PLANNING FOR MIDDLEWARE

R. Gamble, J. Payton, L. Davis
Dept. Mathematical & Computer Sciences
University of Tulsa, Tulsa, OK 74104
gamble@utulsa.edu
(918) 631-2988 voice *(918) 631-3077 fax*

Software products will continue to be built from independent, heterogeneous components distributed across a network with the goals of increasing reusability and decreasing time to market. Software architecture research has been instrumental in facilitating descriptions of the components of these systems [ALL97, MDEK95, SG96], along with producing generic development methods [BOE99, JBR99] and specific analysis techniques such as determining COTS product suitability and integration constraints [PKG99, YBB99].

One major consideration in these component-based systems is their use of middleware; a variety of distributed computing services and application development environments which operate between the application logic and the underlying system [CHA99]. Employing a commercially available middleware product, such as CORBA, DCOM, Java RMI, or MQSeries Integrator, is commonly thought to be an easy solution to resolving interoperability problems among heterogeneous components. However, due to the complexity of middleware products, implementation of an integration solution may be time and resource intensive. In addition, evolving a middleware product may require the assistance of an expert consultant or vendor, or even changing products altogether.

Research trends in software architecture are emerging to address some of the issues surrounding middleware that form the foundation for our research. For instance, the influence of software architecture characteristics on middleware choices [KG99] and vice versa [DIN99] has led to a deeper understanding of the interplay between the design of the system and the need for, and subsequent performance of, its middleware.

Reports on failed integration experiences [BAY98] warrant a way to first identify the middleware framework to resolve interoperability conflicts, followed by choosing a product that implements the desired framework according to traditional criteria, such as cost, vendor reputation, and familiarity with the product. Even better is a method by which to plan for middleware as part of the design of the integrated system and the choice of reusable components. This is the direction in which our research is proceeding.

Our research involves defining an analysis approach that allows the system developer to plan for middleware during the system design. One challenge to developing such a plan is to separate the intricate functionality of the disparate middleware frameworks. In most cases, the full integration capabilities are not even needed, but what to use or what to ignore within a product often must be determined by vendor consultants, as opposed to software developers. Specifically, questions such as the following need to be answered:

- Is a particular middleware framework/product well suited to the system needs?
- Can the middleware perform as expected given its requirements and the system’s requirements?
- How are the middleware choices influenced by the system requirements, the architecture of the overall system, and the participating components?

Our approach is based on providing consistent descriptions across middleware frameworks to facilitate analysis of their reusability, flexibility, evolvability, and interoperability. Our previous research focused on identifying and modeling generic integration element connectors and their common specializations for translation, control, and extension within the core integration capabilities of middleware [KG98]. These integration elements are supplemental to the standard architecture connectors defined for many architecture styles [SG96, AG97].

To broaden this research, we are developing a theory for composing integration elements and standard architecture connectors into an *integration architecture* that specifies the solution needed. As shown in Figure 1, the integration architecture should cover the main interaction points and form the skeleton of the middleware. Hence, the objective is to capture and codify middleware functionality by leveraging architecture concepts. The integration architecture must both satisfy system requirements and be implementable by current/future middleware products. Thus, we do not wish to produce solutions that are incongruous to current middleware products or even to enforce a change in the current industry. Rather, we want to initiate within the design process the consideration of middleware usage to facilitate later implementation. Finally, the description of the integration elements and connectors should be such that an automated composition is possible as a result of architecture and interoperability analysis. While we are not tied to any ADL specifically, the proposed descriptions should be transformable.

