

Using XML for an Architecture Interaction Conspectus¹

J. Payton L. Davis D. Underwood R. Gamble²

Department of Mathematical and Computer Sciences

University of Tulsa

600 South College Avenue

Tulsa, OK 74104 USA

+1 918 631 2988

{payton, davisl, underwood, gamble}@utulsa.edu

ABSTRACT

Architectural descriptions of component style and behavior provide powerful predictive capabilities for interoperability conflicts. In this paper, we describe architectural indicators for a first pass interoperability assessment. These indicators form a set that is minimal and realizable, thus having the potential for automated forecasting. We encapsulate these indicators into a XML *architecture interaction conspectus* (XMLAIC) that fronts each independent software system. XML provides an open, standards-based way to implement the conspectus, affording industry-wide interoperability assessment for both pre- and post-component purchase.

1 INTRODUCTION

The progression of component-based software engineering (CBSE) is essential to the rapid, cost effective development of complex software systems. Much of the research in CBSE targets choosing and adapting components for a composite application. While such reuse requires effective methods of component retrieval, it also requires adequate component assessment. Specifically, this assessment must be geared toward determining potential interactions among components, the composite application requirements, and middleware.

Assessment should make use of readily available properties based on known design information. Unfortunately, there is yet no standard methodology to formulate component comparisons. The methodology should generate usable results pertaining to the problematic nature of the interaction, while maintaining understandability, practicality, and cost-effectiveness. However, a methodology is only as strong as the information foundation on which it relies. Therefore, it is imperative that the proper building blocks are in place, meaning that software interoperability properties must be gathered and quantified in a stable manner.

Our approach is to summarize component interaction

using a set of indicators that reflect the architectural information known about a component [3, 10]. We use XML to formulate these indicators into an *architecture interaction conspectus* (XMLAIC) that can then be used to detect potential interoperability conflicts during application integration. We rely on software architecture for coarse-grained structural and behavioral indicators as derived from architectural style properties and protocols. The XMLAIC can be thought of as a component résumé to accompany or be attached to a component, allowing for a quick assessment of its fit in an integrated application.

2 RELATED BACKGROUND

The *software architecture* of a system is a coarse-grained description of its computational elements, the means by which they interact, and the structural constraints on that interaction [13, 15]. With such abstraction, it is possible to focus on important conceptual system issues without becoming entangled in implementation and deployment details. One facet of this abstraction is architectural style, which provides information regarding pattern-based configuration and coordination constraints. A large number of architectural characteristics have been identified to differentiate individual styles, their extensions, and specializations. These characteristics embody descriptions of the various types of computational elements, connectors, data issues, control issues, control/data interaction, and general component interoperability issues.

Using abstraction and semantic networks, we have delineated a highly connected, generic set of characteristics that are relevant to interoperability [2]. Moreover, we have found that it is feasible and desirable to examine them according to their involvement in different dimensions of integration [3]. First, *component-level characteristics* contribute to an understanding of the exposed interface of each participating component system. *Application-level characteristics* formulate the architectural demands on configuration and coordination

¹ This research is sponsored in part by AFOSR (F49620-98-1-0217) and NSF (CCR-9988320).

² Contact author.

of the component systems into a composite application.

Advances in methodologies for component assessment include specification matching, architecture definition languages (ADLs), and interface definition frameworks (IDFs). Specification matching is directed toward obtaining components that have the appropriate functionality, without focusing on whether control and data can be properly exchanged [17]. ADLs have examined properties for interaction problems among components [1, 12]. These properties are often detailed and specific to the particular analysis goals of the ADL. Research in IDFs focuses on fronting components with representative properties to facilitate their choice, e.g. from a library [8]. A large number of diverse properties can make up an IDF. These methodologies often require expertise in software architecture and formal methods, potentially limiting their usefulness to academia. For industry-wide acceptance, we need tools that are based on a theoretical foundation, yet are simple and extensible.

Though the intent of XML is to provide a means to structure information, it has recently been identified as a useful tool to aid in software description and interoperability conflict resolution. One approach uses XML to specify software architectures by structuring common abstractions found in ADLs within an XML Document Type Definition (DTD). This creates an extensible ADL, or xADL [11]. The motivation behind the development of xADL is to provide an interchange format that can support several ADLs in order to facilitate tool integration. A similar approach uses XML to provide platform independence for exchanging architectural models described in the Unified Modeling Language (UML) between development tools [16]. In [9], the authors propose specifying UML models in XML Metadata Interchange (XMI) to support evolution of UML models. XML can also be used in conjunction with middleware products to facilitate communication among components [7]. For example, distribution middleware can be combined with XML to reconcile data representation differences. The motivation is to alleviate the substantial development overhead associated with expressing complex data structures in the middleware interface definition language. Examining these approaches shows the viability of using XML as a

medium to communicate architectural information.

