ICS

TECHNICAL REPORT

Extending Component Interoperability Standards to Support Architecture-Based Development

Rema Natarajan
David S. Rosenblum

UCI-ICS Technical Report No. 98-43
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

December 1, 1998

Information and Computer Science
University of California, Irvine

Extending Component Interoperability Standards to Support Architecture-Based Development

Rema Natarajan **David S. Rosenblum**
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
1-949-824-6534
{rema,dsr}@ics.uci.edu

ABSTRACT
Components have increasingly become the unit of development of software. In industry, there has been considerable work in the development of standard component interoperability models, such as ActiveX, CORBA and JavaBeans. In academia, there has been intensive research in developing a notion of software architecture. Both of these efforts use software components as the basic building blocks, and both address concerns of structure and reuse. With component interoperability models, the focus is on specifying interfaces, binding mechanisms, packaging, inter-component communication protocols, and expectations regarding the runtime environment. With software architecture, the focus is on specifying systems of communicating components, analyzing system properties, and generating "glue" code that binds system components. Our research involves studying how standard component models can be extended to accommodate important issues of architecture, including a notion of architectural style and support for explicit connectors. For our initial effort in this work, we have enhanced the JavaBeans component model to support component composition according to the C2 architectural style. Our approach enables the design and development of applications in the C2 style using off-the-shelf Java components or "beans" that are available to the designer. In this paper, we describe the techniques underlying our approach, and we identify the important issues that surface when attempting this type of extension.

Keywords
Architectural style, C2, component standards, connectors, JavaBeans, software architecture

1 INTRODUCTION

Components have increasingly become the unit of development of software. In industry, there has been considerable work in the development of component interoperability models, such as ActiveX [1], CORBA [12], and JavaBeans [14]. These models help developers deal with the complexity of software and facilitate reuse of off-the-shelf components. Component interoperability models also make a positive move toward standardization of components, and the creation of a software component marketplace.

Software architecture research deals with the same issues of software complexity and promoting reuse. Software architecture has been the focus of intense research in academia. Architectures help designers focus on system level requirements and the interconnection of components in a large-scale software system.

Both these approaches use software components as the fundamental building blocks. With component interoperability models, the focus is on specifying interfaces, packaging, binding mechanisms, inter-component communication protocols, and expectations regarding the runtime environment. With software architectures and architectural styles, the focus is on specifying systems of communicating components, analyzing system properties, and generating "glue" code that binds system components [9].

We have begun studying how standard component models can be leveraged to accommodate issues of architectural modeling, including a notion of architectural style and support for explicit connectors. As described in this paper, our approach merges component interoperability models with suitable architectural styles to leverage the full benefit from both technologies, and to develop a comprehensive approach to software development.

We have chosen the JavaBeans component interoperability model as our initial platform for investigation. We made this choice for a variety of reasons:

- The Java language and the JavaBeans component model are becoming increasingly popular and have been widely adopted as de facto standards.
- JavaBeans tools and resources are free or have negligible cost.

