

Composing Distributed Components with the Component Workbench

Johann Oberleitner and Thomas Gschwind

Technische Universität Wien
Institut für Informationssysteme
Argentinierstraße 8/E184-1
A-1040 Wien, Austria
{joe,tom}@infosys.tuwien.ac.at
<http://www.infosys.tuwien.ac.at/Staff/joe/>
<http://www.infosys.tuwien.ac.at/Staff/tom/>

Abstract. Although software components have gained importance, support for the composition of distributed components is still limited. Worse, if components implemented for different component models need to interact with each other, the composition process becomes a nightmare. Though, bridging technologies for different component models have been standardized, almost no implementations of these exist so far. In this paper, we present the Component Workbench (CWB), a flexible toolkit for the composition of components. Due to CWB's modular design and a generic component model used for the internal representation of the components, it supports the composition of components implemented for different component models.

1 Introduction

Component models support the developer in the design and implementation of components adhering to a common architectural style [18, 11]. Using components developed for the same component model eases the task of the developer because it helps to avoid architectural mismatch [10] if components developed by different vendors are being used.

Today's component models can be distinguished between desktop component models [16], which are also referred to as local component models [8], such as JavaBeans or ActiveX controls and distributed component models [8], such as the CORBA Component Model (CCM), Enterprise JavaBeans (EJBs), or COM+. Desktop component models are typically used in combination with integrated development environments (IDEs) and supply the programmer with user interface elements such as buttons or list boxes. In the following, however, we focus on distributed component models.

Distributed component models are based on middleware technologies providing fundamental services that conceptually operate between the operating system layer and the application layer [2]. Such services provide transaction

management, persistence, or security services, hence simplifying the implementation of distributed components rather than facilitating the implementation of clients using these components.

Components implemented for different component models cannot easily be mixed due to the models' type systems and APIs that differ considerably. Additionally, some component models, such as the Enterprise JavaBeans (EJB) component model, are tied to a specific programming language [5]. While in theory bridging technologies exist to solve some of these problems, in practice their use is rather limited. For instance, they provide no support for transactions across component model boundaries. To solve these challenges, we have developed the Component Workbench which will be presented in the following sections.

The outline of the paper is as follows. In Section 2, we present our terminology. In Section 3, we present the interworking problem of today's component models. Section 4 presents the Component Workbench and how we have solved the above mentioned problems. An evaluation of the design based on a library administration application is given in Section 5. Future work is presented in Section 6 and related work in Section 7. Finally, we draw our conclusions in Section 8.

2 Terminology

Different researchers have adopted different definitions of a component. Hence, for reasons of clarity, the following paragraphs present the terminology used within this paper. This terminology has been mostly adopted from [4, 22].

A software *component* is a software element that conforms to a component model and can be independently deployed. Distributed components are provided by Enterprise JavaBeans (EJBs), the CORBA Component Model (CCM), or COM+. A component's client interacts with the component through interfaces. These *interfaces* provide an abstraction of the functionality of the component. Typically, such an interface provides a possibility to invoke a component's operation, read or change a property of the component, or to inform the component about event handlers where the component can notify its clients about the occurrence of events. In this paper we refer to operations, properties and events of a component collectively as features.

A *component model* defines the basic architecture of a component, specifying the structure of its interfaces and the mechanisms it uses to interact with its container and with other components. The component model provides guidelines to create and implement components that can work together to form a larger application. Different organizations have defined and realized different component models.

Application builders are tools that can combine components from different developers or different vendors to construct an application. Recently, application builders from commercial vendors provide support for distributed component models along with associated services.

Interoperability denotes how different implementations of the same middle-ware specification work together. *Interworking* means the integration of middle-ware systems that implement different specifications [7].

3 The Interworking Problem

The problem of today's component models is the lack of interworking of components implemented for different component models. While components implemented for the same component model can be composed easily by implementing pieces of glue code, this is not the case for components implemented using different component models.

