Visualizing Software Architecture with Off-The-Shelf Components

Jie Ren, Richard N. Taylor
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
+1 949 824 2776
{jie, taylor}@ics.uci.edu

ABSTRACT

Software Architecture provides a high-level model of the components and connectors that comprise a complex system. Visualizing both its static description and dynamic execution facilitates understanding of its key properties by software architects. Off-the-shelf components provide a rich foundation on which an advanced architecture visualization tool can be constructed. This paper discusses our experience in integrating a set of off-the-shelf components to create an event-based software architecture development environment that supports the visualization of the description and execution of certain architectures. We discuss the benefits and obstacles of integrating Common-Off-The-Shelf (COTS) components, describe the design and implementation of the visualization solution, and report some experiences from this effort.

1. INTRODUCTION

Software architecture has been proposed as an effective solution for producing better and cheaper software [18]. Although there does not exist a common agreement about the concepts and terminologies within the architecture research community, a minimum core of principles could be presented as: 1) a software system is composed of components and connectors, 2) components are loci of computation, 3) connectors are loci of communication, and 4) a specific set of components and connectors form the configuration of the software system [21].

Software architects generally will use an Architecture Description Language (ADL) to model architecture. To fully achieve the potentials of architecture-driven development, software architects need the support of an architecture development environment. The environment should provide architects the capability to model the architecture and manipulate the resulting model.

ArchStudio 3 is an environment we have used as a test bed on which to explore solutions to address problems encountered in architecture-driven software development.

To construct the model of systems in the ArchStudio 3 environment, an editor is needed. The editor should let the architect design the components, connectors, and their connections. If the editor can be graphical, it would greatly ease its usage and boost its applicability.

A useful and interesting feature of an environment is visualization. A visualization tool can display the underlying architecture graphically, facilitating an architect's constructing and understanding of the model. It can provide multiple views, helping an architect manage the complexity. It can even visualize the execution of the architecture, enabling an architect to understand its dynamic properties [12].

A sample system we chose to test our solutions was the AWACS system (Airborne Warning and Control System). The system is large and complex, having hundreds of components and connectors. It is also a legacy system, having been running on a mainframe for many years. There is a need to enhance and upgrade the system so it can utilize the latest technologies to perform more functions.

To test our solution, we tried to model the architecture of the AWACS system and visualize the resulted architecture. The architecture could then be edited graphically. We then executed the architecture and observed whether the desired properties were achieved.

Building a graphical editor is a routine task. However, it's not easy to provide advanced functionalities, even performing the basic operations is easy.

There has been active research on software visualization [5]. However, few of them directly address architectural level software artifacts as defined by the architecture research community. Besides, most of these tools are developed from scratch, duplicating much basic functionality and lacking other advanced features. These custom made tools cost the researchers unnecessary effort that could have been better spent on core research issues. Users of these tools might find some desirable features are missing. These tools are not always applicable to large problems

With the maturity and availability of useful off-the-shelf components, we felt it is possible and cost-effective to develop a software architecture editing/visualization tool with these components.

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 impose 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 [2].

The following criteria was used as the basis for choosing the components of the editing/visualization tool: 1) they must provide both basic and advanced graphical editing/displaying capabilities; 2) they should be scalable, having the capability to handle a system as complex as our AWACS sample system, 3) they should be easily extensible so they can be plugged into the ArchStudio environment and accomplish required functionalities.

In this paper, we present our experience in integrating a set of off-the-shelf components to create a software visualization tool for an event-based software architecture development environment. The resulted environment is shown in Figure 1. Section 2 introduces the background of this research. Section 3 details the integration activity. Section 4 describes the resulting visualization tool. Section 5 discusses related work. Section 6 concludes the paper.

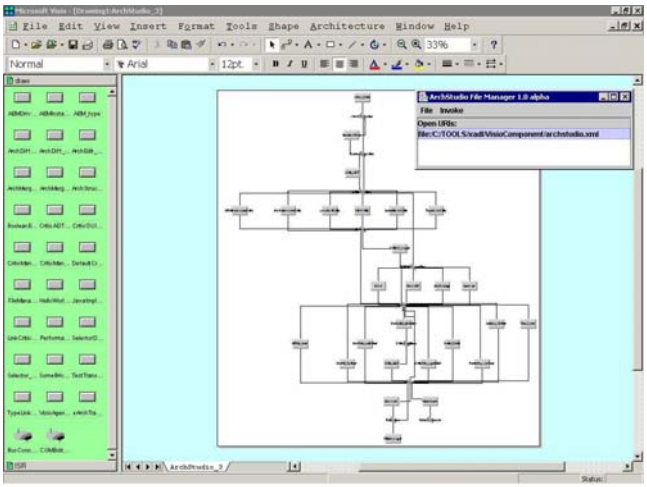

Figure 1, Visio front end for ArchStudio

2. BACKGROUND

After surveying the available products, we chose Microsoft Visio as the foundation to develop a visualization tool that is part of the ArchStudio 3 software architecture development environment. ArchStudio 3 is constructed in accordance with C2 architecture style. It supports architecture modeling using xADL Architecture Description Language. We picked Graphviz as the underlying layout engine.