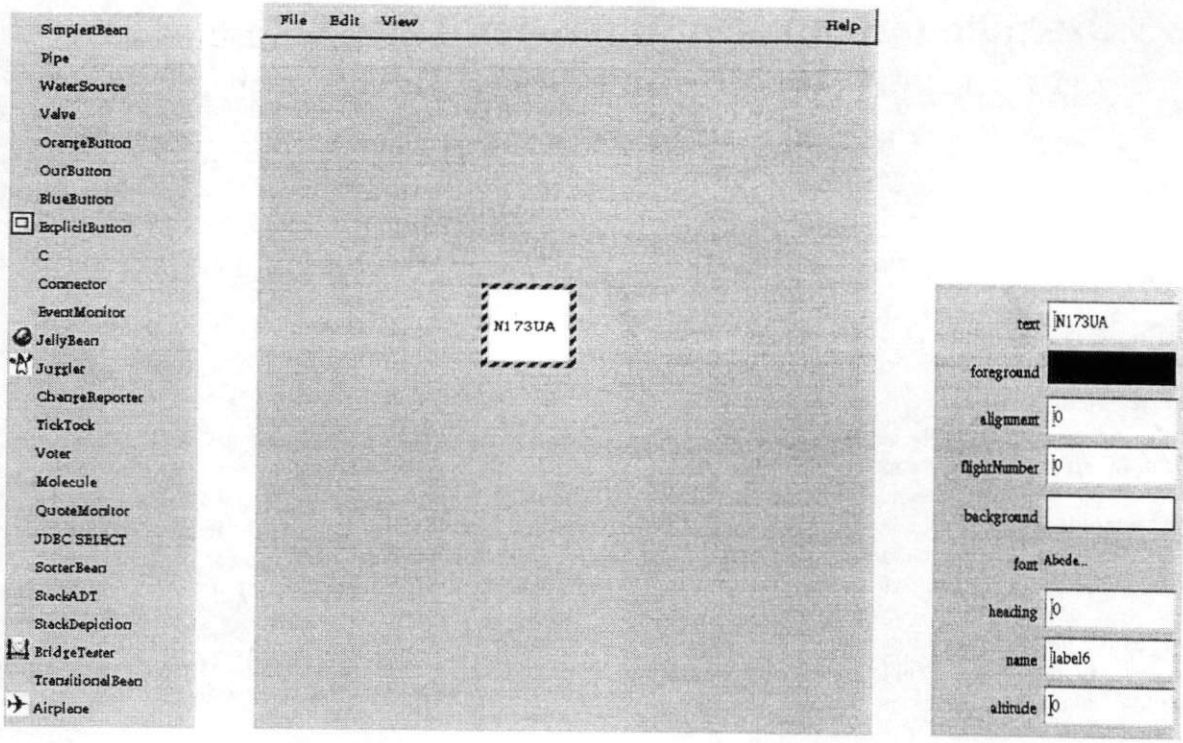

Figure 1. The Sun BDK JavaBeans design environment, with theToolBox of available beans shown on the left, the BeanBox design palette shown in the center with an Airplane bean, and the Property Sheet of the Airplane bean shown on the right.

- The model of composition in JavaBeans is natural and straightforward.
- JavaBeans is a lightweight and flexible framework that lends itself to modification and extension.

In addition, we have chosen the C2 architectural style as our initial architectural technology because it is a novel style that is highly flexible and lends itself naturally to dynamic architectural change. It also supports the property of substrate independence that facilitates reuse and substitutability across architectures with ease of effort.

In this paper, we describe our work in enhancing the JavaBeans component model to support component composition according to the C2 architectural style. Our approach enables the design and development of applications in the C2 architectural style using off-the-shelf Java components or *beans* that are available to the designer. The creation of individual components with their specific interfaces, functionalities and behaviors is a different task from the composition of an architecture of a system that satisfies requirements. The merging of the component interoperability model with the architectural style provides a seamless integration of both activities.

2 THE JAVABEANS COMPONENT MODEL

The JavaBeans component model is a component interoperability model tailored to the Java language. Interoperability is achieved primarily by designing component or *bean* interfaces according to a *component design pattern*.[1] The JavaBeans design pattern defines a naming scheme and interaction protocol to which compliant beans must adhere. The interface constituents governed by this design pattern include properties, methods, and events that together define a bean interface. *Properties* encapsulate key attributes of a bean and can be *simple*, *bound* (meaning they generate events whenever they change values) or *constrained* (meaning their changes can be vetoed by other beans). *Methods* are public operations that form part of the bean interface. Beans communicate with each other through bean *events*; the event handling is based on the Java 1.1 event model.

Figure 1 depicts the JavaBeans design environment that is provided by Sun Microsystems in their Beans Development Kit (BDK) [2]. The environment allows designers to develop beans using the JavaBeans design pattern and to instantiate and test bean compositions. This environment is

[1] The term *design pattern* has been used by many authors to characterize the JavaBeans interface design convention, even though it does not correspond to the usual notion of a design pattern as a frequently-recurring design solution [3].

```
import java.awt.*;
import java.beans.*;

public class Airplane extends Label {
 protected int altitude = 0;
  protected PropertyChangeSupport changes =
    new PropertyChangeSupport(this);

  public int getAltitude() { return altitude; }

  public void setAltitude(int a) {
    int oldAltitude = altitude;
    altitude = a;
    changes.firePropertyChange("Altitude", new Integer(oldAltitude),
                               new Integer(a));
    repaint();
  }

  public void addPropertyChangeListener(PropertyChangeListener l) {
    changes.addPropertyChangeListener(l);
  }

  public void removePropertyChangeListener(PropertyChangeListener l) {
    changes.removePropertyChangeListener(l);
  }
}
```

Figure 2. Partial Java code for the example Airplane bean. The code demonstrates how a bean property is declared plus the methods interested beans can use to listen to bean property change events.

representative of the kinds of visual design environments that can be used to support construction of applications with the JavaBeans component model. As shown in the figure, a JavaBeans design environment includes a palette of available beans (called the ToolBox in the BDK environment), a design tablet on which beans are instantiated and interconnected (called the BeanBox in the BDK environment), and a Property Sheet showing the properties of the bean that is currently selected in the BeanBox.[2]

