

THE UNIVERSITY OF TULSA
THE GRADUATE SCHOOL

NOTATING PROBLEMATIC
ARCHITECTURE INTERACTIONS

by
Gerður Jónsdóttir

A thesis submitted in partial fulfillment of
the requirements for the degree of Master of Science

in the Discipline of Computer Science

The Graduate School

The University of Tulsa

2002

**THE UNIVERSITY OF TULSA
GRADUATE SCHOOL**

**NOTATING PROBLEMATIC
ARCHITECTURE INTERACTIONS**

by

Gerður Jónsdóttir

A THESIS

**APPROVED FOR THE DISCIPLINE OF
COMPUTER SCIENCE**

By Thesis Committee

_____, **Chairperson**

ABSTRACT

Jónsdóttir, Gerður (Master of Computer Science)

Notating Problematic Architecture Interactions (Chapter I – VII, pp 1 – 71)

Directed by Dr. Rose Gamble

(135)

The improvement of component-based software engineering is essential to the rapid, cost-effective development of complex software systems, affording reusability and increased reliability. However, applications developed according to this practice can suffer from difficult maintenance and control problems stemming from improper or inadequate integration solutions. Avoiding such unfortunate results requires the ability to identify and mitigate the causes and effects of interoperability problems prior to integration.

Software architecture provides important coarse-grained information, which can be used to pinpoint interoperability problems in the form of *problematic architecture interactions*. This thesis focuses on specifying a notation and minimization rules that can be used to formally specify problematic architecture interactions. Moreover, we describe the use of the notation and minimization rules in a multi-phase process for pre-integration analysis establishing initial integration requirements based on the problematic architecture interactions identified.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Rose Gamble, for her guidance throughout my graduate studies and for all her assistance and encouragement. I want to thank Dr. Sen for all his help during my stay at The University of Tulsa. I also want to thank him and Dr. Redner for taking the time to serve on my Thesis Committee.

I am very grateful to everyone in the Software Engineering and Architecture Team (SEAT) for all their support and encouragement (Leigh Davis, Daniel Flagg, Jamie Payton, and Dan Underwood). I greatly ‘v’alue all the ‘w’onderful times we had in the office.

Last but not least, I want to thank my husband, Sigurður Ingi Kristinsson, for standing by my side, and supporting and pushing me toward my goal no matter which “disaster” struck. I will be grateful to you for the rest of my life.

This research is sponsored in part by AFOSR (F-49620-98-1-0217), AFSOR (F-49620-01-1-0002) and NSF (CCR-9988320).

TABLE OF CONTENTS

ABSTRACT	ii
ACKNOWLEDGEMENTS	iii
TABLE OF CONTENTS	iv
LIST OF TABLES	vi
LIST OF FIGURES	vii
CHAPTER I	1
INTRODUCTION	1
CHAPTER II	5
BACKGROUND	5
2.1 Properties Describing a Software Architecture	6
2.2 Pre-Integration Assessment	8
2.3 Connectors	9
2.4 Integration Elements	10
2.5 Post-Integration assessment	11
2.6 Thesis Terminology	11
CHAPTER III	13
NOTATION FOR PROBLEMATIC ARCHITECTURE INTERACTIONS ---	13
3.1 Architectural Interactions	13
3.2 Problem Categories	15
3.3 Notation and Associated Rules	17
3.4 Minimization Rules for Problematic Architecture Interactions	18
3.5 Pre-integration Process	25
CHAPTER IV	29
USING THE NOTATION	29
CHAPTER V	48
USING THE NOTATION FOR BLACK BOX INTEGRATION	48
CHAPTER VI	65
CONCLUSION AND FUTURE WORK	65
CHAPTER VII	67

REFERENCES	67
APPENDIX A	72

LIST OF TABLES

Table 1: Application and Component Level Characteristics	8
Table 2: Conflict Categories	16
Table 3: Component Characteristic Values.....	33
Table 4: Filter A – Gzip PAIs	36
Table 5: Gzip – Filter B PAIs.....	36
Table 6: Filter A – Gzip PAIs	37
Table 7: Gzip – Filter B PAIs.....	37
Table 8: Gzip – Filter B: Characteristic Additivity.....	39
Table 9: Filter A – Gzip: Conflict Union	39
Table 10: Application Characteristics Values	44
Table 11: Applying Component Additivity	45
Table 12: Component Characteristic Values.....	51
Table 13: Market – Risk PAIs.....	52
Table 14: Market – Financial PAIs	52
Table 15: Risk – Financial PAIs	53
Table 16: Market – Risk: Characteristic Additivity.....	53
Table 17: Market – Financial: Characteristic Additivity	54
Table 18: Risk - Financial: Characteristic Additivity	54
Table 19: Market – Risk: Conflict Union.....	54
Table 20: Market – Financial: Conflict Union.....	55
Table 21: Financial – Risk: Conflict Union	55
Table 22: TradeMaster Application Characteristics Values	57
Table 23: Application-Component PAIs.....	58
Table 24: Applying Component Union.....	59
Table 25: Applying Component Additivity	59
Table 26: Applying Conflict Union	60
Table 27: Conflict Categories	72
Table 28: Blocking PAIs	74
Table 29: Control Structure PAIs.....	76
Table 30: Control Topology PAIs.....	78
Table 31: Data Storage Method PAIs	79
Table 32: Data Topology PAIs.....	82
Table 33: Identity of Components PAIs.....	83
Table 34: Supported Control Transfer PAIs	84
Table 35: Supported Data Transfer PAIs	86
Table 36: Application Control Structure PAIs.....	88
Table 37: Application Control Topology PAIs.....	91
Table 38: Application Data Topology PAIs.....	94
Table 39: Application Synchronization PAIs.....	96

LIST OF FIGURES

Figure 1: Integrated System Terminology	12
Figure 2: Characteristic Additivity.....	20
Figure 3: Conflict Union	21
Figure 4: Component Additivity	23
Figure 5: Component Union.....	25
Figure 6: The Compressing Proxy	30
Figure 7: Compressing Proxy Architecture.....	33
Figure 8: Bipartite Conflict Graphs.....	34
Figure 9: TradeMaster	50
Figure 10: TradeMaster Architecture.....	51
Figure 11: Bipartite Conflict Graphs.....	52
Figure 12: TradeMaster Integration Solution.....	64

CHAPTER I

INTRODUCTION

The face of business computing is ever changing due to corporate mergers and the proliferation of the Internet. Component-based software engineering and reuse techniques offer assistance during a company's evolution, affording methods to adapt existing software components for increased reliability and reduced time-to-market. The use of these techniques introduces the challenge of coercing heterogeneous components to interoperate seamlessly. The path many companies have taken towards realizing this goal has been to use off-the-shelf middleware products to integrate their components. However, applications developed according to this practice can often suffer from difficult maintenance and control problems that stem from improper or inadequate integration solutions.

Decisions for using middleware (i.e., distributed computing services that facilitate interoperation between components [C99]) are often made in an ad hoc manner, without sufficient understanding of the initial systems chosen or the application into which they will be composed. This is not to say that middleware products are not useful or are 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.

Interoperability between software components is an important problem and has received much attention in the research community. It is, however, a very complex problem and, up to this point, no major breakthroughs have been made towards developing an all-encompassing solution. Interoperability problems can have a substantial impact on integration efforts, raising development costs and creating ad hoc, unmanageable solutions. Because interoperability problems are usually not discovered until the development is well underway, the solution is often last minute, improperly designed glue code, hindering future evolution and maintenance of the integrated application. It is obvious that a method is needed to identify and mitigate the causes and effects of interoperability problems. The appropriate time for this identification and assessment is during the design of the multi-component application.

Coarse-grained architectural characteristics can be used to pinpoint interoperability problems related to control and data interactions between components [D01, DPG02]. Software architecture is defined as the blueprint of a system's computational elements (components), the means by which they interact (connectors), and the structural constraints on that interaction [PW92, SG96]. We use nine high-level

architectural characteristics as a representative set of previously published characteristics [D01, DPG02]. Using a broad-based set of properties is important because off-the-shelf components often tend to be highly encapsulated, not providing much detail about their internal functionality.

In a large-scale integration, a myriad of integration conflicts can be uncovered using architectural information. It is important to present them in a manner that is easily understood and managed. Notating the interoperability problems in a principled way provides the developers with the ability to assess them systematically. If proper documentation is kept so that an interoperability problem is connected with a specific solution, a valuable design history results. This design history is used to make future evolution and maintenance simpler and more robust.

In this thesis, we describe notation for *problematic architecture interactions* as defined below.

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

Our notation describes PAIs and provides minimization rules used to compress them into a smaller, more meaningful set. We demonstrate the use of the notation and the minimization rules in a pre-integration assessment process, which results in the initial requirements needed to mitigate the causes and effects of interoperability problems.

This thesis is organized into the following chapters. Chapter II presents research that is directly relevant to this thesis. In Chapter III, we establish a notation for defining problematic architecture interactions. In Chapter IV, we present a model problem using the notation. Chapter V demonstrates use of the notation in an industrial example with black box components, where only a subset of architectural characteristics is known. Chapter IV closes with final results and future work.

CHAPTER II

BACKGROUND

Interoperability problems are a pressing concern to developers integrating complex component-based systems. Using architectural characteristics to identify interoperability problems is one major effort of ongoing software architecture research. Without a principled means to notate and manage these interoperability problems, their solutions are unclear. To justify our position, it is necessary to look at the overall research in the area of software architecture and its relation to component-based software engineering and integration issues. In this chapter, we overview the following subject areas that are applicable to our research.

- Software Architecture Characteristics- *Architectural characteristics* have been defined with respect to architectural styles and provide an abstract method of describing software components [D01, DPG02].
- Pre-Integration Assessment – Examining the software architecture of components to be integrated can elucidate potential interoperability problems [A96, G97, KCBA97, SIT97, YBB99].

- Connectors – Proper interconnection between components can resolve interoperability problems [AG97, G98, K99].
- Integration Elements – Classifying the entire spectrum of middleware-related connectors using a few generic *integration elements* allows an *integration architecture* to be created [K99, KG98].
- Post-Integration Assessment - Once an integration architecture is in place, analysis can be performed to detect such problems as deadlock, liveness, reliability, evolvability and security [CIW99, DG02b, JG02, IYW00].
- Thesis Terminology – The most important terms used in this thesis are defined.

