

TOWARD IDENTIFYING THE IMPACT OF COTS EVOLUTION ON INTEGRATED SYSTEMS*

L. Davis J. Payton R. Gamble
Dept. of Mathematical & Computer Sciences
University of Tulsa
600 S. College Ave., Tulsa, OK 74104
 [{davisl, payton, gamble} @euler.mcs.utulsa.edu](mailto:{davisl, payton, gamble}@euler.mcs.utulsa.edu)
(918) 631-2988 voice (918) 631-3077 fax

1. INTRODUCTION

Evolution is inevitable when dealing with current software systems. Often it stems from user requests, developer needs, advancements in technology, and component upgrades. COTS products are especially susceptible to evolution, including radical changes, due to the need to attain and keep a broad customer base. Sometimes this evolution is embraced as customers see added functionality or performance in the upgraded product. Other times, the product has changed so drastically that customers may be wary of adopting it because of unknown bugs, missing functionality, or lack of backward compatibility.

Problems created by component evolution are magnified when a COTS product is part of a system built from independent, heterogeneous components. Often an integration solution or middleware is used to bridge the interaction among components. When one component is modified or upgraded in a manner that alters its interaction properties, it can affect the way the middleware performs. Indeed, this is a weighty issue for developers because these changes can require that an expensive integration effort be redone. The reason is that evolution can generate additional component integration issues, while, at the same time, making others obsolete.

How the impact of a component upgrade on interoperability is handled determines the complexity, and ultimately the cost, of changes to the existing middleware. Simple reinsertion of the component may, in fact, work if the evolution is not drastic. However, trial and error is not the best approach, because it will not provide any clues as to what integration solution changes to make when they *are* needed. In contrast, we advocate a priori analysis to predict the severity of new integration problems resulting from component evolution, allowing the developer to assess tradeoffs to replacing the component with its more recent self. This suggests that in some cases, perhaps the best cause of action is no action at all.

To assess the gravity of evolution the developer must first identify those component properties that reflect evolutionary changes. These properties must be such that they can be expressible for COTS products due to the extent to which the products are self-contained. Our approach uses software architecture characteristics of the component as a basis for these properties. When assembling the integrated system initially, these component characteristics provide a meaningful aid to predict interoperability conflict that determines the resulting middleware. Consequently, we hypothesize that the impact on interoperability (both in terms of new and obsolete conflicts) can likewise be predicted, should the values of

* This research is sponsored in part by AFOSR (F49620-98-1-0217).

these characteristics change. In this same vein, processes developed to predict interoperability conflicts in an integration could be modified to handle component upgrades as well [11].

In this paper, we define the architecture characteristics that lead to assessing the severity of the impact on an integrated system when a component evolves. We substantiate these claims by pointing to the importance of these same characteristics in understanding initial interoperability problems. Additionally, we discuss relevant characteristic changes and their effect on an existing middleware solution – whether the fit is maintained. We conclude the paper with further research directions needed for a comprehensive assessment.

2. BACKGROUND

The *software architecture* of a system is a high-level description of its computational elements, the means by which they interact, and the structural constraints on that interaction [12, 14]. The phrase *architecture mismatch* has been used to describe the underlying reasons for interoperability problems among seemingly “open” software components with available source code [6]. Often, mismatch problems can be traced to fundamental characteristics of the architectural styles of the interacting component systems [1, 16]. There is an extensive set of architectural characteristics [15], many of which can play a part in predicting interoperability conflicts without knowing the architectural style [7, 4]. Indeed, characteristics can be traced from integration solutions through to the interoperability problems that resulted in those solutions. Currently, attempts are being fostered to reverse this mapping, making it possible to move directly from conflict to solution [4, 17]. As a result, use of this analysis can be delineated as part of a process to formulate appropriate integration solutions [11].

Principles of software architecture are being used in various ways to study evolving components. One goal is to find an architecture to which other architectures can transition

easily. This same architecture should have properties such that when system requirements change, they do not impact the entire system. An orthogonal architecture is a specialized style that satisfies many of these properties - it is organized into layers and multiple independent threads [13]. Requirements changes are localized within this architecture such that when evolution occurs only certain threads are affected. This research establishes that architecture properties are very important in assessing adaptability to change. Though architecture migration is a plausible solution, it is not always feasible for all systems, especially COTS products where much of the migration properties are hidden.