Theoretically, it should be sufficient to only know the interface of a component and to be able to use it like any other component. In reality, however, the composition requires the developer to be familiar with all the different component models being used. Hence, it requires the developer to deal with different type systems used by different component technologies. To be able to use an enterprise bean in cooperation with a remote COM+ component the developer has to write low-level code such as Java Native Interface (JNI). Microsoft's Java Virtual Machine (MS JVM) used to support COM+ but lacks support for RMI and unfortunately is no longer supported. Hence, implementing such an application is the ideal nightmare even for professional programmers.

To ease the integration of components developed for different component models, we have designed and implemented a new toolkit, the Component Workbench (CWB). While implementing the CWB, we had to solve the challenge to provide a coherent, easy to understand, and powerful interface for developers. No matter of the component model a component belongs to, the developer has to be able to use it using the same interface. The interface has to be as easy to understand as if the programmer were only using CORBA or EJB components. Yet, the interfaces provided have to be powerful enough to accommodate the different features provided by today's component models. Hence, it is not enough to restrict the developer to use only the subset of features available by all of the component models.

4 The Component Workbench

In this section the main parts of the architecture of the CWB are explained. As figure 1 shows the architecture consists of several layers where each layer sits on top of another.

For each component model there is a wrapper that realizes a common interface for the corresponding components. The design of the wrappers as well as the uniform component access are explained in section 4.1 and 4.2. On top of this layer we have arranged the scenario layer and the adapter facility of the CWB. Both are explained in section 4.3 and 4.4. The CWB tool layer consists of those parts of the CWB that are related to the user interface and the configuration of

the application. A description of the design of the user interface is presented in [16].

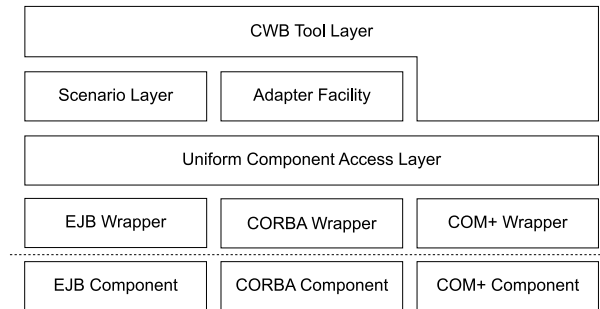


Fig. 1. Component Workbench Architecture

4.1 Component Wrapper

Each component model provides its own mechanism to access the features of a component. To provide uniform access across different component models we have specified *component wrappers*. Component wrappers create a consistent view of different component models onto a concrete internal component representation.

We have defined several categories of functionality that have to be supported by the component wrappers.

Instantiation: The instantiation functions are responsible for the creation of a component or the assignment of an already existing object to a component wrapper. For the instantiation process additional information has to be provided such as the application server that has to be used or the naming information that is required to connect to a component's application server.

Feature Access: Since every component model supports different feature categories the component wrapper has to provide methods to find out about the supported feature categories, and the elements of each category. For example it has to be possible to access all operations a component provides. A filter criteria can be used to restrict the required features to only a subset of the available ones. Additional information about a feature can be retrieved too, such as a set of allowed filter values for a particular feature category.

Graphical Representation: Graphical application builders need a way to show the components on the desktop. Since distributed components typically have no client-side GUI representation, the wrapper should provide a useful representation. The advantage of providing a visual representation by the component wrapper is that it can show additional information to the user that is

dependent on the wrapped component instance such as important component property values.

Configuration Panels: It must be possible to add components to a building tool as well as to configure the settings of a component model. This can be done with configuration panels provided by the component wrappers, too.

The internal design of a component wrapper (Figure 2) is a combination of the factory and the bridge design patterns [9]. The methods that have to be implemented by a component wrapper to provide the above functionality are specified in the `IComponent` interface. Within the CWB, this interface is used to access a component in a standardized way for all supported component models. Hence, if support for a new component model has to be added to the CWB, only the `IComponent` interface has to be implemented.

4.2 The Generic Component Model

The API of the generic component model resembles a reflective API that can be used for components of arbitrary component models. Since this is inconvenient for many situations we have provided a set of classes that help the programmer to find and access a particular feature. The *Uniform Component Access Layer* consists of these classes and a mechanism to load the parameters for the instantiation of a parameter in a consistent way across all component models. The most challenging task of this layer is the provision of a uniform type system.