3 CONSTRUCTING AN ARCHITECTURE INTERACTION CONSPECTUS

In this section, we introduce the indicators for interoperability assessment that are structured using XML into the XMLAIC. These indicators highlight basic, yet relevant, software architecture properties and select functional behaviors, as well as non-functional requirements. We separate the indicators into four main categories: (1) identifying name and type, (2) style related characteristics, (3) implementation relationships, and (4) non-functional requirements [5]. For brevity, we discuss only the first two categories.

3.1 Name and Type

Every entity that participates in the development of an integrated system has a name or identifier that separates it from other systems. Currently, we partition these entities into three types: (1) the *application* that composes independent subsystems, (2) the *components* that are complete, executable systems or components in development, and (3) *middleware* that facilitates the interchange among components in the application. Our focus for this paper is on the two types that have similar indicators: applications and components.

3.2 Style Related Characteristics

There are eleven style-related characteristics identified as indicators for the conspectus [2]. We do not claim that the set is complete, because more characteristics may be introduced with further research. What makes this set essential for interoperability analysis is the encompassing nature of the definitions. The characteristics are course-grained, but reflect functionality present at lower levels of abstraction. That is, each high-level characteristic has multiple semantic links to corresponding low-level characteristics [2, 3, 10]. Two characteristic indicators, control topology and control structure, are defined in Table 1. Though applications and components differ slightly in how their indicators impact integration, both rely on these two.

Once defined, it is a natural fit to express the indicators in XML. Figure 1 shows the syntax of a portion of the conspectus as an XML DTD. In Figure 2, the XML

CHARACTERISTICS	DEFINITION	VALUES
<i>Control Structure</i>	The structure that governs the execution in the system	Single-Thread, Multi-Thread, Decentralized
<i>Control Topology</i>	The geometric form the control flow takes in a system [14]	Hierarchical, Star, Arbitrary, Linear, Fixed

Table 1: Characteristic Indicators

documents depict sample XMLAICs in which we assign values to the indicators in Table 1 for components *A*, *B*, and *C*. It is given that *A* does not interact with *C*. In this example, the required control topology of the integrated application (*APP*) is arbitrary (see Figure 2).

```
<!-- Element declarations -->
<!ELEMENT Conspectus (CharacterAnalysis?)>
<!ATTLIST Conspectus
    name CDATA #REQUIRED
    type (application | component | middleware) #REQUIRED>

<!-- Characteristic Section -->
<!ELEMENT CharacterAnalysis (ControlTopology?, ControlStructure?)>
<!ELEMENT ControlTopology EMPTY>
<!ATTLIST ControlTopology
    value (hierarchical | star | arbitrary | linear | fixed) #REQUIRED>
<!ELEMENT ControlStructure EMPTY>
<!ATTLIST ControlStructure
    value (single-thread | multi-thread | decentralized) #REQUIRED>
```

Figure 1: A Portion of the XMLAIC DTD

4 USING THE ARCHITECTURE INTERACTION CONSPECTUS IN XML

Every participating component will have values assigned to its known characteristic indicators. For each characteristic, the most restrictive, yet applicable value is chosen. Analysis on component-component interactions uses a bipartite graph to examine: (1) same characteristics with like values, (2) same characteristics with mismatched values, and (3) different characteristics. This approach differs from other assessments by expanding the comparison beyond same characteristics with mismatched values.

4.1 Problematic Architecture Interactions

A *problematic architecture interaction* is an interoperability conflict that is predicted through the comparison of architecture interaction characteristics and requires intervention via external services for its

resolution. The notation, $? T?$, means "problematically interacts causing conflict set *T*," where *T* is a subset of predefined conflicts [6]. For illustration purposes, we use two common conflicts: *F1: Restricted points of control transfer* and *F2: Sequencing multiple control transfers*.

Given the values in Figure 2, empirical analysis shows that

$$CT.Hierarchical (B) ? \{F1, F2\}? CS.Decentralized (C)$$

$$CT.Hierarchical (B) ? \{F1, F2\}? CT.Arbitrary (C)$$

$$CT.Hierarchical (A) ? \{?\} ? CT.Hierarchical (B)$$

$$CT.Arbitrary (APP) ? \{F1, F2\}? CT.Hierarchical (A),$$

$$CT.Hierarchical (B)$$

where *CT* and *CS* refer to control topology and control structure, respectively. These relations indicate that both conflicts occur between *B* and *C*. The call-return, point-to-point control flow of a hierarchical topology (*B*) cannot directly pass control to a component (*C*) whose concurrent threads provide no fixed entry point or control flow.

After formulating the set of component-component problematic interactions, we perform a second analysis for application-component interactions. In the example, it appears that no component-component conflicts occur between *A* and *B*. The absence of conflicts indicates the potential for simple connectors to be deployed. However, the requirements for the application's control topology forces components that normally have direct control exchange to communicate in a more decoupled manner. Now, intervention is needed in the form of external integration services, such as a mediator, to conduct the control transfer between components *A* and *B*. Through this notation and assessment process, a static set of problematic interactions can be formulated [6].

