BRIDGING HETEROGENEOUS SOFTWARE INTEROPERABILITY PLATFORMS

Nenad Medvidovic
University of Southern California
Los Angeles, CA
neno@usc.edu

David S. Rosenblum
University of California, Irvine
Irvine, CA
dsr@ics.uci.edu

Rose F. Gamble
University of Tulsa
Tulsa, OK
gamble@utulsa.edu

1. Introduction

The software systems of today are rapidly growing in size, complexity, amount of distribution, heterogeneity of constituent building blocks (*components*), and numbers of users. With rapid increases in the speed and capacity of hardware, its dropping cost, and the emergence of the Internet as a critical national (and worldwide) resource, the demand for software applications is outpacing our ability to produce them, both in terms of their sheer numbers and the sophistication demanded of them. Most notable about current software development practice is the continued preponderance of ad-hoc approaches driven by industry needs, commercial interests, and market pressures, rather than scientific principles.

A recent development that has shown promise to address large-scale software engineering challenges (component-based development, software reuse, software interoperability [P99]) has been the emergence of generic software interoperability services, also referred to as *middleware*. The best-known examples of these include CORBA [Sie96], COM [Box98] and Enterprise JavaBeans [FFC+99]. The ultimate goal of these technologies is to achieve software development and component interoperability with lower development costs, higher developer productivity, and increased off-the-shelf (OTS) reuse. Initial success stories have been used by middleware vendors to support their growing claims of an imminent "plug-and-play" software component marketplace. The expectation has been that their technologies will minimize interoperability problems and that those problems that do arise can be resolved by standard middleware.

Unfortunately, few of these claims have proven true. It is not surprising that competing middleware vendors vying for market dominance have ended up constructing incompatible, proprietary component and middleware standards. The current situation can be characterized as a "component tower of Babel:" components "speaking" the same language are interoperable, while those "speaking" different languages are not. Thus, for example, a COM component can readily communicate with another COM component, but not with a CORBA component; furthermore, components implemented using different flavors of CORBA are often unable to interact. This fragmentation has been coupled with a "one size fits all" mentality, whereby a particular middleware platform is touted as being *the* solution to the problem of component-based development. In reality, each technology has its limitations. The fragmentation has also focused developer training on the idiosyncrasies and implementation-level details of a particular technology, rather than the general principles of component-based development.

This assessment paints a bleak picture of the future and indicates that some of the existing software engineering problems are likely to worsen without fundamental technological improvements in component-based development. To address this state of affairs, we propose an approach whose goal is to establish scientific principles and create technological solutions to support *software multioperability*, the ability of software components to be multiply-interoperable with other kinds of components automatically, seamlessly, and regardless of their underlying architecture or middleware basis. A simple hardware analogy to multioperable software components is the modem. Every modem is built to adaptively interoperate with another modem by dynamically selecting a "middleware" or protocol that both modems can understand. Of course, this middleware between modems is much simpler than what is needed for software components since the choices of protocols are small in number and widely adopted. Additionally, the services offered by the modem protocols are fairly simple and show little variation, allowing the placement of all the adaptive capability in the modems themselves. In contrast, the richness and variety of the services required of middleware for software components are vast and require more than an ability to adaptively tailor the components alone. Indeed, for certain kinds of legacy components, adaptation may not even be a feasible engineering option.

Thus, a major problem in providing support for software multioperability is that of *bridging inter-middleware gaps*. In particular, mechanisms are needed for analyzing the *interactions* between components and automatically constructing the appropriate integration solutions. This is the focus of our work—rather than proffering yet another standard to which all components are (unrealistically) expected to adhere in order to achieve interoperability, we suggest the technological foundations needed to automatically construct the necessary integration solutions between heterogeneous middleware.

