Merging Component Models and Architectural Styles

Rema Natarajan David S. Rosenblum

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA

{rema,dsr}@ics.uci.edu

1. ABSTRACT

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, CORBA and JavaBeans. In academia, there has been intensive research in developing a notion of software architecture. 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. In this paper, we discuss issues arising from our initial effort in this research, where we have extended the JavaBeans component model to support component composition according to the C2 architectural style.

1.1 Keywords

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

2. 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 [4], and JavaBeans [6]. These models help developers deal with the complexity of software and promote reuse. 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 developers focus on system level requirements and the interconnection of components in a large-scale software system.

Both these approaches use software components as the 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 in on specifying systems of communicating components, analyzing system properties, and generating "glue" code that binds system components [3].

It seems intuitive to merge component interoperability models with suitable architectural styles to leverage the full benefit from both technologies, and to develop a comprehensive approach to software development. It also opens up opportunities for researchers in industry and in academia to exchange views and results.

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 developer. 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.

3. THE JAVABEANS COMPONENT MODEL

The JavaBeans component model is a component model tailored to the Java language. The JavaBeans design pattern defines a protocol to which beans must adhere. This interface pattern mainly consists of the properties, methods, and events that together define a bean interface. *Properties*

encapsulate key attributes of a bean and can be read-only, read/write, *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. Thus, the JavaBeans component model concentrates on the interface a Java software building block can or should represent. It does not specify how the building blocks can or should be combined to create any type 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 [6]. The JavaBeans design pattern is designed to make beans tool-aware; in particular, the interface pattern has been defined for a modern software developer who will manipulate beans via visual interactions.

4. THE C2 ARCHITECTURAL STYLE

The C2 architectural style is primarily concerned with high-level system composition issues, rather than particular component packaging approaches [3,5]. 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 before and during runtime. This style places no restrictions on the implementation language or granularity of components and connectors, potentially allowing it to use 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.

5. EXPERIENCE TO DATE

We have begun our investigation of the problem of merging component models with architectural styles by enhancing the BeanBox (the visual composition environment for JavaBeans) that comes with Sun's Bean Development Kit. The BeanBox allows developers to develop beans using the beans design pattern, and instantiate and test the beans in the BeanBox. Our enhancements make the BeanBox C2-aware. In particular, the enhanced BeanBox allows one to build complex compositions of the beans in the C2 style as different instantiations of a given C2 architecture. It is not necessary to do any translation or mapping to convert an existing bean into a C2 component. Introspection mechanisms employed in the 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 developer informs the Beanbox through an appropriate dialog about the events that are to be classified as requests and events. An instantiated bean is wrapped in a C2 Component, which has a wrapper for the bean, and a dialog manager that manages the communication of beans through these requests and notifications.

As shown in Fig. 1, this wrapping is done according to the general model of wrapping that has been defined for components in the C2 style [5]. The visual interface of the BeanBox allows the developer to build C2 architectures composed of beans intuitively and easily. The tool automatically enforces the C2 style constraints and notifies the developer when C2 style constraints are violated. C2 connector beans are used as the connectors in the architecture.

A key advantage of this approach is that our architectural infrastructure is now complete, in the sense that the full range of developmental activities is supported from the design, development and testing of individual components, to the design, development and testing 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.

6. 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 to development of

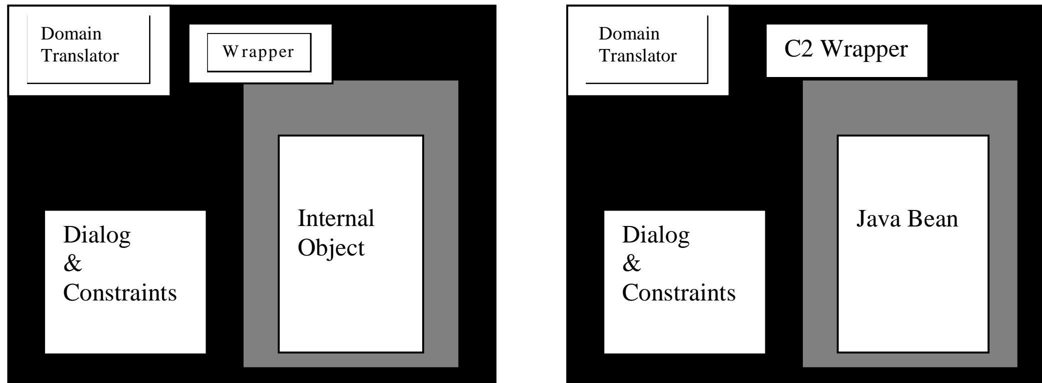

Fig. 1. Wrapping of C2 components; the general model of wrapping is shown in the picture on the left, while the picture on the right shows how the general model has been applied for JavaBeans.

component-based software. In the future, we plan to complete the implementation of this tool, and we plan to further investigate the issues raised and opportunities opened up by this approach. For example, this research opens up interesting possibilities to use an enhanced BeanBox to test runtime behaviors of system architectures before system implementation is completed.

The plug-in capabilities of the JavaBeans environment, and the philosophy of substrate independence in C2, make substituting of components and rearranging of architectures fairly easy. The BeanBox is an example of a tool where the distinction between the design environment and the runtime environment of systems has become blurred. This is an issue that is being studied in greater depth in other work on C2, in the context of designing and instantiating system architectures [2]. Our experience, we believe, will help us expand and develop our understanding of the synergy between component models and software architectures.

7. ACKNOWLEDGMENTS

Discussions with Dick Taylor and Peyman Oreizy helped us improve many of the ideas presented in this paper. This material is based upon work supported by the National Science Foundation under Grant No. CCR-9701973, and by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061. 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 Air Force Office of Scientific Research or the U.S. Government.

8. REFERENCES

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

[2] 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.

[3] 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.

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

[5] 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.

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