

A Notation for Problematic Architecture Interactions

L. Davis R. Gamble J. Payton G. Jónsdóttir D. Underwood

Department of Mathematical and Computer Sciences

University of Tulsa

600 South College Avenue

Tulsa, OK 74104 USA

+1 918 631 2988

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

ABSTRACT

The progression of component-based software engineering (CBSE) is essential to the rapid, cost-effective development of complex software systems. Given the choice of well-tested components, CBSE affords reusability and increases reliability. However, applications developed according to this practice can often suffer from difficult maintenance and control, problems that stem from improper or inadequate integration solutions. Avoiding such unfortunate results requires knowledge of what causes the interoperability problems in the first place. The time for this assessment is during application design.

In this paper, we define *problematic architecture interactions* using a simple notation with extendable properties. Furthermore, we delineate a multi-phase process for pre-integration analysis that relies on this notation. Through this effort, potential problematic architecture interactions can be illuminated and used to form the initial requirements of an integration architecture.

Keywords

Software architecture, interoperability, architecture analysis

1. INTRODUCTION

As software ages and new paradigms for computing emerge, the complex reuse, sharing and integration of information is becoming a paramount concern. Recent methodologies assist in defining the requirements of the integrated application and deciding how the accumulated components will interact [15]. While they have advanced component-based software engineering (CBSE), there is no clear path from an interoperability problem to its solution. Methodologies for interoperability assessment that do exist either are not complete (e.g., [26]), are too

focused (e.g., [3,20,32]), or lack specificity (e.g., [5]), to have wide-ranging usefulness and offer guidance to resolve known defects.

Decisions for using middleware – distributed computing services that facilitate interoperation between components – are often made in an ad hoc manner, without sufficient understanding of the initial systems chosen or the enterprise application into which they will be composed. This is not to say that middleware products are not useful or are always used improperly. In fact, middleware is being used exactly as intended. The problem is that middleware products are built to address the effects of interoperability problems on component integration, but not their causes. Unfortunately, without knowledge of the causes, application maintenance and evolution can lead to chaos. Interim steps are needed to first assess component interaction during application design and then to uncover problems that will determine the essential integration functionality, how to manage it, and how to effectively code it.

We must, therefore, return to basic software engineering principles in which the design is analyzed and understood before determining the best direction for implementation. This type of *pre-integration* analysis of abstract architecture information can illuminate potential interaction problems in an integration. Emerging research in the area of software architecture offers a coarse-grained description of a software system. Moreover, software architecture styles have been directly implicated in interoperability conflicts [1,12,32,33]. Still, there is no accepted standard for software architecture representation. In this paper, we describe a process to determine *problematic architecture interactions*, defined below.

Definition 1. Problematic architecture interaction (PAI). An interoperability conflict that is predicted through the comparison of architecture interaction characteristics and requires intervention via external services for its resolution.

Additionally, our multi-phase *pre-integration* analysis process utilizes a simple notation with extendable properties to describe the problematic interactions manifested in the integration. Our work illustrates how effective assessment can be performed very early in the

application design with minimal specifications. Though the process is constructed with conflict resolution guidance in mind, this topic is beyond the scope of this paper.

This paper includes the following sections. Section 2 presents research that is directly relevant to this paper. In Section 3, we define the characteristics used for analysis. Section 4 presents the notation and its properties such as union and additivity. Section 5 exemplifies the use of the notation in our multi-phase process using a small but relevant integration problem. Finally, Section 6 concludes with results and future work.

2. RELATED RESEARCH

The *software architecture* of a system is a high-level description of its computational elements, the means by which they interact, and the structural constraints on interaction [27,29]. With such abstraction, it is possible to initially focus on important conceptual system issues without becoming entangled in implementation and deployment details. One facet of this description is architectural style, which provides information regarding configuration and coordination constraints on a system's components or modules. Characteristics are defined for architectural styles to delineate different configurations, such as pipe and filter systems, event-based systems, and main/subroutine systems.

The phrase *architecture mismatch* was coined to describe certain underlying reasons for interoperability problems among seemingly "open" software components [12]. On this basis, characteristics have been viewed with respect to their potential impact on interoperability. However, only subsets based on style constraints have been examined for their role in integration issues [1,3,4,10,12,28,30].

To resolve interoperability problems, an *integration architecture* must be constructed. Integration architectures are composed of multiple connectors. Connectors have become increasingly important in software architecture analysis, having been raised to first-class status in many descriptions [11,18,21,23,31]. As their potential for contribution to analysis has been recognized, attempts to capture and describe the functionality of connectors are ongoing. Explicit descriptions of connectors are desirable, as these can allow for design choices among and analysis of existing interaction schemes, plus the specification of new connectors [11]. Formal descriptions of connectors have been expressed in several architectural description languages (ADLs) [3,13,19,20,22], as well as in the Z and Object-Z notations [2,25,31].