Since different component models support different features, we have defined abstractions for the most frequently used features. We have provided interface definitions for the access of properties (attributes), methods (operations) and eventsets (callbacks). Each component wrapper has to provide implementations of these interfaces. The features provided by a component can be queried using the `IComponent` interface that has to be implemented for each component model. The implementations of these interfaces use the meta-information and the access mechanisms provided by the corresponding component model to provide access to the corresponding features. Figure 2 shows the relation between the `IComponent` interface, the interfaces describing various features of a component model and the implementation of the EJB component wrapper. Since the EJB component model does not support events there is no implementation of this feature category.

The `IComponentProperty`, `IComponentMethod` and `IComponentEventset` interfaces provide operations for the access of properties, methods, and eventsets accordingly. Among the functionality of these interfaces are property read/write access, method invocation, and eventset connection. It is possible to implement `IComponent` to add new features at runtime with the feature access mechanism.

The component wrappers shield the developer from having to do the complicated work himself. When user interface elements are placed onto the scenario, the Enterprise JavaBeans component wrapper uses reflection to query the components for the methods and properties they provide. COM+ components that are placed onto a CWB scenario make use of COM type libraries. CORBA ob-

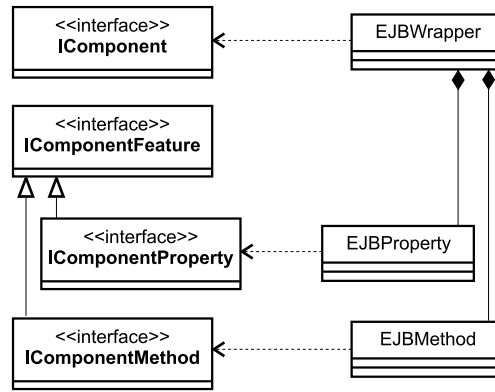


Fig. 2. Internal Design of a Component Wrapper

jects that are used within the CWB use CORBA's interface repository to obtain the properties and methods provided by the component.

4.3 Scenario

Components can be selected, configured, arranged and deployed in scenarios. Hence, a scenario acts as a container for the components. CWB modules such as user interface elements or the export generator access a scenario via the `IScenario` interface. We have provided an interface instead of a class to reduce coupling and provide more flexibility for future extensions. The scenario interface specifies different methods for adding, removing, or querying for a particular component instance.

Furthermore the scenario is responsible for the persistence of component configurations. Some attributes that are important for a component's configuration cannot be stored within a component such as the user-defined name of an instance within CWB or the geometry of the graphical representation. Though these properties are not relevant for the component itself, they are of relevance for the application built from these components. A scenario can attach arbitrary objects to a component within a scenario.

In the CWB a scenario is represented as a graphical diagram that contains all components with their connections. Within this diagram a component can be selected and modified. The appearance can be manipulated too. The components involved in a scenario are already functional when they are instantiated.

4.4 Adapter Facility

Once different components have been selected and placed onto a scenario, these components are connected by the user to interact with other distributed components or with GUIs. For this purpose a flexible adapter facility has been in-

tegrated into the CWB to provide for a variety of connection styles between components.

At the time of writing we support adapters that are capable of reacting on events that are emitted by a component and propagate these events by invoking a method of another component. A user interface wizard can automatically create these adapters. The adapter is compiled from the CWB, instantiated, and the appropriate components are connected with the use of the adapter and an eventset of the component that emits the events. After an adapter has been created it can be reused for other components or for different scenarios at all. At the time of writing we just support adapters that connect one method of a sending event — this is exactly what the untyped CORBA event service provides [13] — to one method of a target component. In future releases, however, we will try to create adapters that support more complex connection patterns, such as the mapping of whole event interfaces to target components, something the COM+ event service provides [15].

Since it cannot be assumed that the parameters of the events of the sender components will fit to method parameters, we have realized different ways for solving this problem. It is possible to change the automatically generated adapter code within a code window before it is stored and compiled, and we support *typed-based adaptation* that can be used to connect a chain of predefined adapters to connect components [12].