2.1 Properties Describing a Software Architecture

The *software architecture* of a system is the blueprint of its computational elements, the means by which they interact, and the structural constraints on their interaction [PW92, SG96]. 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.

Characteristics defined with respect to architectural styles include those that describe the various types of computational elements and connectors, data issues, control issues, and control/data interaction issues [A96, AG97, BCTW96, G97, GMW97, SC97, STI97]. These architectural characteristics provide details that further differentiate

individual styles, their extensions and specializations. Characteristics have been viewed with respect to their potential impact on interoperability [GAO95]. However, only subsets based on style constraints have been examined for their role in integration issues [A96, AG97, BCTW96, G97, GMW97, SC97, STI97].

Our previous research partitions architectural characteristics across two viewpoints: *component-level* and *application-level*. Component-level characteristics contribute to an understanding of the component's exposed interface to other external subsystems. Application-level characteristics address architectural demands on the configuration and coordination of the integrated application [DPG00a].

Table 1 contains the relevant characteristics. Their abbreviated references are in column one, with the associated name in column two. They are distinguished in column three by whether they are component-level (C) or application-level (A). Definitions and values are in columns four and five, respectively. We do not claim that the set is complete; more characteristics may be introduced and partitioned accordingly with further research [D01].

What makes this characteristic set essential for interoperability analysis is the encompassing nature of its definitions [DPG02, DPG00b, KG99]. The nine static, high-level characteristics have multiple semantic links to corresponding low-level characteristics. Therefore, they are a representative set of the characteristics previously published. Highly encapsulated components tend to provide little to no detail about their internal functionality, providing a strong impetus to concentrate on broad-based properties.

ABBRV	CHARACTERISTICS	TYPE	DEFINITION	VALUES
Bk	<i>Blocking</i>	C	Whether or not the thread of control is suspended.	Blocking, Non-Blocking
CS	<i>Control Structure</i>	A, C	The structure that governs the execution in the system.	Single-Thread, Multi-Thread, Concurrent
CT	<i>Control Topology</i>	A, C	The geometric form control flow takes in a system.	Hierarchical, Star, Arbitrary, Linear
DSM	<i>Data Storage Method</i>	C	How data is stored within a system.	Local, Global, Distributed, None
DT	<i>Data Topology</i>	A, C	The geometric form data flow takes in a system.	Hierarchical, Star, Arbitrary, Linear
IC	<i>Identity of Components</i>	C	Awareness of other components in the system.	Aware, Unaware
SCT	<i>Supported Control Transfer</i>	C	The method supported to achieve control transfer.	Explicit, Implicit, None
SDT	<i>Supported Data Transfer</i>	C	The method supported to achieve data transfer.	Explicit, Implicit-Discrete, Implicit-Continuous, Shared, None
Sn	<i>Synchronization</i>	A	Whether or not the components need to rendezvous.	Synchronous, Asynchronous

Table 1: Application and Component Level Characteristics

2.2 Pre-Integration Assessment

Pre-integration assessment aims to detect interoperability conflicts prior to the actual integration [A96, D01, DPG02, KCBA97, SIT97, YBB99]. The phrase *architecture mismatch* was coined to describe certain underlying reasons for interoperability problems among seemingly “open” software components [GAO95]. To be effective, pre-integration assessment should identify all architecture mismatches between the interacting components, notate them, and point to generic functionality needed to resolve them.

Architectural information has been used to determine the ability of heterogeneous software components to interoperate. Previous approaches, such as Architect's Automated Assistant [A96, G97], examine a narrow subset of styles, characterizing the components and connectors to illustrate important mismatch concerns. The tool is only able to analyze a limited set of architectural styles and provides no formal way to specify those interoperability problems identified.

Similarly, Sitarman [SIT97] examined a subset of architectural styles to identify mismatch between different styles. The styles and their characteristics are compared using a decision tree based approach. However, the system cannot be easily expanded, and there is no unified way to present identified conflicts.

Kazman et al., [KCBA97] classify software architectural elements to describe combinations that can easily interoperate (but do not detect problems). Yakimovich et al., compare architectural characteristics to provide an a priori cost estimation for COTS integration [YBB99]. However, both of these methods suffer from inconsistent representation and selection of characteristics. Neither provides a method to represent discovered conflicts, limiting their usability to specific contexts.

2.3 Connectors

Connectors are architectural entities that describe interactions between components. They are becoming increasingly important in software architecture analysis, having been raised to first-class status in many style description techniques [G98, KG98, MGR00, MMP00, STI97]. 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 [MMP00]. Formal descriptions of connectors have been expressed in several architectural description languages (ADLs) [A97, AG97, GMW97, LV95, MRT99, MDEK95], as well as in the Z and Object-Z notations [AAG95, PGKD00, STI97]. As their potential for contribution to analysis has been recognized, research is ongoing to capture and describe architecture integration functionality in terms of connectors [K99, KG98, SG01].

2.4 Integration Elements

Integration elements can be defined as architectural connectors for specifically resolving interoperability conflicts. We use three types of integration elements: the *translator*, the *controller*, and the *extender* [KG98, K99]. These integration element connectors supplement traditional architecture connectors by providing generic classes of functionality that resolve interoperability problems. Moreover, they offer a uniform description of the integration functionality found in particular architecture and design patterns, e.g., adaptor, broker, proxy, etc [BMRSS96, GHJV95]. Using this uniform description simplifies the construction of solutions to architecture interoperability problems [DG01, PJFG02].

An *integration architecture* is defined to be the software architecture description of an integration solution between at least two interacting software components [KG98, K99, M94]. Therefore, an integration architecture describes the strategies and patterns that comprise a specific integration solution. We construct integration architectures by composing integration elements [DG01, PJFG02]. Previous research has developed a taxonomy of middleware frameworks modeled as integration architectures, illustrating the relationship between them [KG98, K99].

2.5 Post-Integration assessment

Post-integration assessment looks at systems after the integration solution is already in place. The CHemical Abstract Machine (CHAM) formalism is one such assessment method. It identifies deadlock problems caused by mismatched behavioral assumptions at the architectural level [CIW99, IWY00]. These assessments offer additional insight into the prescribed integration architecture and can help in uncovering the impetus behind integration solutions. However, they are limited in how they assess interoperability a priori and how they direct the selection of an initial solution that is appropriate.

Post-integration assessments are useful for assessing non-functional properties such as security, guaranteed message delivery, and evolvability. Most non-functional requirements cannot be assessed effectively until an initial integration architecture has been established. Davis et al. [DG02b] examines two approaches for assessing non-functional properties: a pattern-based approach and a formal approach using I/O automata. In this instance, the two approaches are used to assess evolvability and guaranteed message delivery. In [JG02], an integration architecture is examined to discover potential security breaches and to enforce authentication with the goal of creating secure integration architectures.

2.6 Thesis Terminology

For this thesis, we use the following terminology. We refer to the software systems that interact as *components*. An *application* is a multi-component system. The entire abstract functionality that resides external to the component for them to form an application is the *integration architecture* (see Figure 1).

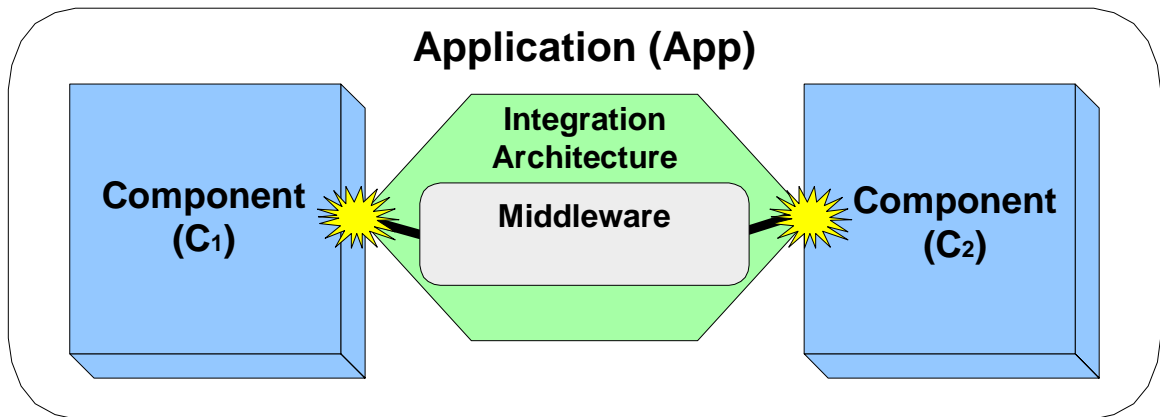


Figure 1: Integrated System Terminology

CHAPTER III

NOTATION FOR PROBLEMATIC ARCHITECTURE INTERACTIONS

Formally notating PAIs affords developers with a unified presentation of interoperability problems, which can simplify their resolution discovery and make their management easier. Identifying PAIs provides the developer with a “heads-up” about potential conflicts, even in the case of black-box component assessment. Furthermore, automation of pre-integration assessment can be realized given this foundation [PDUG01].

Providing minimization rules to condense a set of PAIs is highly beneficial because during a large-scale integration effort, a number of PAIs can be discovered. In addition, it often collapses PAIs according to the same integration solution that is needed. In this chapter, we introduce the notation for PAIs and the associated minimization rules.

3.1 Architectural Interactions

We use the characteristics presented in Chapter II, Section 2.1, Table 1 as the

architectural basis for our approach [DPG02, D01]. There are several interesting things to note that distinguish our research from others in the area. First, we show that conflicts can occur between like-valued characteristics. This is in contrast to the notion that only mismatched values lead to conflict [A96, G97, GAO95]. Second, we not only compare similar characteristic types (e.g., control topology vs. control topology) [A96, G97] but also define conflicts between different types of characteristics (e.g. control structure vs. control topology). This cross comparison goes beyond other research attempts to establish a partial order among values of a single characteristic type to determine the best value for that characteristic in the application to resolve conflict [YBB99].

