Incorporating Off-The-Shelf Components with Event-based Integration

Jie Ren, Richard Taylor

Institute for Software Research

University of California, Irvine

Irvine, CA 92697-3425

{jie,taylor}@ics.uci.edu

ABSTRACT

Event-based Integration (EBI) is a promising technology for constructing large software architectures. It can integrate concurrent, heterogeneous components in dynamic software architecture. This paper discusses our experience in integrating a set of off-the-shelf components to create an event-based software architecture development environment. We discuss the benefits and obstacles of integrating Common-Off-The-Shelf (COTS) components, explain the rationale for choosing event-based integration, and report some experiences from this effort.

1. INTRODUCTION

Software architecture has been proposed as an effective solution for producing bigger, better and cheaper software [6]. Its principles are: a software system is composed of components and connectors, components are loci of computation and connectors are loci of communication, and a specific set of components and connectors form the configuration of the software [5].

Components can be either developed in-house, or acquired off-the-shelf. The advantages provided by the off-the-shelf components include richer functionality, higher reliability, less development time, reduced documentation effort, flatter learning curve, and easier deployment.

However, these advantages do not come for free. The external component may not match the requirements perfectly, the design could bear with them some inflexible decisions, and the uneasy task of understanding could be made worse by lack of proper documentation. The absence of source code, which is common practice in industry, can make integration a very challenging task [1].

The key of composition and integration in architectural-driven component-based development lies in connector technology. Different technology has different capabilities and limitations. Among them, event-based integration (EBI) is very effective in integrating concurrent, heterogeneous components in dynamic environment [7]. In this paradigm, components communicate with each other by sending events, while connectors provide the infrastructure for messaging, include event registration, routing, and monitoring. The components can be written in different languages, reside on different processes, and run on different machines. They don't need to maintain specific pointers about the components that they are communicating with, and they can be easily added or removed from the system without adversely affecting other members.

In this paper, we present our experience in integrating a set of off-the-shelf components to create an event-based software architecture development environment. Section 2 introduces the specific architecture style and development environment we are developing. Section 3 details the integration activity. Section 4 discusses related work. Section 5 concludes the paper.

2. C2 AND ArchStudio

C2 is an architecture style featuring event-based integration [7]. In this style, components communicate with each other only by sending events, which are routed by connectors. Components send request events to upper components for service, and the upper components reply by sending notification events downwards.

We developed a software development environment, ArchStudio, to support the development of software in this style [4]. ArchStudio itself is in C2-style. Its architecture is depicted in Figure 1.

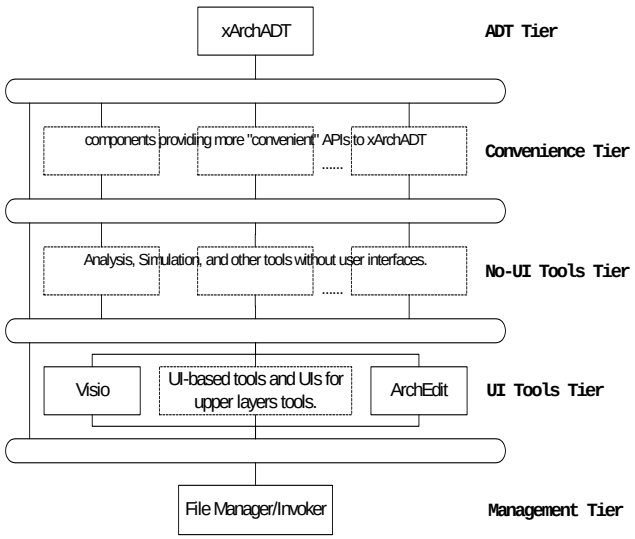

Figure 1, Architecture of ArchStudio

The core of ArchStudio is a component called xArchADT, which stores architectural information expressed in xADL 2.0, an extensible, XML-based Architecture Description Language [2]. A set of UI tools is used to manipulate the information graphically.

Most of the components in the environment are written using a Java-based framework we developed to ease constructing C2-style software. We want to explore the possibility to incorporate non-Java tools using events. Another goal is to enhance the frond end of ArchStudio.

The old front end in ArchStudio is Jargo, a tool based on GEF (Graphics Editing Framework).GEF is an open source Java graphics-editing framework. It provides basic support for graphics editing, but lacks industrial strength capability. We tried Mica, another Java-based GUI toolkit, only to find it still is not mature enough.

