Proceedings of the

1st International Workshop on Incorporating COTS Software into Software Systems

(IWICSS 2004)

http://www.tuisr.utulsa.edu/iwicss/

Redondo Beach, California, USA

February 1, 2004

Editors: Alexander Egyed (Teknowledge Corporation) and
Dewayne E. Perry (University of Texas at Austin)

In conjunction with the
3rd International Conference on COTS-Based Software Systems
(ICCBSS 2004)

Table of Contents

Organizing and Program Committee

Organizers

Alexander Egyed
Teknowledge Corporation, USA

Dewayne E. Perry
University of Texas at Austin, USA

Program Committee

Francis Bordeleau (Carleton University, Canada)

Lisa Brownsword (Software Engineering Institute, USA)

Rose Gamble (University of Tulsa, USA)

Anna Liu (Microsoft Research, USA)

Nenad Medvidovic (University of Southern California, USA)

Maurizio Morisio (Politecnico di Torino, Italy)

Judith Stafford (Tufts University, USA)

Tarja Systa (Tampere University of Technology, Finland)

Ye Wu (George Mason University, USA)

Overview

This workshop explored innovative ways of integrating COTS software into software systems for purposes often unimagined by their original designers. It emphasized tools and techniques for plugging COTS into software systems safely and predictably. The past had predominantly explored how to deal with COTS integration during requirements engineering, risk assessment, and selection. This workshop focused on how to complement ordinary software development with techniques for designing, implementing, and testing COTS integration.

There is empirical evidence that COTS integration is not like ordinary software development. It has been shown that, for example, writing glue code is several times more difficult than writing ordinary application code. Thus the emphasis of this workshop was on software engineering principles for COTS integration. This included but was not limited to the following topics:

- how to write the glue code
- how to implement data and control dependencies
- how to mediate between incompatible interfaces
- how to make the COTS tool aware of its surroundings
- how to architect/design/simulate COTS integration
- how to do code generation
- how to resolve stumbling blocks and risks
- how to integrate user interfaces
- how to handle new COTS releases and other maintenance issues
- how to reverse engineer
- how to design product lines with COTS
- how to build domain-specific architectures with COTS
- how to test COTS-based systems

The workshop discussed:

- ⇨ theoretical approaches to COTS integration and
- ⇨ practical experiences (lessons learned) on COTS integration.

Tools for Commercial Component Assembly

Francis Bordeleau
Carleton University
francis@scs.carleton.ca

Mark Vigder
National Research Council of Canada
(mark.vigder@nrc-cnrc.gc.ca)

Abstract

An emerging model for building software systems is for system developers to licence a suite of components from commercial component developers and then to assemble these components into a working system. Although this model involves a minimal amount of development of source lines of code, it involves significant effort in specifying components, assemblies, configurations, and deployments. Standards for component Deployment and Configuration (D&C) have recently been developed for expressing this type of information about components and component-based systems. This paper discusses the implications of these standards on organizations that use commercial components to build systems. As well, it proposes a toolset that system developers can use to support development using commercial components.

1 Introduction

Component-Based Software Engineering (CBSE) models software applications as an assembly of software components.[5] Many definitions of the term "component" exist in the software engineering literature. However, a common theme that emerges from these different definitions is that a component is an independently deployable piece of software that interacts with other components in a well-defined manner to accomplish a specific task. A software component represents a modular part of a system that encapsulates its content and that can be independently deployed and configured. Since the beginning of the 90's, several component models have been developed. The most widely known are CORBA, J2EE, DCOM, and .NET. These models are now used in large-scale industrial applications.

Increasingly, organizations are using Commercial Off-The-Shelf (COTS) components to build applications, rather than building the components within their own organization.[4] Such an approach involves licencing components from other

organizations and then assembling the application from these components. When using such pre-built COTS components to construct applications, the software engineering process focuses on adapting, assembling, configuring, deploying, and managing large-scale commercial components from third parties rather than on developing components from source lines of code.

Many problems exist when trying to build applications from commercial components:

- Application developers must receive specifications from developers that not only provide the functional interface for the component, but also identify constraints, limitations, and assumptions made by the components on the system, as well as the properties of the component that are configurable.
- Incompatibilities between components from different vendors must be mediated.
- Compatible sets of components must be selected and assembled into working systems.
- Each component must be configured using the vendor supplied configuration parameters.
- The components of the assembly must be deployed within an appropriate environment.

One of the ways to address these problems is to have a standard for describing components and how they are assembled into systems. If both COTS component vendors and COTS component users work towards this standard, the assembly of COTS components into systems can be more automated. These standards are beginning to emerge as the Deployment and Configuration (D&C) standards.

This paper discusses the new Component Deployment and Configuration standards, shows how they can be used to assist the CBSE process when using commercial components, and proposes a set of tools for use by system developers to assist in the assembly.

2 Component Deployment and Configuration

Recently, different specifications have been developed to standardize the deployment and configuration of component-based software applications. One of the main objectives of these specifications is to enable the development of software applications that are independent of the target platforms[1] on which they can be deployed. This enables the development of applications that can be deployed on different target platforms. This is similar to what exists in the PC world where an application developed for a given operating system (OS) can be deployed on any computer using this OS, independently of the fact that different computers may be using different hardware devices. However, one important distinction with the PC world is that deployment and configuration

1 A deployment target is a node or a clustered group of nodes on which an application is to be deployed and configured.

specifications are OS-independent, and thus allow for deployment on heterogeneous target platforms.

The Object Management Group (OMG) has recently adopted a Deployment specification[1] that defines at a platform independent level the elements of an infrastructure that allows for the automated deployment and configuration of distributed component-based applications. This specification can be customized for different application domains, such as embedded systems and web-based systems, and platforms, such as the CORBA Component Model (CCM), J2EE, and .NET. This specification deals with the deployment and configuration of component-based applications onto distributed systems, anticipating that subcomponents might be distributed among a set of independent, interconnected nodes.

The Java community also recently adopted a specification that aims at standardizing deployment and configuration across J2EE servers. The Java 2 Platform, Enterprise Edition (J2EE) Deployment API Specification[2] defines standard APIs to enable a deployment of J2EE components and applications. It is intended to enable the development of platform-independent deployment tools.

These specifications define the main elements that must be provided by the deployment platform and the set of descriptors that must be provided by application developers to enable the deployment and configuration of their applications on a compatible deployment target. The descriptor files contain information concerning the set of components that compose the application, the configuration of the components, the location of component implementation files (containing the executable code), the interconnection between the components, and the deployment requirements of the components. Thus, in order to make a component-based application compatible with a given deployment and configuration specification, an application developer must be produced the set of required descriptor files.

3 Applying D&C to COTS Component Assembly

When an agreed upon Deployment and Configuration standard is employed by both COTS component developers and system developers a number of significant advantages emerge.

First, the D&C standard provides important information through which the component developers and the system developers can communicate. Component developers have a means of describing not only component interfaces, but as well the requirements and dependencies that must be satisfied to use the component. Additionally, component developers can describe the configuration parameters that can be set to customize the component within a particular application. By standardizing this information and having a machine-readable interchange format the transfer of this information from component vendor to system builder can be automated.

Second, system developers have a means of capturing important information about components as they test and develop. For example, if it is found that two components from different vendors are incompatible, this can be recorded as a constraint as part of the component specifications.

Third, it is possible to build tools that system developers can use to configure, assemble and deploy components into systems. By having all specification in standard formats third-party toolsets can be used for the system design and construction process.

Finally, it is possible to automate the verification of systems. Verifications include appropriate configuration parameters, compatibility between components, and satisfaction of all component constraints and assumptions.

4 Tools for COTS Component Assembly

The design of a new component-based application involves the following tasks:

- Design of the application as an assembly of components.
- Validation of the resulting component assembly, which includes the validation of aspects such as compatibility between connected ports and satisfaction of architectural constraints.
- Configuration of components using configuration parameters defined by the COTS component supplier.
- Validation of component configuration, which includes the verification that all required properties have been properly configured and verification that component properties do not violate the properties defined at the application level.
- Production of the complete set of descriptor files for the application (each component is associated with a set of descriptor files describing its different aspects).
- Validation of the content of the descriptor files against the DTDs.

Currently, the tools for the assembly, configuration, deployment and management of component-based applications are quite primitive. Most of the tasks are performed manually by technology experts who are expensive and rare resources. For example, the required descriptor files are usually manually written by deployment experts. This task is both very costly and error prone. The importance of this problem will increase quickly as the number of available components grows over time. Without proper tools these problems will become unmanageable.

There is a need for solutions that help reduce the risk associated with component-based application development. Appropriate tools can significantly reduce the cost and increase the reliability and quality of component-based applications. Moreover, they can increase the productivity of component-based application developers and reduce the level and range of expertise required to build component-based applications.

Among the key features that are needed from such tools are the support for the modeling of component assemblies, the automatic generation of descriptor files, and the validation of the application model.

The following features are essential to support the D&C of Component-Based Systems.

- Support for component-based application modeling
- Automatic generation of descriptor files
- Validation of applications (in terms of both component-assemblies and component configuration)
- Support of different component technologies
- Support of different D&C infrastructures that would allow porting applications to different deployment platforms

Two important software engineering technologies developed in the last few years in the Object Management Group (OMG) provide a basis for developing such tools: the Unified Modeling Language (UML), in particular UML 2.0[3]; and Model Driven Architecture (MDA).

UML is the de facto modeling language in the software industry. It is a general modeling language that aims at being tailored for specific application domains. For this purpose, UML provides a profiling facility that allows defining specialized UML-based languages. The latest version of UML, UML 2.0, which was adopted in June 2003, provides major improvements for component and deployment modeling, profiling, and MDA support.

MDA is the latest technology proposed by the OMG. The main objective of MDA is to enable the portability/reuse of models across different platforms. MDA focuses on the definition of Platform Independent Model (PIM), Platform Specific Model (PSM), and Model Mappings that allows moving from one model to another in a systematic manner. One goal of MDA is to define a set of Model Mappings between standard technologies that can be reused in different contexts. One of the main benefits of MDA is that models can be defined independently of specific implementation platforms and mapped to different platforms using predefined mappings.

We believe that UML 2.0 provides both the necessary core notation for modeling component-based systems, and an appropriate profiling mechanism that allows tailoring the language for specialized application domains, which in general requires specific types of components. The MDA technology enables the definition of mapping between UML 2.0 component models and different D&C infrastructures and deployment platforms. These mappings can be used to automate the generation of the descriptor files required for the deployment and configuration of distributed component-based applications. Finally, the combination of UML 2.0 and MDA technologies allows developing advanced automatic validation techniques.

UML and MDA also provide a means by which COTS component vendors and COTS component users can exchange valuable information regarding the components. By expressing component properties and constraints using the OMG standards, and exchanging the information using the interchange formats, system builders can use tools

assemble and configure commercial components supplied by different vendors, and verify that the assembly, configuration and deployment satisfies all the constraints imposed by the various components.