We base our approach on the assumption that architectural characteristics uncover three types of interaction mismatch. The three types of interaction mismatch are the following:

- Two component systems are required to interact but are inhibited by certain characteristic values (component-component).
- 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).
- The application expectations do not comply with the integration solution needs (application-integration).

We formally notate the PAIs that occur due to the first two interaction types. The formal notation of the third interaction type is a part of future research. The entire set of PAIs that have been identified can be found in Appendix A.

3.2 Problem Categories

The interoperability problems that typically arise from component-component and application-component interactions appear in one of three categories: *control transfer*, *data transfer*, and *interaction initialization*. We classify and define these categories in Table 2. Column one gives the reference number of the conflict used throughout the thesis. The conflict name is in column two, followed by its definition in column three.

#	CONFLICT CATEGORY	CONFLICT 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 is not processed in the expected manner.
9	Invalid data	Data formats are incompatible.
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.

Table 2: Conflict Categories

We use the conflict reference number within the notation for a PAI (defined in the next section). For example

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

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 (conflict #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.

Extensive empirical study has shown that there is a static mapping between the conflicts and the characteristics comparison that can potentially cause them (See Appendix A). We use this mapping as a first-pass assessment to designate potential PAIs.

3.3 Notation and Associated Rules

In defining the notation and its associated rules, we assume there is a single application governing the interaction of at least two participating components.

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

Definition 3. Let **Labels** be a set of component names and App be the application name. Given \mathbf{N} is a non-empty string of names from $\mathbf{Labels} \cup \{App\}$, the notation $C.s(\mathbf{N})$ means that for all the component names in string \mathbf{N} , the value for characteristic C is s . Elements in string \mathbf{N} are separated by commas.

Every component has at most one element of **CVPairs** per characteristic. That is, given $C \in \mathbf{AC}$ such that $C.s$ and $C.t$ are in **CVPairs**, and $Q \in \mathbf{Labels} \cup \{App\}$,

$$C.s(Q) \wedge C.t(Q) \Leftrightarrow s = t$$

Definition 4. Let **Conflicts** be the set of interoperability conflicts in Table 2. Given that $T \subseteq \mathbf{Conflicts}$, $\{C_{1.s}, C_{2.t}\} \subseteq \mathbf{CVPairs}$, and $\{Q, R\} \subseteq \mathbf{Labels} \cup \{App\}$ such that $Q \neq R$, we describe a PAI using the following notation,

$$C_{1.s}(Q) \rightarrow T \leftarrow C_{2.t}(R)$$

which means that Q with characteristic/value pair $C_{1.s}$ problematically interacts with R having the characteristic/value pair $C_{2.t}$, causing all of the conflicts that appear in the set T . It should be noted that C_1 can equal C_2 , and s can be equal to t , i.e., they can be the same characteristic with the same value.

The notation provides a common vocabulary with which to discuss the interactions. By definition of a PAI the relation,

$$\rightarrow T \leftarrow$$

is symmetric for all characteristic/value pairs. Reflexivity and transitivity, however, do not hold for all pairs. Further research is ongoing to define appropriate constraints for transitivity to hold between and across components.

3.4 Minimization Rules for Problematic Architecture Interactions

There are four minimization rules associated with the notation: *characteristic additivity*, *conflict union*, *component additivity*, and *component union*. The rules pertain to particular interaction types. In this section, we define these rules and discuss their use in assessment. It is important to note that the minimization rules are information preserving. No information presented in the original PAIs is lost through application of the minimization rules.

Definition 5. Characteristic Additivity Rule for component-component interactions. Given that $T \subseteq \mathbf{Conflicts}$, $\{C_{1.s}, C_{2.t}, C_{3.v}\} \subseteq \mathbf{CVPairs}$ such that $C_2 \neq C_3$ (i.e., two different characteristics), and $\{Q, R\} \subseteq \mathbf{Labels}$ such that $Q \neq R$.

$$\begin{aligned} C_{1.s}(Q) \rightarrow T \leftarrow C_{2.t}(R) \wedge \\ C_{1.s}(Q) \rightarrow T \leftarrow C_{3.v}(R) \\ \Leftrightarrow C_{1.s}(Q) \rightarrow T \leftarrow C_{2.t}(R), C_{3.v}(R) \end{aligned}$$

Let $\mathbf{ACC}(R)$ be an accumulation of characteristic/value pairs for a component R that are separated by commas (e.g. $\mathbf{ACC}(R)$ above would be $C_{2.t}(R), C_{3.v}(R)$). Using the same additivity rule with $C_{n.r}$ as an element of $\mathbf{CVPairs}$, but not in $\mathbf{ACC}(R)$ we have:

$$\begin{aligned} C_{1.s}(Q) \rightarrow T \leftarrow \mathbf{ACC}(R) \wedge \\ C_{1.s}(Q) \rightarrow T \leftarrow C_{n.r}(R) \\ \Leftrightarrow C_{1.s}(Q) \rightarrow T \leftarrow \mathbf{ACC}(R), C_{n.r}(R) \end{aligned}$$

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). To maintain symmetry and for information preservation, the PAI must have at least one side that contains a single name associated with a characteristic/value pair.

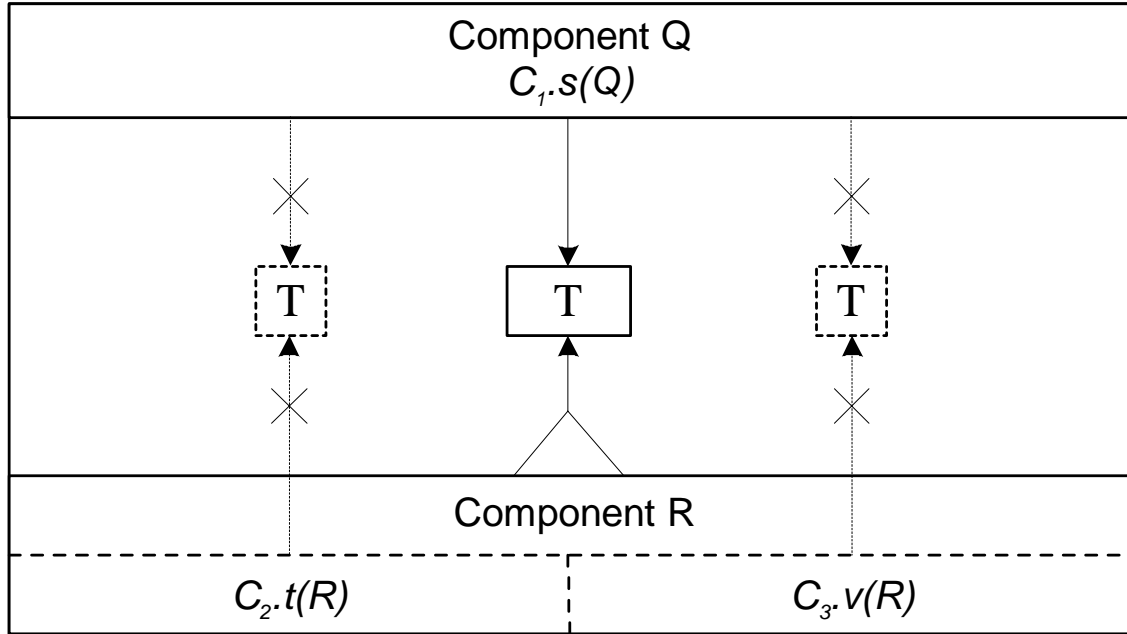


Figure 2: Characteristic Additivity

Figure 2 depicts the Characteristic Additivity Rule that combines PAIs across same conflicts with different contributing characteristic/value pairs. This rule gives a notion of significance to a particular conflict. Thus, there is the potential for focusing on a single solution for that conflict, minimizing complexity with the integration architecture.

Definition 6. **Conflict Union Rule** for both component-component and application-component interactions. Given that $\{T, V\} \subseteq \mathbf{Conflicts}$, $\{C_{1.s}, C_{2.t}\} \subseteq \mathbf{CVPairs}$, and $\{Q, R\} \subseteq \mathbf{Labels} \cup \{App\}$ such that $Q \neq R$,

$$\begin{aligned}
 & C_{1.s}(Q) \rightarrow T \leftarrow C_{2.t}(R) \wedge \\
 & C_{1.s}(Q) \rightarrow V \leftarrow C_{2.t}(R) \\
 \Leftrightarrow & C_{1.s}(Q) \rightarrow T \cup V \leftarrow C_{2.t}(R)
 \end{aligned}$$

Let the right-hand side (**RHS**) of the PAI represent an accumulation of component-related characteristic/value pairs separated by commas. Then it is the case that

$$C_{1.s}(Q) \rightarrow T \leftarrow \mathbf{RHS}_1 \wedge$$

$$C_{1.s}(Q) \rightarrow V \leftarrow \mathbf{RHS}_1$$

$$\Leftrightarrow C_{1.s}(Q) \rightarrow T \cup V \leftarrow \mathbf{RHS}_1$$

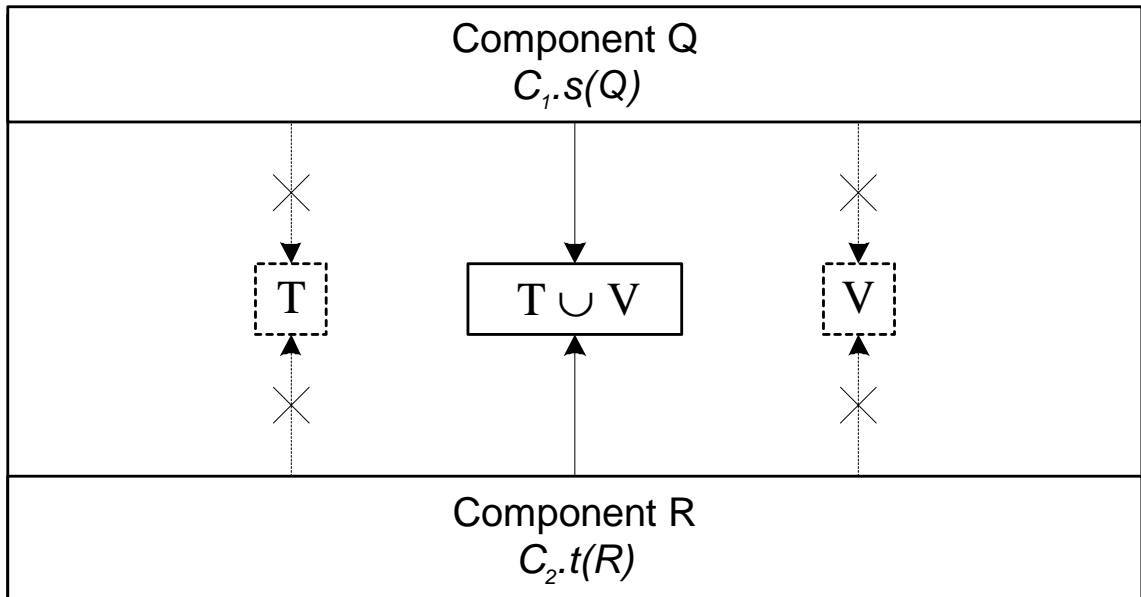


Figure 3: Conflict Union

Figure 3 depicts the Conflict Union rule that combines PAIs across the same characteristic/value pairs. 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. This could lead to redundancy among the solution entities in the integration architecture. Therefore, the rule leads to a smaller PAI set and points to relationships among those conflicts. To maintain symmetry and for information preservation, the PAI must have at

least one side that contains a only a single component label in the string associated with a characteristic/value pair, as well as at most one characteristic/value pair on that side.