3. INTEGRATING VISIO USING EVENTS

3.1 Visio

We decide to use Visio, an industry-strength graphics-editing product, as the basis for our new graphical front end. In addition to standard shape creation and editing functionality, this commercial product provides many advanced features, including dynamic master shape generation, flexible connection between shapes, rich format, and zoom. The resulting environment is shown in Figure 2. The main Visio window shows part of the architecture for AWACS (Airborne Warning and Control System). The architecture is described using xADL 2.0.

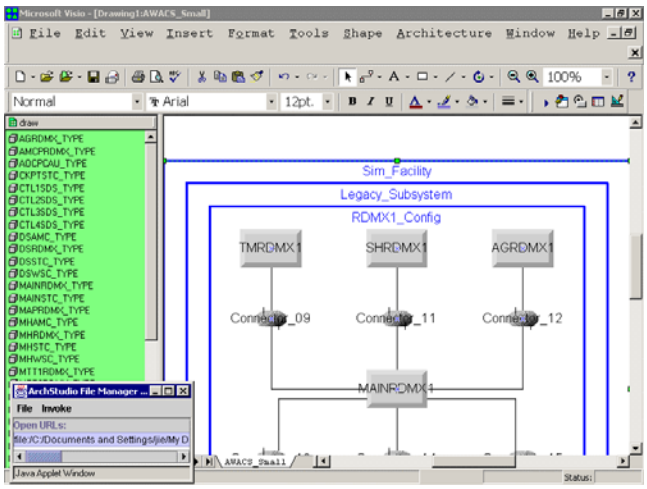

Figure 2, ArchStudio with Visio front end

Visio front end provides the operations to the architect for creating architecture graphically. These editing operations will send requests to xArchADT to insert or remove the relevant xADL elements instantly. Since Visio is not the only editor in ArchStudio, it also needs to get the notification when other editors modify xArchADT.

Visio provides a programmatic interface for its graphics engine, through which the rich functions can be accessed. This facilitates the development of customized solutions for various fields. We use it to integrate Visio into ArchStudio and make it the front end.

Both ArchStudio and Visio are event-based systems. This similarity in the underlying programming paradigms eases the integration. However, they are written in different languages. ArchStudio is a pure Java system. Visio's programming interface is COM-based. We need to bridge the COM world and the Java world.

3.2 Microsoft's Java Virtual Machine

There are several products that enable the interoperation between COM and Java to happen. We choose Microsoft Virtual Machine, because it provides both COM-to-Java and Java-to-COM conversion, and it is readily available with the Microsoft operating system without any further charge or extra installation.

In COM, the central artifact is the interface, a set of abstract functions. The interface is the contract between the client and the server. A class can implement a set of interfaces. A client will create an object from the class and access the object's services. Both interfaces and classes are designated through Global Unique IDs.

Although COM exhibits a lot of influence from C++, its notions about interface and class actually match the corresponding concepts in Java better. This makes COM programming in Java very natural, due to the capability provided by the bridging Microsoft Virtual Machine (VM).

To access a Java object from COM, a COM-compatible interface is needed. This is provided by the Virtual Machine. It automatically constructs a COM-Callable Wrapper around the Java object. The wrapper has a set of standard COM interfaces, in addition to interfaces for the original functions exposed by the object. To standard COM objects, the wrapper looks like a canonical COM object. The call on these interfaces will be translated by the wrapper into a call on the internal Java functions.

To access a COM object from Java, another wrapper is needed. The Java-Callable Wrapper is a Java class that has some Microsoft-specific attributes that tell the Microsoft Virtual Machine how to map the Java object to the COM component that it represents. Microsoft has tools to automatically generate Java source files from COM interface definitions. These source files contain special directives that tell Microsoft compiler to insert certain attributes into the generated class files that represent the COM component. Other compilers and virtual machines will ignore these proprietary directives and attributes.

3.3 First Integration Scheme

To notify ArchStudio of the changes the developer makes, Visio needs to maintain a communication path to ArchStudio. This is achieved through several steps.

First, we write a standard C2 component VisioAgent in Java, which will receive events that Visio sends whenever user modifies the architecture design.