2.1 C2 Architecture Style

C2 is an architecture style featuring event-based integration. Event-based integration (EBI) is very effective in integrating concurrent, heterogeneous components in dynamic environment [19]. 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.

C2's basic tenets include [23]:

- 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.

2.2 xADL Architecture Description Language

xADL 2.0 [3] is a highly extensible XML-based ADL. It supports run-time and design time modeling, architecture configuration management, and model-based system instantiation. xADL 2.0 is defined as a set of XML Schemas, including: Instance (defining run time architecture instances), Structure and Type (defining design time architecture structures (components, connectors and their types)), Boolean Guard/Options/Variants/Versions (defining configuration management support), Implementation (defining implementation mapping from architecture elements to underlying implementations), Message Rules (defining run time event exchanges), and Structural Diffing (defining architecture evolution). An architecture definition written in xADL is an XML document constructed using these schemas. xADL covers many aspects of architecture development. If new modeling needs arise, it can be extended by defining new schemas.

2.3 ArchStudio 3 architecture development environment

A software development environment, ArchStudio 3, was used to support the development of software in C2 style [28]. 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.

The capabilities of ArchStudio are continually expanding. The architecture of the ArchStudio 3 environement is depicted in Figure 2.

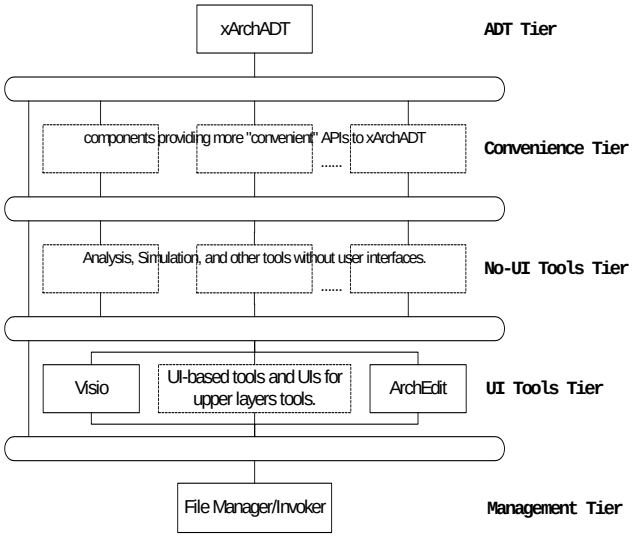

Figure 2, Architecture of ArchStudio

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 the architectural model expressed in xADL 2.0. It contains all available component and connector types, their implementations, and current components, connectors, and their links. 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.

The original front end in ArchStudio was Jargo, a tool based on GEF (Graphics Editing Framework) [26]. 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.

2.4 Visio

There are many drawing products. We decided to use Visio, an industry-strength graphics-editing product, as the basis for our new graphical editing/visualization front end. We chose Visio based on Visio's popularity, support, functionality, and customizability.

Visio has long been a popular drawing tool. It is now part of the Microsoft Office suite, which improves the applicability of our solution. 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 is theoretically possible, the cost of development would be enormous, and there is no guarantee for the quality of the final result.

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 application fields. We used it to integrate Visio into ArchStudio and made the resulted solution the front end.

2.5 Graphviz

We chose Graphviz [8, 27] from AT&T Research as the core layout engine for static visualization. Comparing to other graph drawing tools such as DaVinCi, Graphlet, and VCG, Graphviz is outstanding in both the appearance of the resulting graph drawings and tool usability [17]. It has been widely used in information visualization. Its input is a logical directed graph, into which many software architectures can be mapped. Graphviz can generate aesthetic diagram very fast, and it scales to large and complex diagrams.

Figure 1 depicts the Visio frond end and ArchStudio's portal, the File Manager. The architecture shown in Visio is of ArchStudio 3, visualized by Graphviz from a xADL description of the environment.