Our approach comprises three distinct, but interwoven facets:

- Identification and formalization of the architectural styles a middleware technology induces. Though no middleware platform explicitly recognizes that it imposes a particular style of application composition upon the developers (a direct by-product of the vendors' "one size fits all" mentality), our preliminary studies have indeed shown this to be the case [DR99]. In order to understand the true extent of applicability of a particular middleware platform and the idiosyncrasies of its integration with other platforms, such an analysis is a necessity.
- Classification and formalization of the underlying principles, fundamental building blocks, and compositional properties of middleware. Although existing middleware technologies present a wide diversity of interoperability mechanisms, we have shown that they share certain general characteristics [KG98]. A formal treatment of middleware is needed to better understand their commonalties and potentially automate their interaction.
- Development of a principled method and viable technology to bridge the inter-middleware gaps. The first two facets deal with improving the existing understanding of middleware, as well as formalizing and predicting its nature and impact on applications. This facet strives to provide the inter-middleware "links" in a manner that is consistent, repeatable, automatable, and practical.

Our approach is based on applying and extending the principles of software architecture [PW92, SG96]. We believe that architectures provide the necessary leverage over middleware-based system composition. In many ways, software development based purely on middleware can be regarded as the "assembly programming" of software composition [DMT99]: a middleware technology provides no support for determining the application's structure and behavior, selecting the needed components, or interconnecting them in the appropriate ways. Software architectures supply the needed higher-level abstractions. The remainder of the paper provides additional details of the three thrusts of our approach, presents our initial results, and discusses future work.

2. Middleware-Induced Architectural Styles

In previous research we demonstrated the pervasive influence a middleware has on the architectures of the applications that use it [DR99]. Despite the fact that architectures and middleware are typically viewed as addressing different phases of software development, the usage of middleware can influence the architecture of a system being developed. Conversely, specific architectural choices constrain the selection of the underlying middleware used in the system's implementation phase.

For a system to be implemented in a straightforward manner on top of a middleware, the corresponding architecture must be compliant with the architectural constraints imposed by the middleware. Sullivan et al. corroborate this assertion by demonstrating how a particular style that in principle seems to be easily implementable using the COM middleware is actually incompatible with it [SSM97]. We have also stated this view in discussing the importance of complementing component interoperability models with explicit *architectural models* [OMTR98]. Our initial study of middleware-induced styles [DR99] had the relatively narrow goal of evaluating the ability of architecture description languages (ADLs) [MT00] to model the architectural styles induced by several well-studied middleware infrastructures. The study revealed several deficiencies in the modeling power of existing ADLs.

Thus, it is necessary to broaden this work to fully ascertain the architectural impacts of a middleware on the applications that use it and vice versa, and to account for these impacts in the construction of multioperability solutions. We argue that achieving multioperability within an application must be more than a localized process of piling adapter upon adapter until all component and middleware pieces are "plug compatible." Instead, multioperability must be achieved in a manner that respects the architectural properties and constraints that the designer has established for the application. For instance, if the application architecture is to be centered around a style of event-based interaction, the multioperability solutions that are incorporated must respect the essential properties of such a style—its asynchrony, the anonymity of event publishers and subscribers, the lightweight form of event data, and so forth. One challenge is to create a principled way of first capturing the constraints of middleware-induced architectural styles, and then feeding these constraints into the production of multioperability solutions.

3. A Theory of Middleware Interoperability

In addition to studying and formalizing the impact of middleware on application architectures, we must also study and formalize the fundamental architectural abstractions that middleware itself comprises. This second thrust thus forms the theoretical foundation of multioperability. It is based on our preliminary studies of architectural characteristics [SC97, GAO95, YBB99] that contribute to software interoperability problems and the fundamental integration building blocks that underlie middleware. To

date, we have classified specific architectural characteristics that inhibit component interoperability during system integration and play a part in middleware selection and usage decisions [KG99]. Our research suggests that middleware frameworks can be viewed as compositions of *integration elements* that provide generic ways in which interoperability conflicts cause by component characteristics are resolved [KG99, GPK99]. Subsequently, formalizing the architectures of these integration elements toward developing a theory for their composition will permit guarantees for system-wide multioperability properties, such as required topological profile, absence of deadlock, and information management