Unfortunately, connectors suffer from similar problems in characterization as components and architectural styles. There are many types of connectors that range from generic to complex, with variants at every level. The impetus behind connector taxonomy research is to allow

designers flexibility in choosing the correct interoperability schemes in an integrated application [3,11]. This research suffers from a considerable amount of detail, making the classifications very cumbersome. Currently, large taxonomies are being constructed (e.g., [23]), for the purpose of understanding and analyzing a wide range of connectors and to promote the formation of unique connectors. A taxonomy of *integration elements* (architectural connectors specific to resolving interoperability conflicts), links connectors to functions found in design patterns describing integration techniques. By restricting the connector taxonomy in this way, defining solutions to problematic architecture interactions is simplified.

The interim steps between determining how components should interoperate and detecting the resulting problems are best formulated into a process. Recent approaches to performing such analysis can be divided into pre-integration assessment, like the process presented in this paper, versus post-integration assessment. Approaches close to our own include [16,32], in which characteristics are evaluated to provide an assessment of the architecture. However, the differences between our approach and these others include the inability to compare simple architecture styles in [32], and the limitation of evaluation assessment to only nonfunctional requirements in [16]. Post-integration analysis can offer additional insight into the prescribed integration architecture, as with the chemical abstract machine (CHAM) formalism discussed in [14] and with ADLs such as Wright [3]. While these approaches aid in the assessment of integration solutions, they do not provide a means to assess interoperability a priori and to direct the selection of an initial solution that is appropriate.

3. SOFTWARE ARCHITECTURE CHARACTERISTICS

In this section, we briefly present the characteristics that we use throughout the remainder of the paper. Characteristics defined with respect to architectural styles include those that describe the various types of components and connectors, data issues, control issues, and control/data interaction issues [1,3,4,10,12,28,30]. These architectural characteristics provide details that further differentiate individual styles, their extensions and specializations.

Our previous research suggests that it is feasible and desirable in analysis to partition architectural characteristics across two viewpoints: component-level and application-level [7,9,17]. *Component-level characteristics* contribute to an understanding of the exposed interface of a component to other external subsystems. In turn, *application-level characteristics* address architectural demands on configuration and coordination of the component systems into a single integrated application to satisfy overall requirements.

After extensive review of published characteristics, coupled with empirical analysis, we previously define abstraction levels and use semantic networks to identify the high-level characteristics, presented in Table 1 [7,8,9,17]. Their abbreviated references are in column one, with the name in column two. They are distinguished by the association of the characteristic with a component (C) or with the overriding application (A) in column three. Definitions and values are in columns four and five, respectively.

These nine style-related characteristics are course-grained, reflecting functionality present at lower levels of abstraction. That is, each of these high-level characteristics has multiple semantic links to corresponding low-level characteristics. We do not claim that the set is complete; more characteristics may be introduced with further research.

4. PROBLEMATIC ARCHITECTURE INTERACTIONS

Not all interactions are problematic – some require only a simple connector to facilitate communication. For instance, two UNIX filters that require a typical streaming pipe for communication would not exhibit a PAI. PAIs, in contrast, require additional functionality and processing, beyond simple connectors, to facilitate communication

between components. Incompatible data or control assumptions tend to result in PAIs. It consequently becomes difficult to assign a ready-made connection, as this is not sufficient to negotiate, translate, or buffer the control and/or data necessary in the communication. Thus, PAIs necessitate the use of one or more *integration elements* (controller, translator, extender) packaged into an integration architecture [18,24].

4.1 Defining the Notation

We base our approach on the assumptions that problematic architecture interactions 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 application expectations do not comply with the integration solution requirements. (application-integration)
4. The configuration and coordination requirements of the application are themselves in conflict (application-application).

Table 1: Application and Component Level Characteristics

ABBRV	CHARACTERISTICS	TYPE	DEFINITION	VALUES
Bk	<i>Blocking</i>	C	Whether or not the thread of control is suspended [17]	Blocking, Non-Blocking
CS	<i>Control Structure</i>	A, C	The structure that governs the execution in the system [30]	Single-Thread, Multi-Thread, Decentralized, Concurrent, Sequential
CT	<i>Control Topology</i>	A, C	The geometric form control flow takes in a system [28]	Hierarchical, Star, Arbitrary, Linear, Fixed
DSM	<i>Data Storage Method</i>	C	How data is stored within a system [30]	Repository, Data With Events, Local Data, Global Source, Hidden, Distributed, Stream
DT	<i>Data Topology</i>	A, C	The geometric form data flow takes in a system [28]	Hierarchical, Star, Arbitrary, Linear, Fixed
IC	<i>Identity of Components</i>	C	Awareness of other components in the system [30]	Aware, Unaware
SCT	<i>Supported Control Transfer</i>	C	The method supported to achieve control transfer [1]	Explicit, Implicit
SDT	<i>Supported Data Transfer</i>	C	The method supported to achieve data transfer [1]	Explicit, Implicit, Shared
Sn	<i>Synchronization</i>	A	Whether or not the components need to rendezvous [17,32]	Synchronous, Asynchronous

Each of these four interaction assumptions require distinct characteristic comparisons that potentially result in different problem categories. Thus, a multi-phase process of analysis in which each phase considers one interaction assumption is a natural approach. In this paper, we focus on the first interaction assumption and briefly discuss the impacts of the second and third.