3. INTEGRATING OFF-THE-SHELF COMPONENTS

3.1 Microsoft's Java Virtual Machine

Even though the overall philosophy and underlying implementation of ArchStudio and Visio are quite different, both can be programmed using events. (Visio's events are mapped to synchronous COM procedure calls). 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 chose the built-in Visual Basic for Application) [6], as most other Microsoft products are. We needed to bridge the COM world and Java world.

There are several products that enable this interoperation to happen. One innovative solution is Jintegra from Intrinsyc [29], which implements DCOM in pure Java. However, it is not free, and its deployment requires further setup, which does not match Visio's file-based template solution. We chose Microsoft Virtual Machine (VM) [25], because it provides both COM-to-Java and Java-to-COM conversion, it is readily available with the Microsoft operating system without any further charge or extra installation, and it meets Visio's deployment requirements.

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.

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.

3.2 Integration Schemes

3.2.1 Running ArchStudio within Microsoft VM

At first, we run ArchStudio in Microsoft VM. Visio needs to maintain a communication path to ArchStudio so it can notify ArchStudio of the changes the architect makes and receive information about events happening in ArchStudio. The two-way communication is achieved as following.

First, we wrote a standard C2 component VisioAgent in Java, which would get called whenever the architect modifies the architecture design. A reference to the component is passed to Visio. During the passing process Microsoft Virtual Machine constructs a COM Callable Wrapper for this Java component.

When Visio tries to notify VisioAgent, it calls the COM-Callable Wrapper using standard COM mechanism. The wrapper translates this COM call into a Java function call on VisioAgent. VisioAgent will send an event to the rest of ArchStudio, using C2's event notification service.

Second, to let Visio receive notifications from ArchStudio, we wrote a standard COM component called VisioStub and embedded it in Visio. After Visio receives the reference to VisioAgent, it passes VisioAgent the reference to VisioStub. During this process Microsoft Virtual Machine will create a Java Callable Wrapper around the VisioStub COM object.

When VisioAgent receives events from the rest of ArchStudio, it will make a Java function call on the wrapper, which translates it into a COM call on VisioStub. VisioStub will handle this call by calling Visio using COM mechanisms.

3.2.2 Interoperation with Sun VM through RMI

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.)

We used Remote Method Invocation (RMI) because it was 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 (reuse the VisioAgent component), another in Sun VM (named VisioAgentRMI), are constructed. They communicate with each other using RMI to achieve the two-way event communications afore mentioned. Now the Java/COM integration happens completely in Microsoft VM, and applications in both VMs can evolve independently to accommodate new requirements.

The complete architecture is depicted in Figure 3. The Visio application, Visio VBA, and VisioStub are COM objects. The Microsoft VM is the bridge between COM and Java. It is a COM object itself, and it hosts the special Java object VisioAgent, with the necessary wrappers. The Sun VM is the standard Java VM that runs the rest of ArchStudio.

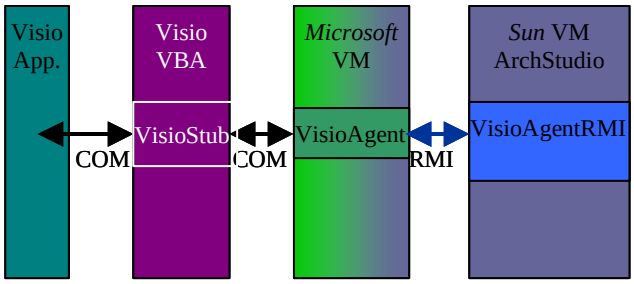

Figure 3, ArchStudio<->Visio Communication

3.2.3 General COM/Event Bridge

The above integration between the Java component and the COM component is custom-made, which requires adding a set of new adapters on both sides for each new function. The new changes do little more than relaying the message to the correct receiver. This process is tedious and labor-sensitive.

Another limit of the above integration is it only supports point-to-point communication, lacking the capability to notify multiple interested event consumers.

We are extending the current connection into a standard, adaptive communication channel, using serialization and reflective technology. The new channel is a general bridge between COM components and event producers/consumers, so it can be utilized easily in other similar integration situations. A first prototype has been developed.

3.3 File-based Integration with Graphviz

Currently the integration with Graphviz is file-based. An input file is generated in Visio for Graphviz, and its output is read back and analyzed. A new COM interface to Graphviz recently became available. We are planning to explore the possibility of integrating Graphviz using its native COM interface.

4. VISUALIZING SOFTWARE ARCHITECTURE

In this section, we first describe how the Visio solution can be used as a graphical editor, and then we describe the static and dynamic visualization features of the solution.