For instance, consider an Airplane bean that represents the control of an airplane. A partial design for an Airplane bean class is presented in Figure 2. As shown in the figure, the class declares properties that will show up in the property sheet of the BeanBox. In particular, the Airplane declares a property called Altitude, which the BeanBox will find because the class exports the pair of methods getAltitude() and setAltitude() (following the naming scheme of the JavaBeans design pattern). As shown in Figure 1, this property appears in the Property Sheet for the Airplane bean (as do other properties, whose implementation is not shown in Figure 2). While it is possible for changes to the property to be effected by manually coding calls to these exported methods, the BeanBox is designed to allow the designer to customize bean properties via the Property Sheet.

The Altitude property is a bound property because it fires a PropertyChange event whenever its value changes during a call to the method setAltitude(). Other beans register and unregister themselves as listeners for this event by calling the methods addPropertyChangeListener() and removePropertyChangeListener(), respectively; these methods are required by the JavaBeans design pattern when bound properties are to be supported. While it is possible for such event-based interactions between the Airplane bean and other beans to be coded by hand, the BeanBox has been designed to support "wiring up" interacting beans in a graphical manner. In particular, the designer would select the Airplane bean with the pointing device, select a menu item corresponding to the PropertyChange event, and then select a target bean to receive the event; the BeanBox would then take care of generating the necessary method calls to effect the interaction. Note that each bean maintains its own set of listeners and manages event notification by itself.

As can be seen in Figure 2, the JavaBeans design pattern paves the way for building tools that can dynamically "introspect" beans and publish their interfaces, in a manner similar to what the BDK BeanBox does. The tools can provide designers with the capability for customizing bean behavior. Additionally, the JavaBeans design pattern

[2] In this paper we use the term BeanBox as a synonym for a complete JavaBeans design environment.

defines a notion of *bean customizers*, which can be built to allow complex customization of a bean's appearance and behavior, and *property editors*, which define custom editors for a specific property. These two mechanisms aid the design and implementation of generic beans that can be easily customized for different applications. Apart from their individual customizability, beans can be composed in the BeanBox to create running applications, and their runtime behavior can be tested during the design phase itself. This novel capability blurs the distinction between "design-time" and "runtime", since manipulating beans in this manner actually has the effect of creating running instances of bean classes that cooperate according to the designer's intent.

In this manner, the JavaBeans component model concentrates on the interface a Java software building block can or should present. It does not specify how the building blocks can or should be combined to create any kind of application. It specifies how two or more beans can communicate information, without imposing any semantic rules on the information exchanged or on the topology of any bean communication network [14]. The JavaBeans design pattern is designed to make development tools better aware of component capabilities; in particular, the interface pattern has been defined for a modern software developer who will manipulate beans via visual interactions.

3 THE C2 ARCHITECTURAL STYLE

The C2 architectural style is primarily concerned with high-level system composition issues, rather than particular component packaging approaches [9,13]. The building blocks of C2 architectures are components (computational elements) and connectors (interconnection and communication elements). This separation of computation from communication enables the construction of flexible, extensible, and scalable systems that can evolve both at design-time and runtime. The style places no restrictions on the implementation language or granularity of components and connectors, potentially allowing the use of multiple interoperability technologies for its connectors. This flexibility has enabled us to use the event-based interoperability of JavaBeans for our purposes.

Central to the C2 style is the principle of limited visibility or *substrate independence*: components are arranged in a layered fashion in a C2 architecture, and a component is completely unaware of components that reside beneath it in the stack of component layers. Substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. Components communicate only by exchanging messages through connectors, which greatly simplifies the problem of control integration issues; this property also facilitates low-cost interchangeability of components to construct different members of the same system family. Two components cannot assume that they will execute in the same address space; this eliminates complex dependencies, such as components sharing global variables and simplifies modification of architectures. Conceptually, components run in their own thread(s) of control, allowing components with different threading models to be integrated into a single application. Finally, a conceptual C2 architecture can be instantiated in a number of different ways. Many potential performance issues or variations in functionality can be addressed by separating the architecture from actual implementation techniques.

C2 components and connectors have a notion of a "top" and a "bottom" interface through which they receive and send messages and communicate with other components in the architecture. This notion of a "top" and a "bottom" is important to ensure substrate independence. Messages that travel up the architecture are called *requests*, and messages that travel down the architecture are called *notifications*. Components execute application logic and communicate with other components in the architecture via requests and notifications. Components do not communicate directly with one another, but instead must communicate through connectors that take care of most of the management of message traffic in the system. Connectors, on the other hand, can be directly connected to each other.