Definition 7. **Component Additivity Rule** for application-component interactions. Given that $T \subseteq \mathbf{Conflicts}$, $\{C_{1.s}, C_{2.t}, C_{3.v}\} \subseteq \mathbf{CVPairs}$ such that $C_2 \neq C_3$ (i.e., two different characteristics), $\{Q, R\} \subseteq \mathbf{Labels}$, and App is the application name,

$$\begin{aligned} & C_{1.s}(App) \rightarrow T \leftarrow C_{2.t}(Q) \wedge \\ & C_{1.s}(App) \rightarrow T \leftarrow C_{3.v}(R) \\ \Leftrightarrow & C_{1.s}(App) \rightarrow T \leftarrow C_{2.t}(Q), C_{3.v}(R) \end{aligned}$$

Let the right-hand side (**RHS**) of the PAI be any set of component-related characteristic/value pairs and the associated component strings, separated by commas.

Using the same additivity rule with $C_{n.r}$ as an element of $\mathbf{CVPairs}$, but not in **RHS**,

$$\begin{aligned} & C_{1.s}(App) \rightarrow T \leftarrow \mathbf{RHS} \wedge \\ & C_{1.s}(App) \rightarrow T \leftarrow C_{n.r}(Q) \\ \Leftrightarrow & C_{1.s}(App) \rightarrow T \leftarrow \mathbf{RHS}, C_{n.r}(Q) \end{aligned}$$

This rule is analogous to Characteristic Additivity with the appearance of App on one side of the relation. There is no limit to the number of characteristics/value pairs that can be accumulated to one side of the relation (e.g., the right-hand side above) for those component names in **Labels**. However, to maintain symmetry and for information preservation, App must be the only label in the string on the other side of the relation (e.g., the left-hand side above) with a single characteristic/value pair.

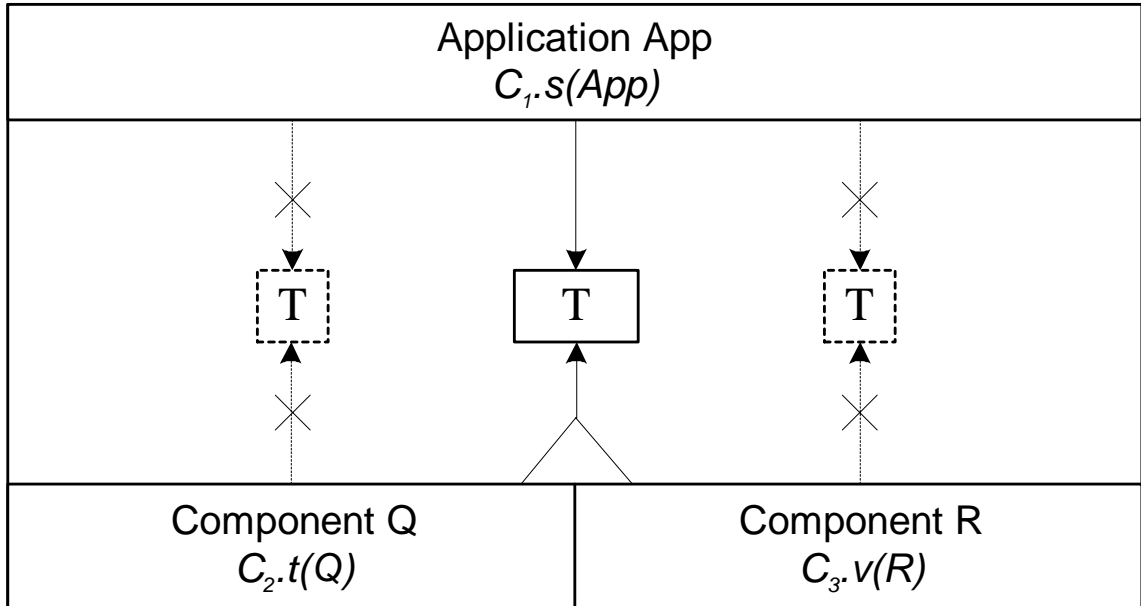


Figure 4: Component Additivity

Figure 4 shows the Component Additivity rule as consolidating the causes of a particular conflict due to the interaction of the application requirements with multiple components. The examination of the conflict set can pinpoint an overriding integration solution that can satisfy both application requirements and component needs.

Definition 8. Component Union Rule for application-component interactions. Given that $T \subseteq \mathbf{Conflicts}$, $\{C_{1.s}, C_{2.t}\} \subseteq \mathbf{CVPairs}$, \mathbf{N} is a non-empty string of names from **Labels**, $R \in \mathbf{Labels}$ such that $R \notin \mathbf{N}$, and App is the application name,

$$C_{1.s}(App) \rightarrow T \leftarrow C_{2.t}(\mathbf{N}) \wedge$$

$$C_{1.s}(App) \rightarrow T \leftarrow C_{2.t}(R)$$

$$\Leftrightarrow C_{1.s}(App) \rightarrow T \leftarrow C_{2.t}(\mathbf{N}, R)$$

Let the right hand side (**RHS**) of the PAI be any set of characteristic/value pairs and their associated component strings, separated by commas. $C_{i,r}(R)$ is a **CVPair** in which R is a component label *not* in **N**. It follows that

$$\begin{aligned}
 C_{l,s}(App) \rightarrow T \leftarrow \mathbf{RHS}, C_{i,r}(\mathbf{N}) \wedge \\
 C_{l,s}(App) \rightarrow T \leftarrow C_{i,r}(R) \\
 \Leftrightarrow C_{l,s}(App) \rightarrow T \leftarrow \mathbf{RHS}, C_{i,r}(\mathbf{N}, R)
 \end{aligned}$$

With this rule we pull together all characteristic/value pairs and components that have some issue with the application via the same conflict set. To maintain symmetry and for information preservation, App is the only label that can appear in the string on the one side of the relation (e.g., the left-hand side above) associated with a single characteristic/value pair.

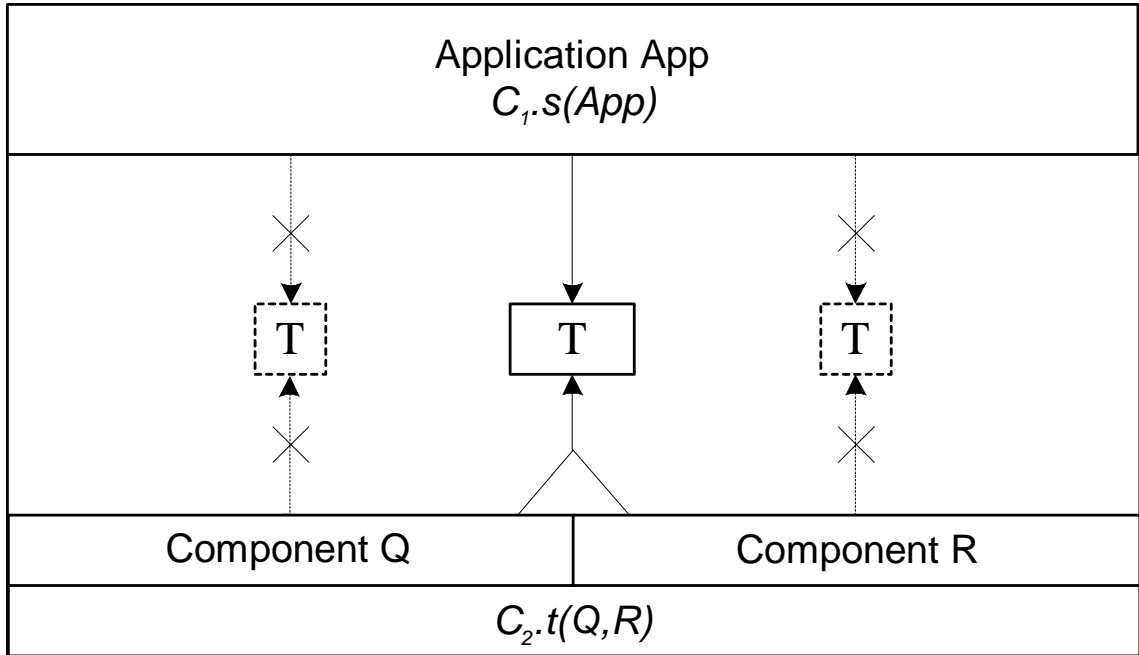


Figure 5: Component Union

Figure 5 depicts the Component Union rule using component names Q and R that form the resulting label string. This rule indicates that there is a common characteristic among a set of components that conflicts with the governing application. Detailed examination of this characteristic can pinpoint which application requirements are most prohibiting, leading to a change in the requirements (if possible), a change in component selection, or the use of a specific integration solution.

