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

Jie Ren
Univ. of California, Irvine
jie@ics.uci.edu

Richard N. Taylor
Univ. of California, Irvine
taylor@uci.edu

April 2003

ISR Technical Report # UCI-ISR-03-2

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
+1 949 824 2776
{jie, taylor}@ics.uci.edu

ISR Technical Report # UCI-ISR-03-2
April 2003

ABSTRACT

Event-based Integration (EBI) is an 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.

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

+1 949 824 2776

{jie, taylor}@ics.uci.edu

ISR Technical Report #UCI-ISR-03-2

April 2003

ABSTRACT

Event-based Integration (EBI) is an 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.

Categories and Subject Descriptors

General Terms

Design, Experimentation, Languages.

Keywords

COM, Java, Event-based Integration

1. INTRODUCTION

Software architecture has been proposed as an effective solution for producing bigger, better and cheaper software [7]. Although there does not exist a common agreement about the concepts and terminologies within the research community, a minimum core of principles could be presented as: 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 [6].

There are many ways to construct components. They can be either developed in-house, or acquired off-the-shelf. While in-house components provide the unique functionalities for a system, what makes Component-based Software Engineering cost-effective are those components acquired externally. The many advantages provided by such 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. Many technologies have been used as connectors, such as pipe-and-filter, remote procedure call, and object request broker. The variants have different capabilities and limitations [9]. Among them, event-based integration (EBI) is very effective in integrating concurrent, heterogeneous components in dynamic environment [8]. 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 ARCHITECTURE STYLE

C2 is an architecture style featuring event-based integration [10]. Its basic tenets include:

- Components communicate with each other only by sending events, which are routed by connectors.
- Components and connectors both have one top interface and one bottom interface.
- Components and connectors are connected in a layered manner.
- Components can be connected to at most one connector at any of its interfaces, while connectors can connect to any number of components and connectors at any of its interfaces.
- Components send request events to upper components for service, 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 [5]. ArchStudio itself is in C2-style. It integrated a set of tools, from both in-house and off-the-shelf, with various degrees of integration.

We are continually expanding the capabilities of ArchStudio. The architecture of the latest version can be depicted as below:

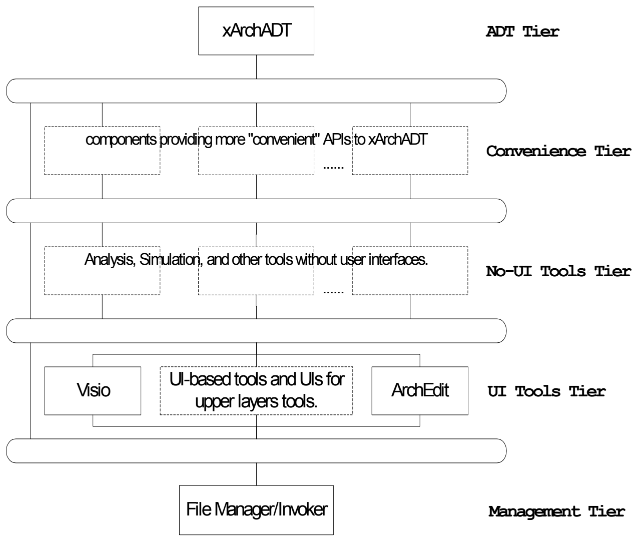
xArchADT
ADT Tier
components providing more "convenient" APIs to xArchADT
......
Convenience Tier
Analysis, Simulation, and other tools without user interfaces.
......
No-UI Tools Tier
Visio
UI-based tools and UIs for upper layers tools.
ArchEdit
UI Tools Tier
File Manager/Invoker
Management Tier

Figure 1, Architecture of ArchStudio

The rectangle designates components. Those in solid line rectangles are complete components, while those in dashed line rectangles are components to be added. The round angle box designates connectors. 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 convenience tier provides easy access to this abstract data repository. Some no-UI tools provide analysis, simulation, and monitoring capability. A set of UI tools is used to manipulate the architectural information graphically. An invoker provides a portal for accessing all integrated tools.

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

The old front end in ArchStudio is Jargo, a tool based on GEF (Graphics Editing Framework) [15].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. While adding these functionalities to Jargo or Mica are theoretically possible, the cost of development would be enormous, and there is no guarantee for the quality of the final result.