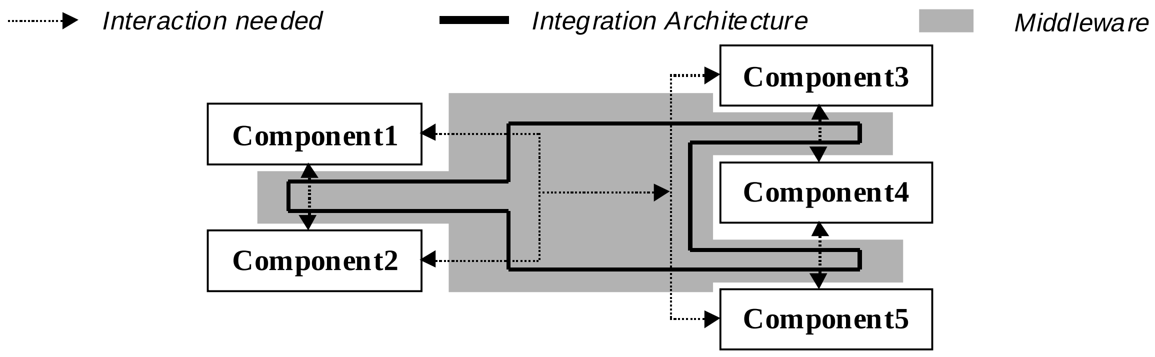

Figure 1: An Integration Architecture and its Relationship to Middleware

Our goal is to develop a principled approach using a combination of formal and semi-formal methods of architecture specification. These descriptions allow guarantees that the resulting integration architecture is correct at two levels. First, composition must maintain all properties satisfied (and conflicts resolved) by the individual integration elements. Second, the resulting architecture must satisfy the requirements of the integrated system application. Unfortunately, the choice among solutions may depend on what other integration elements are needed or the configuration/coordination requirements of the overall system. Two integration architectures may be comprised of the same integration elements, but the *way* they are composed influences their functionality [KPFG99]. This composition is partly impacted by the system requirements and partly by characteristics of the participating components. Hence, a composition theory is not as simple as accumulating the integration elements to resolve conflicts and then directly combining them.

For example, assume that we have the two configurations of components as depicted in Figure 2. The data representations of CS_1, CS_2, and CS_3 are all distinct. This conflict requires a translator (either unidirectional or bi-directional) for each comparison. In the first linear configuration of the data topology, two unidirectional translators are needed to resolve the representation conflict. This results in implementing middleware adapters. In the complete graph, each of the three pairwise conflicts would result in its own bi-directional translator. However, when composing the integration elements into an overall solution, the integration architectures would be different. Here, a translator would likely be used to convert between the data representations into an intermediate language. The intermediate representation would then be translated to the appropriate receiver. In this case, a controller would be used to determine which component system gets the translated information, therefore, requiring a broker framework.

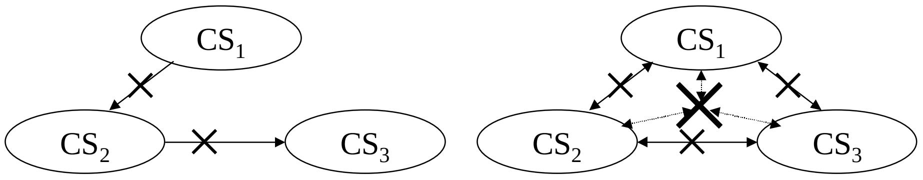

Figure 2: Two System Configurations of the Same Components

As a result of this research, we intend the design plan for middleware within a network based application to be evolvable as the components change, to allow the developer to assess trade-offs between frameworks and products, to provide an initial expectation of performance, and to be reusable as similar integration efforts are performed. This requires advances in software architecture-based development and description that will lead toward standardized assessment of interoperability problems and the impact of architecture evolution on a component-based system.

REFERENCES:

[AG97] R. Allen & D. Garlan, A Formal Basis for Architectural Connection, *ACM TOSEM*, 1997.

[ALL97] R. Allen, A Formal Approach to Software Architecture, Ph.D. Dissertation, TR CMU-CS-97-144, 1997.

[BAY98] B. Bayliss and F. Kilpatrick, Briefing at Schreiver AFB, Nov. 1998.

[BOE99] B. Boehm, D. Port, A. Egyed, M. Abi-Antoun, The MBASE Life Cycle Architecture Milestone Package: No Architecture is an Island, *1st Working Int'l Conf. on Software Architecture*, 1999.

[CHA99] J. Charles, Middleware Moves to the Forefront, *Computer*, 22 (5), 1999.

[DIN99] E. Di Nitto and D. Rosenblum, Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures, *21st Int'l Conf. on Software Engineering*, pp. 13-22, 1999.

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh, *Unified Process Development Model*, Addison Wesley, 1999.

[KG98] R. Keshav and R. Gamble, Towards a Taxonomy of Architecture Integration Strategies, *3rd Int'l Software Architecture Workshop*, Nov. 1998.

[KG99] A. Kelkar and R. Gamble, Understanding the Architectural Characteristics Behind Middleware Choices, *Proc. 1st Int'l Conf. on Information Reuse and Integration*, Sept. 1999.

[KPFG99] R. Keshav, J. Payton, K. Frasier, and R. Gamble, Architecture-Directed Integration, Submitted for publication, Dec. 1999.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, Specifying Distributed Software Architectures, *Proc. of the 5th European Software Engineering Conference*, 1995.

[PKG99] 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.

[SG96] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[YBB99] D.Yakimovich, J. Bieman, and V. Basili, Software Architecture Classification for Estimating The Cost Of COTS Integration, *21st Int'l . on Software Engineering*, pp. 296-302, 1999.