3.5 Pre-integration Process

We follow a generic process to perform pre-integration assessment and apply the minimization rules. The process states the steps that need to be taken to identify and denote PAIs, given a set of components that need to be integrated and the surrounding application requirements. The process is partitioned into three phases, each which corresponds to one of the interaction types described in Section 3.1.

At the heart of this process is the use of the defined minimization rules to compress the set of PAIs into a manageable set while maintaining all pertinent information about the interoperability conflicts. Applying the minimization rules in a particular order can result in a more meaningful set than just applying them randomly. The main constraint is to apply the Conflict Union rule last. This is because when the conflict set of a PAI has been increased, it becomes increasingly difficult to find a matching set of conflicts within different PAIs. It is, therefore, beneficial to apply the other rules first to achieve an optimal combination. The order in which Component Additivity and Component Union are applied is unimportant because the resulting set will always be the same. We state the pre-integration process below as the basis for our assessment in Chapters IV and V.

Phase 1: Component-Component Interactions

Step 1.1 Determine the characteristic/value sets for each component.

Step 1.2 Graph the basic architecture of the application indicating the participating components, their connectivity, and the directionality of data and control flow in the system.

Step 1.3 Notate the component-component PAIs using bipartite graphs.

Step 1.4 Eliminate those PAIs from the pairwise assessments in which there is no data and/or control exchange.

Step 1.5 Apply the minimization rules.

Step 1.5.1 Apply the characteristic additivity rule.

Repeat Step 1.5.1 until characteristic additivity can no longer be applied to the PAI set.

Step 1.5.2 Apply the conflict union rule.

Repeat step 1.5.2 until conflict union can no longer be applied to the PAI set.

Step 1.6 Examine the remaining PAIs in more depth.

Phase 2: Application-Component Interactions

Step 2.1 Determine the characteristic/value sets for the application.

Step 2.2 Notate the application-component PAIs using bipartite graphs.

Step 2.3 Apply the notation rules.

Step 2.3.1 Apply the component union rule.

Step 2.3.2 Apply the component additivity rule.

Repeat step 2.3.1 and 2.3.2 until component union and component additivity can no longer be applied to the PAI set.

Step 2.3.3 Apply the conflict union rule.

Repeat step 2.3.3 until conflict union can no longer be applied to any PAI set.

Repeat step 2.3 until no rule can be applied to the set PAI set.

Step 2.4 Examine the remaining PAIs in more depth.

Phase 3: Application-Integration Interactions

Step 3.1 Assessing the integration requirements as a whole.

Step 3.2 Re-examining the application requirements.

The result of the pre-integration process is the PAI set and an initial integration solution design.

CHAPTER IV

USING THE NOTATION

This section shows the use of PAIs in the multi-phase process presented in Chapter III, Section 3.1. To demonstrate its use, the Compressing Proxy a previously defined model integration problem, is employed [CIW99, IWY00].

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 [CIW99, IWY00]. 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. 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 parameters to the functions. Previously, these stream objects have been referred to as filters [CIW00, IWY00], and we will continue to use this terminology.

Gzip is a file compression utility commonly used as a filter (a type of architectural component) at the UNIX process level. It communicates through UNIX pipes (a type of architectural connector), 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 6).

The Compressing Proxy is used to demonstrate the correctness of a post-integration interoperability analysis method based on the CHAM (CHemical Abstract Machine) formalism [CIW00, IWY00]. By applying CHAM to the Compressing Proxy, a deadlock problem caused by an unsophisticated adapter is correctly identified. CHAM is then reapplied to prove that a modified integration solution using two concurrent adapters is free of deadlock.

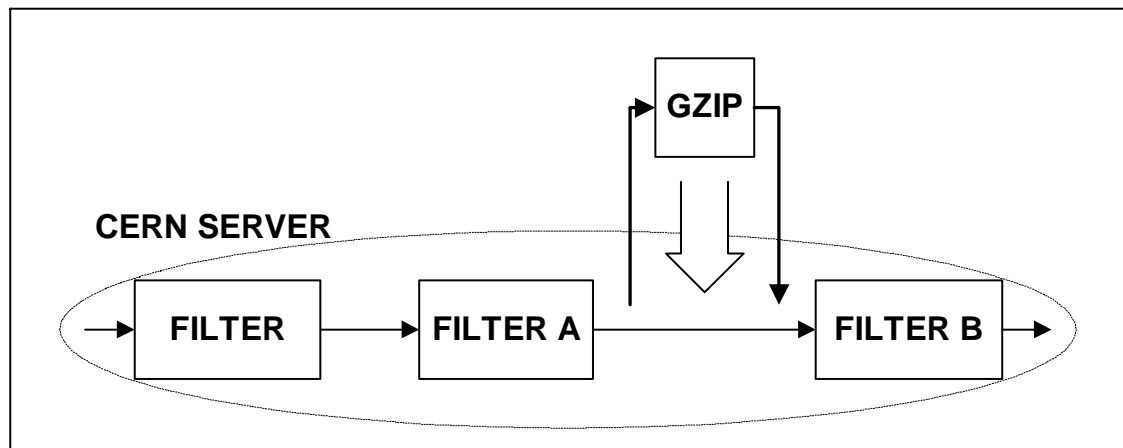


Figure 6: The Compressing Proxy

This chapter illustrates how interoperability problems are identified, notated and planned for, using the Compressing Proxy example, before any integration effort is performed. It depicts the identified PAIs in a discrete and understandable set that leads to

formulating an integration solution design similar to the deadlock-free adapter implemented in Inverardi et al. [CIW00, IWY00].

Phase 1: Component-Component Interactions

This phase covers component-component interaction described in Chapter III, Section 3.1. We divide the phase into six steps to examine the direct interaction among components. In Step 1.6 we examine the PAIs discovered in previous steps and identify generic integration solution requirements. We use the requirements to set up preliminary integration solutions before examining the application-component interaction. This identification gives us important knowledge about the structure of the application that can be used to our advantage in Phase 2.

Step 1.1 Determine the characteristic/value sets for each component. We assign a value to the 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. When no value exists, we simply omit the comparison of that characteristic.

Since the filters in the CERN server are architecturally equivalent, filters A and B in Figure 6 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 exchanges control by invoking the interface function of its sink, passing data as a parameter. Therefore, its 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 its upstream filter initiates it again.

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 blocks when reading from an empty pipe or writing to a full pipe. As is typical of a Unix filter, it does not know the identity of its upstream and downstream filters, thus it is unaware and simply relinquishes its control, implying a supported control transfer. Finally, `gzip` transfers its data in a stream onto its standard output pipe. Hence, its supported data transfer is implicit and continuous [G97]. Table 3 summarizes the component characteristic values identified above.

CHARACTERISTICS	FILTER A	GZIP	FILTER B
<i>Blocking</i>	Blocking	Blocking	Blocking
<i>Control Structure</i>	Single-Thread	Single-Thread	Single-Thread
<i>Control Topology</i>	Linear	Linear	Linear
<i>Data Storage Method</i>	Local	None	Local
<i>Data Topology</i>	Linear	Linear	Linear
<i>Identity of Components</i>	Aware	Unaware	Aware
<i>Supported Control Transfer</i>	Explicit	Implicit	Explicit
<i>Supported Data Transfer</i>	Explicit	Implicit-Continuous	Explicit

Table 3: Component Characteristic Values

Step 1.2 Graph the basic architecture of the application indicating the participating components, their connectivity, and the directionality of data and control flow in the system. It is important to graph the configuration of the application. This illuminates important information relating to connectivity and directionality of data and control flow in the system. The information is utilized in later steps to eliminate unnecessary comparisons between components that never exchange control and data, simplifying the assessment. See Figure 7 for the basic architecture of the Compressing Proxy.



Figure 7: Compressing Proxy Architecture

Step 1.3 Notate the component-component PAIs using bipartite graphs. We first construct a bipartite conflict graph for pairwise assessment of components. Bipartite graphs allow full cross-comparison of characteristics for every pair of components that interact. Hence, we do not compare topologically unconnected components, nor compare a component with itself. For instance, in Step 1.2 we see that Filters A and B do not communicate directly in the application. Therefore, it is unnecessary to perform pairwise assessment between those two components.

For clarity, Figure 8 displays only those lines in the bipartite conflict graphs that represent conflicts between Filter A and gzip and between gzip and Filter B.