5 Conclusions and discussion

Building systems by means of configuring, assembling and deploying components involves methods, techniques, and tools different from that used to build systems by writing large amounts of source code. Components interfaces must be specified, along with the component constraints and limitations, the system assembly from components, and the deployment of components. When the components being used to build the system are acquired from different commercial vendors, the issues become even more difficult due to the large amount of information that must be transferred from component vendor to component user, and because of the potential for incompatibilities between components from different vendors.

A number of emerging standards provide an opportunity for building tools that assist system developers in assembling systems from a diverse set of components. In particular, the OMG standards for MDA, UML and Deployment and Configuration will give component vendors and system builders the following capabilities:

- A standard that allows component builders to specify components, including their interfaces, constraints, assumptions, and configuration parameters and an interchange format to allow component vendors to supply this information to system builders in machine readable form.
- A standard that allows system builders to specify component configurations, assemblies, and deployments.
- The means to develop tools that not only automate the creation of these specifications, but also allow for the automatic verification and validation of the specifications.

Widespread adoption of these standards, and the development of tools based on these standards, will therefore not only facilitate the use of component-based software engineering, but will also facilitate the development of a commercial component marketplace where system builders can standardize and automate the construction and validation of their component-based models.

References

[1] Object Management Group (OMG). *Deployment and Configuration of Component-Based Distributed Applications*. OMG specification ptc/2003-07-08.

[2] Java 2 Enterprise Edition Deployment API Specification, Version 1.0.

[3] Object Management Group (OMG). *Unified Modeling Language: Superstructureversion 2.0*. OMG specification ptc/03-07-06.

[4] Erdogmus, H. and Weng, T. (Eds.), *Proceedings of the Second International Conference on COTS-Based Software Systems, (ICCBSS 2003)*, Ottawa, Canada, Springer-Verlag, 2003.

[5] Heineman, G.T. and Councill W.T. (Eds.), *Component-Based Software Engineering*, Addison-Wesley, 2001.

Container-Managed Exception Handling Framework

Kevin Simons and Judith Stafford

Department of Computer Science
Tufts University
Medford, MA, USA 02155
{ksimons,jas}@cs.tufts.edu

1 Introduction

Commercial component developers are generally not aware of the components with which their software will interact when used in an assembly. Therefore, components are written to be as generic and reusable as possible. The components are not aware of each other, only of the interfaces that they provide and require. The Java™ 2 Enterprise Edition (J2EE™) framework allows for binary implementations of Enterprise JavaBean™ (EJB) components to be directly "wired" together in a deployment without any sort of "glue code". This can be accomplished via EJB metadata and Java reflection. This approach to deployment is the basis of Component-Based Software Engineering (CBSE), but it yields several problems with predicting the behavior of the system once it is assembled [4]. One such predictable assembly problem arises due to current exception-handling practices in component-based systems. Since commercial components are designed with no knowledge of the components with which they interact, they have no knowledge of the exceptional behavior of such components resulting in three possible exception-related situations: (1) Components making calls to other components will be catching generic exceptions with very little useful exception handling. (2) The result of a component operation may be considered exceptional by the system developer in the context of the current application, but the exceptional result is allowed by the calling component. (3) There may be exception results that could be easily handled by the developer without requiring exception propagation back to the calling component, which would most likely handle the exception poorly in the first place.

Component containers have emerged as an area with great promise in aiding the predictable assembly of commercial components. Containers are a receptacle into which components are deployed, providing a set of services that allow component execution [7]. With these services implemented in the container, the developer is allowed to concentrate on writing components in their domain of expertise. Containers, therefore, provide an excellent model for the separation of concerns. Furthermore, all calls made to components must be relayed through the containers, so containers provide an excellent means of crosscutting. System developers configure these services in the container at deploy-time, but the implementation of such services is left up to the container developers [2].

Augmenting the J2EE container with the ability to handle exceptions outside of the components alleviates the problem of improper exception handling in commercial components. Giving the system developer the ability to deal with the exceptions in an

application-specific context leads to more useful handling of exceptions and more predictable and robust system performance. Furthermore, abstracting the exception handling into the container helps alleviate the tangle that generally occurs in exception handling code; the code that controls normal behavior is separate from the code that handles exceptional behavior [3]. For this research, the EJB container used was the container provided as a part of the JBoss open source project[1]. This container is freely available, and is representative of the commercial J2EE containers.

2 Container-Managed Exception Handling

The container-managed exception handling (CMEH) framework facilitates the creation and deployment of modular exception handling components to promote proper separation of concerns in COTS-based systems. The CMEH framework allows system developers to quickly and easily deploy and manage exception handling components on a Java application server, allowing for more appropriate handling of component exceptions than is currently possible. It provides a modular, event-driven model for handling exceptional behavior. The CMEH framework is based on intercepting a component method call and dispatching exception events at a variety of points during the method invocation at these points in order to let event handling code correct the exceptional behavior of the system.

2.1 Handling of component method-exception event

When an exception is thrown by a method of a component, it is caught by the container-managed exception mechanism, giving system developer an opportunity to handle it. This can be done in a variety of ways. In the simplest case, the exception-handling code can simply re-throw the exception, and it will be propagated back to the calling component. This is the generic behavior of the EJB container before the introduction of this new exception mechanism. Another possible use of this mechanism is to stop the propagation of the exception altogether. Rather than propagating an exception back up the stack to the caller, the system developer may instead wish to return a value of the correct type to the caller. This will effectively allow the developer to return a default value to the calling component in the event of erroneous behavior. An exceptionally useful option when monitoring the exceptions thrown by a component is exception translation. The exception handling code catches the thrown exception, extracts any needed information from the exception, and then throws a different class of exception. This method allows for the exception to be translated into a subclass of exception that the calling component knows how to handle properly. Of course, knowing what exceptions a component can handle is not immediately obvious. There is somewhat of a lack of consensus in the way component interfaces should be specified [2], and the exceptions a component expects to receive from components that it requires is often absent from current interface

1 www.jboss.org

specifications. Unfortunately, a great deal of information about the way a component functions must still be discovered through use [4].

2.2 Handling of component method-called and method-returned events

The container-managed exception handling mechanism allows a system developer to check the arguments passed to a component method before the method is executed; providing the developer with several useful options. First, the developer can test the value of the arguments to ensure they are acceptable in the application and to the component being called. If they are, the method call continues as normal with the arguments being passed along to the called method. If the arguments are in any way out of range, the container exception handling code can raise an exception that will be propagated back to the calling component, effectively ending the method call. On the other hand, the system developer can modify the values of the arguments, then allow the method call to proceed as normal, thus eliminating any erroneous behavior in the component receiving the method call. Similar to the monitoring of arguments, this container model also provides the developer with the means to verify all return values returned by a component method. Once again, the return value can be modified by the container exception code in order to ensure correct functioning of the system, or an exception can be propagated back to the caller.

2.3 Handling of test-component-state and recover-component-state events

If a component throws an exception, there is a possibility that the component will be left in an invalid state. By handling the `test-component-state` and `recover-component-state` events, the system developer can logically handle this situation while keeping this exception handling code separate from the business logic of the components. After a component method throws an exception, the exception either 1) propagates back to the caller of the method or 2) is translated or caught by a handler of the method-exception event. In both cases, the container-managed exception framework fires a `test-component-state` event after the method invocation has concluded and before the control flow of the application progresses. This event gives the developer and opportunity to test the state of the component that threw the exception. If the exception event handling code determines that the component is in an invalid state, a `recover-component-state` event is fired. By handling this event, the system developer has an opportunity correct the state of the component before the application flow resumes. Handling of this event generally consists of unloading and reloading the component into the container.

3 Container Implementation

To handle the exception events in the CMEH framework, the system developer need only write Java classes that implement a simple set of interfaces and then deploy their libraries on the application server. To order to deploy their exception event handling

code, the system developer must modify the XML deployment descriptor of the EJB whose methods they want to monitor. Once the `ExceptionHandler` classes have been developed, some additions are needed to the ejb-jar.xml XML deployment descriptor. Each EJB has an XML deployment descriptor that tells the application server how the bean should be deployed into the container. The system developer must add a new tag into the `<assembly-descriptor>` portion of the deployment descriptor. It is perfectly valid to specify the same event handler class for several different component methods, and it is also valid to specify several handlers to handle the same event for the same component method, allowing exception handling code to be further modularized.

3.1 The Interceptor Stack

The CMEH framework is dependent on a feature in the JBoss container known as the interceptor stack. In the JBoss application server, services (such as transaction and security) are wrapped around a client's call via the interceptor stack. The task of an interceptor in the stack is to receive the invocation from the previous interceptor, perform any necessary processing, and then either pass the invocation on to the next interceptor, or raise an exception, effectively canceling the client's method call. The return value of the component method is then passed back up the interceptor stack, giving the interceptors the opportunity to perform operation on the invocation, pass the invocation further up the stack, or throw an exception back to the client. CMEH adds a new interceptor to the chain that is responsible for intercepting method invocations at the appropriate times and dispatching the exception events.

3.2 The JMS-based exception event model

The exception event model in the container-managed exception handling framework is based on the Java Messaging Service (JMS). This service, which is provided as part of the JBoss J2EE application server, provides a means of dispatching and listening for asynchronous messages. When the system developer deploys their exception event handlers, the framework automatically registers them to listen on the appropriate JMS topic. When an event is fired by the framework, a new JMS message is created and then dispatched to the correct topic. Event handlers, deployed to handle the type of event that is carried by the JMS message, receive the event and a new thread is automatically created by JMS for handling the event. Allowing the framework to support asynchronous handling of exceptional behavior. This will prove helpful if several handlers are deployed on the same event for the same component method and some of the exceptional handling behavior can be performed concurrently. Allowable synchronicity is also specified in the XML deployment descriptor.

3.3 The ExceptionHandlerService MBean

The exception handler service, responsible for the deployment of event handlers and dispatching JMS messages, is implemented as a Managed Bean or MBean in the

CMEH framework. Other MBeans in JBoss include services for transactions and security. When an EJB wishing to use CMEH (as specified in the deployment descriptor) is deployed into the component container, the `ExceptionHandlerService` MBean deploys the event and registers them with the appropriate JMS topic so that they can have exception events dispatched to them. If the system developer deploys a new version of the exception handlers when the system is up and running, the `ExceptionHandlerService`'s class loader dynamically replaces the exception event listener object so that the new version will receive the events. When the CMEH interceptor in the interceptor stack receives the invocation, it uses the Java Naming and Directory Interface (JNDI) to look up the `ExceptionHandlerService` and instructs the service to dispatch a JMS message containing the appropriate exception event. Implementing the service as an MBean allows applications running in other JVMs to look up the service via JNDI and register their exception event handlers with the services. This feature allows for exception handling code on entirely different machines to be registered with service in order to handle exceptional behavior in a distributed and parallel fashion.

3.4 Exception event automation

Some exception event handling patterns are useful enough that they have been automated in the CMEH framework. For instance, translation of exceptions can be done with a simple addition to the deployment descriptor specifying which classes to translate. By adding a few lines to the XML deployment descriptor, the system developer does not need to write any code, and the appropriate exception event handlers are created automatically. Other automated patterns include automatically testing the integer and string attributes of EJBs for the `test-component-state` event and automatic component reloading for the `recover-component-state` event.

