How System Architectures Impede Interoperability

Leigh A. Davis Jamie Payton Rose Gamble
Dept. of Mathematical and Computer Sciences
University of Tulsa
600 S. College Avenue
Tulsa, OK 74104 USA
+1 918 631 3283
{davisl, payton, gamble} @euler.mcs.utulsa.edu

ABSTRACT

Interoperability problems arise when complex software systems are constructed by integrating distinct, and often heterogeneous, components. By performing interoperability analysis on the software architecture design of the system and its components, potential incompatibilities can be anticipated early in the design process. In this paper, we focus on an application's structural requirements as reflected in values of particular architectural characteristics, describing how they can be incompatible with individual component system properties.

Keywords

Software architecture, interoperability, requirements

1. INTRODUCTION

To truly resolve interoperability conflicts that impact performance among components, it is necessary to predict incompatibilities during the design of the integrated system. Fortunately, a variety of properties have been published that describe a component's software architecture at varying levels of abstraction. Certain pair-wise comparisons of these characteristics can detect where potential architecture mismatches occur during interaction [5, 1, 2, 11, 12, 6]. This paper examines how requirements decisions, as reflected in defined system-level architecture characteristics, impact interoperability among components, both in why conflicts occur and in how they are resolved.

2. CAUSES OF INCOMPATIBILITY

Software architecture is the description of the computational components of a program or system, the connectors that establish the interactions between the components, the overall configuration of components and connectors, as well as principles and guidelines governing their design and evolution over time [7, 10]. The phrase *architecture mismatch* has been used to describe the underlying reasons for interoperability problems among seemingly "open" software components with available source code [5]. Often, mismatch problems can be traced to fundamental characteristics of the architectures of the interacting systems.

Characteristics include types of components and connectors, data issues, control issues, and control/data interaction issues [9]. Component-level characteristics can be classified for their contribution to architecture interoperability, and can be related using semantic nets [4].

It is possible to formulate an integration strategy by analyzing conflicts between component characteristic values. However, this may result in either an incomplete or overly complex solution. To truly arrive at an initial solution the communication patterns of the overall system must be analyzed. These patterns are evident in stakeholder structural requirements such as decoupling or extensibility. Specifically, in this paper, we are interested in *system-level architecture characteristics*, i.e., those architecture properties that are synthesized from the integrated system's structural requirements. We propose to maintain a small set of characteristics, each of which encompasses a broad range of properties. Basically, these characteristics formulate the architectural demands on configuration and coordination of the participating components in the application [8]. Our approach entails the representative component-level characteristic set, which was paired down from the original 74, being compared against the demands of the system-level characteristics to form an initial integration solution [6, 4].

3. SYSTEM-LEVEL CHARACTERISTICS AND THEIR INFLUENCE

In the following section we detail the characteristics which we feel best represent a system's structural requirements. The findings detailed in each section reflect findings made in case studies from [3]. The two case studies utilized the same components but featured different system requirements, making their system-level characteristic values different. As a general finding, the influence of the system characteristics was less if the component-level characteristic values were equivalent.

WOSP 2000, Ontario, Canada

3.1 Control Topology

By depicting the arrangement of the components according to the desired control interactions, this characteristic directs attention to those pairwise comparisons concerned with control issues [9]. Different layouts regarding how the individual components exchange control can alter the complexity of the interaction. Thus, the value of the control topology provides expectations for the control interaction that can lead to interoperability problems. For example, given an arbitrary system control topology, its influence on highly coupled components requires some sort of coordination device to simulate the dynamic control geometry of an arbitrary topology.

3.2 Data Topology

Like control topology, by depicting the arrangement of the components according to their required data interactions, this characteristic's value defines only those pairwise comparisons concerned with data exchange issues [9]. One common incompatibility is due to the disparate data representation formats supplied by each participating component. Hence, the topology of the components can directly affect how translators are used for data interchange. Should all the values at the system-level interact benignly with properties of the components, if they do not communicate the exact same format of data there still will be problems between interacting components. Therefore, translation devices will always be needed to mediate data between communicating components.

3.3 Control Structure

The value chosen for control structure affects what the system expects as the kind of control flow, including the scope of control, and whether it is simultaneous [11]. To gauge difficulties between component property values and system requirements, information concerning which component governs control at any time during execution, and how that control is governed must be considered. Coordination will be necessary should your system value be decentralized and some of your participating components single thread. In this way, the system can simulate concurrent execution and communication.

3.4 Synchronization

The synchronous or asynchronous values of this characteristic impact the style of communication that can, in turn, affect interaction [12, 6]. For example, should a asynchronous system be required, but most components present in the system block a polling mechanism will be necessary. Therefore, when the blocking components finish their execution and wait to be re-executed there will be a mechanism in place to check who is free to process new data.

4. CONCLUSION

Solutions for integration exist, but their complex nature often leads to a difficult derivation and a lengthy development of the solution. As a result, performance of the application can suffer, even if middleware is used. Using the software architecture as a basis for analysis aids in the prediction of interoperability problems among interacting components. This prediction allows for alternative design decisions, as well as documented understanding of which middleware solutions to consider. Furthermore, it can also indicate where requirements changes or using components compatible to the requirements can alleviate the need for complex middleware.

5. REFERENCES

[1] A. Abd Allah, Composing Heterogeneous Software Architectures, Ph.D. Dissertation, Dep. of CS, USC, August 1996.

[2] R. Allen and D. Garlan. A Formal Basis for Architectural Connection, *ACM TOSEM*, 1997.

[3] L. Davis, J. Payton, and R. Gamble, Under the Influence: How System Architectures Impede Interoperability, Technical Report UTULSA-MCS-2000-10, University of Tulsa, Tulsa, Oklahoma, March, 2000.

[4] L.Davis, J. Payton, and R. Gamble, The Impact of Component Architectures on Interoperability, Submitted for Publication, December 1999.

[5] D. Garlan, R. Allen, and J. Ockerbloom, Architectural mismatch or why it is so hard to build systems out of existing parts, *Proceeding of the 17th International Conference on Software Engineering*, April 1995.

[6] A. Kelkar and R. Gamble, Understanding the architectural characteristics behind middleware choices, *Proc. 1st Conf. on Information Reuse and Integration*, Sept. 1999.

[7] D. Perry and A. Wolf. Foundations for the study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4): 40-52, October 1992.

[8] M. Radestock and S. Eisenbach, Component Coordination on Middleware Systems, *IFIP Int'l Conf. on Distributed Systems Platforms*, 1998.

[9] Mary Shaw and Paul Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. Proc. COMPSAC97 and *1st Int'l Computer Software and Applications Conference*, August 1997, pp. 6-13.

[10] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[11] R. Sitaraman. Integration of Software Systems at an Abstract Architectural Level, M.S. Thesis, Department of Mathematical

[12] D. Yakimovich, J. Bieman, and V. Basili, Software Architecture Classification for Estimating The Cost Of COTS Integration, Proceedings from the 21st International Conference on Software Engineering, 296-302, (1999).