What makes the characteristic set in Table 1 essential for interoperability analysis is the encompassing nature of the definitions. Subsequently, through case study analysis and expert knowledge, we derive a set of PAIs for initial study. We do not maintain that the PAIs, which result from characteristic value comparison, are definitive. They offer a degree of interoperability prediction for a first-pass assessment.

There are several interesting things to note about PAIs that distinguish our research from others in the area. First, conflicts can occur between like values of characteristics. This is in contrast to the notion that only mismatched values lead to conflict [1,10,12]. Secondly, in contrast to only comparing similar characteristics types (control topology vs. control topology), analysis across different types of characteristics may predict a conflict between components [1,10]. This cross comparison goes beyond other research attempts to establish a partial order of the values of a characteristic type to determine the resulting value in the application that resolves the conflict [32,33].

It is important to note that identifying PAIs provides the developer with a “heads-up” about potential conflicts, even in the case of black-box component assessment. Further investigation may be warranted to pinpoint more functional detail within the individual interactions.

4.2 Problem Categories

Repeated interoperability problems appear in one of three categories: control transfer, data transfer, and interaction initialization. Using prose to describe the problems better reflects the course-grained nature of the characteristics whose comparison produces them. Thus, the problem types embody a level of abstraction parallel to a software architecture description.

Category 1: Control Transfer

1. Restricted points of control transfer: *control must be passed to particular interface points*
2. Unspecified control destination: *there is no specified interface point to which control may be passed*
3. Inhibited rendezvous: *dissimilar communication protocols prohibit handshaking*
4. Multiple, unsequenced control transfers: *processes/threads attempt concurrent control communication with a component*

Category 2: Data Transfer

5. Restricted points of data transfer: *data must be passed to particular interface points*
6. Unspecified data destination: *there is no specified interface point to which data may be passed*
7. Unspecified data location: *the data location is obscured*
8. Inconsistent data: *data formats are incompatible*
9. Invalid data: *data is not processed in the expected manner*
10. Multiple, unsequenced data transfers: *processes/threads attempt concurrent data communication with a component*
11. Mismatched data transfer assumptions: *data is required directly/indirectly and the other components expect the opposite*

Category 3: Interaction Initialization

12. Uninitialized control transfer: *control of a participating component cannot be forwarded*
13. Uninitialized data transfer: *data of a participating component cannot be forwarded*

These thirteen conflicts form the current set of PAIs identifiable through architecture characteristic comparison. For a better understanding of their use, refer to the following notated example of an interoperability problem.

$$CS.Single-Thread(A) \rightarrow \{3\} \leftarrow Bk.Non-Blocking(B)$$

which means component A has a single-threaded control structure (CS) that problematically interacts with component B , which is non-blocking (Bk), due to an inhibited rendezvous (3). A single-threaded component requires completion of execution before control can be transferred, but a component that does not block can continue to execute. Hence, the problem exists because both components might not be ready or willing to interact at the expected time.

4.3 Notation and Associated Rules

We explain the notation further, along with its associated rules, using the following definitions. We assume universal quantification when no explicit statement is made.

Definition 2. **AC** is the set of architectural characteristics found in Table 1. Let C_i be a characteristic such that $C_i \in \mathbf{AC}$. Given $S(C_i)$ is its value set and s_r is a value, $s_r \in S(C_i)$ if and only if $C_i.s_r$ is a valid characteristic/value pair. We union all possible pairs into the set **CVPairs**.

Definition 3. Let **Labels** be the set of names for the components composed to form the application. If $Q \in \mathbf{Labels}$, then $C_{i,s_r}(Q)$ means that in component Q , the value for characteristic C_i is s_r . For now, we assume that each component has one characteristic/value pair per characteristic, i.e.,

$$C_{i,s_r}(Q) \wedge C_{i,s_t}(Q) \Leftrightarrow s_r = s_t$$

Definition 4. Let **PAI** be the set of problematic architecture interactions (currently the 13 described in Section 4.2). Given that $T \subseteq \mathbf{PAI}$, $\{C_{i,s_r}, C_{j,s_t}\} \subseteq \mathbf{CVPairs}$, and $\{Q, R\} \subseteq \mathbf{Labels}$ such that $Q \neq R$,

$$C_{i,s_r}(Q) \rightarrow T \leftarrow C_{j,s_t}(R)$$

means that component Q with characteristic/value pair C_{i,s_r} problematically interacts with component R having the characteristic/value pair C_{j,s_t} causing all of the conflicts that appear in the set T . It should be noted that i can be equal to j , i.e., they can be like characteristics.

We use this notation to express the presence of a PAI. For instance, assume that $\{Q, R, A, B\} \subseteq \mathbf{Labels}$, such that $Q \neq R$ and $A \neq B$, then interoperability problems between these components would be denoted as

$$DT.Arbbitrary(Q) \rightarrow \{6\} \leftarrow DT.Arbbitrary(R)$$

and

$$CS.Decentralized(A) \rightarrow \{1\} \leftarrow CT.Hierarchical(B)$$

$$CS.Decentralized(A) \rightarrow \{4\} \leftarrow CT.Hierarchical(B)$$

This notation provides a common vocabulary with which to discuss the interactions, and can lead to a better understanding of the problem. Furthermore, in the future the PAI notation from Section 4.2 can be extended to show the existence of a direct link from a problem to solution, and to catalog these discoveries. For instance, should a set of interactions be grouped under *restricted points of control transfer* and they all require a controller, it can be surmised that any other interaction causing this conflict will as well. Further research may reveal the need for more distinct categorization of conflicts to clarify the solutions that can result.