The advantage of explicit connectors is that they encapsulate the logic for message broadcasting, message filtering, and other interaction logic, and they reduce the complexity involved in composing components. This is different from the JavaBeans model where every bean manages its event listeners and event propagation on its own. Additionally, the notion of connectors supports a more generic structure whereby it becomes easier to substitute one component or connector with another, and enables reuse of individual components or connectors across different architectures. It also becomes easier to support dynamic alteration of the architecture. The C2 style constraints (which we have only briefly summarized here) help preserve these C2 properties.

4 A C2-AWARE COMPOSITION ENVIRONMENT

We have begun our investigation of the problem of merging component models with architectural styles by enhancing the BDK BeanBox described in Section 2. In our approach, we create beans using the same JavaBeans design pattern, thus retaining all the advantages of the beans component model. However, we extend the JavaBeans model to incorporate the notion of "components" and "connectors" as defined in the C2 architectural style, and we extend the BeanBox composition functionality to enforce the rules of the C2 style. Thus, we have enhanced the BeanBox and made it "C2-aware".

C2 Style Beans

Our C2-Aware BeanBox provides two mechanisms to

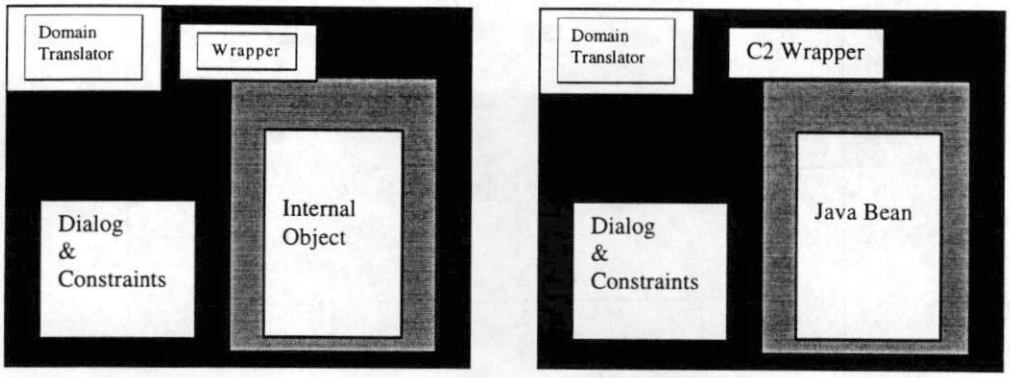

Figure 3. Wrapping of C2 components; the general C2 model of wrapping is shown in the picture on the left, while the picture on the right shows how the general model has been applied for JavaBeans.

instantiate C2 components and C2 connectors as beans. In our first approach, we have provided C2 component and connector "framework beans" that can be subclassed by developers interested in creating beans for specific applications. When these beans are instantiated in the BeanBox, they automatically publish their C2 interface and can thus be hooked up "as is" to compose applications. Beans created in this fashion are fully C2 compatible, and this approach is convenient for developers to create new beans in a way that incorporates the characteristics of a C2 architecture described in Section 3.

In our second approach, we have provided a C2 wrapping mechanism to create C2 components from off-the-shelf beans. Figure 3 presents the model component wrapping we use, which follows the general model of wrapping that has been developed for the C2 style [7,13]. Off-the-shelf beans that have already been developed by other vendors can be instantiated into the BeanBox. Upon this instantiation, a wrapper object is created for the bean to make it C2 compliant. This C2 wrapper is created with the help of interactive dialogs that publish the interface of the bean and guide the designer in mapping the bean's events into C2 requests and notifications. The wrapper then uses this information to build the internal dialog component that is responsible for converting incoming requests and notifications into bean events. The *domain translator* component of the C2 wrapper is used to resolve incompatibilities between communicating components such as mismatches between message names, parameter types and ordering of parameters. The *constraints* component specifies constraints that cannot be violated by the component, provides recovery mechanisms when constraints are violated and exceptions are raised, and provides mechanisms to customize the bean so that constraints are satisfied without raising exception conditions.

Most of the translation required for converting beans into C2 components involves mapping bean events to requests and notifications in the C2 style. The properties that a bean publishes in its property sheet are used "as is" after the bean has been wrapped as a C2 component. Other tools provided for manipulation of beans such as property editors and bean customizers can also be used "as is" in the C2 aware BeanBox.