The resulting environment is shown in Figure 4. The ArchStudio File Manager is the central portal to the various tools of ArchStudio environment. The main Visio window shows part of the architecture for AWACS (Airborne Warning and Control System). The architecture is described using xADL 2.0. The graphical layout is generated from this description with the help of AT&T Research's open-source Graphviz Dot tool [16].

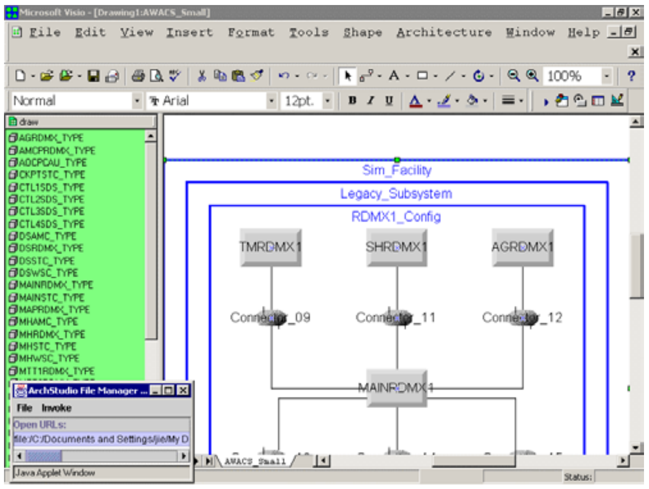

Figure 2, ArchStudio with Visio front end

Visio front end provides the following operations to the architecture designer:

- Create component and connector types, define their interfaces
- Create a component or a connector
- Connect a connector to a component or another connector
- Disconnect a connector from a component or another connector
- Delete a component or a connector and its connected links
- Undo the editing operations
- Group a set of components and connectors into a larger group
- Ungroup a group into its constituents
- Create a sub architecture for a component type or a connector type to support large software architecture

These editing operations will send requests to xArchADT to insert or remove the relevant xADL elements instantly. For example, when a component is created, an element for it is created, with sub elements for its identifier, interfaces, and type.

Since Visio is not the only editor in ArchStudio, it also needs to get the notification when other editors modify xArchADT. For example, when ArchEdit, a generic syntax-directed editor, deletes a connector, Visio will get the notification from xArchADT and delete the corresponding shape. ArchEdit will not delete any connected links of the connector because of its simplicity and generality, but Visio will enhance the correct semantics and delete those links, telling xArchADT to remove the corresponding elements.

In a word, Visio front end maintains a graphical, high-level snapshot of xArchADT, and it can be used to view and modify the states of xArchADT instantly.

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 is now a Microsoft product, and its programming interface has long been COM-based (Any COM-compliant language can be used to develop the customized solution, we choose the built-in Visual Basic for Application), as most other Microsoft products are. We need to bridge the COM world and the Java world.

3.2 Microsoft's Java Virtual Machine

There are several products that enable this interoperation to happen. Sun has an ActiveX Bridge for JavaBeans [13], which enables Java beans to be used in a COM container. It also has an Enterprise Edition Client Access Services COM Bridge [14], which allows COM clients to access EJB components. Linar's J-Integra [11] is an innovative pure Java-COM bridge, which implements COM services in Java. It provides both Java-to-COM and COM-to-Java bridging capability.

We choose Microsoft Virtual Machine [12], 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 [3], the central artifact is the interface, which contains nothing but 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 (IUnknown for COM identity, IDispatch for automation, IMarshal for marshaling and unmarshaling, and IConnectionPointContainer for COM event, etc.), 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 COM interfaces will be translated by the wrapper into call on the internal Java functions. For example, here is a regular Java class:

```
public class VisioCOM {
       private Object      agent;

       public void setAgent(Object _agent) {
              agent = _agent;
       }

       public Object getAgent() {
              return agent;
       }
}
```

An instance of this class can be accessed in VBA as a regular COM object in the following way:

```
Dim visioCOM as Object, visioAgent as Object
Set visioCOM = GetObject(, "VisioCOM")
Set visioAgent = visioCOM.getAgent
```

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. For example, the wrapper COM object for the previous Java class can be turned into the following Java interface and class, with special directives:

```
/** @com.interface(iid=...) */
public interface VisioCOM_DispatchDefault
{
  /** @com.method(...)
      @com.parameters(...) */
  public void setAgent(Object a);

  /** @com.method(...)
      @com.parameters(...) */
  public java.lang.Object getAgent();
  ...
}

/** @com.class(classid=...) */
public     class     VisioCOM     implements
VisioCOM_DispatchDefault
{
  /** @com.method()
      @hidden */
  public native void setAgent(Object a);

  /** @com.method()
      @hidden */
  public native java.lang.Object getAgent();
  ...
}
```

Notice in the previous examples, the Java class is wrapped in a COM object, which can then be accessed in both COM and Java environment. When the access happens in the Java environment, both COM-Callable Wrapper and Java-Callable Wrapper are used. That is, the Java class is accessed not through the original Java class, but through a Java-Callable Wrapper on a COM object that is a COM-Callable Wrapper for the Java class.

