

Understanding the Architectural Characteristics behind Middleware Choices[†]

A. Kelkar

R.F. Gamble

Department of Mathematical and Computer Sciences

University of Tulsa

600 S. College Ave.

Tulsa, OK 74145

Keywords: Software architecture, interoperability.

ABSTRACT

Integrating heterogeneous components to form a distributed system can manifest difficult interoperability problems among the components. Contributing to this problem is a lack of understanding of the underlying software architecture of the components, and, possibly, the distributed architecture in which they participate. Middleware to resolve these problems may be difficult to choose and implement, often resulting in integration solutions that are not complete or evolvable. This paper discusses software architectural characteristics that underlie choices for integration solutions to interoperability problems in distributed component architectures. The 4+1 view model of architecture is used to represent the characteristics at their appropriate level of abstraction. We use two distinct case studies to illustrate how comparisons among characteristic values of participating components in the integrated system could have predicted interoperability problems.

1. INTRODUCTION

One of the major industrial efforts in software development is to build distributed systems from heterogeneous components. Such components, commonly called sub-systems, include legacy software, commercial-off-the-shelf (COTS) products, and software under design. The integrated system may involve migration of an existing system to new functionality, the linking or merging of multiple existing systems, and/or the additional utilization of COTS. This style of software development can facilitate reusability, adaptability, and evolvability, as applied to the individual components and the integrated system overall.

Interoperability problems among distributed, heterogeneous components poses a high degree of risk. There are many factors that contribute to this. One factor is that the participating components may not be designed to work in the integrated environment. Another factor is that integration is still relatively new to many programmers. This newness contributes to a lack of design decisions when defining the architecture of the

overall integrated system. Therefore, interoperability problems may come as a surprise at implementation when only ad hoc approaches are possible. An additional factor is that “what to look for” during the design is not readily known to make it possible for predicting interoperability problems [10]. This in turn contributes to a lack of understanding as to what middleware to include in the design of the distributed architecture. By the term middleware, we refer to preexisting in-house software, commercially available products, and known integration strategies or design patterns.

Available commercial middleware solutions may require a large amount of coding effort to deploy and possibly still result in an incomplete integration solution. Until recently, most middleware has been focused on addressing interprocess communications within a sub-system [3]. As more complex integrated systems are proposed, the flexibility and interoperability requirements for middleware should receive new focus.

At the highest level of abstraction is the software architecture description. A general problem with integration is a lack of understanding of the underlying software architecture of the components, and, possibly, the distributed architecture in which they participate. This can result in interoperability issues that are not addressed as part of the overall system. Subsequently, middleware may be difficult to choose and implement, often resulting in solutions that are not complete or evolvable. If interoperability problems are predicted at the software architecture level, then the middleware needs can be determined and become directly part of the overall design [10]. In this paper, we examine the aspects of software architectural characteristics, and their relevance to middleware selection.

2. SOFTWARE ARCHITECTURE

Software architecture emerged from the need for design and specification of large and complex software systems. Perry and Wolf [15] define a software architecture as structure of the components of a program or system, their interrelationships, and principles and guidelines governing their design and evolution over time. Shaw and Garlan [17] define architectural styles that

[†] This research is supported in part by the Air Force Office of Scientific Research, (F49620-98-1-0217). All correspondence regarding this paper should be directed to R.F. Gamble at the address above, phone (918) 631-2988, fax (918) 631-3077.

show design and implementation decisions among software systems.

Shaw and Clements [16] provide additional details that further differentiate individual styles, their extensions and specializations, by defining four categories of characteristics embodied in each style: components and connectors, data issues, control issues, and control/data interaction issues. Kazman [9, 8] defines characteristics for evaluation of quality attributes of software architecture.

Kruchten [12] describes the software architecture of a system using 4+1 view model of architecture. Embodied within each view are the architectural abstractions (components, connectors, etc.), the forms and patterns that work for that view, and the rationale and constraints. The view definitions below are taken directly from [12].

The *logical view* describes end-user services using architecturally significant components, their high-level descriptions and interactions. The *development view* focuses on the organization of the system detailing import/export relationships. The *process view* is represented by the decomposition of the system into independent tasks. The *physical view* represents the different configurations of the system for development and testing. Examining the first four views, one finds their representations at different levels of abstraction for the architecture of the system.

Analyzing the construction of Aesop [5], Garlan et al, coined the phrase *architecture mismatch* to describe the underlying reasons for interoperability problems among seemingly “open” software components with available source code. Garlan traces the mismatch problem back to the fundamental characteristics of the architectural styles of the interacting systems. The case study established an important result that knowledge of the software architecture might aid in identifying interoperability problems. Further research applies this result to the design and implementation of integrated system [18, 10, 7, 1].