Second, to pass the reference for VisioAgent to Visio, we write another proxy object, VisioCOM. The VisioCOM

is written in Java, so it can preserve a standard Java reference for VisioAgent. Instead of creating a Java instance of VisioCOM, a COM Callable Wrapper containing the Java object is created by Virtual Machine. When VisioAgent initializes, it creates this COM Callable Wrapper through a Java Callable Wrapper, and put the COM Callable Wrapper into COM Running Object Table.

When Visio initializes, it retrieves the COM Callable Wrapper for VisioCOM from the Running Object Table using COM services, and get the internal Java reference to VisioAgent, from which a COM Callable Wrapper is constructed by Microsoft Virtual Machine.

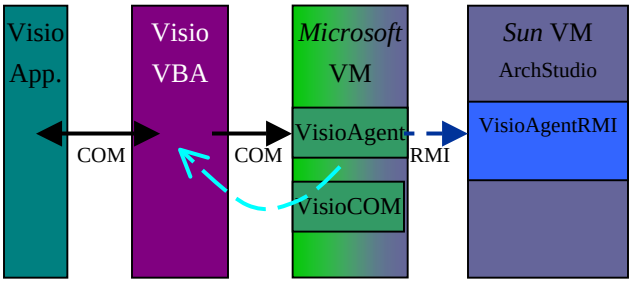

Figure 3, Visio->ArchStudio Communication

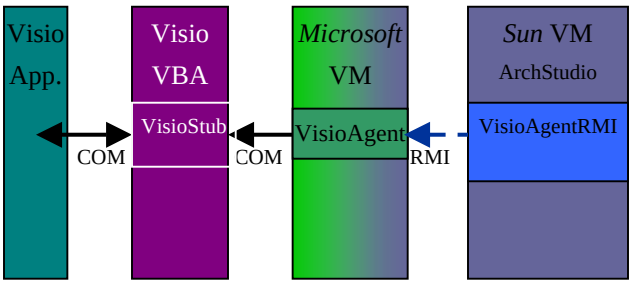

Figure 4, ArchStudio->Visio Communication

This initial setup is designated by the curved dash line in Figure 3. The Visio application and the Visio VBA are COM objects. The Microsoft VM is the bridge between COM and Java. It is a COM object itself, and it hosts two Java objects, VisioAgent and VisioCOM, with the necessary wrappers. The Sun VM is the standard Java VM that runs the rest of ArchStudio.

When Visio tries to notify VisioAgent, it sends a COM message to the COM-Callable Wrapper, and the wrapper translates it into a Java message for VisioAgent. VisioAgent will send an event to the rest of ArchStudio. COM messages are designated as solid arrows in Figure 3.

Figure 4 describes the situation when a notification originates from ArhcStudio. To let Visio receive these notifications, we write a standard COM object called VisioStub and embed it in Visio. VisioStub is in charge of processing events that come from ArchStudio, such as notifications sent when other editor deletes a connector. It will modify the display of Visio to reflect those changes.

During initialization, after Visio retrieves the reference to VisioAgent, it tells VisioAgent the reference to VisioStub. Since VisioAgent is a Java component and VisioStub is a COM component, Microsoft VM will create a Java Callable Wrapper around the COM component and that wrapper will be referenced in VisioAgent.

When VisioAgent receives events from the rest of ArchStudio, it will send a Java message to the wrapper, which translates it into a COM message for VisioStub. VisioStub will send the event to Visio using COM services.

3.4 Second Integration Scheme

The approach outlined above solves the COM/Java integration problem, with a major limitation. The solution requires Microsoft VM, which is only JDK 1.1.4 compliant (Due to the legal dispute between Microsoft and Sun, it will not be updated to accommodate the latest technology.) and runs only on Windows operating system. We would like to eliminate this limitation so the portability and latest development of Java technology will not be compromised.

A standard socket connection can be used to achieve the interoperation. Due to the primitiveness of socket communication, we choose to use Remote Method Invocation (RMI), the only high-level distributed computing primitives available to JDK 1.1.

Two separate RMI servers, one in Microsoft VM (VisioAgent), another in Sun VM (VisioAgentRMI), are constructed. They communicate with each other using RMI to achieve the two-way event communications afore mentioned. The communication is depicted in Figure 3 and Figure 4 by the straight dashed lines.

Now the Java/COM integration happens completely in Microsoft VM, and applications in both VMs can evolve independently to accommodate new requirements.

