitate portable modification of source code after its generation. However, modifications that are applied to the UML diagram are portable to the models.

Our framework supports modification within a component by the use of metaclasses implemented in Java. Although our framework does not support the use of UML diagrams it has the advantage that the template code is a Java class that can be tested and debugged before exporting its functionality to other component models.

8. Conclusions

In this paper, we have presented a framework that enables the composition of software components that have been developed for multiple different component models. From a practitioner's point of view, such a framework allows software developers to choose from a wider range of software components without having to know the details of all the component models available today. They only need to become familiar with a single component framework,

giving them more time to focus on the task at hand. From a researcher's vantage point, such a framework provides a basis for the detailed comparison of different component models, both at an implementation level and from a performance point of view. Although component models are conceptually similar, the implementation requirements impose significant differences in their usage, performance, and inherent interoperability. Most research work to date only considers individual component models, rather than cross-model issues.

To evaluate our claims that VCF is able to integrate most of the component models available today, we have implemented plugins for COM, CORBA, Enterprise JavaBeans, and JavaBeans as we have shown in Section 4. Using this infrastructure, we have been able to implement a small application using components that have been implemented for different component models.

We have evaluated the performance of the VCF. As shown in Section 6, VCF imposes only an overhead of—depending on the server-side component model—about 5–20% for server side components compared to accessing the components using their native component models. Additionally, as we have shown with the COM plugin, it should be possible to optimize the generated stubs and subsequently to reduce the performance overhead to about 2–5%.

In future versions, we plan to focus on the optimization of accessing components through VCF and to focus on distributed services such as transactions and security. Furthermore we plan to build additional plugins for web services accessible via SOAP [2] and for classes built for Microsoft's .NET framework [22].

Acknowledgments

We gratefully acknowledge the financial support provided by the European Union as part of the EASYCOMP project (IST-1999-14151) and an IBM University Partnership Award from IBM Research Division, Zurich Research Laboratories. We appreciate the helpful comments and suggestions of Heinz Appoyer, Uwe Zdun and the anonymous reviewers for their helpful comments and suggestions.

References

- [1] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, 2002.
- [2] D. Box et al. *Simple Object Access Protocol (SOAP) 1.1*. W3C, May 2000.
- [3] K. Brockschmidt. *Inside OLE*. Microsoft Press, second edition, 1995.
- [4] R. DeLine. Avoiding packaging mismatch with flexible packaging. *IEEE Transactions on Software Engineering*, 27(2):124–143, Feb. 2001.
- [5] L. G. DeMichiel, L. Ü. Yal cinalp, and S. Krishnan. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems, Apr. 2001. Proposed Final Draft 2.
- [6] Eclipse platform technical overview. Technical report, Object Technology International, Inc., 2001. <http://www.eclipse.org/whitepaper/eclipse-overview.pdf>.
- [7] <http://www.eclipse.org>.
- [8] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.
- [9] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, 2000.
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [11] G. Hamilton, editor. *JavaBeans*. Sun Microsystems, <http://java.sun.com/beans/>, July 1997.
- [12] G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [13] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley Longman, Inc., 1999.
- [14] B. Jadhav. *Caboom White Paper*. Campbell, CA 95008, 2001. <http://www.calkey.com>.
- [15] C. Lüer and A. van der Hoek. Composition environments for deployable software components. Technical Report UCI-ICS-02-18, Department of Information and Computer Science, University of California, Irvine, Aug. 2002.
- [16] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fashioned java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '01)*, pages 211–222, 2001.
- [17] Microsoft Corporation. *The Component Object Model Specification*, 1995.
- [18] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., first edition, June 1999.
- [19] J. Oberleitner and T. Gschwind. Composing distributed components with the component workbench. Technical Report TUV-1841-2002-17, Technische Universität Wien, Jan. 2002. Accepted for publication in the Proceedings of the 3rd Software Engineering and Middleware Workshop.
- [20] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.4 edition, 2001.
- [21] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, and C. Burt. A unified approach for the integration of distributed heterogeneous software components. In *Proceedings of the Monterey Workshop on Engineering Automation for Software Intensive System Integration*, pages 109–119, 2001.
- [22] J. Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.
- [23] J. Siegel. *CORBA 3: Fundamentals and Programming*. John Wiley & Sons, Inc., second edition, 2000.
- [24] N. N. Siram, R. R. Raje, A. M. Olson, B. R. Bryant, C. C. Burt, and M. Auguston. An architecture for the uniframe resource discovery service. In *Proceedings of the 3rd International Workshop on Software Engineering and Middleware 2002 (SEM 2002)*, pages 22–38, 2002.