In a similar vein, researchers are examining constraints on reconfiguring architectures to assess how they are impacted by evolution. Not all architectural styles have been found to be easily reconfigured [10]. In fact, different styles are more flexible than others with respect to adaptability depending on what properties change and when. For instance, object-oriented architectures, such as C2, which use high-level dynamic languages to describe them, are flexible concerning evolution. This is because, again, certain properties of components (such as unawareness of other components) and connectors (such as asynchronous message exchange) lessen the impact of change [9].

Another research thrust is to examine architecture definition languages (ADLs) with respect to describing and supporting an evolvable architecture [8]. As a result, C2SADEL was created to support architecture-based evolution. This ADL is based on architecture subtyping and type checking. Specifically, it addresses the evolution of components and their topology.

In each of these research areas, an architecture and its properties were evaluated for its potency in solving problems produced by changing components. For each, the hypothesis was that an architecture-based approach to the problem

of evolution would insure a viable solution the greatest chance of being applicable across a broad set of systems. We can conclude from these approaches that examining architecture adds understanding to system evolution beyond a "quick fix" approach. It is further apparent that the architectural characteristics of a component system provide insight into how it responds to evolution.

3. THE EVOLUTIONARY CHARACTERISTICS OF ARCHITECTURE

Characteristics from three recognized aspects of software description (control, data, system) comprise a representative set regarded for evolution. This set, *evolutionary characteristics of architecture* (ECOAs), is described by the characteristic values corresponding to properties readily available without benefit of source code. Particularly with COTS, this information tends to be illuminating in an abstract way. For example, if you know the style in which the product was developed, or the language used to implement it, some assumptions about its topology, both data and control, can be made [2]. Indeed, the nature of the software sometimes provides useful information about its interface, such as operating systems which need to fulfill consumer profiles of personal user, network user, server, etc. Guidelines for

determining values for characteristics such as control structure, supported data transfer, and data storage method often coincide with need for parallel computing or a decentralized system. Maintaining a high level of characteristic scrutiny aims to express available information regarding the component systems and their interaction. However, due to the nature of COTS components it is not always possible to delineate values for every characteristic in the set. It is simply representative of what will be most easily described, and, as it turns out, are what is needed to make an initial assessment.

ECOAs were established by first examining characteristics that have an impact on component interoperability [7]. It was a natural extension to consider these characteristics given previous research about their involvement in constructing integration solutions [5]. As an added benefit, the characteristics influencing interoperability were further stratified through the use of semantic relationships [4]. ECOAs also crown the semantic networks; any changes made at the lower levels of architectural description will propagate up to ECOAs, insuring that the brunt of predictive evolution analysis can fall there. These characteristics are defined in Table 1.

SYSTEM CHARACTERISTICS	
<i>Identity of Components</i>	Knowledge or awareness of other components in the system [16].
<i>Naming</i>	If the components in the system are uniquely named [3].
<i>Blocking</i>	Whether or not the thread of control is suspended [7].
DATA CHARACTERISTICS	
<i>Supported Data Transfer</i>	The method supported by a particular architectural style to achieve data transfer [1].
<i>Data Topology</i>	The geometric form the data flow takes in a system [15].
<i>Data Storage Method</i>	The details about how data is stored within a system [16].
CONTROL CHARACTERISTICS	
<i>Control Structure</i>	The structure that governs the execution in the system [16].
<i>Control Topology</i>	The geometric form the control flow takes in a system [15].

To appear in *ICSE-2000 Workshop Proceedings of Continuing Collaborations for Successful COTS Development*, May 2000.

Table 1: Evolutionary Characteristics of the Architecture

Suppose the value for a component's method of communication (a more implementation-based characteristic) is altered, e.g., modified from call and return to event broadcast. Both the flow and scope of data and control in the component are affected, subsequently causing the data and control topologies to change values. The underlying reasoning is that most likely you will see a broadcast value related to an arbitrary topology [15]. (Figure 1 depicts these transitive control relationships in a vertical slice of the semantic net). As a result of the evolved nature of the component, the integration solution originally implemented may be impacted, thus requiring assessment and upgrade to accommodate the change, potentially causing a ripple effect throughout the entire system.