4.1 Visio as a graphical editor

The 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 sole 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 enforce 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.

4.2 Visualization of Static Structure

4.2.1 Extensible visual notation

The first decision about architecture visualization we needed to make is how the elements of software architecture should be visually represented. Using home-made visualization solution the developer has free choices, but the developers have to implement all features they want. In a solution based on a commercial drawing product, the notations should conform to the established rules of the product, so the general user could reuse their knowledge about the existing product. As a return, the notations can utilize many of the built-in features of the product to enhance its appearance and interaction capability.

Visio uses a stencil to hold a set of masters. Each master is a representative of a certain type of shape. When the user drags a master from the stencil and drops it onto the drawing page, a new shape, based on the master type, is created for the user. The user can connect the shapes later on. This master/shape relationship matches very well into xADL's concept of type/instance. That is, each component or connector type is represented by a master. To create one instance of the type, the user can drag and drop a master onto the drawing page.

Visually the three most important constituents of an architecture are the components, connectors, and the links between them. A master is designed to represent a component type. Another master is designed to represent a connector type. We reused a master from one of Visio's built-in stencils because it provides the functionality.

The three of the masters are "meta" masters, because they are used to generate other masters. When an architect designs a new architecture, she can have many component and connector types collected from different sources, either developed in-house or acquired externally. Each type should be represented by a distinct master, so the architect can easily grasp the properties of each architectural constituent by simply looking at the visual representation. The Visio visualization solution will generate one master for each type held within xArchADT. Currently all components types are represented using the meta component master's icon. So do all connector types. For example, in Figure 2, the left pane is the stencil generated from ArchStudio 3's type descriptions. The rectangles in it are masters for component types, one for each. The two round boxes at the bottom row are masters for connector types. It is possible for each master to have its own visualization icon, after the designer of each component and connector type provides its own visualization in addition to the functionality. Generating stencils based on available types provides architects with visual notations that can be extended to suit their specific needs.

The meta masters use some format features of Visio to enhance its visual appearance, such as the roundness of the corner and the shade of the shape. There are many other advanced built-in features, all available through simple customization. This is a great advantage for commercial product-based solution.

There is little customization on the master for link. In traditional box-and-line diagrams, a box stands for a component and a line stands for a connector. A line can have many visual annotations to designate its properties. Here, a link is a simple connection between components and connectors. All complex properties that used to associate with links should be transferred to connectors, which server as the loci of communication aspects of an architecture.

4.2.2 Visualizing architecture description

While the Visio solution can be used to construct an architecture description graphically, it can also be used to visualize an existing architecture description. The description can come from various sources, such as through reverse engineering.

We chose Graphviz as the underlying layout engine. While Graphviz has many other capabilities, we only used it to layout the constituents of an architecture so Visio can get the information that is necessary for it to position the components and connectors.

Graphviz accepts an input file that is a description for a directed graph. The description uses the form "A->B" to designate there is an arc from node A to node B. We used this to express the link between a component and a connector.

This differs from using Graphviz to visualize an architecture description in a traditional box-and-line fashion. For example, if there is a connection between component A and component B, it can be simply expressed as "A->B". Graphviz would then generate a visualization in which there is a box for component A, another box for component B, and a line between them for the connection. But in a xADL description, all architectural connections are expressed through connectors, and they would be visualized as boxes to designate their architectural importance. A central tenet of architecture driven development is the explicit expression of architectural connections as first-class connectors. It is natural and important to retain this explicit notion in the visualization. While visualization eventually produces a box-and-line diagram of one kind or another, the difference of visualization between the two architecture description models is summarized in Table 1.

Architecture Concepts to be Visualized	Visualized In Box-And-Line-like Description	Visualized In xADL-like Description
Components	Box	Box
Connectors	Line	Box (different type)
Links	(Implicit)	Line
Input to Graphviz	Component-> Component	Component-> Connector; Connector-> Component

Table 1, Visualization of two types of descriptions

In some architecture the direction in "A->B" does not matter, but in others the direction has a significant impact on the final result of visualization. For example, C2 has a notion of top and bottom (see Section 2.1), so it is best for the visualization to retain this notion. Since the visualization aspect is not generally considered architecturally important, no modeling notions have built-in support for it. We used the ordering of the components in an architecture description to carry this extra directional information, assuming the component that appears first as the one at the top of the relation. This hint works well for our current solution, achieving the desired effect without obscuring architecturally important concerns. We are exploring the suitability of extending the language to explicitly support visualization.