Dashofy, et al [4] suggest that off-the-shelf middleware fails to address the complete set of interoperability problems. Applying such solutions is not yet well understood when considering the software architecture. There is an inability of middleware to address the architecture level issues in integration. Thus, we need to describe the software architecture in way that relates directly to potential interoperability problems, and guides the developer to a viable middleware selection. The first step in this process is to accumulate a set of relevant architectural characteristics. The next step is to show how the characteristics can predict interoperability problems and why. The remaining steps include relating the problems to high-level solutions and their connection to middleware alternatives [14].

We have documented 76 published characteristics of a system’s software architecture [11]. Relating architectural characteristics to interoperability problems and middleware solutions is not easy. Though Shaw and Clements [16] provide basic categories, there is no relationship defined among the different levels of abstraction represented by the characteristics. Not all characteristics are relevant for interoperability concerns, and it is necessary to narrow the scope to those that offer the most insight into potential problems. Thus, there remains a large gap between architecture integration understanding and appropriate middleware selection.

3. ARCHITECTURE CHARACTERISTICS FOR INTEROPERABILITY

Comparison of the architectural characteristic values can predict potential interoperability problems. These problems occur at different levels of abstraction. Some can be seen at a higher level of architectural description while others require more details. To gain insight into these problems, the architectural characteristic values must be analyzed at the same abstraction level as the conflicts. Currently, the characteristics are given as a flat set that, as a whole, cannot be applied at the single level of abstraction. What is needed is a better representation of the comprehensive software architecture that expresses the different levels of abstraction at which the characteristics reside. The 4+1 view model of architecture provides analogous representation to this [12]. Using this view model we classify the characteristics into the view that most closely corresponds to its abstraction level.

Our initial characteristic set was that given by Shaw and Clements [16]. From this foundation we compared and contrasted characteristics published by other researchers [1, 18, 5, 9, 8]. This process incrementally added characteristics to the foundational set. Employing semantic nets, we examined interdependencies among the characteristics. This allowed us to observe characteristic clusters from which we could chose representatives that encompassed the values of the cluster.

The remaining part of this section presents the characteristics in the 4+1 view. We use the Logical, the Development and the Process views. We do not have characteristics that are part of the Physical view at this time. In each view, we partition the characteristics into *data*, *control* and *system* characteristics to facilitate more explicit and modular resolution of conflicts. Data and control characteristics predict interoperability problems that occur due to transfer of data and control among the system components. The characteristics capture the problems that arise when the components have different ways of dealing with data and control. The system characteristics predict problems caused by different overall system constraints and requirements that must be satisfied during integration.

Characteristic	Definition
Supported data transfer[1]	The method supported by a particular architectural style to achieve data transfer
Data storage method [18]	The details about how data is stored within a system
Data topology [16]	The geometric form the data flow takes in the system.

Table 1: Logical view data characteristics

Characteristic	Definition
Control topology [16]	The geometric form the control flow takes in the system.
Control structure [18]	The structure that governs the execution in the system.

Table 2: Logical view control characteristics

Characteristic	Definition
Distribution [1]	How the functional requirements are divided among the components in the system
Identity of components [18]	Knowledge or awareness of other components in the system
Naming [2]	The components in the system are uniquely named

Table 3: Logical view system characteristics

3.1 Logical View Characteristics

The characteristics whose values relate to the logical view are defined in Tables 1, 2, & 3. These characteristics describe the functionality and decomposition of the components as black boxes, without incorporating the exact details of implementation.

3.2 Development View Characteristics

Because of the development view's focus on component organization, distribution, programming language constraints, and import/export relationships, the characteristics classified within the development view

focus on many of the data and control scope issues. Tables 4, 5, and 6 define these characteristics.

3.3 Process View Characteristics

The characteristics related to the process view are more implementation related, and, thus, are lower level characteristics. They express the run time issues in the integrated system. Thus, many of the characteristics focus on how components execute internally and how directly communicate with other components [ref Tables 7,8,9]. Method of communication is relevant to passing of data and control and thus appears in both the tables.

Characteristic	Definition
Data mode [16]	The way data is made available throughout the system
Data scope [9]	Where in the system a particular set of data can be used
Data flow [2]	How the data flows between the system components.

Table 4: Development view data characteristics

Characteristic	Definition
Control scope [9]	Where in the system the control can reach
Control flow [2]	How the control flows between system components

Table 5: Development view control characteristics

Characteristic	Definition
Isomorphic control and data shapes [16]	If the data topologies and control topologies are isomorphic
Encapsulation [1]	The presence of embedded components in the system that can only be accessed by a well-defined interface.
Configuration [2]	The structural arrangement and the complete functional layout of the system
Directionality [16]	The relationship between the direction of data flow and the direction of control flow

Table 6: Development view system characteristics

Characteristic	Definition
Data representation [18]	The data structures and the ontology used by the system
Data binding time [16]	The time when the identity of a partner in the transfer of a data is established
Continuity [16]	The continuous nature of the data flow through the system.
Time of data acceptance[9]	The time when multiple components accept data simultaneously.
Method of Communication^[18]	The methods by which the components provide the means to communicate.