4 Performance Costs

The event-based nature of CMEH, along with required method invocations pauses, indicates that there is a definite performance cost for using CMEH. Empirical results as to exactly how much slower applications will run has not yet been collected, but the increased ease of development, the added system predictability and the proper separation of concerns is expected to outweigh any costs.

5 Related Work

The use of containers as a means of crosscutting is directly related to work being done in the field of Aspect-oriented Programming (AOP). Much like the CMEH approach, AOP for exception handling stresses the detangling of exceptional code, as well as support for multiple configurations, incremental development and dynamic reconfiguration [3]. CMEH is also influenced by research in the field of multi-

dimensional separation of concerns [6]. The hyperslicing mechanism that is a part of the Hyper/J™ tool [5] is a large scale abstraction of the type of crosscutting mechanism used in this component container research. The Actor-based approach also influenced both the separation of concerns focus and the asynchronous model of this research [1]. To the best of our knowledge, the work most closely related to CMEH is Vecellio et al.'s [7] research in augmenting the JBoss container. Assertion capabilities (including watchdog timers and software firewalls) were added to the EJB containers in order to better predict the behavior of component-based systems.

6 Conclusions and Future Work

The CMEH framework provides a simple and highly automated system for allowing system developers to deploy modularized event handlers for dealing with exceptional behavior in COTS-based systems. This framework promotes a separation of concerns and more appropriate handling of exceptions and other exceptional behavior. While the framework was developed mainly for use with COTS components, it will also help to alleviate code-tangle in proprietary component-based systems. Our work is ongoing and new features are being added. Exception handling will be more easily parallelized by adding support for automatically deploying exception event handlers on remote Java application servers.

References

1. Agha, G. and W. Kim. "Actors: A Unifying Model for Parallel and Distributed Computing." Journal of Systems Architecture, 45(15):1263-1277, 1999.
2. Bass, L. et al. "Volume I: Market Assessment of Component-based Software Engineering." Technical Report CMU/SEI-2001-TN-007, Software Engineering Institute, May 2000.
3. Lopes, C. et al. "Using AspectJTM for Programming the Detection and Handling of Exceptions." Proceedings of the ECOOP Exception Handling in Object Oriented Systems Workshop, June 2000.
4. Stafford, J. and K. Wallnau. "Predicting Feature Interactions in Component-Based Systems." Proceedings of the ECOOP Workshop on Feature Interaction of Composed Systems, June 2001.
5. Tarr, P., and H. Ossher. "Hyper/JTM: Multi-Dimensional Separation of Concerns for JavaTM." Proceedings of the 22nd International Conference on Software Engineering, June 2000.
6. Tarr, P et al. "N Degrees of Separation: Multi-Dimensional Separation of Concerns." Proceedings of the 21st International Conference on Software Engineering, May 1999.
7. Vecellio, G., W. Thomas and R. Sanders. "Containers for Predictable Behavior of Component-based Software." Proceedings of the Fifth ICSE Workshop on Component-Based Software Engineering, May 2002.

Dynamism Gloom and Doom?

T. Gamble D. Flagg R. Baird R. Gamble[1]

Software Engineering and Architecture Team
Department of Mathematical and Computer Sciences
The University of Tulsa
600 South College Avenue
Tulsa, OK 74104 USA
{gamble}@utulsa.edu

Abstract. In the areas of application integration, mobility, and agent technology, dynamism is an increasingly desirable quality. However, deploying dynamism is problematic due to constraints on and limitations of the applications and middleware used in such distributed systems. In order to realize dynamism in these contexts, it must be well defined and it must work with available features in middleware. The application components, objects, and agents involved must themselves be equipped to perform dynamically. In this paper, we outline issues for incorporating dynamism into application integrations that rely on middleware.

1 Introduction

Dynamism is a key aspect of application integration design and development. When COTS products are integrated, commercially-supported middleware generally forms the backbone of the system. With respect to the dynamic manipulation of components, developers are limited to not only the features the middleware provides, but also to those features that translate into functionality that is understandable, usable, and reliable within the context of the integration. Conversely, when expecting COTS products to exhibit dynamic behavior, the component is likewise constrained by its communication protocols, service descriptions, management policies, and data types. These problems are not lesser issues by any means. In fact, these constraints may prohibit the component from interacting dynamically without augmentation.

Thus, the middleware must be adaptive (something that commercial vendors have not yet delivered), the component must be adaptive (difficult for COTS components), or the connector providing the interaction must provide the capability (a more likely scenario). For true dynamism, connector properties should not just be adaptable, but automatically adaptable. However, in this evolving area there is, as yet, no set of best practices.

For this paper, we focus on the pertinent features of middleware that COTS systems must accommodate. We examine three commercially-supported frameworks within the middleware "genres" of CORBA, Message Queuing, and the Grid. Specifically, we examine an implementation of each of these frameworks, all of which are widely used: Borland VisiBroker [14], IBM Websphere MQ [13], and the Globus Toolkit [17].

[1] This work is supported in part by AFOSR (F49620-98-1-0217) and NSF (CCR-9988320).

2 Where Dynamism is Found

Dynamism represents system change, mainly during run-time. The term "dynamic" can imply changes to the network topology, processor mobility, participating client processes, processor and network failures, and timing variations [1]. Key aspects of dynamism are how much change can occur, how much is known before the change, and both where and how the change is controlled. Researchers approach the accommodation of change by restricting it, planning ahead, and/or implementing predetermined change management solutions wherever possible. True dynamism, though, should allow for limited stored and predefined information by the component or the middleware.

When one or more of the components involved in an integration activity changes, it affects the architecture of the system, causing a dynamic reconfiguration. Many solutions include describing a configuration manager, which assures that no change occurs that is not already specified. Using a predetermined set of allowable state changes eases the task of assuring that the system remains in an architecturally permissible state [2, 3, 4]. These predefined constraints on the types of change provide for the dynamic creation and deletion of processes and clients [5].

Connectors have been designed to facilitate reconfiguration. By specializing components and providing a way for them to manage the changes to incoming and outgoing information through a connector, components and connectors can be swapped to accommodate those changes. The connectors are context-aware so they can provide the interaction functionality the components need as they change [6, 7].

For web services, dynamism often incorporates aspects of discovery, binding, composition, and configuration. User requirements may be leveraged to predetermine service inclusion. Abstractions can be inserted into a predefined configuration to be bound at runtime to a service in order to satisfy the constraints [8, 9]. Here, the components have some context-awareness to provide services dynamically [10]. It is recognized that some form of adaptive ability is needed within the component, its connector, or the middleware to support dynamic web services [11]. One goal is that service-oriented integration would permit the automatic composition of services given a client's description of the needed tasks and available components to provide the services [12].

3 Is Middleware Prepared to be Dynamic?

Modern middleware systems span many technologies and programming environments. Some, such as IBM's WebSphere MQ message queuing system, have been on the market for many years and have matured through a number of development iterations. Others, like VisiBroker, are a blend of technologies from middleware standards such as CORBA [15] and Sun's J2EE framework [16]. We now see a class of newer middleware for the Internet to support grid computing. The most prominent example is the Globus Toolkit. In its current incarnation (version 3 of the toolkit), Globus has merged a number of

characteristics from earlier Globus versions that focused on distributed high performance computing with the emerging standards for Web Services [18]. Having examined these middleware products, we outline certain fundamental features that impact dynamism (Table 1). These are registration, data typing, asynchrony, and security. It is not a comprehensive set of features, but it shows that COTS components in dynamic integrations have a responsibility for connection and further operation. This is especially true if the component participates in multiple middleware frameworks simultaneously.

Registration is the act of the component and the middleware becoming aware of each other. Registration is needed for interacting components to reliably identify one another, generally in the form of establishing a representation of a remote computer with a local context for later reference. While most middleware frameworks focus this activity on the component (whether a client or a service) establishing itself with the middleware, it can be a bilateral process whereby the component also retains information identifying the middleware. This is usually required prior to further interaction between the two entities. It may involve inclusion in a directory service, establishment of a programmatic reference, publication of an interface specification, or another mechanism that binds identifying information for later use. To facilitate dynamic behavior, registration must support binding and rebinding at runtime.

There are a number of implications for dynamism based on how, or if, registration is performed and its scope or depth. For example, IBM's WebSphere MQ[2] requires a large amount of information about the communication channels between servers, the queues residing on those channels, and the queue managers handling message delivery across queues. Applications reading or writing queues need only identify themselves in a more limited fashion. When using the MQ publish-subscribe interface, components subscribe to topics of interest and then receive messages only when other components publish on those topics. Components also specify how they should be alerted when a new event appears on the topic "channel" to which they subscribed. In this case, registration is a two step process involving subscription and event management. This behavior is actually similar to the other frameworks when using asynchronous interfaces. For example, in the current incarnation of Globus, grid services are registered by creating and transmitting a Web Services Description Language (WSDL) document specifying their service definition.

We define *data typing* as the collection of capabilities for type definition. It is required for enforcement of type safety across component interactions and is particularly important when interactions are only loosely defined, if at all, prior to runtime, as is the case with all dynamic systems. Facilitating this must be a function of the middleware or must occur at the integration endpoints, either in the component itself or in a connector to the middleware. Thus, the particular realization of these capabilities depends largely on the programming model that is assumed within the component-level interfaces to middleware. VisiBroker uses CORBA's interface definition language (IDL) to register types for interfaces. Delayed binding of typing is supported via the "ANY" type in IDL. The

[2] For brevity, we henceforth refer to this product as simply MQ.

Globus Toolkit is increasingly dependent on Java and J2EE run-time features as well as Web Services standards for interface specification. The use of WSDL as a definition mechanism is becoming the predominate method for interface definition. These type definitions must be submitted when services register in Globus. MQ has few data typing facilities and treats data payloads in messages as simple strings. There is limited matching done between publish-subscribe channels based on topic names.

Asynchrony is a middleware feature that is based on the context of interactions between components. This style of interaction must allow the components to communicate but operate independently, in a loosely coupled fashion. Loose coupling reduces dependencies between components and makes changes due to dynamic behavior easier to manage. This normally means a component sends a message to another component and expects a reply, but doesn't know when that reply will come and, more importantly, it does not wait for it. Most middleware frameworks that allow asynchronous behavior provide mechanisms for message transmissions to be reliable, requiring an implementation of persistence in the form of queues. If the model of message delivery is event-based, then the receiving components must be prepared to process the incoming events.

Within MQ, asynchronous behavior is inherent. There are no specific mechanisms for clients and servers to rendezvous. For VisiBroker, components may implement the CORBA Asynchronous Method Invocations (AMI) to support polling and callback of services by clients. This requires the *a priori* agreement of both client stubs and server skeletons. In Globus, asynchronous behavior is largely centered on mechanisms for batch processing, which reflect Globus' heritage in scientific data processing. Clients asynchronously submit batch *jobs* and may be alerted via *notifications* when batch jobs complete. In all cases, it is important to note that the components' interaction with the asynchronous features of middleware must provide appropriate behavior for that programming model. If they cannot, then some form of explicit mediation is required.