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 just creating a standard Java instance of VisioCOM, a COM Callable Wrapper containing the Java object is created by Microsoft Virtual Machine. When VisioAgent initializes, it creates this COM Callable Wrapper through a Java Callable Wrapper, as described in the previous section, and put the COM Callable Wrapper into the 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. After this step, VisioCOM has accomplished its mission.

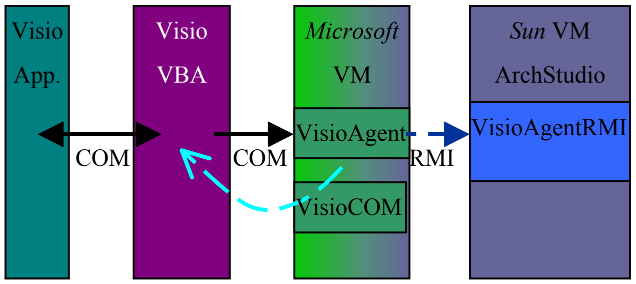

Figure 3, Visio->ArchStudio Communication

The reason we use two objects is that VisioCOM is a simple object whose sole purpose is to transfer the VisioAgent reference to Visio. We want to separate this functionality from other complex and evolving functionalities.

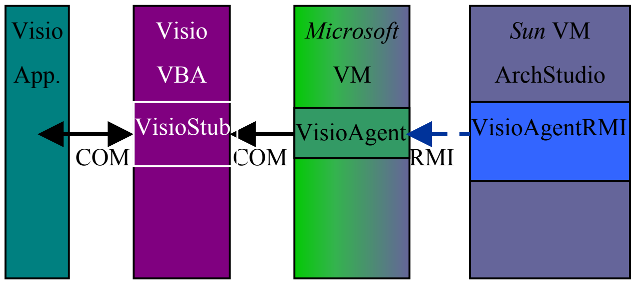

Figure 4, ArchStudio->Visio Communication

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. The 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 ArchEdit 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 deliver 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. The next step of integration is to let the Microsoft VM interoperate with the Sun VM, on which other parts of ArchStudio run. (As a matter of fact, ArchStudio keeps exploiting the continuous developments in Java technology, and the latest version of ArchStudio needs JDK 1.4 to run correctly.)

A standard socket connection can be used to achieve the interoperation. However, it is a low-level programming interface, which means significant development effort is required to provide and maintain the needed communication capability. We choose to use Remote Method Invocation (RMI), the only high-level distributed computing primitives available to JDK 1.1. (RMI support for Microsoft VM is "unofficially" provided through a separate download.)

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, depicted in Figure 3 and Figure 4 by the straight dashed lines.

VisioAgent will send the following requests to VisioAgentRMI whenever the designer performs the corresponding operations in Visio: create a type for component/connector/interface, create a sub architecture for a type, create or remove a component/connector, connect a connector to a component, disconnect a connector from a component, group a set of components and connectors, ungroup a group, and get current components/connectors/connections from the architecture.

VisioAgentRMI will send the following notifications to VisioAgent whenever other tools modify xArchADT in ArchStudio: a component or connector is created or deleted, a connection is created or deleted, and a group is added or removed.

Generally speaking, there will be event traffics whenever the designer modifies the design to change some aspects of the architecture. Comparing to other high volume real time events, most of these design events occur much less infrequently and require less processing due to the little associated data.

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.

There is still some room to improve the performance. One possibility is to replace the late binding automation used in Visio-to-VisioAgent communication by early binding automation. Another possibility is to change the RMI communication primitive to some lightweight solutions. While RMI provides a good balance between productivity and performance, it might be overkill for the current problem. We are not sure whether these technical solutions could result in any significant improvement in user perceived performance.