This second approach has several advantages. Existing beans can be used off-the-shelf simply by plugging them into the BeanBox with the help of the C2 wrapper. The wrapper handles the translation of bean events into C2 messages. Additionally, the dialog and constraints can be used to build in constraints for the component as specified in an architecture specification to ensure that the bean is properly customized for the specific architecture. It is also the facility that lends itself most naturally to providing dynamic testing, analysis and instrumentation mechanisms. We plan to focus on these issues much more in the near future.

The C2-Aware BeanBox

C2 compliant beans, which can be created using any of the two approaches described above, can be instantiated into the C2-Aware BeanBox, as shown in Figure 4. The C2-Aware BeanBox has all the C2 style rules and constraints built into it. It provides a C2 Style Dialog that notifies designers whenever stylistic constraints are violated and thus guides the designer through the composition process. Components and connectors are hooked up using the bean event wiring mechanism, where request events and notification events become the two kinds of events that beans use to interoperate. As required by the C2 style, components cannot be connected to other components, and connectors handle the propagation of events. Thus, unlike in the traditional beans model, beans composed in the C2-Aware BeanBox do not maintain lists of other bean

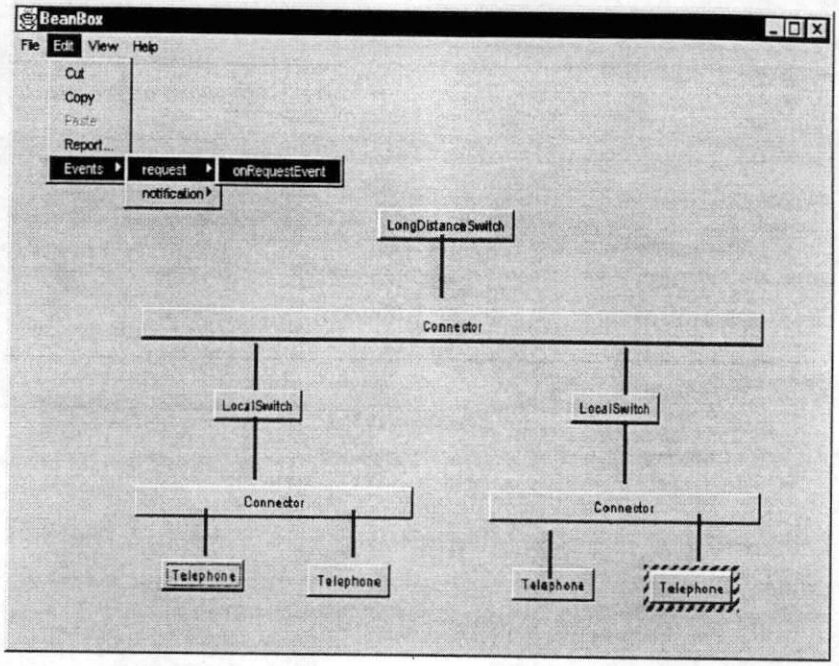

Figure 4. The C2-Aware BeanBox. The figure depicts the composition of a telephone network application as a group of beans interacting in the C2 style.

listeners or handle notification of events to those listeners. Instead, event notification is handled by C2 connector beans. Hence, component bean behavior is better confined to the execution of application logic.

The C2-Aware BeanBox thus allows one to build complex compositions of beans in the C2 style as different instantiations of a given C2 architecture. Introspection mechanisms employed in the C2-Aware BeanBox are used to extract the properties, methods and events that form the public interface of the bean. Conceptually, beans communicate using bean events; these events then become the requests and notifications in the C2 architecture. The designer informs the C2-Aware BeanBox through an appropriate dialog about how events are to be classified as requests and notifications and then manages the communication of beans through these requests and notifications.

5 AN EXAMPLE JAVABEANS-BASED C2 ARCHITECTURE

We have chosen a telephone network system as an example to illustrate our approach; the instantiation of the application in the C2-Aware BeanBox is depicted in Figure 4. Our hypothetical system consists of telephones, local switches and long distance switches. Each of these components is represented as standard beans that publish events (such as ring, hang up, busy) and properties (such as phone numbers and area codes). The properties are bound properties, and thus they fire PropertyChange events. In our C2 architecture for the telephone system, the telephones form the lowest layer of the architecture (i.e., the "interface elements", as is typical of C2-style architectures), with local switches in a layer above the telephones, and the long distance switches at the highest layer in the design.