<pre><Conspectus name="A" type="component"> <CharacterAnalysis> <ControlTopology value="hierarchical"/> <ControlStructure value="single-thread"/> </CharacterAnalysis> </Conspectus></pre>	<pre><Conspectus name="B" type="component"> <CharacterAnalysis> <ControlTopology value="hierarchical"/> <ControlStructure value="single-thread"/> </CharacterAnalysis> </Conspectus></pre>
<pre><Conspectus name="C" type="component"> <CharacterAnalysis> <ControlTopology value="arbitrary"/> <ControlStructure value="decentralized"/> </CharacterAnalysis> </Conspectus></pre>	<pre><Conspectus name="APP" type="application"> <CharacterAnalysis> <ControlTopology value="arbitrary"/> </CharacterAnalysis> </Conspectus></pre>

Figure 2: Example XMLAICs

4.2 The XMLAIC

The XMLAIC offers options for pre- and post-purchase interoperability assessment of components. We are currently in the process of developing a tool based on our methodology to automate the comparison of XMLAICs. It is conceivable that the XMLAIC could be posted on a vendor website, so potential customers can download a XMLAIC to assess the fit of a commercial product in any integration. Also, the XMLAIC can be shipped with the product and archived by the customer. By having and comparing XMLAICs associated with multiple versions of a product, the negative impacts of product evolution on an integrated application can be discovered prior to upgrade [4].

5 DISCUSSION AND CONCLUSION

The intended use of the XMLAIC is primarily for a first pass assessment of potential problematic architecture interactions. Providing high-level indicators ensures the consistent effectiveness of our approach from which further, more detailed assessment can be performed. XML facilitates the realizable use of the defined indicators as a way to perform this multi-dimensional analysis. Furthermore, storing architecture information in XML allows for the addition and modification of existing characteristics, without rendering previous versions unusable as the set of characteristics evolve.

REFERENCES

1. R. Allen. A Formal Approach to Software Architecture. Ph. D. Dissertation, Department of Computer Science, Carnegie Mellon University, 1997.
2. L. Davis, R. Gamble, J. Payton. The Impact of Component Architectures on Interoperability. *to be published in Journal of Systems and Software*, 2002.
3. L. Davis, J. Payton, R. Gamble. How System Architectures Impede Interoperability, *To appear in the 2nd International Workshop On Software and Performance*, 2000.
4. L. Davis, J. Payton, R. Gamble. Toward Identifying The Impact Of COTS Evolution On Integrated Systems. In, *2nd Int'l Workshop on the Successful Development of COTS*, 2000.
5. L. Davis, J. Payton, R. Gamble, D. Underwood, D. Flagg. Component Indicators for Architectural Interaction Assessment TR UTULSA-MCS-00-15. Department of Mathematical and Computer Sciences, University of Tulsa, 2000.
6. L. Davis, J. Payton, G. Jonsdottir, R. Gamble, D. Underwood. A Notation for Problematic Architecture Interactions. under review, 2001.
7. M. Emmerich, W. Schwatz, A. Finkelstein. A Markup Meets Middleware. In *Proceedings of the 7th International Workshop on Future Trends in Distributed Systems*. Capetown, South Africa: IEEE Computer Society Press., 261-66, 1999.
8. J. Han. A Comprehensive Interface Definition Framework for Software Components. In, *Asia Pacific Software Engineering Conference*, 1998.
9. F. Keienburg, A. Rausch. Using XML/XMI for Tool Supported Evolution of UML Models. In *Proceedings of the 34th Hawaii International Conference on System Sciences*, 2001.
10. A. Kelkar, R. Gamble. Understanding the Architectural Characteristics behind Middleware Choices. In, *1st International Conference in Information Reuse and Integration*, 1999.
11. R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, R. Taylor. xADL: Enabling Architecture-Centric Tool Integration with XML. In *Proceedings of the 34th Hawaii International Conference on System Sciences*, 2001.
12. J. Magee, N. Dulay, S. Eisenbach, J. Kramer. Specifying Distributed Software Architectures. In, *The 5th European Software Engineering Conference*. Barcelona, Spain, 1995.
13. D. Perry, A. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT*,17(4): 40-52, 1992.
14. M. Shaw, P. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In, *1st International Computer Software and Applications Conference*. Washington, D.C., 6 17, 1997.
15. M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall, 1996.
16. J. Suzuki, Y. Yamamoto. Toward the Interoperable Software Design Models: Quartet of UML, XML, DOM and CORBA. In *Proceedings of the 4th IEEE International Symposium and Forum on Software Engineering Standards*. Curitiba, Brazil, 1999.
17. A. Zaremski, J. Wing. Specification Matching of Software Components. *TOSEM*,16(4): 333-69, 1997.