Table 7: Process view data characteristics

Characteristic	Definition
Trigger [1]	An action associated with the reception of data component by control component
Concurrency [18]	The constraint on the number of concurrent threads that may execute within a system
Invocation [18]	How the components call each other.
Synchronicity [16]	The dependencies of component actions upon each others' control states
Control binding time [16]	The time when the identity of a partner in the transfer of control operation is established
Time of control acceptance [9]	The time when multiple modules accept control simultaneously.
Method of Communication*[18]	The methods by which the components provide the means to communicate.

Table 8: Process view control characteristics

Characteristic	Definition
Protocol [18]	The mechanism of sending the messages that must follow a particular message structure.
Parameter passing [18]	How parameters are passed between the modules during the transfer of data

Table 9: Process view system characteristics

4. CASE STUDIES ANALYSIS

In this section, we present two published case studies that implement middleware solutions to integrate heterogeneous, distributed components. Notkins, et al [13] discusses the implementation of a heterogeneous computer system. This system implements a Face-Finger distributed service (ffinger). The system was developed so that the clients residing on different machines can use the ffinger system on the server. The server for the ffinger is implemented on a UNIX operating system using RPC. The clients who call the server reside on workstations including VAX, XEROX, and SUN machines. Clients use their own RPC facility with the server using the native RPC facility. Some of the interoperability problems faced during the implementation are as follows.

- The RPC facilities of the clients are distinct.
- Because the individual clients are named, naming conflicts occur in the integrated environment.
- Conflicts occur as some components are aware of other components at run time, while others are bound at compile time.

- The clients and the server use different data formats and data representations so the conflicts occur during the data transfer.

A priori knowledge of the characteristics listed in Table 10 would have provided information toward predicting the above problems, which, in turn, may have facilitated the construction of the solution.

Notkins, et al [13] implements the following middleware solutions.

- The HRPC (Heterogeneous RPC) system is used as the underlying communication facility provided for all the clients. It implements a single communication protocol in the integrated system.
- The HRPC system solves the binding time conflicts by delaying the binding of the clients and server to run time.
- The HNS (Heterogeneous Name Service) creates a global name space to resolve naming conflicts.
- The HNS also resolves the syntax and semantic issues by using the global name space and by providing additional mappings to attain consistency.

Characteristic	Involvement
Protocol [18]	Different protocols among the components predict the need for a middleware solution that makes communication transparent.
Data binding time Control binding time [16,17]	Modules bound at run time cannot communicate with the modules bound at compile time. Due to such differences, the identity of one of the modules in the transfer of control or data may not be known.
Naming [18]	A middleware solution is needed to resolve duplicate names, as well as the situation when one component using names and another does not.
Syntax and semantics [18]	Analysis of syntax and semantic values of the individual systems indicate problems due to lack of syntactic and semantic consistency.

Table 10: Case Study #1 Interoperability Analysis

Gruhn & Wellen [6] analyze their software migration effort done in telecommunications. In contrast to the first case study, Gruhn and Wellen informally consider architecture properties when defining their process of software migration. Therefore, more characteristics are directly observable than in the integration effort described by Notkin, et al [13]. The telecommunications project involves developing the migration path towards an integrated software architecture starting from independent, heterogeneous software modules. Their definitions are taken directly from [6].

A SAP/R3 module manages both finance and control operations. The SAP/R3 module structure and its relationships with the other modules are not changeable. SAM is the administrative tool for customer master data. It is a client server architecture and uses ORACLE as the RDBMS. The data access is carried out with the SQL-net. PRIS is a provisioning system for external sales partners. It is also a client server system based on a 3-tier architecture. CURT is an application for providing reports and statistical analysis results. BICOS is a billing system

that has an interface with PRIS. All the systems have their own data repository.

The integration involved the following problems.

- Each module had shared data, but due to differences in their data transfer methods, data integration was problematic.
- Each module has a different data representation.
- Due to differences in the data flow in the individual modules, data integration created problems.
- Due to differences in the control flow of the system, establishing a single control path through the distributed system was problematic.
- Conflicts also occur due to differences in the methods by which the modules communicate.

Analysis of the architectural characteristics given In Table 11 shows how these problems could have been predicted. In the actual implementation of the integrated system, the problems mentioned above are resolved through progressive migration phases that involve data and control integration and the implementation of the underlying system. These phases are as follows.