4.4 Relations on Problematic Interactions

By definition of a problematic architecture interaction,

$$\rightarrow T \leftarrow$$

is symmetric for all characteristic/value pairs. Reflexivity and transitivity, however, do not hold for all pairs. Further research is needed to constrain properties such that transitivity holds between and across components.

We introduce two composition rules, union and additivity. We define these next and discuss their usefulness for assessment.

Definition 5. Union Rule for Problematic Architecture Interactions. Given that $T \subseteq \mathbf{PAI}$, $V \subseteq \mathbf{PAI}$, $\{C_{i,s_r}, C_{j,s_t}\} \subseteq \mathbf{CVPairs}$, and $\{Q, R\} \subseteq \mathbf{Labels}$ such that $Q \neq R$

$$\begin{aligned} C_{i,s_r}(Q) \rightarrow T \leftarrow C_{j,s_t}(R) \wedge \\ C_{i,s_r}(Q) \rightarrow V \leftarrow C_{j,s_t}(R) \\ \Leftrightarrow C_{i,s_r}(Q) \rightarrow T \cup V \leftarrow C_{j,s_t}(R) \end{aligned}$$

This means that conflicts can be combined across similar characteristics. By gathering the conflicts in this manner, they can be assessed as a unit, potentially leading to a single solution. Without union, it would appear that there are multiple unrelated conflicts between the same characteristic comparisons. Therefore, the rule leads to a smaller conflict set and may point to relationships among conflicts.

Definition 6. Additivity Rule for Problematic Architecture Interactions. Given that $T \subseteq \mathbf{PAI}$, $\{C_{i,s_r}, C_{j,s_t}, C_{k,s_v}\} \subseteq \mathbf{CVPairs}$ such that $j \neq k$ (i.e., two different characteristics), and $\{Q, R\} \subseteq \mathbf{Labels}$ such that $Q \neq R$,

$$\begin{aligned} C_{i,s_r}(Q) \rightarrow T \leftarrow C_{j,s_t}(R) \wedge \\ C_{i,s_r}(Q) \rightarrow T \leftarrow C_{k,s_v}(R) \\ \Leftrightarrow C_{i,s_r}(Q) \rightarrow T \leftarrow C_{j,s_t}(R), C_{k,s_v}(R) \end{aligned}$$

Let **ACC** be an accumulation of characteristic/value pairs separated by commas, using the same additivity rule with C_{n,s_w} as an element of **CVPairs**, but not in **ACC**,

$$\begin{aligned} C_{i,s_r}(Q) \rightarrow T \leftarrow \mathbf{ACC} \wedge \\ C_{i,s_r}(Q) \rightarrow T \leftarrow C_{n,s_w}(R) \\ \Leftrightarrow C_{i,s_r}(Q) \rightarrow T \leftarrow \mathbf{ACC}, C_{n,s_w}(R) \end{aligned}$$

There is no limit to the number of characteristics that can be accumulated to one side of the relation (e.g., the right-hand side above). However, only a single characteristic/value pair must remain on the other side (e.g., the left-hand side above).

The significance of this rule is that in general, given the same conflict across multiple characteristics, there is a sense of importance associated with this conflict. Moreover, there is the potential for focusing on a single solution for the conflicts across these related characteristics. This could lead to a smaller solution set.

5. USING THE NOTATION

This section describes our multi-phase process for pre-integration. We address the interaction assumptions 1 through 3 described in Section 4.1. The majority of the section covers the first interaction assumption, as is consistent with our example. We employ the characteristics from Table 1 and the notation described in Section 4.

5.1 The Compressing Proxy

The Compressing Proxy is an HTTP server that dynamically compresses and uncompresses data sent across a network, with the goal of improving web browser performance [6,14]. The Compressing Proxy is constructed by integrating gzip into the W3C HTTP daemon (CERN server). The CERN server is a generic, full-featured hypertext server, which can be used as a regular HTTP server. According to the HTML documents distributed with the source code, the CERN server processes data by passing it to a stack of “stream objects.” Processing, such as filtering or parsing, can be performed within each stream. The stream objects contain pointers to interface functions and a pointer to the next stream object on the stack. Data is passed into the stream as a parameter to the functions. Previously, these stream objects have been referred to as filters [6,14], and we will continue to use this terminology.

Gzip is a file compression utility that is commonly used as a filter at the UNIX process level. It communicates through UNIX pipes, thus performing incremental reads and writes. To build the Compressing Proxy, the gzip utility is inserted into the CERN server stream stack at the appropriate point (see Figure 1).