3.5 Evaluation

Through this two-segment integration approach, we have an integrated event-based software architecture development environment, with capabilities provided from both latest Java technology and commercial graphics editing product. The footprint of the solution is small. While the integration imposes some overhead resulting from several stages of conversion, it still performs reasonably well under an interactive environment. The solution can be freely downloaded with source code. Initial feedback from first users is positive.

The current connector between the Java component and the COM component is custom-made, which requires adding a set of new adapters for each new function. We plan to extend the connector into a standard, adaptive communication channel, so it can be utilized easily by other users to integrate similar components.

The messaging capability of the connector is still rudimentary. The communication pattern is point-to-point, and some explicit references are still needed. While the C2

framework provides rich event functionalities in the Java side, COM's support for events is limited. COM+ provides the capability to dynamically define events and change the subscribers and publishers of events, which greatly loose the coupling of involved parties. We plan to use the features to enhance the capability of the connector.

3.6 Object Identity Problem

We found an anomaly in the Microsoft Virtual Machine. When it generates wrappers at run-time, it does not always preserve the identity for the original object. That is, two different wrappers may be generated for the same original object.

We believe a more transparent translation, which can keep the identity unchanged and generate the same wrapper for the same original object, is possible, using some session information. Since we don't have the source code of Microsoft VM to implement this identification-preserving translation, we have to tag a streamable string identifier with such objects to circumvent this problem. The identity of the string could also be changed, but its content is preserved, and used as a pseudo identifier for the wrapper.

4. RELATED WORK

Researchers have been using COM as a platform for developing new tools and an environment for exploring issues in component-based software development. Neil Goldman and Robert Balzer [3] extended PowerPoint to create a visual design editor generator. Their technology focuses on the generation of a new COTS-based environment given a set of specifications. Our research explores the issues encountered in integrating a desired COTS component to become an integral part of a pre-existing environment. Our component tools benefit from the event notification provided by C2 style.

David Coppit and Kevin Sullivan [1] point out there are three problems in pursuing successful component-based software development models: lack of appropriate models, absence of knowledge about conditions under which such models can succeed, and shortage in understandings for specific promising models. They view Goldman and Balzer's approach as a model using a single component as a platform upon which to build a system, and they propose an alternative model, Package-Oriented Programming, that employs multiple components and integrates them tightly into a single application. They verify the model's potentials for success by integrating several Microsoft products into a reliability research tool. Our research applies the same general model. We agree with their conclusions about the potentials, returns, and risks involved in using the model. Technically Our emphasis lies on integrating two different technologies, COM and Java, within an event-based integration framework.

5. CONCLUSION

Our work demonstrates that event-based integration can be an effective way to integrate off-the-shelf, heterogeneous components to create software architectures. We use it to extend the capability of ArchStudio to include the vast functionalities provided by a COM-based product. While still limited and open for future improvement, the solution we propose shows its usefulness and could be applied in similar integrations.

6. ACKNOWLEDGEMENTS

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. 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 (DARPA), the Air Force Laboratory, or the U.S. Government.

7. REFERENCES

[1] D. Coppit, K. J. Sullivan; **"**Multiple mass-market applications as components". Proceedings of the 2000 International Conference on Software Engineering, p.273-82, 2000

[2] E. M. Dashofy, A. van der Hoek, R. N. Taylor; **"**A highly-extensible, XML-based architecture description language". Proceedings of Working IEEE/IFIP Conference on Software Architecture, p.103-12, Aug. 2001.

[3] N. M. Goldman, R. M. Balzer; "The ISI visual design editor generator". Proceedings of 1999 IEEE Symposium on Visual Languages, p.20-7, 1999.

[4] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, R. N. Taylor; "xADL: enabling architecture-centric tool integration with XML". Proceedings of the 34th Annual Hawaii International Conference on System Sciences, p.9-17, Jan. 2001

[5] N. Medvidovic, R. N. Taylor; **"**A classification and comparison framework for software architecture description languages" IEEE Transactions on Software Engineering, vol.26, no.1, p.70-93, Jan. 2000.

[6] D. E. Perry, A. L. Wolf; "Foundations for the study of software architecture"**.** SIGSOFT Software Engineering Notes, vol.17, no.4, p.40-52, Oct. 1992.

[7] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, D. L. Dubrow; "A component- and message-based architectural style for GUI software". IEEE Transactions on Software Engineering, vol.22, no.6, IEEE, p.390-406, June 1996.