Besides the "A->B" notation, there is other information that needs to be passed to Graphviz. One is [shape=box] for components. Connectors will have the default value, which is [shape=circle]. What is important is not whether it is a box or a circle, but two distinct types of shapes are utilized. Graphviz will then use this information to generate more appealing layout. Another information is the visual dimensions of each component and connector. This information is extracted from the masters of each type.

Based on the architecture model contained in xArchADT, Visio solution generates an input file that contains the links between components and connectors, the order of components and connectors, the type, the width and the height of components and connectors. Graphviz reads the file, lays out the components and connectors using its internal algorithm, and outputs a file that contains the arrangements of the components and connectors. Visio visualization solution will create a shape for each component and connector using a master for its type, read the output file, position the component and the connector with the coordinates contained in the file.

While Graphviz generates both nodes and links between them, Visio visualization solution does not use any output for links. The solution connects the components and connectors with the Visio's built-in flexible links, which is easier to use and manipulate in Visio.

The following are excerpts of the xADL description for the architecture of AWACS. The architecture is reverse engineered from an AWACS simulator. The complete architecture contains more than 40 component types, more than 120 components, more than 200 connector, and more than 400 links. The xADL file has more than 10000 lines.

```
<componentType id="MAINRDMX_TYPE">
</componentType>
......
<connectorType id="CONNECTOR_TYPE_1">
</connectorType>
<component id="MAINRDMX1">
        <type href="#MAINRDMX_TYPE"/>
        <interface id=”MAINRDMX1_I_AOCPCAU1”/>
</component>
<connector id="Connector_02">
        <type href="#CONNECTOR_TYPE_1"/>
        <interface id=”Interface_B_02”/>
</connector>
<link id="Link_MAINRDMX1_02">
<point>
<anchorOnInterface href="#MAINRDMX1_I_AOCPCAU1"/>
</point>
<point>
<anchorOnInterface href="#Interface_B_02">
</point>
</link>
......
```

Figure 4, Excerpts of xADL description

The following figure is the result of visualization:

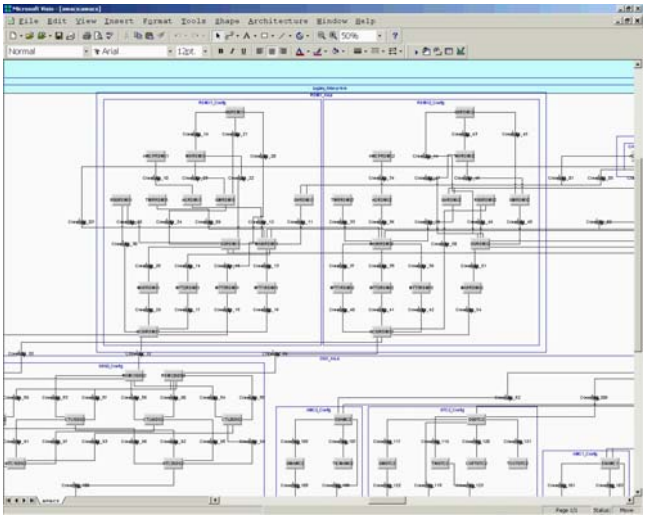

Figure 5, Visualization of AWACS architecture

It is worthy to note that this visualization process only happens when the architecture description needs to be visualized for the first time. After the initial visualization, in subsequent graphical editing it becomes the architect's responsibility to maintain the graphical layout of the architecture. To maintain the unique appearance of C2 architecture style, the solution has some support in aligning the graph according to the canonical C2 guidelines, *after* the architect edits the architecture.

4.2.3 Supporting complex architecture

A good ADL language and a good set of support tools should enable an architect to master the inherent complexity of the architecture. xADL provides some constructs that help architects organize the architecture elements in an easy to understand way. Visualization solutions should provide graphical support for them

One such feature of xADL is group. A group can contain any architectural elements (including sub groups), for any purpose. For example, a set of components and connectors that will be deployed in one physical machine can be grouped together. A set of components that share a single repository can also be grouped together. Visio provides a built-in support for grouping shapes. The visualization utilizes it to support the group of architectural elements. That is, all shapes that correspond to elements in an architecture group will be organized under a Visio group. To achieve better visualization effect for architectures that contain groups, the solution puts all members of a group into a subgraph when it constructs the input file for Graphviz, so the layout engine will lay out these members close to each other. This illustrates the close relationship between the underlying architectural model, the operation of the layout algorithm, and the final rendering by the drawing product. In Figure 5, the bounding large boxes designate groups.