When looking at *security* in integration, there are many considerations. For the most part, these involve authentication, confidentiality, integrity, and non-repudiation. Dynamism results in change, which normally translates into added risks for secure systems. We focus on confidentiality and integrity, where the middleware provides them at either the transport level or the message level. For transport level security, the middleware need only assure that the overall connection and all data that travels over it be secured. In many cases, this means encrypting data along a communications channel between instances of the middleware on multiple operating systems.

The use of secured channels between middleware server processes is a common behavior across the middleware products in Table 1. MQ does transport encryption in the processes that implement channel endpoints. VisiBroker uses the broker implementations deployed to the operating systems where CORBA clients and/or servers reside. Globus has a "gatekeeper" process that is spawned in the local OS (operating system) to handle connection management for components. In each case, there is actually a physical process instance associated with one or more components running within the context of the same OS. This process represents a connector provided by the middleware that the component

Table 1. Middleware Features Impacting Dynamism

Feature	IBM MQ	Borland VisiBroker	Globus Grid Toolkit
Registration	Identification of queues and pub/sub topics. Establishment of callbacks or polling mechanisms for event processing.	Components must specifically state what is transferred and which services they offer or require, respectively. Registration requests are processed by local instances of broker servers.	A service component submits a WDSL document that describes its services and what is transferred, while clients submit a WSDL document in order to submit jobs.
Data types	There is no explicit checking of data types, as message payloads consist of raw bytes.	Registration includes checks to make sure the information to be transferred between components matches predefined data types.	Registration checks to make sure the information transferred between components matches predefined data types.
Asynchrony	Communication is inherently asynchronous.	Communication is inherently synchronous. Components may use supplied connectors to support polling and callback of services.	Clients can submit and be asynchronously notified by the services when processing has completed.
Security (transport level)	Communication channels between queue managers may be encrypted.	SSL is used between ORB instances on separate host environments.	A private protocol (httpg) is implemented between Globus containers (aka servers) on remote machines.
Security (message level)	No inherent capability.	Encryption is at the transport level, but message level integrity may be use checksums.	The message level security is based on WS-Security, XML Encryption and XML Signature standards using either a pre-established security or credentials supplied inside the calling parameters.

must use. The connector is context-aware in the sense that it supports the programming model(s) available on the OS. Communication between the component and the middleware through that connector is often not considered to be within the scope of transport concerns. Thus, it is not explicitly secured by middleware features and any dynamic connectors could cause a security breach.

Rather than protecting all interactions across a secure channel, the middleware may allow each component to component interaction to be selectively secured. For MQ, message level security isn't included. User programs may do their own encryption if they have consistent logic with peer programs. VisiBroker provides for limited message level security in the form of integrity checking via secure checksums. For encryption, only transport level is supported. Globus tends toward a potpourri of implementation choices,

and the security features it provides are no exception. Message level security in Globus is similarly influenced by Web Services standards and is based on WS-Security, XML Encryption and XML Signature standards.

4 Conclusions

Commercially-supported middleware still has many difficulties with dynamism. Features central to allowing basic interaction are often at odds with dynamism. Much *a priori* knowledge is needed to support interconnection. Also, middleware sets expectations, (e.g., code customization and interface interoperability), which are impractical for most dynamic COTS applications. More research is required to define automated, adaptable connectors and marry them with the architectural analysis to assure correct run-time behavior.

References

1. N. Lynch, A. Shvartsman, Communication and Data Sharing for Dynamic Distributed Systems, *Future Directions in Distributed Computing*, LNCS, Vol. 2584, 2003.
2. R. Allen, R. Douence, D. Garlan, Specifying Dynamism in Software Architectures, *Proceedings of Foundations of Component-Based Systems Workshop* (Sept. 97).
3. E. L. White, "General strategies for dynamic reconfiguration," *ACM SIGSOFT Software Engineering Notes*, Volume 25, Issue 1 (January 2000) pg. 93.
4. M. Wermelingerl, J. L. Fiadeiro, Algebraic Software Architecture Reconfiguration, *FSE'99*.
5. R. Allen, R. Douence, D. Garlan, Specifying Dynamism in Software Architectures, *Proceedings of Foundations of Component-Based Systems Workshop* (Sept. 97).
6. M. Mikic-Rakic, N. Medvidovic, Adaptable Architectural Middleware for Programming-in-the-Small-and-Many, *ACM/IFIP/USENIX Int'l Middleware Conference*, June 2003.
7. M. Mikic-Rakic N. Medvidovic, A Connector-Aware Middleware for Distributed Deployment and Mobility, *ICDCS Workshop on Mobile Computing Middleware* (MCM03). May 2003.
8. L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, Q. Z. Sheng, Quality Driven Web Services Composition, *WWW2003,* May 20–24, 2003.
9. C. Ferris, J. Farrell, What Are Web Services? *CACM*. June 2003/Vol. 46, No. 6.
10. A. Arsanjani, B. Hailpern, J. Martin, P. Tarr, Web Services: Promises and Compromises, *Queue*, March 2003.
11. G. Piccinelli, W. Emmerich, C. Zirpins, K. Schütt, Web Service Interfaces for Interorganisational Business Processes, (EDOC2002).
12. M. Turner, D. Budgen, P. Brereton, Turning Software into Service, *IEEE Computer*, 2003.
13. http://www.ibm.com/software/integration/mqfamily/library/manualsa/.
14. http://www.borland.com/besVisiBroker/.
15. http://www.omg.org/technology/documents/formal/corba_2.htm.
16. http://java.sun.com/j2ee/docs.html.
17. http://www-unix.globus.org/toolkit/documentation.html.

Reengineering Large-Scale Polylingual Systems

Mark Grechanik[1], Dewayne E. Perry[2], and Don Batory[1],

[1] Department of Computer Sciences,
UT Center for Advanced Research In Software Engineering (UT ARISE)
University of Texas at Austin
{gmark,batory}@cs.utexas.edu

[2] Department of Electrical and Computer Engineering,
UT Center for Advanced Research In Software Engineering (UT ARISE)
University of Texas at Austin
perry@ece.utexas.edu

Abstract. Building systems from existing applications written in two or more languages is common practice. Such systems are polylingual. Polylingual systems are relatively easy to build when the number of APIs needed to achieve language interoperability is small. However, when the number of distinct APIs become large, maintaining and evolving them becomes a notoriously difficult task. We present a practical and effective process to reengineer large-scale polylingual systems. We offer a programming model that is based on the uniform abstraction of polylingual systems as graphs and the use of path expressions for traversing and manipulating data. This model enables us to achieve multiple benefits, including coding simplicity and uniformity (where neither was present before) that facilitate further reverse engineering. By performing control and data flow analyses of polylingual systems we infer the schemas used by all participating programs and the actions performed by each program on others. Finally, we describe a tool called *FORTRESS* that automates our reverse engineering process. The contribution of this paper is a process that allows programmers to reverse engineer foreign type systems and their instances semiautomatically at the highest level of design. We know of no other approach with comparable benefits.

1 Introduction

Building software systems from existing applications is a well-accepted practice. Applications are often written in different languages and provide data in different formats. An example is a C++ application that parses an HTML-based web page, extracts data, and passes the data to an EJB program. We can view these applications in different ways. One way is COTS integration problem where a significant amount of code is required to effect that integration. Or we can view them as instances of architectural mismatch, specifically as mismatched assumptions about data models [1]. Or we can view them, as we do in this paper, as instances of polylingual interoperable [2,3] applications that manipulate data in foreign type systems (FTSs) i.e., type systems that are different from the host language.

Consider an architecture for polylingual systems as shown in the directed graph in Figure 1. Graph nodes correspond to programs P_1, P_2, …, P_n that are written in different languages and may run on different platforms. Each edge $P_i \rightarrow P_j$ denotes the ability of program P_i to access objects of program P_j. $P_i \rightarrow P_j$ is usually implemented by a complex API that is specific to language of the calling program P_i, the platform P_i runs on, and the language and platform P_j to which it connects. (In fact, there can be several different tools and APIs that allow P_i to access objects in P_j). Note that the APIs that allow P_i to access objects in P_j may be different than the APIs that allow P_j to access objects in P_i.

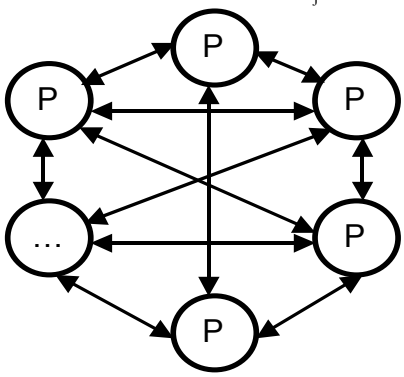


Figure 1. Architecture of Polylingual Systems.

The complexity of a polylingual program is approximately the number of edges in Figure 1 that it uses. That is, when the number of edges (i.e., APIs needed for interoperability) is miniscule, the complexity of a polylingual system is manageable; it can be understood by a programmer. But as the number of edges increases, the ability of any single individual to understand all these different APIs and the system itself rapidly diminishes. In the case of clique of n nodes (Figure 1), the complexity of a polylingual system is $O(n^2)$. This is not scalable. Of course, it is hard to find actual systems that have clique architectures. In fact, people want them, but these systems are too complex to build, maintain, and evolve. A large-scale polylingual system is a polylingual system where the number of edges (APIs) is excessive. Such systems are common and are notoriously difficult to develop, maintain, and evolve.

We propose a combined forward and reverse engineering process consisting of four steps to reverse engineer foreign type systems and their instances semiautomatically at the highest level of design. First, we offer a programming model that is based on abstracting constituents of polylingual systems as graphs of objects and providing language-neutral specifications based on path expressions, coupled with a set of basic operations, for traversing these graph to access and manipulate their objects [4]. This step allows us to achieve multiple benefits, including coding simplicity and uniformity that facilitate further reverse engineering. Next, we perform control flow and data flow analyses of polylingual programs in order to infer schemas (e.g. a schema is a set of artifact definitions in a type system that defines the hierarchy of elements, operations, and allowable content) and actions performed by FTSs. Then we transform the schema and actions into plain English description. Finally, we implement tool called *FORTRESS (FOReign Type*

Reverse Engineering Semantic System) that partially automates our reverse engineering process.

The main contribution of this paper is a combined forward and reverse engineering process based on a programming model that enables programmers to recover high-level design from complex polylingual systems with a high degree of automation.

2 The Reengineering Process

We propose a reengineering process that enables programmers to recover a high-level design of polylingual systems with a high degree of automation. The first and most important step on this process is the use of our programming model that normalizes FTS-based code into structured sequences of operations on type graph objects. These operations are provided by reification operator objects that are defined in Reification Object-Oriented Framework (ROOF) [4]. Program statements that contain reification operator objects are called reification statements. Our model reduces the complexity of polylingual programs by improving code simplicity and uniformity.

We achieve these results as naturally obtained benefits of normalized FTS-based programs. Our programming model significantly reduces the number of syntactical constructs that otherwise must be present in polylingual code, and the normalized programs have much simpler semantics.