Upon instantiation of the beans into the C2-Aware BeanBox, each bean gets wrapped in a C2 wrapper. As described in Section 4, the dialog component of the C2 wrapper dynamically introspects the bean and then displays the bean's events in a list and lets the user select the events that should get published as requests and those that should get published as notifications for that bean component. For example, for the telephone component, we would select the "dial" event as a request that needs to travel "up" in the architecture, and the "ring" event as a notification event that needs to travel "down" the architecture. We use standard connectors that are provided as part of the C2 framework to link the components of the telephone network together. The connector propagates request events fired by a component connected to its bottom interface to all components attached to its top interface. Thus a request event for dialing a number by a telephone is propagated through the connector above it to the local switch that handles requests for that area in the architecture. The local switch in turn forwards the request to the long distance switch above it. The long distance switch forwards the

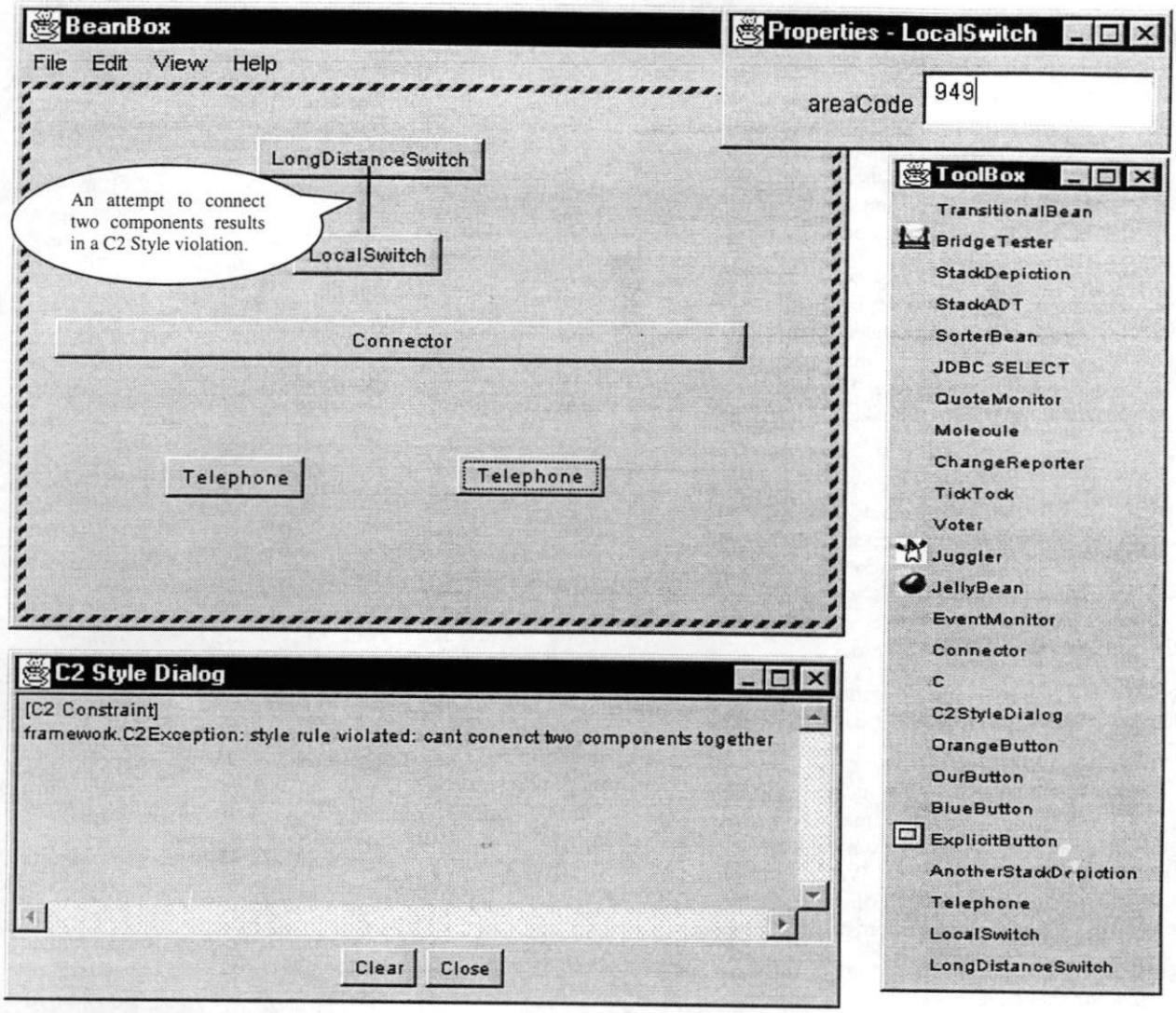

Figure 5. Wiring up the Long Distance Switch and the Local Switch in the C2-Aware BeanBox. This throws a C2 Style Violation exception, which appears in a C2 Style Dialog. The Property Sheet of the Local Switch shows its Area Code property.

message "down" the architecture as notifications to the local switches below it. The local switch with the area code for the dialed number processes the notification by generating a "ring" event as a notification for the telephones below it. The telephones receive the notification, and the telephone with the correct number responds to the notification by processing it. The beans themselves retain their interfaces as before, but the wrapper ensures that beans effectively communicate with the rest of the C2 architecture.