Building on these results will enable the understanding of those properties that inhibit system interaction *across* middleware platforms. One challenge is to model these properties in a manner that is consistent with the architectural properties imposed by application requirements and induced by the middleware (recall Section 2). Our approach will seeks to identify, capture, and separate the "core" software integration functionality from vendor-specific extensions. The extensions inhibit interoperability by reinforcing customer reliance on a single product. We hypothesize that there are patterns of extensions that can be elucidated through empirical study. The patterns include the manner in which these extensions are employed and whether they are linked to a particular interoperability problem or application domain. Consequently, where and why conflicts arise among middleware is an essential part of this research. Commonalties and distinctions in the various middleware frameworks' underlying integration elements can already be observed [KFGP99]. Further comparisons are needed of the underlying (implicit) usage assumptions made about a middleware framework - whether those assumptions are the same across all products that employ the same underlying framework (e.g., different implementations of CORBA).

Concluding this research thrust is the formation of a taxonomy of middleware that depicts its similarities, differences, extensions, and refinements. Naturally, this task is quite similar to (though a level of abstraction above) identifying the basic components and services provided by operating systems. By using a taxonomy, fundamental features of middleware functionality, proprietary extensions, and application assumptions can be modeled uniformly using combinations of semi-formal languages (e.g., the UML [RC99]) and formal languages (e.g. Object-Z [DRS94]) for architectural descriptions [AAG95, RMRR98, GSP99, GPK99, MR99], inheritance, refinement, temporal reasoning, and proofs. Based on these models, a theory will emerge to define the middleware conflicts and the essential parts needed to connect component-based systems, providing a formal underpinning for our third and final thrust.

4. Facilitating Software Multioperability with Software Connectors

The predominant focus of component-based development has been on designing, selecting, adapting, implementing, and integrating software components. Component development is controlled by industry demands for rich interoperability platforms and services and is unlikely to be driven by a single, universal standard. We thus believe that it is best to achieve multioperability among the (innumerable) heterogeneous components by integrating the (comparably very few) interaction mechanisms they employ. Existing middleware technologies have addressed component interaction via a predefined set of capabilities (e.g., RPC) that is typically not intended to be extensible. These capabilities are usually packaged into a facility, such as an object request broker (ORB), a message broker (MOM), or a software bus [Rei90, ISG97, IMA98].[1] Our approach to achieving software multioperability employs *software connectors* [SG96, MT00] and directly leverages ORBs.

Connectors are architecture-level abstractions and facilities that isolate all interaction details in a system and separate them from the functionality. In large, and especially distributed systems, connectors become key determinants of system properties, such as performance, resource utilization, global rates of flow, and security [MMP99]. Encapsulating interaction details within connectors has shown a lot of promise in helping address traditional software development challenges: scalability, distribution, concurrency, runtime adaptability, code mobility, and so forth [SDK+95, SG96, AG97, OMT98, KM98]. We have extensively employed connectors to support software modeling, analysis, generation, evolution, reuse, and heterogeneity [TMA+96, MOT97, OMT98, MRT99]. Our use of connectors in the context of multioperability is based on the recognition that, though different, ORBs and connectors share several key characteristics. Indeed, an ORB can be viewed as an implementation of a sophisticated connector supporting a large set of interaction protocols and services [DMT99]. This perspective allows a software architect to design an application in the most appropriate way, map the architecture to a particular set of middleware-induced styles (recall Section 2), and use the appropriate ORBs to implement its connectors.

[1] In the interest of simplicity, and as is commonly done in literature, we will refer to the different interaction facilities provided by middleware as "ORBs."

An example that illustrates the role of connectors in achieving multioperability is shown in Figure 1. On the left, a conceptual architecture of a system modeled in a particular style, C2 [TMA+96], is shown. Assume we want to implement the architecture with components bound to a given middleware and to distribute the implementation over three locations. The middle diagram in Figure 1 depicts the resulting solution: the single ORB ensures the cross-machine interaction of its attached components, but not the topological and interaction constraints imposed by the style. Also note that, if the four components are not all built on top of the same middleware infrastructure, the engineers will depend on existing point solutions or will have to develop the needed inter-middleware bridge themselves.