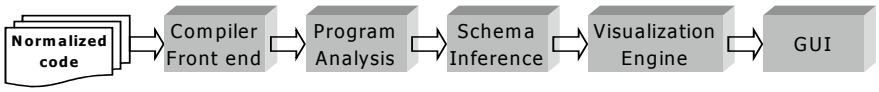


Figure 2. The illustration of steps in our reengineering process.

The reverse engineering process is illustrated in Figure 2. After the code is normalized, the next step in the reverse engineering process is to use control and data flow analyses on the simplified polylingual code. Control flow analysis (CFA) relates the static program text to its possible execution sequences [5]. Data flow analysis (DFA), on the other hand, computes relationships between data objects in programs [6]. We apply the CFA to build graphs of possible execution sequences of polylingual programs, and then we run the DFA on each execution graph to analyze FTS reification statements. The results of program analysis is used to infer schemas that describe the FTS models and operations executed against them by polylingual programs. The schemas and operations fed into high-level design description driver as shown in Figure 6 that produces plain English description of and visualizes structural and behavioral aspects of polylingual systems.

Finally, we describe the FORTRESS tool that enables programmers to reverse engineer polylingual systems using our process, browse high-level descriptions of FTS-based systems, visualize their impacts on schemas and their instances, and map elements of these descriptions and schema definitions to the source code.

3 ROOF Programming Model

ROOF is designed in light of principles of interoperable polylingual systems described in [2,3,4]. The goals of the ROOF programming model is to enable easily maintainable and evolvable polylingual interoperability by removing the need for elaborate name management solutions and allowing programmers to make decisions about sharing objects at the megaprogramming stage. The maintainability and evolvability of polylingual systems are achieved by using foreign objects by their names as they are defined in FTSs thereby eliminating the need for creation of isomorphic types in a host programming language and enabling programmers to share objects at the megaprogramming stage. We also provide a comprehensive mechanism for type checking that allows programmers to verify semantic validity of operations on foreign types both statically and dynamically with certain limitations.

ROOF is based on three assumptions. First, we deal with recursive type systems. Even though it is possible to extend our solution to higher-order polymorphic types, such as dependent types, we limit the scope of this paper to recursive types and imperative languages to make our solution clearer. Second, we rely on reflection mechanisms to obtain access to FTSs. Third, the performance penalty incurred by using reflection is minimal since the low-level interoperating mechanisms such as transmission, marshaling and unmarshaling network data has the largest overhead common to all interoperable solutions.

Suppose we have a handle to an object that is an instance of a foreign type. We declare this handle as an instance `R` of a `ReificationOperator` class. `R` enables navigation to an object in the referenced type graph by calling its method `GetObject` with a path expression as a sequence of type or object names t_1, t_2, ..., t_k as parameters to this method:

```
R.GetObject(t1)...GetObject(tk)
```

`R` implements a reification operator (RO) that provides access to objects in a graph of foreign objects. We give all ROs the same interface (i.e., the same set of methods) so that its design is language independent; reification operators possess general functionality that can operate on type graphs of any FTS. By implementing `R` as an object-oriented framework that is extended to support different computing platforms, we allow programmers to write polylingual programs using a uniform language notation without having to bother about peculiarities of each platform. That is, for Java we have separate extensions of the framework that allows Java programs to manipulate C# objects, another extension to manipulate XML documents, etc. Similarly, for C# we have separate extensions of an equivalent framework that allows C# programs to manipulate Java objects, another extension to manipulate XML documents, etc. *Foreign Object REification Language (FOREL)* is a user interface provided by ROOF to enable programmers to write interoperable polylingual programs.

4 Program Analysis

Assuming that polylingual programs are normalized to conform to the ROOF programming model, we can perform program analysis as the next step in our reverse engineering process. Since programs that constitute polylingual systems communicate by changing each other's data and structures, program analysis allows us to recover these changes from normalized polylingual code. Thus, program analysis is an integral part of our reverse engineering process whose goal is to produce a high-level description of these changes.

We analyze programs in two steps. First, we run control flow analysis (CFA) to build execution graphs, and then we perform data flow analysis (DFA) on these graphs to compute relationships between data objects in polylingual programs [5,6].

DFA is the most significant part of a program analysis. We are interested in finding all definitions and uses of reification operator objects. There are three types of statement that can use RO objects:

- Navigation statements in which RO objects point to a certain object in the FTS type graph. For example, statement `R["CEO"]["CTO"]` denotes a collection of objects of type `CTO` contained in type `CEO`.
- Assignment statements in which values of type objects are set or retrieved. For example, the following statement `R["CEO"]["CTO"]<<5.0` sets the value of an object of type `CTO` to `5.0`.
- Structural statements that invoke operations that modify the structure of other polylingual programs. Given two RO objects `R` and `Q` the structural statement has a form of `R` ⊗ `Q` where ⊗ is a structural operation, for example, append a new object from RO `Q` to a branch in a type graph represented by RO `R`.

DFA enables us to answer the following five questions.

- What is the structure of FTSs operated upon by the analyzed program?
- How do FTSs affect the control flow of the analyzed program?
- What is the relationship between RO objects and program variables?
- What are the operations performed by the analyzed program on foreign type objects?
- What are the operations performed by the analyzed program on the structure of FTSs?

Answering these questions allows us to recover a high-level design from polylingual code. By determining the structure of FTSs we present a unified schema of the system. This operation is sound but not complete since we recover only a part of the schema that is operated upon by polylingual code. However, in the majority of cases the complete schema does not exist for a variety of reasons (e.g., it has never been created or it is rendered obsolete), and a part of a schema that describes type definitions operated on by polylingual code is a good enough approximation.

Knowing the structure of FTSs can help to determine how they affect the control flow of the analyzed program. Consider the following segment of FOREL code that contains a reification statement.

```
if( R[“CEO“].Count() == 1 ){...} else{  ...}
```

Suppose that we retrieve a schema of the FTS designated by `R`. If we establish that the number of `CEO` objects is always equal to one then we do not have to analyze the `ELSE` branch of the `IF` statement since the boolean condition is always true.

Unfortunately, we cannot limit the DFA to the analysis of RO objects. Type names and values of type objects can be assigned to program variables. Tracking uses and definitions of these variables may lead to establishing more complete schemas describing FTSs and improving the analysis of polylingual programs subsequently raising the quality of the recovered high-level design.

The answers to the last two questions can be derived from the type of a reification statement. Navigation statements give us path expressions on type graphs. Assignment statements coupled with the knowledge of relationships between RO objects and program variables tell us what values we assign to or retrieve from destination type objects and what path expressions navigate to them.

Recall that structural reification statements modify the structure of FTSs, and such operations are reduced to modifications of abstract type graphs. Since the ROOF provides a unified set of operations on type graph objects we can detect them using elementary DFA algorithms.

5 Schema Inference

Schema inference is the process of deriving structure information from the query that generated data. This process is more complex than schema extraction that finds the most specific schema for a particular data instance [7]. Some languages make it easy to infer schemas just by looking at a statement that access or modifies data. Consider the following SQL statement.

```
SELECT u.Name, c.Course FROM User u, Courses c WHERE u.ID
= c.ID;
```

Just by looking at this statement we can infer the following information. There are

- two tables: `User` and `Courses`;
- attributes `Name` and `ID` in `User` table;

- attributes `Course` and `ID` in `Course` table;

- declaration of attribute `ID` in both tables is the same or compatible.

A diagram for the inferred schema is shown in Figure 3. It allows programmers to view the design of a database at a high level.

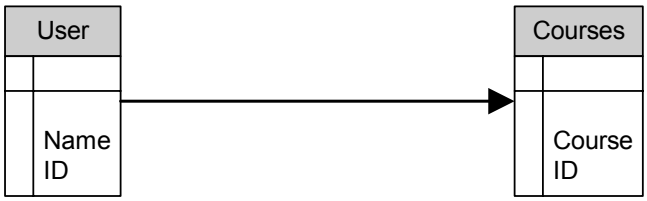


Figure 3. A schema diagram inferred from an SQL statement.

In general, polylingual code is suited very poorly for the schema inference process. Consider a fragment of C++ code shown below.

```
HRESULT hr = CoInitialize( NULL );
if( FAILED(hr) ) return( NULL );
CComPtr<IXMLDOMDocument> spDomDoc;
hr = spDomDoc.CoCreateInstance( __uuidof(DOMDocument40) );
if( FAILED(hr) ) return( NULL );
CComPtr<IXMLDOMNode> node;
node = spDomDoc;
BSTR b = AsciiToBSTR( "CEO" );
hr = node->selectNodes( b, &childList );
SysFreeString( b );
if( FAILED(hr) ) return( NULL );
CComPtr<IXMLDOMNode> nodeCEO;
hr = childList->get_item((long)0,&nodeCEO );
if( FAILED(hr) ) return( NULL );
```

We retrieve a handle to `CEO` node located in some XML data using the MS XML parser, COM and the Active Template Library. These tools and libraries are among the best available and are widely used today. The complexity of this fragment of code is clearly evident. Note that if we use a different XML parser (e.g., for improved performance), we must rewrite this code because their low-level APIs would be different. It is clear that automatic schema inference is very difficult and impractical for this and similar code. There should exist detailed information about the semantics of low-level APIs and their compositional semantics should also be defined. These conditions increase the complexity of program analysis algorithms significantly.

As a result of our normalization process, the semantics of polylingual programs is much simpler than the ones of original programs. It is noteworthy that reification statements do

not lead to increased complexity of program analysis algorithms that model this semantics. All high level design can be extracted directly from reification statements by analyzing their type and performing the DFA.

An additional benefit of inferring schemas from polylingual programs is the ability to compute relationships between FTSs that are manipulated by some polylingual program.

This concept is illustrated in Figure 4. Polylingual program `P` manipulates FTS_I and FTS_J using RO objects R_I and R_J. The interactions between program `P` and FTSs are shown with solid arrows. Suppose that R_I reads a value of some type object from FTS_I and stores it in a program variable. Then, R_J assigns a value of this variable to some type object in FTS_J. By performing the DFA on program `P` we can compute the relationship between FTS_I and FTS_J as it is shown in Figure 4 with a dashed arrow.

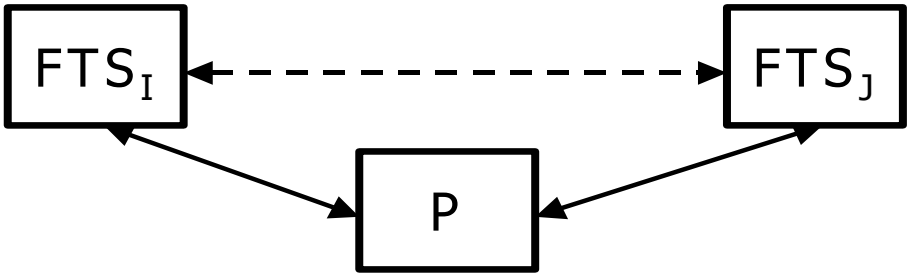


Figure 4. A computed relationship between FTSs.