4.5 Export Generator

Once a component scenario has been designed and tested export generators can be used to create applications from scenarios that are capable of running without starting the CWB. We have provided a general interface to support a variety of export generators to cover different application types.

A simple variant of a code generator creates Java code that interacts with the application servers in a similar way as the CWB would do. In this case the necessary glue code to connect the wrappers is created and compiled. Optionally this code is packaged together with the required wrappers into a Java archive file.

A more sophisticated code generator simplifies how the components are accessed: instead of using the access functions of the component wrappers, code can be created that accesses a component directly. Depending on the model, Java code is generated or in case of COM+ Java code with some related C++ code accessed via JNI is created. Depending on the relevant component models different libraries have to be deployed for compilation and runtime access of the components and/or the calling code between two components.

Another export generator creates scripts in XML form that can be interpreted by a tool that acts as a player for these scripts. These scripts contain the configuration and connections of the involved components in a similar way as provided by the BeanMarkup Language (BML) [23] or the long-time persistence format [21] of the JDK 1.4. The advantage of our format is that it allows not only

the configuration of JavaBeans components but also the configuration of components of arbitrary component models. In all cases the exported functionality can be deployed on hosts that are used for business logic.

5 Evaluation

To verify the design goals of the Component Workbench, we have implemented a small business application that uses COM+ components and EJB components. For the developer using the CWB, however, all the properties, methods, and other features of a component look the same regardless of the component model a component adheres to. To demonstrate how the CWB supports the composition of components that have been built and designed for different component models, we implemented a library administration tool for our department. Figure 3 shows the architecture of the library administration tool.

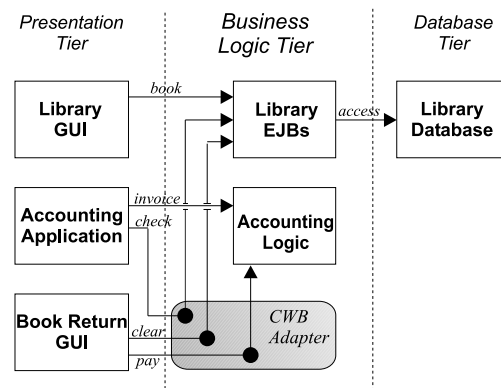


Fig. 3. Library Administration Architecture

For the implementation of the library administration tool, we used Enterprise JavaBeans handling the interaction between the library database and a web service used by students. The web service is driven by these enterprise beans. The library administration tool also has to interact with an accounting program based on COM and running on Microsoft Windows. The accounting program is used to deal with the fees applied to students that do not return books within the predefined time-limit. In addition it supports to enter the payments for newly ordered books.

The Accounting Application periodically emits events with its COM+ interface whenever the library's database should be checked for users that have exceeded the allowed lending time. These events can be accessed within CWB to connect them to methods of other components. We forward these events to an

EJB session bean taking care of overdue books. For each student with an overdue book a notice is generated and the overdue flag is set within the database that forbids these students to borrow new books until the flag is cleared. The flag is cleared when the student returns all overdue books and pays the late return fee. In this case the accounting logic is used to enter the payment. After the money has been paid the flag is cleared. For this task the GUI responsible for returning books comes into play and emits another event that clears the user's overdue flag.

EJB components can be integrated easily with the CWB by specifying a naming service host name and a name that denotes the bean within the namespace of the naming service. COM+ components are integrated using the *programmable* name of such a component [15].

After the components have been selected, the user only has to setup the connections between them. When the user sets up the connection between the COM+ component and the EJBs the CWB generates an adapter that uses the EJB component wrapper to invoke the methods of the enterprise beans. Since there is no way to access an enterprise bean's reflection API without having access to the bytecode of the EJB's interfaces we have to install the compiled versions of these interfaces on the host where the adapters reside.

After we have defined the scenario of our library application we have exported it as Java source code that has been compiled and deployed on a machine where we had access to our COM+ components and on our EJB components.

6 Future Work

We have implemented component wrappers for CORBA, COM+ and EJBs. JavaBeans, as desktop components are supported as well. Interestingly, it has been possible to implement a component wrapper around web services accessed using the Simple Object Access Protocol (SOAP) [3] in a similar way. It would be interesting if a component wrapper for Microsoft's .NET architecture is also possible. Since *.NET remoting*, a .NET API similar to Java RMI, supports SOAP as communication protocol this should be possible [20].