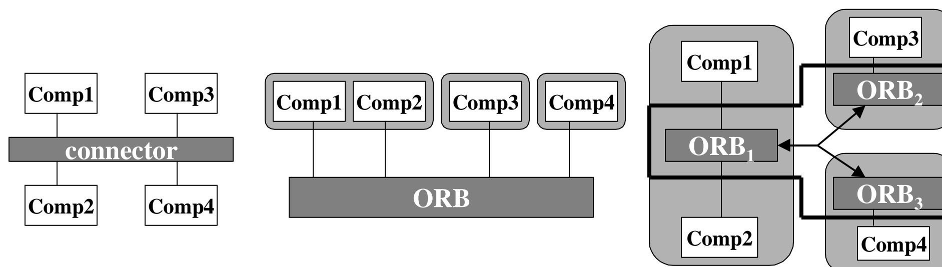

Figure 1. Realizing a software architecture (*left*) using a middleware technology (*middle*) and explicit, middleware-enabled software connectors (*right*).

Our approach, depicted on the right side of Figure 1, enables a more principled way of bridging middleware. We keep connectors an explicit part of a system's implementation infrastructure [MOT97].[2] Each component thus only exchanges information with a connector to which it is attached; in turn, the connector (re)packages that information and delivers it to its recipients. This approach minimizes the effects on a given component of varying application deployment profiles and using components that adhere to heterogeneous middleware standards. Note that, unlike the "middleware-only" solution, the right-hand diagram of Figure 1 also preserves the topological and stylistic constraints of the application.

To date, we have successfully completed a set of feasibility studies, using ORBs from five different middleware technologies to implement connectors that enable the interaction of C2 components built in multiple languages and for multiple platforms [DMT99]. We have also shown that multiple interacting ORBs can be used to implement a single C2 connector. Recently, we have successfully performed a preliminary experiment in which a connector enabled the interaction of C2 and CORBA components [Med00]. The construction of each such middleware-enabled connector has required careful study of the characteristics of the interoperating middleware technologies. However, each connector is constructed only once, after which it is usable indefinitely.

A number of critical issues remain open. To generalize our approach, we must expand our focus to include other (arbitrary) types of software connectors and architectural styles. Thus, we must fully understand the underpinnings of middleware platforms, the assumptions they make, the types of interaction facilities they provide, and the stylistic constraints they impose upon an application. The work outlined in the preceding two sections is a necessary complement to this part of our task. Another key challenge is avoiding pairwise ORB integrations, such that N^2 connectors are required for N middleware technologies. Our proposed theory of middleware interoperability (Section 3) and our on-going work to understand the foundations of software connectors [MMP99] will guide us in choosing the appropriate mechanisms for achieving effective, principled, and repeatable software multioperability solutions.

5. Summary

The work discussed in this paper has the potential to produce an important scientific advance—a general approach to component-based software development that accommodates wide flexibility in the selection and use of underlying interoperability mechanisms. The benefits will accrue from the ability to integrate large software systems of the future in a manner that is less complex and more flexible, makes better use of resources (time, money, personnel), and is based on sound engineering principles. Another benefit is the ability to train students in many aspects of software interoperability, component-based software development, and middleware *concepts* (rather than *products*). Along the way, students will be exposed to the fundamental principles of software architecture and integration, resulting in a workforce that better understands this daunting problem and has the skills necessary to overcome it.

[2] It has been argued in literature that our architecture implementation infrastructure is similar to existing middleware platforms [DR99, YBB99].

6. References

[AAG95] G. Abowd, R. Allen, and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM TOSEM*, 1995.

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

[Box98] D. Box. *Essential COM*. Addison Wesley, 1998.

[DR99] E. Di Nitto and D.S. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. *ICSE'99*, May 1999.

[DMT99] E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. *ICSE'99*, May 1999.