Consider a fragment of FOREL code shown in Figure 5. This small fragment of code allows us to infer key definitions of some organizational schema. When analyzing the boolean condition of the first `IF` statement we can conclude that type `CEO` contains type `CTO` that has attribute `Salary` of primitive type `float`. When analyzing the boolean conditions of the second and third `IF` statements we infer that type `CTO` contains types `Geeks` and `Test`. We can also infer various schema constraints, for example, if a CTO makes more than $100,000 annual salary then there should be Geeks and Test departments in the company reporting directly to him.

```
if( R[“CEO“][“CTO“](“Salary“) > 100000 )
{

   if( R[“CEO“][“CTO“][“Geeks“].Count() < 1 )
   {.....}

   if( R[“CEO“][“CTO“][“Test“].Count() < 1 )
   {.....}

}
```

Figure 5. A fragment of FOREL code containing reification statements.

6 FORTRESS

We are currently in the process of creating a tool that reverse engineers normalized polylingual applications. This tool is called FORTRESS (FOReign Type Reverse Engineering Semantic System). Figure 6 shows the architecture of FORTRESS. We envision FORTRESS as a framework for building a set of tools that help programmers to reengineer and reverse engineer polylingual systems and enable their effective evolution and maintenance.

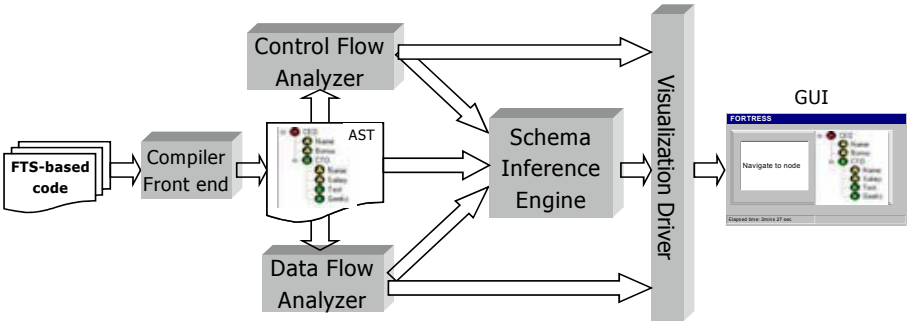


Figure 6. FORTRESS architecture.

The architecture reflects the steps of our reverse engineering process. Once a polylingual program is loaded into the tool we use a parser for the language in which this program is written to create its abstract syntax tree (AST). Currently we plan to support polylingual programs written in C++ and Java. We use EDG front end compilers [8] to parse source code and produce ASTs. Then we perform CFA and DFA on the AST to infer schemas of FTSs that this program accesses and manipulates. Finally, we transform operations on FTSs into plain English and present the listing of these operations in the FORTRESS GUI.

7 Related Work

We know of no existing technologies that fully address the problem of reverse engineering polylingual systems. A variety of reverse engineering tools has been reviewed in [9,10,11,12]. While they are effective in the problems that they intended to solve, none of the existing tools fully addresses the problem of reverse engineering software that consists of applications based on distinct and different type systems. Some tools are developed to reverse engineer multilanguage systems. For example, EER/GRAL approach to graph-based conceptual modeling is used to build models representing relevant aspects of single language [13]. These models are later integrated in a common conceptual model. The other paper [14] introduces GRASP - a software engineering tool designed to provide visualization of multilanguage software control structure, complexity, and architecture. Both approaches for reverse engineering multilanguage systems fall short of producing effective high-level designs by reverse engineering polylingual code.

Program comprehension techniques play important role in the normalization of source code for subsequent reverse engineering. Indeed, if a program is easy to understand by a human then it is likely to be effectively and automatically reverse engineered. Fundamental mechanisms of program comprehension are studied in [15,16].

8 Conclusions

We have sketched a simple and effective way to reverse engineer polylingual systems. We accomplish this by introducing a semiautomatic process to reverse engineer polylingual systems and implement a tool called FORTRESS to automate this process. We offer a programming model that is based on the uniform abstraction of FTSs as graphs and the use of path expressions for traversing and manipulating data. This step allows us to achieve multiple benefits, including coding simplicity and uniformity, and to facilitate further reverse engineering. Next, we perform control flow and data flow analyses of polylingual programs in order to infer schemas describing all participating FTSs and actions performed by each FTS on others.

The contribution of this paper is a process that allows programmers to reverse engineer foreign type systems and their instances at the highest level of design automatically. We are working on a refactoring tool that would process legacy software and output the normalized polylingual code. If we are successful, we expect to make this functionality a part of FORTRESS.

We believe in practicality of our approach. The capability to write uniform and compact programs that work with applications based on different type systems enables better program comprehension and effective and automatic reverse engineering. As a result of our solution, developers concentrate on reasoning about recovered schemas that describe properties of applications without the need to understand low-level APIs and apply complex knowledge rules to recover high-level design from source code. Since the semantics of the reification languages is simple and easily parseable, it may enable various techniques and algorithms to improve reverse engineering process of ROOF-based source code. We know of no other approaches that achieve similar benefits or potential.

References

1. D. Garlan, R. Allen, and J.Ockerbloom, "Architectural mismatch: Why reuse is so hard," IEEE Software, vo. 12, no. 6, November 1995, pp. 17-26.
2. D. Barrett, A. Kaplan, and J.Wileden, “Automated support for seamless interoperability in polylingual software systems,” *Fourth Symposium on the Foundations of Software Engineering*, October 1996.
3. A. Kaplan and J.Wileden, “Software interoperability: principles and practice,” *ICSE* 1999.
4. M.Grechanik, D.Batory and D.Perry, "Design of Large-Scale Polylingual Systems," *ICSE* 2004.
5. S.Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
6. L. Moonen, “A Generic Architecture for Data Flow Analysis to Support Reverse Engineering,” *Second International Workshop on the Theory and Practice of Algebraic Specifications*, November 1997.
7. S. Abiteboul, P.Buneman, D.Suciu, *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kauffman Publishers, 2000.
8. Edison Design Group, http://www.edg.com.
9. R.Kollman, P.Selonen, E.Stroulia, T.Systä, and A.Zündorf, “A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering,“ *IEEE Ninth Working Conference on Reverse Engineering*, October-November 2002.
10. A. van Deursen and L.Moonen, “Exploring Legacy Systems Using Types,“ *IEEE Seventh Working Conference on Reverse Engineering*, November 2000.
11. T.Systä, “Understanding the Behavior of Java Programs,“ *IEEE Seventh Working Conference on Reverse Engineering*, November 2000.
12. B.Bellay and H.Gall, “A Comparison of Four Reverse Engineering Tools,“ *IEEE Fourth Working Conference on Reverse Engineering*, October 1997.
13. B.Kullbach, A.Winter, P.Dahm and J.Ebert, “Program Comprehension in Multi-Language Systems,“ *IEEE Fifth Working Conference on Reverse Engineering*, October 1998.
14. T.Hendrix, J.Cross II, L.Barowski and K.Mathias, “Tool Support for Reverse Engineering Multi-Lingual Software,“ *IEEE Fourth Working Conference on Reverse Engineering*, October 1997.
15. Y.Deng and S.Kothari, “Using Conceptual Roles of Data for Enhanced Program Comprehension,“ *IEEE Ninth Working Conference on Reverse Engineering*, October-November 2002.
16. R.Clayton, S.Rugaber and L.Wills, “On the Knowledge Required to Understand a Program,“ *IEEE Fifth Working Conference on Reverse Engineering*, October 1998.

State Consistency Strategies for COTS Integration

Sven Johann
University of Applied Sciences Mannheim
Windeckstrasse 110
68163 Mannheim, Germany
sven_johann@gmx.net

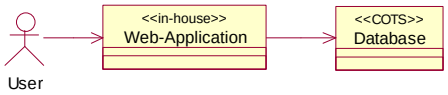
Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA
aegyed@ieee.org

Abstract. The cooperation between commercial-off-the-shelf (COTS) software and in-house software within larger software systems is becoming increasingly desirable. Because COTS software is typically integrated into standalone applications, they do not provide subscription mechanisms that inform other components about internal changes. This, in combination with semantic differences in how COTS data is perceived within the integrated system, causes consistency problems. This paper will first present a scalable consistency approach for the COTS software IBM Rational Rose. The paper will then discuss this approach in context of a set of strategies for COTS integration that is heavily based on incremental transformation.

Introduction

Incorporating COTS software into software systems is a desirable albeit difficult challenge. It is desirable because COTS software typically represents large, reliable software that is inexpensive to buy. It is challenging because software integrators have to live with an almost complete lack of control over the COTS software product.

To date, COTS software integration is not uncommon. It has become common practice to build software systems on top of COTS software. For example, a very common integration case is in building web-based technologies on well-understood and accepted web server COTS software. Indeed, COTS integration is so well accepted in this domain that virtually no web designer would consider building a web server or a database anew.



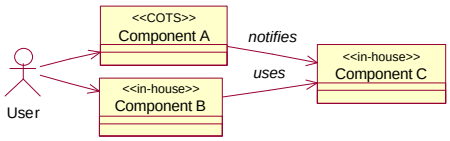
While there are many success stories that point to the seaming ease of COTS integration there are also many failures. We believe that many failures are the result of not understanding the role COTS software is supposed to play in the software system. In other words, the main reason of failure is architectural.

Many software systems, such as the web application above, use the COTS software as a back-end; its use is primarily that of a service providing component. Only some form of programmatic interface (API) is required for COTS software to support this

use. COTS software has become increasingly good at providing programmatic interfaces to data and services they provide.

COTS integration is less trivial if the COTS software becomes (part of) the front end; e.g., with its native user interface exposed and available to the user. These cases are rather complex because the COTS software may undergo user-induced changes (through its native user interface) that are not readily observable through the programmatic interface. In other words, the challenge of COTS integration is in maintaining the state (e.g. data model, configuration, etc) of the COTS software consistent with the overall state of the system *even while users manipulate the system through the COTS native user interface*.

This paper discusses this issue from the perspective of using COTS design tools such as IBM Rational Rose, Mathwork's Matlab, or Microsoft PowerPoint. These design tools are well accepted COTS tools; they exhibit commonly understood graphical user interfaces. This paper will show how to use these COTS tools, with their accepted user interfaces, to build a functioning software system where the architectural integration problem is more like this:



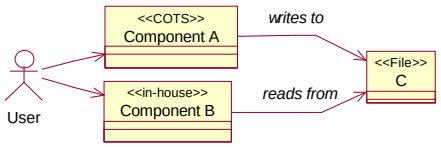
The COTS integration challenge is to make software systems aware of when and how incorporated COTS software undergoes changes. This problem does not exist with in-house developed components because they can be re-programmed to notify other, affected parts of a system (e.g., other components). The lack of access to the source code of COTS software makes this approach impossible.

Because COTS software is not originally designed to be components of a new system, they offer at most interfaces to extend their functionality with plug-ins. Their purpose is to be *the* system and not being only a part of it.

If we want to incorporate COTS software into a system, we have to ensure that the overall system state (data, control) is maintained consistently with that of the COTS software. Therefore, we have to "observe" the COTS software and then forward all events to the system. The system decides if the changes are relevant and updates itself if necessary.