Within Table 1, each of the characteristics represents some aspect of a component that, when modified impacts the current integration solution. Below we present examples of this impact.

- *Identity of components.* The integration solution will be affected by a component migration to an object-oriented language should the original system employ explicit call and return policies. No longer will each component have the addresses of the components with which it interacts. However, it is possible that each component system relies on these addresses for communication. Therefore, an intermediary will be needed on top of the existing integration solution to provide the appropriate direction.
- *Naming.* Consider a travel advisory system which communicates directly with a named component that provides information on air, auto, and hotel services. Suppose that this component is upgraded to multiple individual service components that no longer retain the original name. For the original integration solution to remain functional, a broker is needed to coordinate communication between the travel advisory system and the new service components.

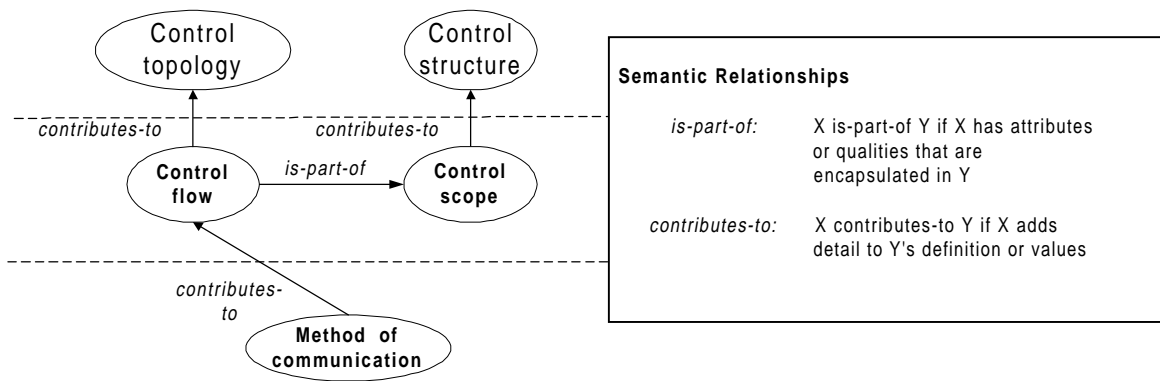


Figure 1: A Vertical Slice of the Semantic Control Relationships between Characteristics

- *Blocking*. Should a COTS product that blocks be present in a system, it requires that there be a mechanism in place that will arrest it from its wait state. If, in turn, the component receives an upgrade changing its blocking status to non-blocking, excess functionality is likely present which the middleware in place need no longer address.
- *Supported data transfer*. Should the supported data transfer of a component reflect a main subroutine style, and then change to accommodate an event system style, this modification can require an additional wrapper to simulate the previous form of data transfer for the rest of the system in order for the integration solution to perform correctly.
- *Data storage method*. An integration solution mandates direct passing of data among components. When a component is modified to use a shared repository, this repository may need to be available to other components. The integration solution then requires an additional mediator to coordinate access to the repository.
- *Control structure*. An integrated system developed for a single-user on one computer, which employs a single-thread of control, is to be evolved to a distributed system making decentralization imperative. In order for this to occur some mechanism must be added which will allow the components to execute independently and communicate asynchronously.
- *Control and data topologies*. Given that a COTS product allows access to its database to a restricted set of components in the integrated system. Components that are restricted may obtain access via an unrestricted third party built into the integration solution. Eventually the COTS vendor upgrades the product to allow direct access to more components, consequently changing both the control and data topologies if better performance can be achieved. This may cause a deletion of the

third party or a different solution for those components still without access.

Requirements and technical advancements will further define ECOA. Such things as migrating an application to a higher-level language, upgrading to a multi-processor unit, becoming part of an inter-office network, implementing parallel computing, etc. are all evolutionary influences that impact the systems involved. Consideration as to which of the characteristics would be affected by these changes narrow the set for analysis. The resulting set depicts what are common changes to COTS products that may impact the integrated system.