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. For example, when the functionality of adding a component is needed, VisioAgentRMI needs to be modified so the request can be understood, and VisioAgnet also needs modification so the corresponding notification, a component is added, can be sent back. The changes do little more than relaying the message to the correct receiver. This process is tedious and labor-sensitive. We plan to extend the connector into a standard, adaptive communication channel, using reflective technology, 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 explore these advanced features to enhance the messaging 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. During initialization, a reference to the COM object VisioStub needs to be stored in VisioAgent, and a Java-Callable Wrapper is generated for this COM object. During finalization, this reference needs to be released, but when passed the same VisioStub object, Microsoft VM generates another Java-Callable Wrapper, instead of reusing the original wrapper. These wrappers are different Java objects, making the comparison for identity generating surprising results.

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 each COM object has its identify (the interface pointer for IUnknown), the VM can maintain a map between the identify of the COM object and the corresponding wrapper. When asked for a wrapper of a COM object, instead of always generating a distinct Java object, the VM can ask the COM object for identity and consult with the map. If the identify of the COM object can be found, then the corresponding wrapper in the map should be used. Otherwise a new wrapper is generated and the association is stored in the map. The association can be removed when the objects are no longer needed.

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. When telling the reference for VisioStub to VisioAgent for the first time, in addition to the reference (which will be wrapped in a Java-Callable Wrapper), a unique random string identifier is also passed. During finalization, instead of using the reference, the string identifier is passed to tell VisioAgent that the specific VisioStub is no longer needed. The identity of the string could also be changed, but its content is preserved, and thus used as a pseudo identifier for the wrapper.

4. RELATED WORK

COM is probably the most widely used component technology so far. Researchers utilize it as a platform for developing value-added tools and an environment for exploring issues in component-based software development.

Neil Goldman and Robert Balzer of Information Science Institute(ISI) [4] extended PowerPoint to create a visual design editor generator. The editor consists of two parts. One is the designer, which creates the entities, specifies their properties and connections. The domain engineer can specify the ontology of the design in the generator by providing a set of samples, and the generator can generate the corresponding designer. The other part is the analyzer, which provides the distinctive semantics, such as analysis, execution, and monitoring of the design. This part has to be manually crafted. To coordinate the cooperation between the generic editor and the specific analyzer, they use an analysis router between the designer and the registered analyzers, design a designer-analyzer protocol based on DCOM to support incremental analysis, transactional analysis, and result report. Their technology focuses on the generation of a new COTS-based environment given a set of specifications. The architecture is shown in the following diagram.

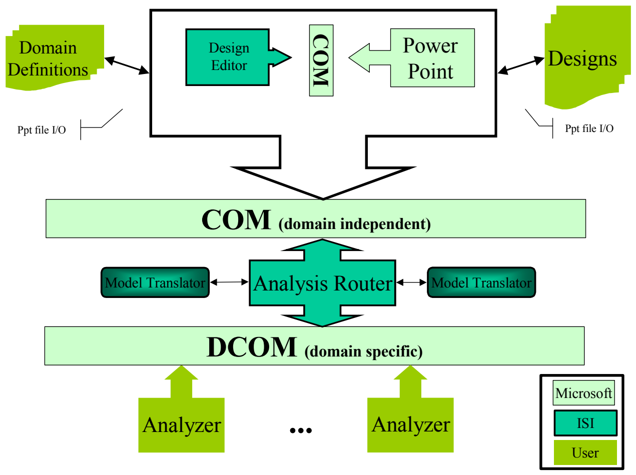

Figure 5, ISI Design Environment Architecture

Our research explores the issues encountered in integrating a desired COTS component to become an integral part of a pre-existing environment. The communication pattern in their research emphasizes the mapping between a generic front end and various specific backends. The control pattern they follow is one in which the dominant thread of control, Analysis Router, dispatches analysis task to analyzers that could reside on different processes or even different computers. Our component tools benefit from the event notification provided by C2 style. It is interesting to note they used an event observation hook at the operating system level to circumvent the lack of event notification in PowerPoint 97, and they planed to investigate Visio since it provides better event notification mechanisms.

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 that uses 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 evaluate their model by a successful case study that builds a computational tool for reliability engineering, which can be seen as an industrially strong representative of an important class of systems. They conclude the model has potential to succeed, and even today it can produce significant returns, although it has certain risks.

The environment, Galileo, is a dynamic fault tree tool for reliability analysis. Its architecture is shown below. The main mediator coordinates the views and the analysis engines. Visio is used as a graphical editor, Word is used as a textual editor, and Internet Explorer is used to provide online help. All the COTS packages have a COM-based application programming interface that is used for integration. During the development process, they had also used Excel and Access.

