Beyond Documentation

R. Gamble[*] L. Davis D. Flagg G. Jonsdottir

Department of Mathematical and Computer Sciences
University of Tulsa
600 South College Ave
Tulsa, OK 74104
Phone: (918) 631-2988 Fax: (918) 631-3077
gamble@utulsa.edu

Abstract

Software component integration is increasingly permeating product development, impacting everything from government, e-commerce and enterprise application systems. While companies and government agencies attempt to streamline their business and provide better customer service, software and middleware vendors are all vying for dominance to bridge the gap between legacy software and COTS, and the new technology that is available to integrate them. There are a number of major problems with this scenario, particularly those with respect to determining the fit of a component in a multi-component application, instantiating the medium by which components should be connected, assuring that the integration meets security requirements, and defining the enterprise infrastructure that enfolds them.

We seek to demystify the integration process. The objective of our research is to efficiently assess qualified software properties, as well as other factors that constrain the multi-component application. To commence the assessment process, we use a software component conspectus that embodies static, architecturally significant properties. These properties can be easily described in the XML, making them portable and extensible, as well as available as part of online documentation. The evaluation produces the major architecture conflicts among the components and any requirements that govern the overall application. The conflicts can then be used to define the basic integration functionality needed for the middleware architecture. Hence, the result of this evaluation is an integration architecture that encompasses the application and the technology that it comprises.

As we move toward further characterization of the technology at work in a multi-component application, we envision empowering developers to quickly assess their software components and integration needs. However, a critical issue is getting software and middleware vendors to embrace the conspectus idea: to go beyond traditional documentation toward qualifying a uniform set of properties that make evaluation easier. To develop the assessment technology, we must define the relevant properties of components, communication mediums, middleware, infrastructures and security concerns in a normalized and uniform way. The values of these properties should be easily discernable from documentation, external interfaces, functional paradigms, or application domains. With the advent of this technology businesses can achieve a large payoff by lessening time and developer resources.

[*]Contact author.

1 Introduction

Government agencies and corporations with multiple business units require the consolidation of certain processes, the centralization of some of their key operations, and the integration of components developed in-house and by contractors. Dynamic assessment and on-the-fly composability may be necessary for real-time, reactive systems. In many instances, the incorporation of a middleware product alone has proven to be unsatisfactory. Often, it does not address all of the problems, it cannot grow to accommodate changes in the architecture or participating components, or it may not coincide with the current infrastructure set by a company for constructing multi-component applications. Understanding integration, empowering in-house developers and maintainers, and capturing a design history are essential. We are attempting to uniformly address these issues by assessment, tool development, and defining a practical theory of integration.

An ongoing problem is striking the balance between research technology and industrial use of that technology. To provide an integration support tool we must minimize the detail needed for interoperability analysis. Our approach is to formulate consistent descriptions across the various factors involved in composability assessment, e.g., software systems, application requirements, interoperability conflicts, and middleware. We use principles of software architecture as a basis for this description. From this basis, we define problematic architecture interactions (PAIs) and a process by which to predict them. This includes investigating the feasibility and contents of an architecture interaction conspectus (AIC) and developing an assessment prototype, called Collidescope. In light of recent events, we are exploring issues of security in multi-component applications as they relate to architecture and component mismatch.

2 Foundational Characteristics

Hypothetically, the composition of existing software components and the migration of legacy components to other environments or applications should cut development cost and time. Unfortunately, components often are not as adaptable to new applications or environments as developers assume they will be. Moreover, there is a lack of experience in addressing interoperability problems and therefore, they are often postponed until implementation, in favor of the use of vendor provided middleware. To empower inexperienced developers an useful and usable assessment technology must be employed at design time.

The main challenges to developing this assessment technology are:

- Accumulating and identifying relevant properties
- Organizing properties into a uniform representation,
- Developing a process to analyze properties to produce interoperability conflict descriptions
- Defining resolution techniques that can express potential integration solution frameworks.

To date, we work with nine architectural style-related characteristics identified as indicators for a component conspectus (Davis and Gamble 2002). We do not claim that the set is complete, because more characteristics will 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 (Keshav 1999; Davis, Payton et al. 2000; Davis and Gamble 2002). Though they share properties (see Table 1), application requirements (A), components (C) and middleware (M) differ in how their indicators impact integration.

Table 1: Application and Component Level Characteristics

ABBRV	CHARACTERISTICS	TYPE	DEFINITION
Bk	*Blocking*	C, M	Whether or not the thread of control is suspended
CS	*Control Structure*	A, C	The structure that governs the execution in the system
CT	*Control Topology*	A, C	The geometric form control flow takes in a system
DSM	*Data Storage Method*	C	How data is stored within a system
DT	*Data Topology*	A, C	The geometric form data flow takes in a system
IC	*Identity of Components*	C, M	Awareness of other components in the system
SCT	*Supported Control Transfer*	C, M	The method supported to achieve control transfer
SDT	*Supported Data Transfer*	C, M	The method supported to achieve data transfer
Sn	*Synchronization*	A, M	Whether or not the components need to rendezvous

Once defined, it is a natural fit to express the conspectus in XML. Figure 1 shows the syntax of a portion of a conspectus as an XML DTD. Also in Figure 1, we depict sample XML conspectuses in which we assign values to indicators in Table 1. For example, imagine *component A* is a legacy component written in FORTRAN. Its control topology and control structure should match that programming paradigm, e.g. hierarchical and single-threaded, respectively. The application (*APP)* in which *component A* will be inserted requires its interaction to simulate an arbitrary control topology (Shaw and Clements 1997).

```
<!-- 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>
```

```
<Conspectus name="A" type="component">
  <CharacterAnalysis>
    <ControlTopology value="hierarchical"/>
    <ControlStructure value="single-thread"/>
  </CharacterAnalysis>
</Conspectus>
```

```
<Conspectus name="APP" type="application">
  <CharacterAnalysis>
    <ControlTopology value="arbitrary"/>
  </CharacterAnalysis>
</Conspectus>
```