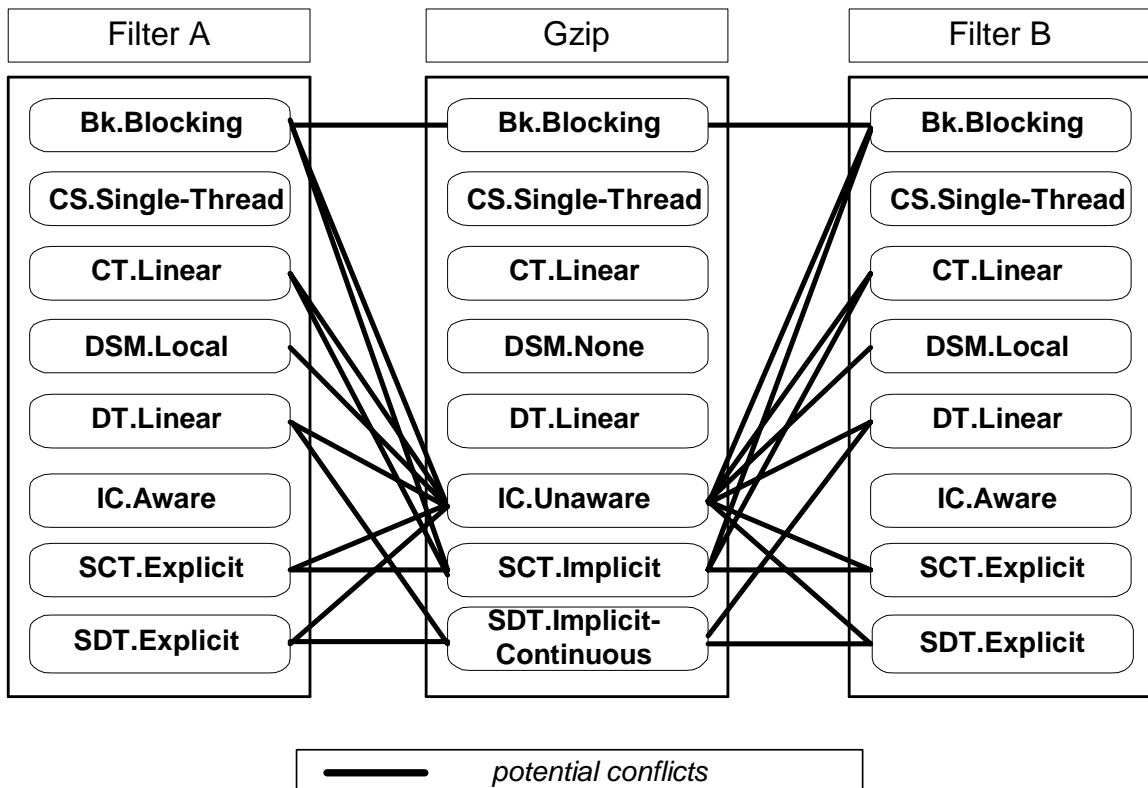


Figure 8: Bipartite Conflict Graphs

The potential conflicts illustrated in Figure 8 are notated as PAIs in Tables 4 and 5. We refer to the components as A, G, B for Filter A, gzip, and Filter B, respectively. App is maintained as the application label. It is important to point out that some conflicts indicated in Figure 8 (e.g, the line from Filter A's blocking characteristic to gzip's unaware characteristic) are notated as multiple, distinct PAIs in Table 4. The characteristic comparison results in multiple conflicts.

Step 1.4 Eliminate those PAIs from the pairwise assessments in which there is no data and/or control exchange. A global analysis of the direction that data and control flow in the system can render certain PAIs extraneous, especially if there is only unidirectional interaction between components.

Consider the following PAI at the top of Table 4,

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

Basically, this PAI means that 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. By examining Figure 7 in Step 1.2, control flows linearly from Filter A to gzip and from gzip to Filter B. Therefore, Filter A is not dependent on gzip's execution. Neither is gzip execution dependent on Filter B. PAIs crossed out in Tables 6 and 7 are eliminated for similar reasons.

FILTER A – GZIP
<i>Bk.Blocking(A) → {3} ← Bk.Blocking(G)</i>
<i>Bk.Blocking(A) → {12} ← IC.Unaware(G)</i>
<i>Bk.Blocking(A) → {13} ← IC.Unaware(G)</i>
<i>Bk.Blocking(A) → {3} ← SCT.Implicit(G)</i>
<i>CT.Linear(A) → {1} ← IC.Unaware.(G)</i>
<i>CT.Linear(A) → {1} ← SCT.Implicit(G)</i>
<i>DSM.Local(A) → {7} ← IC.Unaware(G)</i>
<i>DT.Linear(A) → {5} ← IC.Unaware.(G)</i>
<i>DT.Linear(A) → {5} ← SDT.Implicit-Continuous(G)</i>
<i>SCT.Explicit(A) → {1} ← IC.Unaware(G)</i>
<i>SCT.Explicit(A) → {12} ← IC.Unaware(G)</i>
<i>SCT.Explicit(A) → {1} ← SCT.Implicit(G)</i>
<i>SCT.Explicit(A) → {2} ← SCT.Implicit(G)</i>
<i>SDT.Explicit(A) → {5} ← IC.Unaware(G)</i>
<i>SDT.Explicit(A) → {13} ← IC.Unaware(G)</i>
<i>SDT.Explicit(A) → {5} ← SDT.Implicit-Continuous(G)</i>
<i>SDT.Explicit(A) → {6} ← SDT.Implicit-Continuous(G)</i>
<i>SDT.Explicit(A) → {11} ← SDT.Implicit-Continuous(G)</i>

Table 4: Filter A – Gzip PAIs

GZIP – FILTER B
<i>Bk.Blocking(G) → {3} ← Bk.Blocking(B)</i>
<i>IC.Unaware(G) → {12} ← Bk.Blocking(B)</i>
<i>IC.Unaware(G) → {13} ← Bk.Blocking(B)</i>
<i>IC.Unaware(G) → {1} ← CT.Linear(B)</i>
<i>IC.Unaware(G) → {7} ← DSM.Local(B)</i>
<i>IC.Unaware.(G) → {5} ← DT.Linear(B)</i>
<i>IC.Unaware(G) → {1} ← SCT.Explicit(B)</i>
<i>IC.Unaware(G) → {12} ← SCT.Explicit(B)</i>
<i>IC.Unaware(G) → {5} ← SDT.Explicit(B)</i>
<i>IC.Unaware(G) → {13} ← SDT.Explicit (B)</i>
<i>SCT.Implicit(G) → {3} ← Bk.Blocking(B)</i>
<i>SCT.Implicit(G) → {1} ← CT.Linear(B)</i>
<i>SCT.Implicit(G) → {1} ← SCT.Explicit(B)</i>
<i>SCT.Implicit(G) → {2} ← SCT.Explicit(B)</i>
<i>SDT.Implicit-Continuous(G) → {5} ← DT.Linear(B)</i>
<i>SDT.Implicit-Continuous(G) → {5} ← SDT.Explicit (B)</i>
<i>SDT.Implicit-Continuous(G) → {6} ← SDT.Explicit (B)</i>
<i>SDT.Implicit-Continuous(G) → {11} ← SDT.Explicit (B)</i>

Table 5: Gzip – Filter B PAIs

FILTER A – GZIP
<i>Bk.Blocking(A)</i> → {3} ← <i>Bk.Blocking(G)</i>
<i>Bk.Blocking(A)</i> → {12} ← <i>IC.Unaware(G)</i>
<i>Bk.Blocking(A)</i> → {13} ← <i>IC.Unaware(G)</i>
<i>Bk.Blocking(A)</i> → {3} ← <i>SCT.Implicit(G)</i>
<i>CT.Linear(A)</i> → {1} ← <i>IC.Unaware.(G)</i>
<i>CT.Linear(A)</i> → {1} ← <i>SCT.Implicit(G)</i>
<i>DSM.Local(A)</i> → {7} ← <i>IC.Unaware(G)</i>
<i>DT.Linear(A)</i> → {5} ← <i>IC.Unaware.(G)</i>
<i>DT.Linear(A)</i> → {5} ← <i>SDT.Implicit-Continuous(G)</i>
<i>SCT.Explicit(A)</i> → {1} ← <i>IC.Unaware(G)</i>
<i>SCT.Explicit(A)</i> → {12} ← <i>IC.Unaware(G)</i>
<i>SCT.Explicit(A)</i> → {1} ← <i>SCT.Implicit(G)</i>
<i>SCT.Explicit(A)</i> → {2} ← <i>SCT.Implicit(G)</i>
<i>SDT.Explicit(A)</i> → {5} ← <i>IC.Unaware(G)</i>
<i>SDT.Explicit(A)</i> → {13} ← <i>IC.Unaware(G)</i>
<i>SDT.Explicit(A)</i> → {5} ← <i>SDT.Implicit-Continuous(G)</i>
<i>SDT.Explicit(A)</i> → {6} ← <i>SDT.Implicit-Continuous(G)</i>
<i>SDT.Explicit(A)</i> → {11} ← <i>SDT.Implicit-Continuous(G)</i>

Table 6: Filter A – Gzip PAIs

GZIP – FILTER B
<i>Bk.Blocking(G)</i> → {3} ← <i>Bk.Blocking(B)</i>
<i>IC.Unaware(G)</i> → {12} ← <i>Bk.Blocking(B)</i>
<i>IC.Unaware(G)</i> → {13} ← <i>Bk.Blocking(B)</i>
<i>IC.Unaware(G)</i> → {1} ← <i>CT.Linear(B)</i>
<i>IC.Unaware(G)</i> → {7} ← <i>DSM.Local(B)</i>
<i>IC.Unaware.(G)</i> → {5} ← <i>DT.Linear(B)</i>
<i>IC.Unaware(G)</i> → {1} ← <i>SCT.Explicit(B)</i>
<i>IC.Unaware(G)</i> → {12} ← <i>SCT.Explicit(B)</i>
<i>IC.Unaware(G)</i> → {5} ← <i>SDT.Explicit(B)</i>
<i>IC.Unaware(G)</i> → {13} ← <i>SDT.Explicit(B)</i>
<i>SCT.Implicit(G)</i> → {3} ← <i>Bk.Blocking(B)</i>
<i>SCT.Implicit(G)</i> → {1} ← <i>CT.Linear(B)</i>
<i>SCT.Implicit(G)</i> → {1} ← <i>SCT.Explicit(B)</i>
<i>SCT.Implicit(G)</i> → {2} ← <i>SCT.Explicit(B)</i>
<i>SDT.Implicit-Continuous(G)</i> → {5} ← <i>DT.Linear(B)</i>
<i>SDT.Implicit-Continuous(G)</i> → {5} ← <i>SDT.Explicit(B)</i>
<i>SDT.Implicit-Continuous(G)</i> → {6} ← <i>SDT.Explicit(B)</i>
<i>SDT.Implicit-Continuous(G)</i> → {11} ← <i>SDT.Explicit(B)</i>

Table 7: Gzip – Filter B PAIs

As seen in Table 6, only three conflicts remain after examining control and data exchange. However, Table 7 has fifteen conflicts remaining. This is mainly due to the implicit nature of gzip control and data transfer expectations. The filters both expect to be passed and to pass data and control explicitly. Enabling gzip to transfer explicitly to Filter B involves more integration functionality than simply simulating an implicit transfer from Filter A to gzip.

Step 1.5 Apply the minimization rules. Applying the minimization rules compresses the initially detected set of PAIs into a minimal, understandable set. Sometimes the minimization rules can be reapplied to the final set.

Step 1.5.1 Apply the characteristic additivity rule. To illustrate the characteristic additivity rule, we apply it to the remaining PAIs from Tables 6 and 7. Examining the set, we observe that the rule cannot be applied to PAIs resulting from comparing Filter A and gzip.

However, characteristic additivity can be applied to PAIs resulting from interaction between gzip and Filter B (see left-hand column of Table 8). We obtain the PAIs in the right-hand column.