Inverardi et al. [6,14], use the Compressing Proxy to demonstrate the correctness of a post-integration interoperability analysis method based on the CHAM (CHemical Abstract Machine) formalism. Their primary goal is to examine the behavioral expectations among integrated components to detect deadlock. By applying CHAM to the Compressing Proxy, they are able to correctly identify a deadlock problem caused by an unsophisticated adaptor. CHAM is then applied to the Compressing Proxy with a modified adaptor to prove that it is free of deadlock.

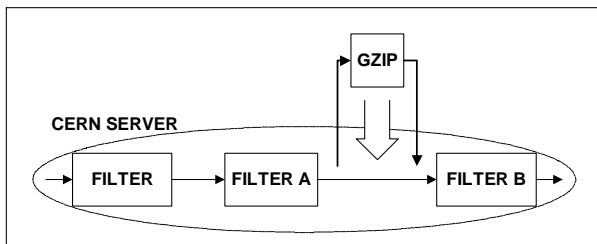


Figure 1: The Compressing Proxy

5.2 Phase 1: Component-Component Interactions

This phase is divided into three steps to examine the direct interaction among components.

Step 1.1 Determine the characteristic/value sets for each component. We begin by assigning a value to the known characteristics for each component participating in the integration. When multiple values exist for a characteristic, the most restrictive value is chosen depending on the architecture information available.

Since the filters in the CERN server are architecturally equivalent, filters A and B in Figure 1 have identical characteristic values. According to the source code, a filter is defined as a *struct* with a pointer to a stream-class object (which contains pointers to the interface functions), a pointer to a sink stream (the downstream filter), and several other stream-specific members. This *struct* reveals much about the filter’s architecture. A filter passes control by invoking an interface function of its sink and passing data as a parameter. The control structure is single-threaded. A function call explicitly transfers control and data to be stored locally. Because of the limited functionality of each filter, the data and control topologies are linear. A filter knows the identity of the downstream filter to allow an interface function to be invoked, and then blocks until the function completes.

According to its code and documentation, gzip uses a one-pass algorithm and is implemented in a single thread. This indicates linear control and data topologies. The fact that gzip is a UNIX-level filter and communicates via bounded pipes yields the values of several characteristics. First, it must block when reading from an empty pipe or writing to a full pipe. It does not know the identity of its upstream or downstream filter [29], thus it is unaware and supports implicit control and data transfer. Finally, its data storage method is a stream [30].

The Compressing Proxy has its own application characteristic/value set for a limited number of characteristics. Because gzip is being “inserted,” it is natural to assume the application values will mimic those of the CERN server. However, gzip cannot be run in the same process as the CERN server. Therefore, a typical assignment for an initial application control structure is sequential. The architecture of the CERN server is asynchronous [28].

Table 2 summarizes the component and application characteristic values as we have just identified.

Table 2: Characteristic Values

CHAR	FILTER A	GZIP	FILTER B	INITIAL APP
Bk	Blocking	Blocking	Blocking	N/A
CS	Single-Thread	Single-Thread	Single-Thread	Sequential
CT	Linear	Linear	Linear	Linear
DSM	Local	Stream	Local	N/A
DT	Linear	Linear	Linear	Linear
IC	Aware	Unaware	Aware	N/A
SCT	Explicit	Implicit	Explicit	N/A
SDT	Explicit	Implicit	Explicit	N/A
Sn	N/A	N/A	N/A	Asynchronous

Step 1.2 Notate the PAIs using bipartite graphs. We first construct a bipartite conflict graph (BCG) for pairwise assessment of components. Bipartite graphs allow full cross-comparison of characteristics. Their use eliminates any interaction assessment among a component's own characteristics. There should be a comparison for every pair of components, i.e., A-B, A-gzip, gzip-B. In phase two (Section 5.3), the graph for A-B would be eliminated because A and B do not directly communicate in the application. For brevity, we exclude it here. Figure 2 displays only those lines in the BCG that represent potential problematic interactions.

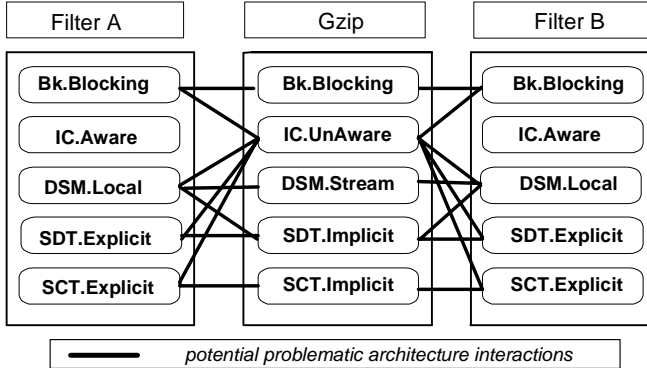


Figure 2: Bipartite Conflict Graphs

We notate the potential PAIs referring to the set of conflicts defined in Section 4.2, where A and B refer to the filters and G refers to gzip.