[DRS94] R. Duke, G. Rose, and G. Smith. Object-Z: A Specification Language Advocated for the Description of Standards, Software Verification Research Centre, Dept. of CS, University of Queensland, Technical Report 94-45, 1994.

[FFC+99] D. Flanagan, J. Farley, W. Crawford, and K. Magnusson. *Java Enterprise in a Nutshell*. O'Reilly, 1999.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architecture Mismatch or Why it is So Hard to Build Systems Out of Existing Parts. *ICSE17*, April 1995.

[GPK99] R. Gamble, J. Payton, and S. Kimsen. The Potential for Formally Modeling Integration Strategies. Technical Report UTULSA-MCS-99-21, Dept. MCS, Univ. of Tulsa, September 1999. Submitted for publication.

[GSP99] R. Gamble, P. Stiger, & R. Plant. The Rule-Based Architectural Style and its Application to Contemporary Knowledge-Based Systems. *Journal of Knowledge-Based Systems,* 12:13-26, 1999.

[ISG97] International Systems Group, Inc. Middleware: The Essential Component for Enterprise Client/Server Applications. February 1997.

[KG98] R. Keshav and R. Gamble. Toward a Taxonomy of Integration Strategies. *ISAW-3*, November 1998.

[KG99] A. Kelkar and R. Gamble. Understanding the Architectural Characteristics Behind Middleware Choices. *1st International Conference on Information Reuse and Integration,* November 1999.

[KFGP99] R. Keshav, K. Frasier, R. Gamble, and J. Payton. Architecture-Directed Integration. Technical report UTULSA-MCS-99-25, Dept. Mathematical & Computer Sciences, Univ. of Tulsa, November 1999. Submitted for publication.

[KM98] J. Kramer and J. Magee. Analysing Dynamic Change in Software Architectures: A Case Study. In *Fourth International Conference on Configurable Distributed Systems*, May 1998.

[Med00] N. Medvidovic. On the Role of Middleware in Architecture-Based Software Development. Submitted for publication, January 2000.

[MMP99] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. Submitted for Publication. Technical Report, USC-CSE-99-529, Center for Software Engineering, USC, November 1999.

[MOT97] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. *SSR'97* and *ICSE'97*, Boston, MA, May 1997.

[MR99] N. Medvidovic and D. S. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. *First Working IFIP Conference on Software Architecture (WICSA1)*, February 1999.

[MRT99] N. Medvidovic, D.S. Rosenblum and R.N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. *ICSE'99*, May 1999.

[MT00] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. Accepted for publication in *IEEE TSE*, 2000 (to appear).

[OMT98] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. *ICSE98*, April 1998.

[OMTR98] P. Oreizy, N. Medvidovic, R. Taylor, and D. Rosenblum. Software Architecture and Component Technologies: Bridging the Gap. *OMG-DARPA-MCC Workshop on Compositional Software Architecture*s. January 1998.

[P99] President's Information Technology Advisory Committee. Information Technology Research: Investing in Our Future. National Coordination Office for Computing, Information, and Communication, February 1999.

[PW92] D.E. Perry and A.L. Wolf, Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, October 1992.

[RC99] Rational Corporation. UML Version 1.3. http://www.rational.com/uml/. 1999.

[Rei90] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, July 1990.

[RMRR98] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. *ICSE'98*, April 19-25, 1998.

[SC97] M. Shaw & P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. Proc. COMPSAC97 and *1st Int'l Computer Software & Applications Conf.*, Aug. 1997.

[SDK+95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE TSE*, April 1995.

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

[Sie96] J. Siegel. *CORBA Fundamentals and Programming*. Wiley, 1996.

[SSM97] K. Sullivan, J. Socha, and M. Marchukov. Using Formal Methods to Reason about Architectural Standards. *ICSE'97*, May 1997.

[TMA+96] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE TSE*, June 1996.

[YBB99] D. Yakimovich, J. M. Bieman, and V. R. Basili. Software Architecture Classification for Estimating the Cost of COTS Integration. *ICSE'99*, May 1999.