Another complexity management feature of xADL is its support for sub architecture. For each component and connector type, a sub architecture can be defined, containing the internal structure of the component and connector type. The internal structure contains its own components and connectors. In the higher level, the whole structure will be represented by a single component or connector type. Thus the complexity is greatly reduced. Visio solution displays each sub architecture in its own drawing page, utilizing Visio's multiple-page support. The first page is the top level architecture, the other pages define sub architectures for complex component and connector types.

Visio solution provides an architecture browser to help architect manage the complexity and browse the architecture [22], as shown in Figure 6. On the left pane, the top level architecture and all sub architectures are displayed. On the right pane, the components and connectors of the current architecture are displayed, organized according to the group structure. Architect can select an element from the pane, and the corresponding architecture, component or connector will be highlighted in the visualization.

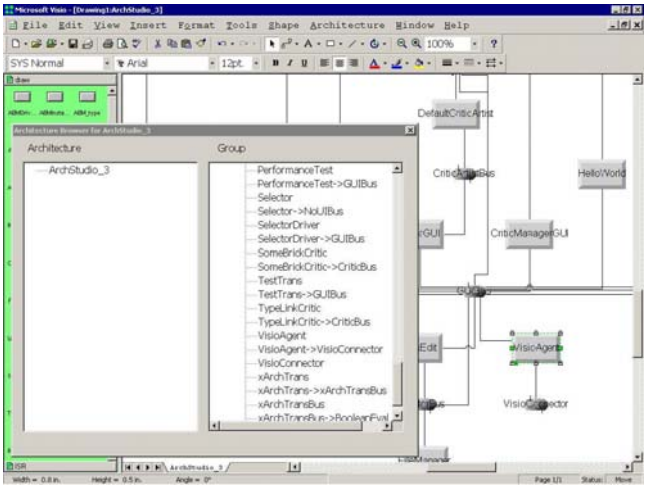

Figure 6, Architecture Browser

4.3 Visualization of Dynamic Execution

Visualization of dynamic execution is important, because it provides information that cannot be derived from static structure. This information can be used in understanding, analyzing, and debugging the architecture.

xADL supports the mapping between architectural elements and their implementations. Currently, a Java implementation is defined so that each component type and connector type has a Java class as its implementation. When ArchStudio 3 is presented with a xADL description, it instantiates the corresponding Java object for each component and connector, using the Java class that is defined for the associated type. It connects the components and connectors together, puts them into execution after the architecture is fully instantiated.

ArchStudio supports event-based architectures. During execution of the architecture, each component achieves its functionality by exchanging events, while the connectors provide the necessary event routing and filtering capability. Using the integration technology described in section 3.2, a

component in Visio is developed, listening for events that happen in ArchStudio. ArchStudio's execution engine will send out events during the execution of the architecture. The listening component will display each occurrence, showing the sender, the receiver, and the information for each event. When a new event comes, it replaces the current display with the newly arrived one, visualizing the execution in an animation. In Figure 7, the highlighted component (the rectangle) sends an event to the highlighted connector (the rounded box). The name of the event is displayed along the link between them.

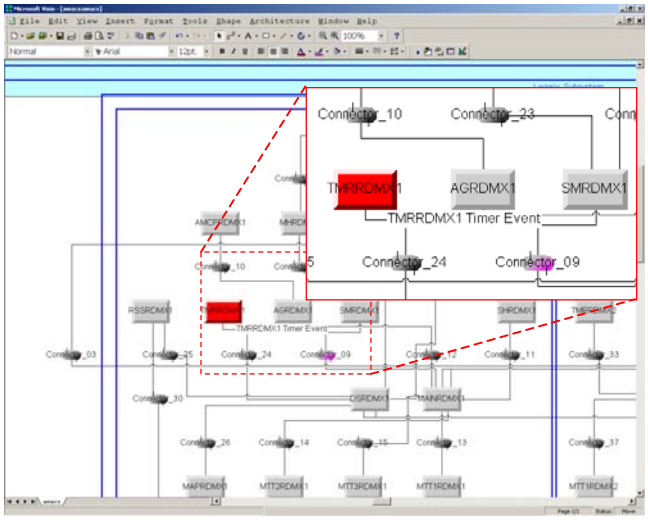

Figure 7, Visualization of Architecture Execution

4.4 Summary

Besides serving as a general purpose graphical editor, our solution can visualize an architecture description, using graphical notations defined by an architect, thus the architect can more easily understand the structures and relationships contained in a description. The solution uses sub architectures, grouping of related elements, and browsing to facilitate the architect in dealing with a complex architecture. Its animation of architecture execution reduces obstacles that architects need to overcome in analyzing and debugging the architecture. In short, the visualization solution is useful in helping architects construct, understand, analyze, and maintain complex software architectures.