As shown in Figure 5, beans are instantiated and removed from the architecture easily using C2-Aware BeanBox. As the telephone network is buiit by plugging beans into the architecture, the C2-Aware BeanBox makes automatic checks to ensure that C2 stylistic rules are honored. For example, an attempt to link two telephones directly will raise an exception message in a popup window, helping the designer through the process of composing the system. Figure 5 shows how an attempt to link two components—the Long Distance Switch and the Local Switch—throws an exception that brings up the C2 Style Dialog.

6 DISCUSSION AND RELATED WORK

A lot of interesting issues came up in our effort to create plug-and-play functionality with off-the-shelf beans in our C2-Aware BeanBox. Here we discuss these issues with respect to some of the Component Integration Heuristics for C2 [7].

- If the OTS (off-the-shelf) component does not contain all of the needed functionality, its source code must be altered. While it is interesting to think of situations where other components might be used in conjunction with the OTS component (without altering its source code) to provide the needed functionality, this would be a complex task to attempt, and would depend on the type of functionality that is required.
- If the OTS component does not communicate via messages, a C2 wrapper must be built for it. This facility has already been provided in our C2-Aware BeanBox for all OTS beans that are used as C2 components in a C2 architecture. The wrapper does all the translation necessary to make the OTS bean C2 compliant.
- If the OTS component is implemented in a programming language different from that of other components in the architecture, an IPC (interprocess communication) connector must be employed to enable their communication. As we have solely dealt with the Java language and the BeanBox environment, this issue does not arise in our work.
- If the OTS component must execute in its own thread of control, an inter-thread connector must be employed.
- If the OTS component communicates via messages, but its interface does not match interfaces of components with which it is to communicate, a domain translator must be built for it. We have done some preliminary work in providing domain translation, and we are currently working on improving this support.

Apart from these heuristics described in [7], there are other interesting issues that have arisen:

- If an OTS component provides the functionality required in the C2 architecture, there is still the necessity, in a development and testing environment such as the C2-Aware BeanBox to provide mechanisms to test and validate the architecture instantiation against an architecture specification. Right now the Dialog and Constraints component of the C2 Wrapper provides no support for this. As we discuss in Section 7, we tend to explore this problem in the future.
- The C2-Aware BeanBox facilitates design and composition of systems using any OTS beans that are available. There are exciting possibilities to be explored in strengthening support for design at the architectural level, apart from the work we have already done for the C2 style. For example, we could use the same design environment to create "architecture template beans" for a required C2 architecture, specifying the interfaces of the C2 components and connectors. The template beans could then be used directly, and the same visual environment could be used to populate the templates with OTS beans. This then reduces the work needed for creating the wrapper for OTS beans, and we can use the same architecture templates to create different instantiations of an architecture family. As we discuss in Section 7, this could be done by leveraging the ADL and environment described in [8].

There has been little work to date on supporting architectural modeling in conjunction with standard design technologies. C2, Darwin and UniCon are examples of ADLs that provide a proprietary implementation infrastructure to support an associated ADL. C2 has its class framework as its infrastructure, and this class framework is implemented in multiple programming languages [13]. Darwin is supported by an infrastructure called Regis for distributed programs that are configured using Darwin [4,5]. And UniCon supports implementation generation for a predefined collection of connectors [11]. There has also been recent work in the Darwin project on supporting architectural modeling of CORBA-based systems [6].

In addition to providing ADL-specific infrastructure support, there has been recent work on incorporating substantial support for architectural modeling into the Unified Modeling Language (UML), an emerging standard design notation [10].