Our component wrappers, however, are not yet completed since we do not yet support the conversion of all datatypes used in method calls between component models. We support the conversion of almost all primitive datatypes such as number and string types, and the conversion of some complex datatypes, but we have no actual implementation for constructed types. We plan to map constructed types of the different component models to types of the implementation language. The user can provide the conversions within the user interface wizard responsible for adapter conversions.

Another problem concerning component type systems are references passed via method calls across component model boundaries. There has to be a transparent conversion of references of one component model to a proxy reference. This problem is rather complex when only two component models are involved, and grows when more models are involved.

Currently, the Component Workbench is able to compose components implemented for different component models. Almost all distributed component models, however, specify services such as naming services, or transaction services that can be used by components implemented using these component models. These services, however, are not yet abstracted by the CWB. Hence, it is not yet possible to let components implemented for different component models participate within the same transaction context.

Since most component models provide support for the two-phase commit protocol, it should be possible, to let the COM+ Transaction Processing System and the Java Transaction Service cooperate. This would enable COM+ components to take part in EJB transactions and vice-versa. All modern distributed component models support declarative security and declarative transactions. Hence, the support of these services has to be realized for the deployment of components and for the composition of components.

The set of components available to the CWB is defined in XML-files that are written by users. These files contain data such as the available component models, the name of the components, and if necessary which naming service has to be used to access a component. We plan to integrate different naming services such as the CORBA naming service, and directory services such as Microsoft's Active Directory Service into the CWB to support developers in selecting and configuring appropriate components.

In the actual release every interaction between components has some part that is executed using wrapper code that converts messages between the component models. This can be of advantage at development time because the developer can notice every interaction between components on his machine. Obviously, this leads to a loss of performance that is not tolerable at deployment time. Therefore we plan to integrate bridging technologies in cases when performance is important and no loss in functionality can be expected. The generators that are used at deployment time have to be changed to support different bridging technologies.

The Component Workbench is a tool for creating scenarios of existing components of different component models. Our prototype has rather restricted support for creating full applications. We are evaluating if an integration of the CWB into a full-grown integrated development environment is desirable. Recently IBM initiated the *Eclipse* [6] project that provides an open source framework that supports plug-ins for various utilization. Since many vendors support this project it would be interesting to integrate CWB into Eclipse.

7 Related Work

PolySPIN [1] is an approach that tries to solve the interworking problem of components written in different programming languages. PolySPIN attacks the problem by modifying the implementation of the object methods. The modified methods consult a language arbiter at each invocation that converts the call to the call semantics of the target component's implementation language. Since

different programming languages use different type systems PolySPIN uses a matcher responsible to match types from different languages. Unlike the CWB, PolySPIN does not address the problem of objects that could interoperate on a conceptual level but whose interfaces have significant differences. The CWB solves this problem using component adapters. Additionally, the CWB does not require the original components to be modified which might be impossible in case of distributed components.

Interworking specifications support the integration of middleware systems of different kinds [7]. Interworking between CORBA and COM is specified in [17]. Implementations of this specification make use of compiler tools that automatically create mappings of the different component models [7]. Up to now only a few CORBA implementations support this form of interworking.

For instance, many different vendors of either CORBA or EJB servers support interworking between CORBA and EJB. The main reasons for better interworking are the availability of a Java language mapping for CORBA and the support of the CORBA communication protocol (IIOP) through EJB application servers in addition to Java RMI. Originally COM has been available on Microsoft's Java Virtual Machine. Since Microsoft no longer supports Java for COM+ development, there are only some approaches to let COM+ objects be accessed from Java. One successful approach seems to be Intrinsyc's J-Integra which is a Java-COM bridge that provides COM+ access to and from Java objects [14] running on any operating system. Microsoft has integrated COM interoperability into its .NET environment. The .NET architecture uses wrapper classes that mediate between .NET and COM, both as client and server [19]. Unfortunately, these bridging technologies support just the interworking between only one pair of component models and unlike the CWB, not an arbitrary number of component models.