1. $Bk.Blocking(A) \rightarrow \{3\} \leftarrow Bk.Blocking(G)$
2. $Bk.Blocking(A) \rightarrow \{12\} \leftarrow IC.Unaware(G)$
3. $Bk.Blocking(A) \rightarrow \{13\} \leftarrow IC.Unaware(G)$
4. $DSM.Local(A) \rightarrow \{11\} \leftarrow SDT.Implicit(G)$
5. $DSM.Local(A) \rightarrow \{11\} \leftarrow DSM.Stream(G)$
6. $DSM.Local(A) \rightarrow \{7\} \leftarrow IC.Unaware(G)$
7. $SCT.Explicit(A) \rightarrow \{12\} \leftarrow SCT.Implicit(G)$
8. $SCT.Explicit(A) \rightarrow \{2\} \leftarrow IC.Unaware(G)$
9. $SDT.Explicit(A) \rightarrow \{13\} \leftarrow SDT.Implicit(G)$
10. $SDT.Explicit(A) \rightarrow \{6\} \leftarrow IC.Unaware(G)$

11. $Bk.Blocking(G) \rightarrow \{3\} \leftarrow Bk.Blocking(B)$
12. $IC.Unaware(G) \rightarrow \{12\} \leftarrow Bk.Blocking(B)$
13. $IC.Unaware(G) \rightarrow \{13\} \leftarrow Bk.Blocking(B)$
14. $IC.Unaware(G) \rightarrow \{2\} \leftarrow SCT.Explicit(B)$
15. $IC.Unaware(G) \rightarrow \{6\} \leftarrow SDT.Explicit(B)$
16. $IC.Unaware(G) \rightarrow \{7\} \leftarrow DSM.Local(B)$
17. $SDT.Implicit(G) \rightarrow \{11\} \leftarrow DSM.Local(B)$
18. $DSM.Stream(G) \rightarrow \{11\} \leftarrow DSM.Local(B)$
19. $SCT.Implicit(G) \rightarrow \{12\} \leftarrow SCT.Explicit(B)$
20. $SDT.Implicit(G) \rightarrow \{13\} \leftarrow SDT.Explicit(B)$

Step 1.3 Apply the union and additivity rules. Applying these properties compresses the set of PAIs into a minimal, understandable set in which characteristics and their relationships are clarified. Using the union rule, we express PAIs 2 and 3 and PAIs 12 and 13 as the following:

$$Bk.Blocking(A) \rightarrow \{12,13\} \leftarrow IC.Unaware(G)$$

$$IC.Unaware(G) \rightarrow \{12,13\} \leftarrow Bk.Blocking(B)$$

Using the additivity rule, we combine together the PAIs 4 and 5 and the PAIs 17 and 18 into the following:

$$DSM.Local(A) \rightarrow \{11\} \leftarrow \begin{matrix} SDT.Implicit(G), \\ DSM.Stream(G) \end{matrix}$$

$$SDT.Implicit(G), DSM.Stream(G) \rightarrow \{11\} \leftarrow DSM.Local(B)$$

Step 1.4 Examine the remaining PAIs in more depth. In this step we take each remaining PAI and describe why the problem exists in the context of our example. From this examination, we produce generic requirements or issues for PAI resolution. In some cases, a deeper assessment is needed.

We group the PAIs to simplify the discussion of the integration solution requirements. Generally, there is no preset method for grouping. In this case, we first consider the PAIs between filter A and gzip, followed by the PAIs between gzip and filter B.

Group 1

$DSM.Local(A) \rightarrow \{11\} \leftarrow DT.Implicit(G), DSM.Stream(G)$

Description: Local data is normally passed discretely and explicitly. Gzip uses incremental data transfer via pipes.

Requirements: The integration solution needs to accept filter A's explicit data transfer and transform the data into an implicit and incremental stream for gzip to accept via pipes. Since gzip must receive incremental data, it runs in a separate process independent of the integration solution.

Group 2

$SCT.Explicit(A) \rightarrow \{12\} \leftarrow SCT.Implicit(G)$

$SDT.Explicit(A) \rightarrow \{13\} \leftarrow SDT.Implicit(G)$

Description: Filter A makes direct function calls. However, gzip is called implicitly with a standard UNIX command and is passed data via pipes.

Requirements: The integration solution needs to generate and execute a UNIX command that transfers control over to gzip. A UNIX pipe must be created so that filter A can pass its data.

Group 3

$IC.UnAware(G) \rightarrow \{2\} \leftarrow SCT.Explicit(B)$

$IC.UnAware(G) \rightarrow \{6\} \leftarrow SDT.Explicit(B)$

$IC.UnAware(G) \rightarrow \{7\} \leftarrow DSM.Local(B)$

Description: Filter B expects to be passed control and data explicitly, but gzip has no knowledge of other components in the system. After gzip has completed execution, it will simply relinquish its control and pass its data onto the standard output.

Requirements: An integration solution is needed that reads data from the standard output, identifies when gzip has completed its execution and then passes control and data explicitly to filter B.

Group 4

$SDT.Implicit(G), DSM.Stream(G) \rightarrow \{11\} \leftarrow DSM.Local(B)$

Description: Gzip expects to transfer data implicitly and incrementally using a bounded pipe. Filter B's local data storage expects to be passed data discretely.

Requirements: The integration solution needs to consolidate gzip's output incrementally as the pipe fills. It then must pass it on discretely and explicitly to filter B's local data storage upon gzip's completion.