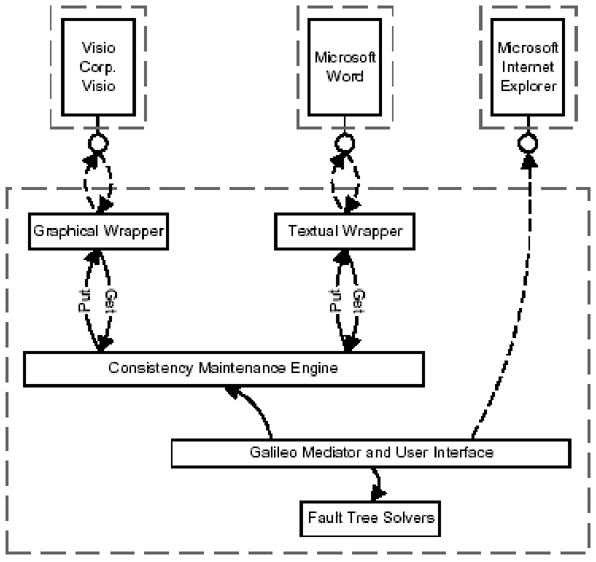

Figure 6, Galileo Architecture

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.

During years of research, their group has been exploring issues related to component capabilities and limitations, evolution of functions and user interfaces, and unexpected and undocumented behaviors. They found early versions of Access did not support Active Document Interface, which made it architecturally incompatible with other members of the environment, and they had to rely back on file system for data storage. But the newly available Internet Explorer proves to be architecturally compatible and very useful. For us, while Microsoft VM provides an easy integration path between COM and Java, due to the lawsuit between Microsoft and Sun it is clear that no improvement can be expected for the VM, and the integration between newer COM and Java technologies thus becomes more difficult.

We encountered some similar problems as they did, but we were happy to find that some of their early troubles have been solved during the evolution of the COTS products. We believe this demonstrates one advantage of using commercial COTS products: continuous support and upgrade from the vendor.

For example, they found providing high-level operations (such as connecting two gates) based on Visio primitives is not easy. We have some difficulty in providing undoing of high-level operations, because the user interface exposes all the underlying low-level operations, which requires the user to understand the implementation details to perform a semantically correct undoing. They found the length of shape identifier limited. We suffer from the same obstacles.

They found Visio did not expose the "delete" event adequately, which forced them to give up an incremental update scheme and turned to a batch-oriented editing. We didn't experience this problem. Similarly they reported problems of multiple-page support in early Visio versions, while we did not encounter such issues during our exploration.

An interesting issue is layout. While they gave up their own layout algorithm and used Visio's built-in layout functionality due to its speed improvement over cross-application communication, we turned to an external layout package, Graphviz, due to Visio's limitation in processing extensive use of groups. Visio can group a set of shapes under a parenthood of a group, and the child's coordinates are parent-relative. When there are many shapes and several levels of groups, Visio spends much time in coordinates recalculating when there is a position change. We found that unacceptable, and decided to calculate and set the coordinates using Graphviz.

It is instructional to notice the difference in data consistency models between their tool and ours. They use a batch-oriented consistency scheme instead of an incremental one: one tool modifies the data, and submits the change so others can learn about it. They adopt this model because the reliability data lack a regular structure, the editing tool does not provide all required events notification, and they feel the automation performance is not adequate. We choose incremental update based on event notification, because the data we are dealing with, the architecture description, has a clear structure, which greatly reduces the amount of information, thus the burden of processing, needed for each event. Through experimentation we find the responsiveness of the tool is good enough for interactive operations on the scale of data we process, thus vindicating the COTS product has good support for notifications and performance.

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. REFERENCES

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

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

[3] Eddon, G.; Eddon, H. Inside COM+ Base Services, Microsoft Press, 1999

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

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

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

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

[8] Reiss, S.P. Connecting tools using message passing in the Field environment. IEEE Software, vol.7, no.4, July 1990. p.57-66.

[9] Shaw, M.; Garlan, D. Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996

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

[11] http://www.linar.com/

[12] http://www.microsoft.com/java

[13] http://java.sun.com/products/plugin/1.3/docs/script.html

[14] http://developer.java.sun.com/developer/earlyAccess/j2eecas

[15] http://gef.tigris.org

[16] http://www.research.att.com/sw/tools/graphviz