There are two ways in doing that: 1) batch notification and 2) incremental notification.

Batch notification is to externalize all relevant COTS state information (e.g., by exporting design data from Rational Rose) so that it becomes available to other components. This is probably the easiest solution but has one major drawback in that it is a loose form of incorporating COTS software into software systems. State information has to be replicated with the drawback that even minor state changes to COTS software may pose major synchronization problems (e.g., complete re-export of all design data from Rational Rose).



Batch notification is computationally expensive and thus only then feasible if the integration between the COTS software and the rest of the system is loose. The following will present our approach to incremental notification. Our approach wraps COTS software to identify changes as they occur with several beneficial side effects:

- Only changes are forwarded
- Changes are forwarded instantly
- Detecting changes and forwarding them does not require manual overhead

Scoping Change Detection

There is a trivial albeit computationally infeasible approach to identifying changes in COTS software by caching its state and continuously comparing its current state with the cached state. This approach obviously does not work well if the COTS state consists of large amounts of data (e.g., design data in tools such as IBM Rational Rose, Mathwork's Matlab, or Microsoft PowerPoint). To identify changes in COTS software smarter it has to be understood how and when changes happen.

Take, for example, IBM Rational Rose. In Rose a user may create a new class by clicking on a toolbar button ("new class") followed by clicking on some free space in the adjacent class diagram. A class icon appears on the diagram and the icon is initially marked as selected. By then clicking on the newly created, selected class icon once more, the name of the class can be changed from its default and class features such as methods and attributes may be added. These changes may also be done by double-clicking on the class icon to open a specification window. There are two patterns worth observing at this point:

- Changes happen in response to mouse and keyboard events only
- Changes happen to selected elements only

The first observation is critical in telling *when* changes happen. It is not necessary to perform (potentially computationally expensive) change detection while no user activity is observed. The second observation is critical in telling *where* changes happen (Egyed and Balzer 2001). It is not necessary to perform (potentially computationally expensive) change detection on the entire COTS design data (state) but only on the limited data that is selected at any given time.

Both observations are the key for scalable and reliable change detection in GUI-driven COTS software.

Change Detection in Time and Space

Changes are detected by comparing a previous state of a system with its current state. Generally, this implies a comparison of the previously cached state with the current one. Knowing the time when changes happen and the location where changes happen limits what and when to compare. Our approach therefore uses the programmatic interface of the COTS software to elicit and cache state information. For example, in IBM Rational Rose this may include all its design data such as classes, relationships, etc. The caching is limited to "relevant" state information that is of interest to other components of the system. For example, if it is desired to integrate some class diagram analysis tool with Rose, then only change information for class diagrams are needed. Thus, it is not necessary to cache and compare other diagrams such as sequence or use-case diagrams.

Basic Change Detection

After every mouse/keyboard event, we ask Rose via its programmatic interface what elements have been selected to compare these elements with the ones we cached previously. If we find a difference (e.g. a changed name, a new method) between the cached elements and the selected ones then we notify other components (e.g., in-house developed components) about this difference. Thus, our approach notifies other components on the behalf of the COTS software. If we find a difference, we also update the cache because we want to find and report a difference once only. Obviously the effort of finding changes is computationally cheap because a user tends to work with few design elements at any given time only.

This approach detects changes between the cached and current state. But there are two special cases: 1) new elements cannot be compared because they have never been cached and 2) deleted elements cannot be compared because they do not exist in the COTS software any more.

The creation and deletion problem can be addressed as follows. If we cannot find a cached element for a selected one then this implies that it was newly created (otherwise we would have cached it earlier). Thus, we notify other components of the newly created element and create a cached element for future comparisons. In reverse, if an element in the cache does not exist in the COTS software then it was deleted. Other components are thus notified of this deletion and the cached element is deleted as well. Note that a deletion can only be detected after de-selection (i.e., a deleted element is a previously selected element that was deleted) and creation can only be detected after selection.

Ripple Effect of Change Detection

Until now, we claimed that changes happen to selected elements only. This is not correct always. Certain changes to selected elements may trigger changes to "adja-

cent", semantically-related elements. For example, if a class X has a relationship to class Y then the deletion of class X also causes the deletion of the relationship between X and Y and it also causes a change to class Y (i.e., it now does not have a relationship to X any more) although the latter are never selected.

There are two ways to handle the ripple effect. The easiest way is to redefine selection to include all elements that might be affected by a change. For example, if a class is selected we could define that also all its relationships are selected. Then change detection will compare the class and its relationships. This approach works well if the ripple effect does not affect many adjacent elements (e.g., as in this example) but it could be computational expensive.

The harder but more efficient way of handling the ripple effect is to implement how changes in selected elements affect other, non-selected elements. For example, we can implement the knowledge that the deletion of a class requires its relationships to be deleted also. In this case, neither the creation of a class nor its change does have the same ripple effect.

Anomalies

We found that basic change detection and the ripple effect cover most change detection scenarios. However, there may be exceptions that cannot be handled in a disciplined manner. We found only few scenarios in Rose that had to be handled differently.

For example, state machines in Rose have a peculiar bug in that it is possible to drag-and-drop them into different classes while the programmatic interface to Rose does not realize this. If, in the current version of Rose, a state machine is moved from class A to class B then, strangely, both classes A and B believe they own the state machine although only one of them can. We thus had to tweak our approach to also consider the qualified name of a state machine (a hierarchical identifier) to identify the correct response from Rose. Obviously, this solution is very specific to this anomaly. Fortunately, not many such anomalies exist.

Consistency between Different Domains

It is generally easier to maintain consistency between COTS software and the system it is being integrated with if the semantics of the COTS data is similar to the semantics of the system data. For instance, the above example integrated Rational Rose design information with UML-compatible design information and both are conceptually similar. Consistency becomes more complicated if the data of COTS software is re-interpreted into a semantically different domain. This is not uncommon. For example, many applications exist that use Rose as a drawing tool. In those cases, the meaning of boxes and arrows may differ widely.

This section discusses how to "relax" change detection depending on the difficulty of the integration problem. This problem is motivated by our need to have a domain-specific component model, called the ESCM (Embedded Systems Component Model) (Schulte 2002), integrated with Rational Rose. While it is out of the scope to discuss

the ESCM, it must be noted that its elements do not readily map one-to-one to Rose elements. As such, there are cases where the creation of an element in Rose may cause deletions in ESCM and there are cases where overlapping structures in Rose may relate to individual ESCM elements. This integration scenario is more problematic because it is very elaborate to define how changes in Rose affect the ESCM.

Previously, we solved the integration problem by comparing Rose data with cached data. Batch transformation was used to create a, initial, cached copy of the Rose data (transformation also re-interpreted the Rose data into UML data). User actions, such as mouse and keyboard events, triggered partial re-transformations to compare the current Rose state with the cached copy. The comparison itself was trivial; so was update. The key was transformation.

The main difficulty of integrating the ESCM is in determining what to re-transform and what to compare. This is a scoping problem and it becomes more severe the more complex the relationship between system data (e.g., ESCM) and COTS data becomes. A simplification is to implement change detecting with the possibility of reporting false positives (Rose change does not affect the ESCM) but the guarantee of not omitting true positives (Rose change affects the ESCM). In case of integrating ESCM with Rose, it was not problematic to err on the side of reporting changes that actually did not happen since it only lead to some unnecessary but harmless synchronization tasks. The ability to relax the quality of change detection to also allow false positives strongly improved computational complexity.

There are two simple strategies in doing change detection with false positives: 1) we delete all ESCM elements which could be affected through a change in the COTS tool and simply re-transform all deleted elements or 2) we compare all possibly affected elements with the cached data and re-transform only the changed ones. The first strategy is the cheapest but produces more false positives than the second strategy. The more detailed discussion of these strategies is out of the scope here.

Conclusion

Consistency between commercial-off-the-shelf software (COTS), their wrappers, and other components is a pre-condition for a working COTS-based system. Our experience is that it is possible to get notification messages about changes from GUI-driven COTS software even if the COTS software vendor did not provide a (complete) programmatic interface for doing so. This paper briefly discussed several strategies for adding change detection mechanisms to COTS software.

1. Egyed, Alexander and Balzer, Robert. Unfriendly COTS Integration - Instrumentation and Interfaces for Improved Plugability. Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)San Diego, USA; 2001 Nov.

2. Schulte, Mark . MoBIES Application Component Library Interface for the Model-Based Integration of Embedded Software Weapon System Open Experimental Platform [Technical Report, Boeing]. 2002 Jun.

Black-box Testing for Evolving COTS-Based Software

Ye Wu

Information and Software Engineering Department
George Mason University
Fairfax, Virginia, USA, 22030-4444
wuye@ise.gmu.edu

Abstract. Black-box testing methods are widely used in system level and integration testing. But in the context of COTS-based engineering, when newly modified COTS components are adopted into the system and the source code is not available, black-box testing is a necessary and feasible approach to assure that these COTS products do not adversely affect the software. In this paper, we first discuss challenges that we are facing when adopting evolving COTS products. Then we describe different black-box testing techniques and how to adjust them to effectively test evolving COTS-based software systems.

1 Introduction

One of the most important objectives of adopting COTS products is to improve the maintainability of COTS-based software systems. Ideally, when COTS products are updated, minimal effort is required to re-evaluate COTS-based software systems. However, to achieve both test effort reduction and quality preservation is extremely difficult, due to many unique features of COTS-based engineering.

COTS products can evolve in many different ways. When faults are discovered in a delivered product, corrective maintenance activities need to be performed. When new features are added in the product or enhancements are made to the previous product, perfective maintenance activities need to be performed. Other activities such as preventive and adaptive activities may also be periodically performed. Maintaining COTS-based software is significantly different from maintaining traditional software. The major difference is that, when maintaining traditional software systems, quite often the same company or even the same group of people who develop the product will maintain the system. They have full control of the maintenance process and complete knowledge of the system. On the other hand, for COTS-based software, COTS products are usually maintained by the COTS provider, while COTS product users have neither control of the maintenance process, nor complete knowledge of the product or the future course of the product enhancements.

Because of these unique features of COTS products, black-box testing is intuitively the most applicable testing strategy for COTS-based software systems. To adequately apply black-box based testing techniques on COTS-based software, many

new challenges that have never been investigated before need to be explored. Therefore, in Section 2, we first analyze the new challenges we are facing when dealing with evolving COTS products. Because of these new challenges, direct adoption of black-box testing is possible, but may induce too much overhead and may also overlook potential faults in the new integrated systems. To adequately apply black-box testing strategies, adjustments need to be made. In Section 3 of this paper, we focus on the issues of adopting black-box testing, and use adjusted partition and boundary value testing to provide an effective way to test evolving COTS-based software systems. Some preliminary results that are described in Section 4 have shown the great potential of this strategy.