Characteristic	Involvement
Supported data transfer [1]	Because the modules have different styles to support data transfer, there are conflicts during this operation.
Data representation [17]	There are conflicts because the data representation for each module is different.
Data topology [16, 17]	Data topology gives the shape of the flow of data within each module. Because the data flows differently within each module, conflicts arise during data integration.
Control flow [2]	During control integration the problems can occur due to differences in the flow of control throughout the system.
Control scope [9]	As the system requires a singular flow of control, conflicts can occur if there are constraints due to whether control can reach particular modules.
Protocol [18]	Examining the modules and their protocol values indicates potential communication conflicts during the transfer of data and control between modules.

Table 11: Case Study #2 Interoperability Analysis

Data exchange support: The customer master data was made available to all modules using a global data exchange scheme. Also, the differences in the data formats were resolved by translating the data formats into an intermediate data format.

Data integration: Decomposition of the system architecture led to identifying the common components, and delegating the responsibility of shared data to them to perform data integration.

Control integration: In this case, control integration needed a singular path of control in the distributed system to carry out the required functionality. A separate, independent component that controlled the overall functionality of the system modules solved this problem.

Implementation by distributed modules: The final step in the integration was implementing the underlying transport mechanism to allow the distributed systems to communicate. CORBA was implemented due to the component based integration.

CORBA offered much of the required functionality and resolved most of the problems within the fourth phase. However, Gruhn and Wellen [6] point out that CORBA required a lot of coding effort within the PRIS system for its distributed database and SQL net protocol. In addition, separate integration schemes were needed within the three previous phases for data and control integration and, also, data exchange.

5. CONCLUSION

In this paper, we have isolated architectural characteristics that can predict potential interoperability problems in a distributed component architecture. Once isolated, they can be classified accordingly at an appropriate level of abstraction to facilitate analysis. These characteristics may have values that are derived from the architectural style, if it is known. However, it is not necessary to know all characteristic values to predict interoperability problems in the integration of heterogeneous components. We are not claiming that this is a complete set of characteristics, as further research may indicate additional ones.

It can be seen from the case studies described that architectural characteristics directly underlie the interoperability problems experienced. Thus, access to these characteristics and their relationship to interoperability problems is important because understanding the foundation for choosing middleware allows for determining its completeness and evolvability.

REFERENCES

- [1] A. Abd-Allah, Composing heterogeneous software architectures, Ph.D. Diss. Dep. of CS, USC, 1996.
- [2] R. Allen & D. Garlan. A formal basis for architectural connection, *ACM TOSEM*, 1997.
- [3] J. Charles, Middleware moves to the forefront, *IEEE Computer*, May 1999.
- [4] E. Dashofy, N. Medvidovic & R. Taylor. Using OTS middleware to implement connectors in distributed software architectures. *Proc. ICSE-99*, 1999.
- [5] D. Garlan, R. Allen, & J. Ockerbloom, Architectural mismatch or why it is so hard to build systems out of existing parts, *Proc. ICSE-95*, 1995.
- [6] V.Gruhn & U.Wellen, Integration of heterogeneous software architectures. *First Working IFIP Conf. on Software Architecture*. Feb. 1999.
- [7] K. Hasler, R.Gamble, K. Frasier, & P. Stiger. Exploiting inheritance in modeling architecture abstractions, position paper for *1st Working IFIP Conf. on Software Architecture*, Feb. 1999.
- [8] R.Kazman & S.J.Carriere View extraction and view fusion in architectural understanding, *Proc. ICSR5*, June 1998.
- [9] R.Kazman, P.Clements, L.Bass & G.Abowd Classifying architectural elements as foundation for mechanism mismatching, *Proc. COMPSAC*, 1997.
- [10] R. Keshav & R. Gamble. Towards a taxonomy of architecture integration strategies, *3rd International Software Architecture Workshop*. Nov. 1998.
- [11] A. Kelkar & R. Gamble, Analysis of architectural characteristics to predict interoperability problems, Univ. of Tulsa, TR-99-15, Dept. Math/CS, 1999.
- [12] P. Kruchten, The 4+1 View Model of Architecture, *IEEE Computer*, Nov. 1995.
- [13] D. Notkins, A.Black, E.Lazowska, H.Levy, J.Sanislo & J.Zahorjan Interconnecting Heterogeneous Computer Systems *CACM*, 31:8, 1988.
- [14] J. Payton, R. Keshav, & R. Gamble. System development using the integrating component architectures process, *ICSE 99 Workshop on Successful COTS Development*, May 1999.
- [15] D. Perry & A. Wolf. Foundations for the study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4): 40-52, Oct.1992.
- [16] Mary Shaw & Paul Clements. A field guide to boxology, *Proc. COMPSAC*, 1997.
- [17] M. Shaw & D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
- [18] R. Sitaraman. Integration of software systems at an abstract architectural level, M.S. Thesis, Department of Mathematical and Computer Sciences, University of Tulsa, 1997.