4. ECOA ASSESSMENT TOOLS

Examination of the impact of COTS evolution on an integrated system suggests some form of conflict prediction tool would be helpful. The identification and substantiation of the ECOA is one step towards that end. Nonetheless, elaboration on these characteristics is needed to prepare them for participation in a tool; such as discovering a possible hierarchy among the characteristics of ECOA. A hierarchy of this type might establish a weight associated with each characteristic and/or value as it relates to the severity and/or frequency of impact on integration when change occurs. A predictive analysis tool of this sort would have dual benefits for COTS product upgrades – the non-descriptive nature of the products makes them harder to analyze and there are often competing products that can be more easily inserted.

5. CONCLUSION

Ultimately, it is necessary to know how and why an existing integration solution is impacted by evolution in order to design it better to withstand changes. This requires an understanding of patterns of change and their resultant effect, thereby offering some insight into flexible middleware designs. A first step toward this goal is identifying the properties of distinct components in an integrated system

To appear in *ICSE-2000 Workshop Proceedings of Continuing Collaborations for Successful COTS Development*, May 2000.

that, when changed, effect the functioning of middleware solution.

Utilizing broad, but descriptive, architectural characteristics of the component systems provides dual benefits to discover initial interoperability problems and to assess inevitable future conflicts brought on by evolution. Through our research we hope to pinpoint advantageous designs for consistent and flexible interactions of component systems. This approach disavows anachronistic design by allowing developers insight into evolutionary changes, making it easier to implement software that can withstand change.

REFERENCES

1. A. Abd-Allah, Composing Heterogeneous Software Architectures, Ph.D. Dissertation, Dept. of CS, USC, August 1996.
2. G. Abowd, R. Allen and D. Garlan, Formalizing Style to Understand Description of Software Architecture, *ACM TOSEM*, 4(4):319-364, 1995.
3. R. Allen and D. Garlan, A Formal Basis for Architectural Connection, *ACM TOSEM*, 1997.
4. L. Davis, and R. Gamble, The Impact of Component Architectures on Interoperability, Submitted for publication, Dec. 1999.
5. L. Davis, J. Payton, and R. Gamble, Examining the Role of System Requirements on Interoperability, Submitted for publication, Feb. 2000.
6. D. Garlan, R. Allen, and J. Ockerbloom, Architectural mismatch or why it is so hard to build systems out of existing parts, *17th Int'l Conf. on Software Engineering*, April 1995.
7. A. Kelkar and R. Gamble, Understanding the architectural characteristics behind middleware choices, *Proc. 1st Conf. on Information Reuse and Integration*, Sept. 1999.
8. N. Medvidovic, D.S. Rosenblum, and Richard N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pp. 44-53, Los Angeles, CA, May 16-22, 1999.
9. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pp. 177-186, Kyoto, Japan, April 19-25, 1998.
10. P. Oreizy, R. N. Taylor. "On the Role of Software Architectures in Runtime System Reconfiguration". *Proceedings of the Fourth International Conference on Configurable Distributed Systems (ICCDs 4)*, 61-70, Annapolis, Maryland, May 4-6, 1998.
11. J. Payton, R. Keshav, and R.F. Gamble, System Development Using the Integrating Component Architectures Process, *Proceedings of the ICSE-99 Workshop on Ensuring Successful COTS Development*, May 1999.
12. D. Perry and A. Wolf, Foundations for the study of Software Architecture, *SIGSOFT Software Engineering Notes*, 17(4): 40-52, October 1992.
13. V. Rajlich, J. H. Silva Evolution and Reuse of Orthogonal Architecture. *IEEE Transactions on Software Engineering*, Vol. 22, No. 2, February 1996.
14. M. Shaw, Some Patterns for Software Architectures, *2nd Annual Conference on the Pattern Language of Programmers*, May 1995.
15. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
16. R. Sitarman. Integration of software systems at an abstract architectural level, M.S. Thesis, Dept. of Mathematical & Computer Sciences, University of Tulsa, 1997.
17. D. Yakimovich, J. Bieman and V. Basilli Software Architecture Classification of

To appear in *ICSE-2000 Workshop Proceedings of Continuing Collaborations for Successful COTS Development*, May 2000.

Estimating the Cost of COTS Integration.
ICSE-99, May 1999.