2 Challenges facing COTS-based systems

A significant difference between software maintenance for traditional systems and for component-based systems lies in the fact that, for traditional systems, software implementation and maintenance activities will be carried out by the same company. The company has the control to evaluate the impact of the changes and has full access to the source code, so that they can minimize the impact of the change. COTS providers do not have the real context for how their products are used by COTS users, and therefore COTS providers cannot effectively evaluate the impact of change. Intuitively, the best way to restrict the impact to a minimal scope is to maintain the same set of interfaces to the services that will be provided. However, practices have demonstrated that this cannot always be achieved, due to issues such as architectural and technology changes. In this section, we will use some real enterprise COTS products to demonstrate the challenges we could face in maintaining evolving COTS-based systems.

Changes in COTS products can happen in different ways and in different contexts; most of them can be classified as the following:

1. COTS-user transparent changes

   Most of the changes in corrective maintenance activities, and a small part of the changes in perfective and adaptive maintenance activities, belong to this category. Corrective maintenance often corrects component internal logic errors or errors due to inadequate interface-interface interaction. These errors are not caused by wrongly defined interfaces. Therefore, very often, the fix for such errors often leaves the interfaces of the COTS product unchanged. Sometimes perfective maintenance may also end up with the similar scenario. For instance, XML parsers are very popular COTS products that have been built into many different enterprise systems, such as WebLogic, Tomcat, the Sun ONE application server, etc. XML parsers have gone through many different versions to improve performance issues for each and every version; the same interfaces are kept to ensure stability for all of its users.

2. Changes that affect the COTS product user, but do not affect user system architecture

Unfortunately, not all changes in COTS products can be classified as user-transparent changes that require minimal efforts for COTS users. There are many causes for these kinds of changes, such as poor design in the previous version, changes in the technology, changes in the standard, or sometimes a strategic move to lock in customers. For instance, the WebLogic Enterprise Service (WLSE) components that have been integrated with WebLogic went through dramatic changes from WebLogic 6.1 to 7.0, the main reason being that version 6.1 does not conform with the mainstream security architecture. Some of the changes in this component require minor changes in COTS users' code. For instance, the authentication service underwent a significant structure change in the WLSE component, but client code only need to change the APIs that will be called. But changes in some other services require extensive maintenance activities as described below. In general, new COTS products will provide new interfaces to substitute the deprecated interfaces or to provide additional services. Sometime, original interfaces are still provided just for backward compatibility.

3. Changes that affect COTS user architecture

If an interface's name change is the only change for the COTS products, programmers only need to change the interface name of their code; then their system is ready to be integrated with the new COTS products. But very often changes in interface names also mean architecture changes. In other words, the way that the new component is to be integrated, and how different interfaces communicate with each other, could be changed as well. If this is the case, major revision for the client program is required. For instance, the architecture of the assess control module service within the WLSE component is completely changed. It changed from access control list (ACL), which is resource oriented, to security policy, which is user oriented. To adopt this revision, the WLSE client component may have to completely rewrite all access control-related code.

3 Black-box testing for evolving COTS-based software

3.1 Issues in black-box testing for evolving COTS-based software

While testing evolving COTS-based software, black-box testing techniques can be directly adopted as the techniques that only rely on specifications. Nevertheless, COTS products have usually been tested, and very often a full scale retesting of the integration is usually not acceptable to COTS users. To adequately apply traditional black-box testing techniques for software components, the following issues need to be taken into consideration:

1) *COTS information unavailability*: COTS products usually do not provide source code to their customers, and detailed changes to their products are very often un-

available as well. Without these types of information, COTS users will encounter great difficulties when attempting to determine the test adequacy for black-box testing when they upgrade COTS products

2) *Component customization*: When adopting COTS products into a software system, what the system requires and what the COTS product provides usually do not perfectly match with each other. Therefore, when adopting a component into the application, customization is required to reconcile the differences between the two specifications. Consequently, customization is the focal point of the testing
3) *Component interface*: Unlike traditional software systems, interactions can be conducted in a flexible way. For instance, we can access an object through its public methods, or sometimes we can access public data members directly. As for component-based software, interfaces are the only point of contact. In other words, the interactions among components have to go through interfaces. Therefore, the specifications of interfaces play a key role in testing software components. If properly conducted, efficient and effective component testing can be achieved through interface-based black-box testing strategies.

3.2 Partition testing and boundary value testing

Black-box testing has been widely used for a long time, and different approaches can be used for different situations. Basically, the techniques can be classified as: 1) usage-based black-box testing techniques, such as random testing or statistical testing, 2) error-based black-box testing techniques, which mainly focus on certain error-prone instances, according to users' experiences with respect to how the program behavior usually deviates from the specification. This type of approach includes equivalence partitioning testing, category-partition testing, boundary-value analysis, decision table based testing, etc., and 3) fault-based black-box testing techniques which focus on detecting faults in the software. In this section, we use partition testing and boundary value testing as examples to demonstrate how to test evolving COTS-based software.

Equivalence partition testing tries to divide the input domains into k different disjoint partitions p1, p2, …,pk, where $p1 \cup p2 \cup \ldots \cup pk$ = input domain, and $pi \cap pj = \Phi$ for any i and j where $i \neq j$. Values from each partition have the "all-or-none" property. I.e. if an element is selected as the input for the component and that element fails the component, all other elements in that partition will also fail the component. On the other hand, if the component succeeds with that element as input, all other elements in the same partition will succeed as well. Therefore, if partitions which satisfy this characteristic can be generated, then test cases can be easily generated by randomly selecting one element from each partition. In addition, the test cases generated by the equivalence partition testing strategies can provide us with confidence about the entire domain without any redundancy. Unfortunately, there is no systematic way to generate equivalence partitions, and many times it is impossible to develop equivalence partitions. To overcome these difficulties, many systematic partitioning approaches are developed. If formal specifications exist, sometimes partitions can be

automatically derived. Given the non-formal functional specification, Ostrand and Balcer proposed a systematic partition testing strategy called category partition testing [6]. Category partition testing starts with an original functional specification and continues through the individual details of each subprogram to derive a formal test specification.

For partition testing, input domain will be classified into different disjointed partitions. Ideally, every element in each partition has the same possibility to either reveal or hide a fault. But based on programming experiences, this is may not usually be true. Values that are close to the boundary of the partition are more likely to expose errors. Thus, when boundary value testing each partition, not only one element will be selected. Instead, additional elements close to the boundary will be tested as well [4]. Boundary value analysis addresses test *effectiveness* – focusing on the high-risk areas. Partition testing addresses test *efficiency* – reducing the number of test cases needed to obtain a certain level of confidence for the software under test. Therefore, boundary value testing strategies should be used along with partition testing strategies.

3.3 Partition testing and Boundary value testing for evolving COTS-based application

As discussed in Section 2, COTS products can evolve in many different ways. Each scenario requires a different strategy to assure the quality of the integrated system:

- User transparent changes. These types of changes often leave the original partition unchanged. According the software quality requirements and other constraints, different adequacy criteria can be adopted. If the COTS user knows where the fault is, and which interfaces have been changed for each modified interface, at least one test case needs to be selected to rerun. However, the modified interfaces can be invoked in different locations, under different contexts. So a stricter criterion can require a test from each partition, which involves an execution of a changed interface in order to rerun. If a changed interface is unknown to the COTS user, similar criteria can be adopted by assuming that all interfaces are potential candidates for changed activities
- Changes that involve new interfaces or even new architecture. These types of changes can potentially change the original partition. This can occur in the following different situations:
    1) *Generalization repartition*: The effects of generalization repartition can cause one partition to expand while the other shrinks. An extreme scenario is to merge the two partitions. To test this scenario, each changed partition needs to be retested. In addition, test cases need to be generated to validate those areas for which ownerships have been changed
    2) *Specialization repartition*: The effects of specialization repartition could lead to the generation of additional partitions. The testing of specialization requires all affected partitions to be retested, and new test cases be generated for newly derived partitions

3) *Reconstruction repartition*: Reconstruction repartition is potentially more complex than the first two types of repartition. If this is the case, substantial retesting is required
4) *Boundary value and repartition*: When partitions are changed, correspondingly boundary values for each partition will be changed. To test the changed partition, in general, at least one test is required to be selected to re-run. To be more careful, more test cases should be selected to ensure that the values around partition boundaries work properly.

In addition to the approaches that are discussed in 3.1 through 3.3, there are many other black-box testing methodologies, for instance, decision table testing, mutation testing [2][3], syntax testing [1], finite-state testing [1], cause-effect graphing [5], error guessing [5], etc. These black-box testing techniques can be adjusted in similar fashion.

4 Pilot Study

To demonstrate how black-box testing strategy can be applied and how effective these black-box testing strategies are, we conduct a pilot study on coffee-machine software, which includes a coffee-machine server component and a coffee-machine client. The server component contains 12 classes and exposes seven interfaces, and the coffee-machine client will invoke all seven interfaces. Two versions of the coffee-machine server component are developed. The second version introduces 10 modifications to the first version. Seven of the modifications are corrective maintenance activities, while three of the modifications are perfective maintenance activities. To test the first version, we adopted Ostrand and Balcer's category partition-testing [6] strategy and obtained 48 partitions. After the modifications, 17 partitions were changed or added. To validate the effectiveness of the testing strategy, we created total 34 faulty versions of the program, each with one inserted fault. After randomly selecting 17 test cases from each of the partitions, 21 faults were identified, while 13 faults remain undetected.

By analyzing the testing result, we discovered that five out of the 13 faults, which were not revealed, are not detectable through black-box testing. They can, however, be detected through unit-level white-box testing [7]. The other eight faults are either caused by the boundary values which are not covered in our test, or by inaccurate partition.

The pilot study has demonstrated that, with minimal testing effort, 76.5% ((21+5)/34) of the fault can be detected, but for safety-critical software systems, more rigorous retesting is required.

5 Conclusion

When COTS components are modified, effectively and efficiently re-evaluated COTS-based software is necessary. Due to many unique features of COTS-based software, black-box testing is one of the most feasible strategies. In this paper, we have demonstrated the potential problems of applying traditional black-box testing strategies to COTS-based systems, and provided a solution for conducting the testing through partition testing.

References

[1] Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold Inc., New York, 2nd Edition, 1990.

[2] Edwards, S.H., et al, "A Framework for Detecting Interface Violations in Component-Based Software", *In Proceeding of 5th International Conference on Software Reuse*, IEEE CS Press: Los Alamitos, CA, 1998, pp. 46-55.

[3] Ghosh, S. and Mathur A.P., "Interface Mutation", *Journal of Software Testing, Verification and Reliability*, Vol. 11, No. 4, December 2001, pp. 227-247.

[4] Hoffman, D., Strooper P. and White, L., "Boundary Values and Automated Component Testing", *Journal of Software Testing, Verification and Reliability*, Vol. 9, No 1, 1999, pp. 3-26.

[5] Myers, G.J., *The Art of Software Testing*, John Wiley and Sons Inc., New York, 1979.

[6] Ostrand, T.J. and Balcer, M.J. "The Category-partition method for specifying and generating functional tests", *Communication of the ACM*, Vol. 31, No. 6, 1988, pp. 676-686.

[7] Zhu, H., Hall, P., and May, J., "Software Unit Testing Coverage and Adequacy", *ACM Computing Surveys*, Vol. 29, No. 4, December 1997, pp.366-427