Original PAI Set	Applying Characteristic Additivity
$IC.Unaware(G) \rightarrow \{12\} \leftarrow Bk.Blocking(B)$ $IC.Unaware(G) \rightarrow \{12\} \leftarrow SCT.Explicit(B)$	$IC.Unaware(G) \rightarrow \{12\} \leftarrow Bk.Blocking(B),$ $SCT.Explicit(B)$
$IC.Unaware(G) \rightarrow \{13\} \leftarrow Bk.Blocking(B)$ $IC.Unaware(G) \rightarrow \{13\} \leftarrow SDT.Explicit(B)$	$IC.Unaware(G) \rightarrow \{13\} \leftarrow Bk.Blocking(B),$ $SDT.Explicit(B)$
$IC.Unaware(G) \rightarrow \{1\} \leftarrow CT.Linear(B)$ $IC.Unaware(G) \rightarrow \{1\} \leftarrow SCT.Explicit(B)$	$IC.Unaware(G) \rightarrow \{1\} \leftarrow CT.Linear(B),$ $SCT.Explicit(B)$
$SCT.Implicit(G) \rightarrow \{1\} \leftarrow CT.Linear(B)$ $SCT.Implicit(G) \rightarrow \{1\} \leftarrow SCT.Explicit(B)$	$SCT.Implicit(G) \rightarrow \{1\} \leftarrow CT.Linear(B),$ $SCT.Explicit(B)$
$IC.Unaware.(G) \rightarrow \{5\} \leftarrow DT.Linear(B)$ $IC.Unaware(G) \rightarrow \{5\} \leftarrow SDT.Explicit(B)$	$IC.Unaware.(G) \rightarrow \{5\} \leftarrow DT.Linear(B),$ $SDT.Explicit(B)$
$SDT.Implicit-Continuous(G) \rightarrow \{5\} \leftarrow$ $DT.Linear(B)$ $SDT.Implicit-Continuous(G) \rightarrow \{5\} \leftarrow$ $SDT.Explicit(B)$	$SDT.Implicit-Continuous(G) \rightarrow \{5\} \leftarrow$ $DT.Linear(B), SDT.Explicit(B)$

Table 8: Gzip – Filter B: Characteristic Additivity

Step 1.5.2 Apply the conflict union rule. Using the conflict union rule, we combine the two PAIs resulting from Filter A and gzip interaction (see left-hand column of Table 9). The right-hand column holds the transformed PAIs.

Original PAI Set	Applying Characteristic Additivity
$SDT.Explicit(A) \rightarrow \{6\} \leftarrow SDT.Implicit-$ $Continuous(G)$ $SDT.Explicit(A) \rightarrow \{11\} \leftarrow SDT.Implicit-$ $Continuous(G)$	$SDT.Explicit(A) \rightarrow \{6,11\} \leftarrow SDT.Implicit-$ $Continuous(G)$

Table 9: Filter A – Gzip: Conflict Union

The union rule cannot be applied to any PAIs resulting from interaction between gzip and Filter B.

Step 1.6 Examine the remaining PAIs in more depth. Given a minimal set of PAIs it is easier to examine their implications and potential resolutions. In this step we describe the different PAIs and identify preliminary integration solutions needed to resolve the conflicts. We group the PAIs to simplify the discussion of the integration solution requirements. Generally, there is no preset method for grouping. We first consider the PAIs between Filter A and gzip, followed by the PAIs between gzip and Filter B.

Group 1

$$SDT.Explicit(A) \rightarrow \{6,11\} \leftarrow SDT.Implicit-Continuous(G)$$

Description: Filters A's data is passed discretely and explicitly. Gzip uses incremental data transfer via pipes. The conflict occurs because gzip expects to be passed data via pipes in a stream and processes the data according to that assumption but Filter A passes the entire set at once. Moreover, Filter A must direct its pass but gzip does not expect to receive data from a known source.

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

Group 2

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

Description: Filter A makes direct function calls. However, gzip is called implicitly with a standard UNIX command. Since gzip does not expect direct calls, Filter A has no destination for its control exchange.

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

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

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

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

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

Description: Filter B blocks and expects to be passed control and data explicitly. Thus, it will attempt to handshake with Gzip. Gzip has no knowledge of other components in the system. After it 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, allowing a simulated handshake with gzip.

Group 4

$$SDT.Implicit-Continuous(G) \rightarrow \{11\} \leftarrow SDT.Explicit(B)$$

Description: Gzip expects to transfer data implicitly and incrementally using a bounded pipe. Filter B expects to be passed data discretely. The conflict occurs because Filter B expects to be passed the entire set, but gzip simply outputs its data incrementally onto its standard output pipe.

Requirements: The integration solution needs to consolidate gzip's incremental output as the pipe fills. Once gzip completes its output data to the pipe, the integration solution must pass it discretely and explicitly to Filter B's local data storage.

Group 5

$$IC.Unaware(G) \rightarrow \{1\} \leftarrow CT.Linear(B), SCT.Explicit(B)$$

$$SCT.Implicit \rightarrow \{1\} \leftarrow CT.Linear(B), SCT.Explicit(B)$$

$$IC.Unaware(G) \rightarrow \{5\} \leftarrow DT.Linear(B), SDT.Explicit(B)$$

$$SDT.Implicit-Continuous \rightarrow \{5\} \leftarrow DT.Linear(B), SDT.Explicit(B)$$

Description: A linear topology typically has a single input point and a single output point. In this case, Filter B assumes a direct pass of control and data to its input point from a known source. However, gzip does not exhibit that behavior. It has no knowledge of Filter B and has no means to directly target the input point. 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 and data to Filter B's interface using the expected function call after gzip has fully completed its processing.

Upon completing phase one, we have identified the reasons underlying the PAIs. This serves two purposes. The first is that it transforms the PAIs from "potential" to "actual". In some cases, additional PAIs will be eliminated because the assumptions implied in the characteristic values do not hold. The second purpose is to identify preliminary integration solution requirements. The requirements are directly connected to the cause of the PAIs. Moreover, they can be viewed in a segmented way to minimize complexity in an effort to obtain significant functional requirements.

Had we not grouped the PAIs, we would need a step that examines the integration solution requirements and eliminates redundancy. By grouping as we did, across analogous conflicts, we address most of the redundancy issues.

Phase 2: Application-Component Interactions

Problematic interactions that result from component-component comparisons do not always comprise the final set of conflicts. Phase 2 can uncover additional PAIs that result from application-component interaction.

Step 2.1 Determine the characteristic/value sets for the application. We assign values to the characteristics of the application that represent its configuration and coordination requirements. Often requirements are malleable, especially if the application is an initial integration effort. However, within an established integrated system, the application

requirements may be solidified to minimize cascading conflicts when a new component is inserted. Generally, this phase follows the same steps as in Phase 1 but is targeted toward application analysis.

Prior to any component-component interaction analysis, it is natural to assume the Compressing Proxy application values will mimic those of the CERN server, since gzip is being “inserted.” Therefore, we set these values appropriately in Table 10. The architecture of the CERN server is synchronous. One filter does not begin execution until its predecessor finishes [SC97].

CHARACTERISTICS	COMPRESSING PROXY
<i>Control Structure</i>	Single-Thread
<i>Control Topology</i>	Linear
<i>Data Topology</i>	Linear
<i>Synchronization</i>	Synchronous

Table 10: Application Characteristics Values

Step 2.2 Notate the application-component PAIs using bipartite graphs. By using bipartite conflict graphs, we identify the following PAIs,

$$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow IC.Unaware(G)$$

$$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow SCT.Implicit(G)$$

$$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow SDT.Implicit-Continuous(G)$$

Step 2.3 Apply the notation rules. Applying the notation rules to the application PAIs highlights the importance of certain conflicts, while at the same time making the assessment simpler and its representation clearer. Though Steps 2.3.1 and 2.3.3 are addressed in the subsequent sections, neither the conflict union rule (Section 2.3.3) nor

the component union rule (Section 2.3.1) can compress the current set of application-component PAIs.

Step 2.3.1 Apply the component union rule. Using the component union rule, we cannot combine any PAIs. An example of component union can be seen in Chapter V.

Step 2.3.2 Apply the component additivity rule. Using the component additivity rule, we combine together the three PAIs identified in Step 2.2 as a single PAI represented in the right column of Table 11. This immediately highlights a problem between the synchronous requirement of the application and gzip.

Original PAI Set	After Applying Component Additivity
$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow IC.Unaware(G)$ $Sn.Synchronous(App) \rightarrow \{3\} \leftarrow SCT.Implicit(G)$ $Sn.Synchronous(App) \rightarrow \{3\} \leftarrow SDT.Implicit-Continuous(G)$	$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow$ $IC.Unaware(G), SCT.Implicit, SDT.Implicit-Continuous(G)$

Table 11: Applying Component Additivity

Step 2.3.3 Apply the conflict union rule. The conflict union rule cannot be applied because the final result of Step 2.3.2 is a single PAI.

Step 2.4 Examine the remaining PAIs in more depth. In this step, we take the PAI at the end of Step 2.3 and analyze it further. As in component-component interaction analysis, it is possible for a PAI to be irrelevant. However, in the Compressing Proxy, the PAI in Table 11's right column leads to the integration solution requirements set by the previous phase. A synchronous application requires its components to handshake. Because gzip passes its data and control implicitly, unaware of the filters, it does not behave as

required. The integration solution must, therefore, wrap gzip to simulate a synchronous interchange of data and control.

Phase 3: Application-Integration Interactions

The results of Phases 1 and 2 include multiple, generic, integration solution requirements. In Phase 3, we take a global view of these requirements toward consolidating them into an integration architecture that provides a consolidate functional resolution for PAIs identified.

Step 3.1 Assessing the integration requirements as a whole. Groups 1 and 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 too must run in a separate process from gzip.

The final integration solution is therefore comprised of two separate processes (one for input and one for output) that communicate with gzip. Gzip must be able to read and write incrementally from its input and output, and these processes must execute concurrently. After all the output has been gathered by the output process, it simulates a synchronous exchange with Filter B (according to the assessment in Phase 2).