Figure 1: A Portion of the XMLAIC DTD and Example XMLAICs

3 Problematic Architecture Interactions

We have developed a process that compares various conspectuses and generates a high-level set of *problematic architecture interactions* (PAI) (Davis, Gamble et al. 2001). We define a PAI as an interoperability conflict that is predicted through the comparison of architecture interaction characteristics and requires intervention via external services for its resolution. We base our approach on the assumptions that PAIs occur when

1. Two component systems are required to interact but are inhibited by certain characteristic values (component-component)
2. The configuration and coordination requirements of the application impose demands for a certain style of interaction that one or more components cannot satisfy (application-component)
3. The configuration and coordination requirements of the application are themselves in conflict (application-application).
4. The application expectations do not comply with the integration solution requirements (application-integration).
5. The application requirements do not coincide with the business integration infrastructure (application-infrastructure).
6. Security aspects of a component that are not supported by the integration architecture will affect the level of security in the application (security-integration).

For example, a PAI related to control transfer problems can be readily seen by the comparison of conspectuses in Figure 1. Component A expects a call-return style of control transfer. Yet that is not guaranteed within the application. Therefore, intervention to perform the appropriate simulation is needed.

4 Using an Architecture Interaction Conspectus

There exist patterns that aid developers in resolving coarse-grained integration problems among components (Gamma, Helm et al. 1995) (Buschmann, Meunier et al. 1996). These patterns are assembled from functional entities that resolve various communication issues. However, little attention has been paid to how specific interoperability problems and their resolution are embodied in these patterns. Detailing this understanding as a pattern promises insight into composing integration functionality by illuminating the consistent, high-level solutions that resolve individual conflicts.

We compare architecture characteristics of heterogeneous components to generate a set of PAIs where each element of the set has a direct mapping to an architectural solution pattern. The produced set of solution patterns constrains the selection of the integration solution framework to only the functionality that can resolve the interaction problems. We combine and extend existing pattern languages to accommodate the information needed to express an *interoperability conflict pattern*. We use an Extender-Translator-Controller (ETC) classification scheme within the pattern to describe a typical solution for the particular conflict (Davis, Gamble et al. 2000).

Given these patterns and the conspectuses, we can derive abstract solution pieces to the component-component and application-component PAIs generated by the comparison process. This path, starting from a limited amount of information beyond traditional documentation, results in a powerful understanding of the integration solution design. Moreover, the process can be implemented. Our prototype, Collidescope[1], takes XML conspectuses and detects the potential PAIs. It can be used to assess the fit of a component into an evolving application (Davis and Gamble 2002). As our technology matures, developers can use this assessment tool without vendor assistance or influence.

5 Security Concerns in Integration

Managing and coordinating multiple security policies is an intricate problem currently faced by all businesses. For a variety of reasons, multiple business units, once separate entities, require the shared access to information systems. By performing integration that allows information sharing, the level of security in such a multi-component application decreases (Whiting and Chabrow 2001). These systems are more open to attack than their individual components were. For example, when different access control policies are involved, the policies must be allowed to remain independent, and some mediation between them must occur.

One concern is that the separate analysis of architectural differences and security issues between software components can lead to redundant and overly complex integration solutions. To resolve interoperability problems, they must be uniformly characterized so that a joint *integration architecture* can be appropriately designed.

6 Conclusion

We describe a process that is a first step in evaluating what influences and constrains interoperability. There is further research to be performed that not only addresses theoretical aspects of constructing a multi-component application, but also the organizational and security-related issues. Indeed, it may be the case that a seamless integration can be achieved given the participating components, but it might not fit within the organization infrastructure. Therefore, we advocate a new paradigm for component-based software development that accounts for the business model.

To gain wide-range acceptance of the conspectus idea and the use of the assessment technology, there must be an effort on the part of developers to share conspectuses and maintain a history of their use within integrated applications. This is much like the successful adoption of patterns started by that community of experts. Only then will software providers go beyond documentation to include component information essential to integration.

7 References

Buschmann, F., R. Meunier, et al. (1996). Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons.

Davis, L. and R. Gamble (2002). "The Impact of Component Architectures on Interoperability." Journal of Systems and Software.

Davis, L., R. Gamble, et al. (2001). A Notation for Problematic Architecture Interactions. ESEC/FSE, Vienna, Austria.

Davis, L., R. Gamble, et al. (2000). Conflict Patterns: Toward Identifying Suitable Middleware. Tulsa, OK.

Davis, L. and R. F. Gamble (2002). Identifying Evolvability for Integration. Internation Conference on COTS-Based Software Systems, Orlando, Florida, Springer-Verlag.

Davis, L., J. Payton, et al. (2000). How System Architectures Impede Interoperability. 2nd International Workshop On Software and Performance.

Gamma, E., R. Helm, et al. (1995). Design Patterns Elements of Reusable Object-Oriented Software, Addison-Wesley.

Keshav, R. (1999). Architecture Integration Elements: Connectors that Form Middleware. M.S. Thesis, Department of Mathematical and Computer Sciences, University of Tulsa.

Shaw, M. and P. Clements (1997). A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. 1st International Computer Software and Applications Conference, Washington, D.C.

Whiting, R. and E. Chabrow (2001). Safety In Sharing. InformationWeek.Com.

[1] You can visit our website at http://www.seat.utulsa.edu to see a Flash presentation of Collidescope.