Group 5

$SCT.Implicit(G) \rightarrow \{12\} \leftarrow SCT.Explicit(B)$

$SCT.Implicit(G) \rightarrow \{13\} \leftarrow SDT.Explicit(B)$

Description: Gzip transfers control without directing it to filter B. Gzip transfers data implicitly and outputs its compressed data onto a pipe. Filter B expects to receive control and data explicitly via a function call.

Requirements: An integration solution must be built to transfer control to filter B after gzip has completed.

5.3 Phase 2: Application-Component Interactions

Problematic interactions that result from component-component comparisons do not always comprise the final set of conflicts. When considering application requirements in phase 2, PAIs previously discovered in phase 1 can be eliminated. Furthermore, phase 2 can uncover additional PAIs that result from application-component interaction. In this section, we discuss those PAIs that can be eliminated based on of application requirements. Though they would normally be discussed more fully in phase 1, we did not introduce them earlier for the sake of brevity.

Step 2.1 Eliminate those PAIs from pairwise assessments in which there is no control or data exchange. A global analysis of application requirements can render certain PAIs as extraneous due to unidirectional interaction between components. Consider the following PAI identified in phase 1,

$Bk.Blocking(A) \rightarrow \{3\} \leftarrow Bk.Blocking(G)$

Description: If both filter A and gzip are waiting for the other to initiate execution they will not be able to communicate.

In the Compressing Proxy application, each component completes execution before passing its control. Control flows only from filter A to gzip and gzip to filter B. Therefore, the application requirements dictate that this potential PAI is irrelevant because filter A is not dependent on gzip's execution. The PAIs listed below are also eliminated for similar reasons.

$SCT.Explicit(A) \rightarrow \{2\} \leftarrow IC.UnAware(G)$

$SDT.Explicit(A) \rightarrow \{6\} \leftarrow IC.UnAware(G)$

$DSM.Local(A) \rightarrow \{7\} \leftarrow IC.UnAware(G)$

$Bk.Blocking(A) \rightarrow \{12, 13\} \leftarrow IC.UnAware(G)$

$IC.UnAware(G) \rightarrow \{12, 13\} \leftarrow Bk.Blocking(B)$

$Bk.Blocking(G) \rightarrow \{3\} \leftarrow Bk.Blocking(B)$

Step 2.2 Append conflicts influenced by the application requirements. Application requirements can influence interoperability by dictating a context in which certain configuration and coordination issues become problematic, depending on how compatible they are with the expectations of the components [7]. Thus, application-component conflicts are highly variable, depending on the current application characteristics. In some instances, they can affect the resolution of a component-component conflict. However, in this example the application requirements do not produce any new PAIs.

5.4 Phase 3: Application-Integration Interactions

The results of phases 1 and 2 normally include multiple, generic, integration solution requirements per PAI or PAI group. In the Compressing Proxy, we obtain these

requirements only from phase 1. The first step of phase 3 takes a global view of these requirements toward consolidating them to create a solution.

Step 3.1 Assessing the integration requirements as a whole. Group 1 and group 2 requirements are based on the communication between filter A and gzip. These requirements support the addition of a unique process that spawns gzip and builds the pipe for filter A's data transmission. From the requirements of groups 3, 4, and 5, we see that an additional process must be able to read the data from gzip's output pipe incrementally. The same process must also consolidate all of the output before passing it to filter B. Therefore, it must run in a separate process from gzip. The need for two separate processes (one for input and one for output) and the understanding that gzip must be able to read and write incrementally dictates that these processes must execute concurrently.

Step 3.2 Re-examining the application requirements. Even when an integration solution can be constructed to satisfy the requirements, there is still a potential for conflicts to occur between the integration solution requirements and the original application requirements. When this occurs, either changes can be made to the application requirements, given the extent of their malleability, or the integration solution requirements must be amended. However, such modifications must be controlled so they do not pose new conflicts. We demonstrate the former case below.

In the Compressing Proxy, the need for the two additional processes as a result of step 3.1 is in direct conflict with the application requirements for a sequential control structure and a linear data topology. In this instance, the application requirements can (and should) be changed to allow concurrent processes and an arbitrary data topology without any detrimental effects. The end result is the implementation of two concurrent processes to direct input to and output from gzip.

6. DISCUSSION AND CONCLUSION

The practice of adhering to the precepts of CBSE is gathering momentum. However, the causes of interoperability problems must be thoroughly addressed, along with the effects, to make this form of system design reliable. It is obvious that analysis of interoperability problems early in design can aid in the formation of integration architectures.