5. RELATED WORK

5.1 Visualization in other Architecture Description Languages

Software architecture is an active ongoing research field. Several architecture description languages have been proposed as foundations for modeling, analysis, and execution. In this section we survey the various languages and the support they provide for architecture visualization.

AcmeStudio is a design tool for Acme [11]. It is developed from scratch. It supports visual editing of Acme design, with user-definable visual styles. It tightly couples the model under design and its visualization. It does not support architecture execution and its visualization.

Aesop [10] is a system to build customized software architecture design environments for domain specific styles. Its graphical capability is based on Tcl/Tk, and it provides a library of pre-built visual classes that can be used in constructing environment for new style. It provides some support for architecture execution, but does not enable its visualization.

UniCon[20] enables designer to provide new icons for components, connectors, and roles. It defines its own set drawing commands to support this. It does not support execution visualization.

Wright [1] is a language based on communicating sequential process that can be used to model and check the properties of a dynamic architecture precisely. Its tool support does not provide visualization capability.

Rapide [13] is an executable language that uses partial order events to model the interaction among components. One of its tools, Partial Order Viewer, visualizes the log of events that happened in architecture execution. It has a facility to filtering out uninterested events from the visualization. Its visualization of execution is post-mortem, while our solution animates execution during its occurring.

SADL [16] is a language to define hierarchical structures that can be checked formally. Its toolset focuses on the formal checking and analysis functionality, providing no support for graphical editing or visualization.

The toolset of Darwin [14], Architect's Assistant, supports visual constructing, analysis, and execution of distributed, dynamic architecture whose behavior is expressed in finite state process. In addition to display structures, it can visualize problems found during analysis and state transitions happening in execution. It is a tool developed in-house.

In summary, visualization efforts so far are based on tools developed from scratch, only a few of them provide enough coverage of both static architecture and dynamic execution.

5.2 Integrating Commercial Products

There are some other research efforts that explored utilizing off-the-shelf components for developing value-added tools.

Neil Goldman and Robert Balzer [9] 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.

David Coppit and Kevin Sullivan [2] 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.

Our research applies the same general model. 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.

They also used Visio as the graphical editor for their environment, Galileo. An interesting issue is about 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.

Alexander Egyed and Philippe Kruchten designed Rose/Architect [7]. It extends Rational Rose, a UML-based modeling tool, using its APIs. While it provides some interesting features, such as plane for organizing architecture views and heuristic rules for constructing missing relationships, its UML basis limits its suitability in software architecture modeling and visualization. As discussed in [15], while UML can be extended to support architecture description concepts, it is not intended to be used as an architecture description language. It lacks direct support for components and connectors, buries possible extensions in other intrinsic constructs, and makes architects' modeling tasks unnecessarily complex and difficult. Rose/Architect only supports one of the several types of UML diagrams, class diagram. This further limits its applicability.

5.3 Other forms of visualization for software architecture

John Grundy and John Husking developed SoftArch [12], an environment that can visualize high-level static and dynamic aspects of software structure. It focuses on continuous refinement from object-oriented analysis to design and its final implementation. They used an extensible visual language that is UML-like. The language has special associations that support mapping between high-level models and low-level designs and implementations. We feel their work has two limitations. First, it does not use any first-class architecture description language to model software architecture. Its UML-like notation does not provide full and natural support for architectural-level concerns. Second, its visualization of dynamic execution is rather low-level, focusing on events such as object creation and method invocation. While they are easier to achieve, given the reflection support provided by the Java run time system, these events do not provide enough information and insight for the execution at architecture level.

Some researches have a focus on reverse engineering, such as [22, 24]. They try to extract design information from source code and visualize the retrieved design. While it is useful in helping developers understand an existing system, generally they are concerned about module structures and the call relations between them, lack an explicit architecture model, and do not cover dynamic aspects. They are like canonical software visualization research that centers on programs, such as FIELD [19].

6. CONCLUSION

We used a set of off-the-shelf components (Microsoft Java Virtual Machine, Microsoft Visio, and Graphviz) to develop a novel visualization solution that supports visualizing both static and dynamic properties of software architecture. By reusing and integrating functionality provided by these components, the focus of the new solution can be concentrated on the really unique and innovative issues. The contribution of this research lies in its demonstration of the effectiveness of development with COTS components, its exploration of different integration technologies, and the software architecture visualization solution resulted.