Another approach to address the interworking problem is SOAP [3], but since it relies heavily on HTTP and XML as communication protocol and format it is no alternative when high-performance is mandatory.

8 Conclusions

In this article we have presented the interworking problem between distributed components of different models. Based on this we have explained the Component Workbench, our solution to this problem.

While implementing the CWB, we had to solve the challenge to provide an abstract component model. This model had to be easy to understand and powerful enough to accommodate the different features provided by today's component models without restricting developers to use only the subset of features provided by all of the component models.

This challenge was solved using component wrappers. No matter of the component model a component belongs to, the component wrappers map it to our abstract component model, hence providing the same interface to developers. The interface of the wrappers is easy to understand because it provides an API

that is similar to CORBA's Dynamic Invocation Interface (DII) [17] and the Java's Reflection API.

To evaluate our approach, we implemented and presented a small library application that demonstrates that the CWB can be used for the composition of applications from components implemented for different component models. Compared to today's bridging technologies, the advantage of our approach is that it allows the translation between arbitrary component models whereas bridging technologies can only be used to bridge between pairs of component models.

Acknowledgements

This work was supported in part by an IBM University Partnership Award from IBM Research Division, Zurich Research Laboratories and the European Union as part of the EASYCOMP project (IST-1999-14191).

References

1. Daniel J. Barrett, Alan Kaplan, and Jack C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *ACM SIGSOFT Software Engineering Notes, Proceedings of the fourth ACM SIGSOFT symposium on Foundations of software engineering*, volume 21. ACM, October 1996.
2. Philip A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, February 1996.
3. Don Box et al. *Simple Object Access Protocol (SOAP) 1.1*. W3C, May 2000.
4. Bill Councill and George T. Heineman. Definition of a software component and its elements. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering*, chapter 1, pages 5–19. Addison-Wesley, May 2001.
5. Linda G. DeMichiel, L. Ümit Yalcinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems, April 2001. Proposed Final Draft 2.
6. <http://www.eclipse.org>.
7. Wolfgang Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, 2000.
8. Wolfgang Emmerich and Nima Kaveh. Component technologies: Java beans, com, corba, rmi, ejb and the corba component model. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 311–312, September 2001.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*, pages 151–161. Addison-Wesley, 1995.
10. David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the Seventeenth International Conference on Software Engineering*, April 1995.
11. David Garlan and Mary Shaw. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing, 1993.

12. Thomas Gschwind. Type based adaptation an adaptation approach for dynamic distributed systems. Technical Report TUV-1841-01-11, Technische Universität Wien, September 2001.
13. Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*, chapter 20, pages 923–964. Addison-Wesley, 1999.
14. <http://www.intrinsyc.com/products/bridging/jintegra.asp>.
15. Mary Kirtland. *Designing Component-Based Applications*. Microsoft Press, 1999.
16. Johann Oberleitner. The Component Workbench: A Flexible Component Composition Environment. Master's thesis, Technische Universität Wien, October 2001.
17. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.4 edition, 2001.
18. Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.
19. David S. Platt. .net interop: Get ready for microsoft .net by using wrappers to interact with com-based applications. *MSDN Magazine*, Aug 2001. <http://msdn.microsoft.com/msdnmag/issues/01/08/Interop/Interop.asp>.
20. Paddy Srinivasan. An introduction to microsoft .net remoting framework. Technical report, Microsoft Corporation, 2001. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/introremoting.asp>.
21. Sun Microsystems. *JSR-00057 Long-term Persistence for JavaBeans™ Specification*, November 2001. <http://jcp.org/jsr/detail/57.jsp>.
22. Anne Thomas. *Enterprise JavaBeans Technology Server Component Model for the Java Platform*. Patricia Seybold Group, 1998. Prepared for Sun Microsystems, Inc.
23. Sanjiva Weerawarana, Francisco Curbera, Matthew J. Duftler, David A. Epstein, and Joseph Kesselman. Bean markup language: A composition language for java-beans components. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technology Systems (COOTS 2001)*, pages 173–187. USENIX, January 2001.