In this paper, we illustrate how *pre-integration* assessment of fundamental architecture characteristics can illuminate potential problematic interactions. We have presented a simple notation to be used in conjunction with the static analysis of problematic architecture interactions to provide initial requirements of an integration solution. By borrowing principles from software architecture, our process benefits from a proven methodology.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

- [1] Abd-Allah, A. Composing Heterogeneous Software Architectures. Ph. D. Dissertation, Computer Science, University of Southern California, 1996.
- [2] Abowd, G., Allen, A., Garlan, D. Formalizing Style to Understand Descriptions of Software Architecture. *ACM TOSEM* (1995), 4(4): 319-64.
- [3] Allen, R., Garlan, D. A Formal Basis for Architectural Connection. *ACM TOSEM* (1997), 6(3): 213-49.
- [4] Barret, D., Clarke, L., Tarr, P., Wise, A. An Event-Based Software Integration Framework 95-048. Laboratory for Advances Software Engineering Research, Computer Science Dept., Univ. of Massachusetts, 1995.
- [5] Boehm, B., Port, D., Egyed, A., Abi-Antoun, M. The MBASE Life Cycle Architecture Milestone Package: No Architecture is an Island. In, *1st Working International Conference on Software Architecture*, (1999).
- [6] Compare, D., Inverardi, P., Wolf, A. Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming* (1999), 33: 101-31.
- [7] Davis, L., Gamble, R., Payton, J. The Impact of Component Architectures on Interoperability, *Journal of Systems and Software*, (to appear 2002).
- [8] Davis, L., Payton, J., Gamble, R. How System Architectures Impede Interoperability, *2nd International Workshop On Software and Performance*, (2000).
- [9] Davis, L., Payton, J., Gamble, R. Toward Identifying the Impact of COTS Evolution on Integrated Systems. *2nd Workshop on Successful COTS*, (2000).
- [10] Gacek, C. Detecting Architectural Mismatches During Systems Composition USC/CSE-97-TR-506. Center for Software Engineering, USC, 1997.
- [11] Garlan, D. Higher-Order Connectors, *Workshop on Compositional Software Architectures*, (1998).
- [12] Garlan, D., Allen, A., Ockerbloom, J. Architectural Mismatch, or Why it is hard to build systems out of existing parts, In *ICSE-95* (Seattle, WA, 1995).
- [13] Garlan, D., Monroe, R., Wile, D. ACME: An Architectural Description Language. In *CASCON* (1997).
- [14] Inverardi, P., Wolf, A., Yankelevich, D. Static Checking of System Behaviors Using Derived

- Component Assumptions. *ACM TOSEM* (2000), 9(3): 239-72.
- [15] Jacobson, I., Booch, G., Rumbaugh, J. *Unified Process Development Model*. Addison Wesley, 1999.
- [16] Kazman, R., Klein, M., Clements, P. ATAM: Method for Architecture Evaluation, CMU, 2000.
- [17] Kelkar, A., Gamble, R. Understanding the Architectural Characteristics behind Middleware Choices. *1st Int'l Conf. on Information Reuse & Integration* (1999).
- [18] Keshav, R., Gamble, R. Towards a Taxonomy of Architecture Integration Strategies, *3rd ISAW* (1998).
- [19] Luckham, D., Vera, J. An Event-Based Architectural Definition Language. *IEEE TSE* (1995), 21(9): 717-34.
- [20] Magee, J., Dulay, N., Eisenbach, S., Kramer, J. Specifying Distributed Software Architectures. *5th European Software Engineering Conference*, (1995).
- [21] Medvidovic, N., Gamble, R., Rosenblum, D. Towards Software Multioperability: Bridging Heterogeneous Software Interoperability Platforms, *4th ISAW* (Limerick, Ireland, 2000).
- [22] Medvidovic, N., Rosenblum, D., Taylor, R. A Language and Environment for Architecture-Based Software Development and Evolution. *21st ICSE* (Los Angeles, CA, 1999).
- [23] Mehta, N., Medvidovic, N., Phadke, S. Towards a Taxonomy of Software Connectors. *22nd ICSE*, (Limerick, Ireland, 2000).
- [24] Mularz, D. Pattern-Based Integration Architectures. In, *PLoP*, (1994).
- [25] Payton, J., Gamble, R., Kimsen, S., Davis, L. The Opportunity for Formal Models of Integration. *2nd Int'l Conf. on Information Reuse and Integration*, (2000).
- [26] Payton, J., Keshav, R., Gamble, R. System Development Using the Integrating Component Architectures Process. *1st Workshop on Ensuring Successful COTS Development* (1999), 49-51.
- [27] Perry, D., Wolf, A. Foundations for the Study of Software Architecture. *ACM SIGSOFT* (1992), 17(4):40-52.
- [28] Shaw, M., Clements, P. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems, *1st Int'l Computer Software and Applications Conference* (1997), Washington, D.C., 6-17.
- [29] Shaw, M., Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*, Englewood Cliffs, NJ, Prentice Hall, 1996.
- [30] Sitaraman, R. Integration of Software Systems at an Abstract Architectural Level. CS M.S. Thesis, University of Tulsa, 1997.
- [31] Stiger, P. An Assessment of Architectural Styles and Integration Components. CS M.S. Thesis, University of Tulsa, 1997.
- [32] Yakimovich, D., Bieman, J., Basili, V. Software Architecture Classification for Estimating the Cost of COTS Integration. *21st ICSE* (Los Angeles, CA, 1999), 296-302.
- [33] Yakimovich, D., Travassos, G., Basili, V. A Classification of Software Components Incompatibilities for COTS Integration, (2000).