7 CONCLUSIONS

Having considered and explored the possibility of combining a popular component interoperability model with a useful software architectural style, we are convinced of the advantages of this approach in the development of component-based software. The philosophy of substrate independence in C2 makes substitution of components and reconfiguration of architectures fairly easy. These modifications to the architecture are done with the least amount of effort because we leverage the strengths of the JavaBeans component model and the ability of the C2-Aware BeanBox to dynamically publish the interface of a beans component and map it to C2-style interactions. The use of a wrapper separates the application logic in the bean component from the translation and dialog with other architectural components. Our C2-Aware BeanBox is a powerful design environment that lets us develop different architectural instantiations with the ease of using a visual environment. It is an example of a tool where the distinction between the design environment and the runtime environment of systems has become blurred.

A key advantage of our approach is that our architectural infrastructure is now complete, to the extent that the full range of developmental activities is supported from the

design, implementation and adaptation of individual components, to the design, implementation and integration of architectures that are compositions of these individual elements. Another advantage is that all these activities are now integrated into a single environment, and this leads the way to a seamless, comprehensive development philosophy that facilitates easy shifting of focus from one activity to another. Sophisticated architectural development tools built along these lines will tie in neatly with component-based software development.

In the future, we plan to further investigate the issues raised and opportunities opened up by this approach. The ability to test the runtime behavior of bean components in a design environment is extremely useful for test different architectural configurations. As we discussed in Section 4, a natural place to provide instrumentation support for testing is in the dialog and constraints portion of the C2 wrapper shown in Figure 3. As also discussed in that section, we would like to begin supporting checking of component semantic constraints in a manner that respects emerging approaches to architectural modeling and emerging standards for component interoperability. An excellent starting point for this work would be to leverage two recent additions to the C2 arsenal, the ADL C2 SADEL and its associated environment DRADEL, which support modeling and evolution of architectures according to a rich model of heterogeneous subtyping of component interfaces [8]. Finally, we need to find ways of adapting our visual approach for architectural construction to distributed architectures. A current limitation of the JavaBeans model is that it does not support composition of distributed beans that must communicate via remote procedure call (e.g., using Java Remote Method Invocation). With the C2-Aware BeanBox, we can naturally incorporate such mechanisms for distributed interaction within C2 connectors, yet we must provide additional support for specifying deployment of beans across distributed hardware.

Our experience with JavaBeans and C2, we believe, are helping to us expand and develop our understanding of the synergy between component models and software architectures.

ACKNOWLEDGEMENTS

Discussions with Dick Taylor, Peyman Oreizy, Elisabetta Di Nitto and Alfonso Fuggetta helped us improve many of the ideas presented in this paper. This effort was sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602- 97-2-0021; by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number F49620-98-1-0061; and by the National Science Foundation under Grant Number CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

[1] D. Chappell, *Understanding ActiveX and OLE*. Redmond, WA: Microsoft Press, 1996.

[2] A. DeSoto, "Using the Beans Development Kit 1.0: A Tutorial", JavaSoft, Sun Microsystems, Inc., Mountain View, CA November 1997.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[4] J. Magee, N. Dulay, and J. Kramer, "Regis: A Constructive Development Environment for Distributed Programs", *IEE/IOP/BCS Distributed Systems Engineering*, vol. 1, no. 5, pp. 304–312, 1994.

[5] J. Magee and J. Kramer, "Dynamic Structure in Software Architectures", *Proc. ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, pp. 3–14, 1996.

[6] J. Magee, A. Tseng, and J. Kramer, "Composing Distributed Objects in CORBA", *Proc. Third International Symposium on Autonomous Decentralized Systems*, Berlin, Germany, pp. 257–263, 1997.

[7] N. Medvidovic, P. Oreizy, and R.N. Taylor, "Reuse of Off-the-Shelf Components in C2-Style Architectures", *Proc. 19th International Conference on Software Engineering*, Boston, MA, pp. 692–700, 1997.

[8] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor, "A Language and Environment for Architecture-Based Software Development and Evolution", in submission August 1998.

[9] P. Oreizy, N. Medvidovic, R.N. Taylor, and D.S. Rosenblum, "Software Architecture and Component Technologies: Bridging the Gap", Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, CA January 1998.

[10] J.E. Robbins, N. Medvidovic, D.F. Redmiles, and D.S. Rosenblum, "Integrating Architecture Description Languages with a Standard Design Method", Department of Information and Computer Science, University of California, Irvine, Irvine, CA, Technical Report 97-35, November 1997.

[11] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314–335, 1995.

[12] J. Siegel, *CORBA Fundamentals and Programming*. New York, NY: Wiley, 1996.

[13] R.N. Taylor, N. Medvidovic, K.M. Anderson, J. E. James Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow, "A Component- and Message-Based Architectural Style for GUI Software", *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.

[14] L. Vanhelsuwe, *Mastering JavaBeans*: SYBEX Inc, 1997.