General COTS integration could suffer from some limitations, such as the inflexibility of the COTS component and the vicissitudes of the vendor. Open source components are well suited to deal with the two factors. They can be modified to suit the specific needs, even if the original author no longer supports them. Graphviz is such a component. If no such components are available or their functionality does not meet the basic need, a component with ample programming interface and documentation will be the second best choice to lessen the first limitation. Choosing a component from a company with long history can reduce the risk of the component. When everything technically is equivalent, business factors may decide which component will be a better choice for integration.

Visio has good adaptability and assured support, making it a suitable candidate for integration.

As outlined in [5], more visualization research is needed in abstract execution and evolution of static structures. While this research explores the visualization of high-level structures and its dynamic aspects, further is needed. The current visualization of dynamic execution animates each occurrence of event between a component and a connector. Two improvements are under exploration: 1) Use filtering mechanism so only interested events are animated, 2) Use analysis techniques to show the events as being sent from one component to another component (This is what is significant at the application level), hiding the display of the messaging infrastructure.

Another area that is worth exploring is to visualize architecture analysis. Currently ArchStudio uses a set of critics to perform architecture-related analysis. Its result can be visualized so the analyst can be more effective in determining the relevant properties.

The dynamic and static evolution of software architecture is an important issue [4]. xADL defines schemas for expressing evolution. ArchStudio has support mechanism for expressing, evaluating, and enacting the changes. We are exploring the possibility to visualize these activities.

7. 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.

8. REFERENCES

[1] Allen, R.; Garlan, D. A Formal Basis for Architectural Connection, ACM Transactions on Software Engineering and Methodology, July 1997, vol.6, no.3, p.213-249.

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

[3] 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.

[4] Dashofy, E.M.; van der Hoek, A.; Taylor, R.N. Towards Architecture-based Self-Healing Systems, Proceedings of Workshop on Self-Healing Systems ’02.

[5] Diehl, S., Software Visualization, Lecture Notes in Computer Science 2269, Springer, 2002, p.347-353.

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

[7] Egyed, A.; Kruchten, P. Rose/Architect: a tool to visualize architecture, Proceedings of the 32nd Hawaii International Conference on System Sciences, 1999.

[8] Gansner, E.R.; North, S.C. An open graph visualization system and its applications to software engineering. Software: Practice and Experience, vol.30, no.11, September 2000, p.1203—1233.

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

[10] Garlan, D.; Allen, R.; Ockerbloom, J. Exploiting Style in Architectural Design Environments, Proceedings of SIGSOFT '94, p.175-188.

[11] Garlan, D.; Monroe, R.; Wile, D. ACME: An Architecture Description Interchange Language, Proceedings of CASCON’97.

[12] Grundy, J.; Hosking, J. High-Level Static and Dynamic Visualization of Software Architectures, Proceedings of 2000 IEEE Symposium on Visual Languages, p.5-12.

[13] Luckham, D.C.; Kenney, J.J.; Augustin, L.M.; Vera, J.; Bryan, D.; Mann, W. Specification and Analysis of System Architecture Using Rapide, IEEE Transactions on Software Engineering, vol.21, no.4, April 1995, p.336-355.

[14] Magee, J.; Kramer, J. Dynamic Structure in Software Architectures, Proceedings of SIGSOFT ’96, p.3-14.

[15] 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.

[16] Moriconi, M.; Qian, X. Riemenschneider, R.A. Correct Architecture Refinement, IEEE Transactions on Software Engineering, vol.21, no.4, April 1995, p.356-372.

[17] Paffendor, K.; Souvaine, D.L. Automating Software Documentation, Technical Report DCS-TR-354, Department of Computer Science, Rutgers University, April 1998

[18] 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.

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

[20] Shaw, M.; DeLine, R.; Klein, D.V.; Ross, T.L.; Young, D.M.; Zelesnik, G. Abstractions for Software Architecture and Tools to Support Them, IEEE Transactions on Software Engineering, vol.21, no.4, April 1995, p.314-335.

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

[22] Sim, S.E.; Clarke, C.; Hold, R.C; Cox, A.M. Browsing and Searching Software Architectures, Proceedings of 1999 Intl. Conference on Software Maintenance, p.381-390.

[23] 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.

[24] Tran, J.B.; Holt, R.C. Forward and Reverse Repair of Software Architecture, Proceedings of CASCON ’99.

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

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

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

[28] http://www.isr.uci.edu/projects/archstudio

[29] http://www.intrinsyc.com/products/j-integra