Step 3.2 Re-examining the application requirements. Even when an integration solution can be constructed to satisfy the requirements, there is still potential for conflicts between

the integration solution requirements and the original application requirements. When this occurs, either changes are made to the application requirements, given the extent of their malleability, or the integration solution requirements are amended. However, such modifications must be controlled, so they do not cause new conflicts.

In the Compressing Proxy, the need for the two concurrent processes as a result of Step 3.1 is in direct conflict with the application requirements for a single-threaded control structure and a linear data topology. Recall, that these requirements were formulated based on the CERN server architecture and they are not necessarily binding. In fact, the application requirements can be changed to allow concurrent processes, as well as an arbitrary data topology, without any detrimental effects. The end result is the implementation of two concurrent processes to correctly direct input and output to and from gzip. This is exactly the solution that CHAM analyzed and which was proven to be deadlock free [CIW99, IWY00].

In larger applications, any changes to the requirements would require further assessment to ensure no new PAIs are detected. However, only the changed characteristics need to be reevaluated. Once characteristics cease to change in a process iteration, it reaches a stable state and the final solution can be constructed. In the Compressing Proxy, the integration solution is able to fulfill the application requirements with no further conflicts.

CHAPTER V

USING THE NOTATION FOR BLACK BOX INTEGRATION

The software architecture of black box components is often hidden from the view of users. However, a partial subset of the characteristics can often be valued from documentation, component interfaces and their observed behavior. In this chapter, we show how our notation and process can also be used to assess integration problems when only limited component information is available. This might not provide the developer all of the pertinent interoperability problems but it does identify those related to available characteristics values – still providing a savings in development time and cost.

In this chapter, we demonstrate the usefulness of our pre-integration analysis given partial characteristics for closed software components. We use an industrial example that we have analyzed and implemented given company specifications. This problem demonstrates the use of minimization rules more thoroughly than previous example. The same process is used as for the model problem in Chapter IV.

TradeMaster is an application to adapt internal processes for managing commodity trading online. Many of the needed independent software components (which are black box) are in place, but are not well connected. TradeMaster must be able to

accommodate customers with real-time trading while still processing the information from manual trades. Prior to integrating the above components, all information passing was done via phone, fax, email, and FTP.

The components to be integrated are *Market*, *Risk*, and *Financial* – all independently executing, black box components. Market is comprised of a national commodities market information module, a database to house all of the bid/ask requests put to the market, and an SMTP server for dynamic messaging. Risk uses a history of bid/ask requests to assess the financial exposure of the company. Financial handles customer billing information, as well as, corporate-wide financial information.

To perform Risk analysis, the Risk component needs to call Market to request the current store of bid/ask requests, sequencing any concurrent communications such as a SMTP packet being sent. Market must then route these records back to Risk for financial exposure analysis. The Financial component needs to be updated by the Market component when a purchase has been made. Risk and Financial must be able to exchange information about former purchases and risk analysis. Figure 9 shows how the user should interact with the integration solution instead of a specific component.

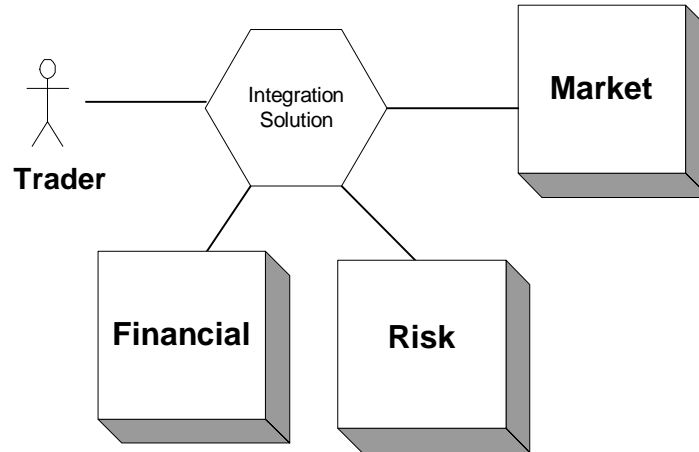


Figure 9: TradeMaster

In the remainder of the section, we demonstrate the use of our notation and minimization rules in the pre-integration process using the TradeMaster example. This illustrates how a limited set of characteristics can identify interoperability problems that need specific resolution in the integration architecture.

Phase 1: Component-Component Interactions

This phase covers the first interaction type described in Section 3.1. As before the phases encompasses six steps, which examine the direct interaction among components.

Step 1.1 Determine the characteristic/value sets for each component. We assign a value to the known characteristics for each component participating in the integration. Only two characteristics can be assigned values with some certainty given the available information (Table 12). The components are all non-blocking, suspending their thread of control when they finish execution. The Risk and Financial components only allow single-threaded control, while Market has the ability to execute multiple, concurrent threads.

CHARACTERISTICS	FINANCIAL	MARKET	RISK
<i>Blocking</i>	Non-Blocking	Non-Blocking	Non-Blocking
<i>Control Structure</i>	Single-Thread	Concurrent	Single-Thread

Table 12: Component Characteristic Values

Step 1.2 Graph the basic architecture of the application indicating the participating components, their connectivity and directionality of data and control flow in the system. We graph the connectivity and directionality of the interaction between Market, Risk and Financials. The graph is a complete graph and can be seen in Figure 10.

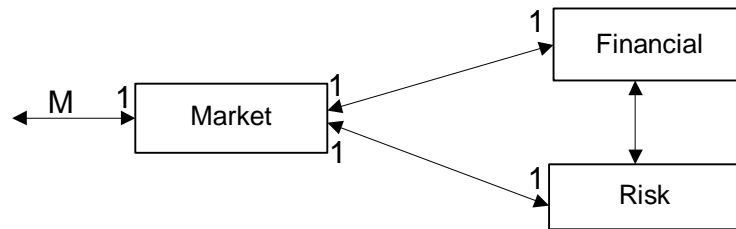


Figure 10: TradeMaster Architecture

Step 1.3 Notate the component-component PAIs using bipartite graphs. We now construct a bipartite conflict graph for pairwise assessment of components. There should be a comparison for every pair of components directly communicating; in this instance, all components communicate. Figure 11 displays only those lines in the bipartite conflict graphs that represent conflicts between Market and Risk, Market and Financial, and Risk and Financial.

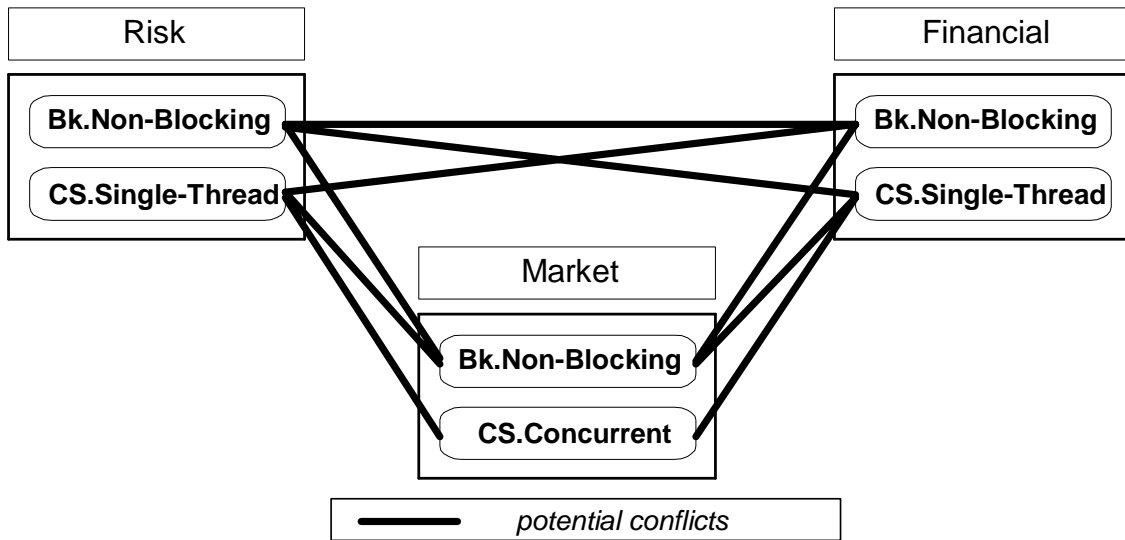


Figure 11: Bipartite Conflict Graphs

The potential conflicts identified by the bipartite graphs in Figure 11 are presented as PAIs for each component-component interaction in Tables 13, 14 and 15

Market - Risk
$CS.Concurrent(M) \rightarrow \{4\} \leftarrow CS.Single-Thread(R)$ $Bl.Non-Blocking(M) \rightarrow \{3\} \leftarrow Bl.Non-Blocking(R)$ $Bl.Non-Blocking(M) \rightarrow \{8\} \leftarrow Bl.Non-Blocking(R)$ $Bl.Non-Blocking(M) \rightarrow \{3\} \leftarrow CS.Single-Thread(R)$ $Bl.Non-Blocking(M) \rightarrow \{8\} \leftarrow CS.Single-Thread(R)$

Table 13: Market – Risk PAIs

Market – Financial
$CS.Concurrent(M) \rightarrow \{4\} \leftarrow CS.Single-Thread(R)$ $Bl.Non-Blocking(M) \rightarrow \{3\} \leftarrow Bl.Non-Blocking(R)$ $Bl.Non-Blocking(M) \rightarrow \{8\} \leftarrow Bl.Non-Blocking(R)$ $Bl.Non-Blocking(M) \rightarrow \{3\} \leftarrow CS.Single-Thread(R)$ $Bl.Non-Blocking(M) \rightarrow \{8\} \leftarrow CS.Single-Thread(R)$

Table 14: Market – Financial PAIs

Risk – Financial
$Bl.Non-Blocking(R) \rightarrow \{3\} \leftarrow Bl.Non-Blocking(F)$ $Bl.Non-Blocking(R) \rightarrow \{8\} \leftarrow Bl.Non-Blocking(F)$ $Bl.Non-Blocking(R) \rightarrow \{3\} \leftarrow CS.Single-Thread(F)$ $Bl.Non-Blocking(R) \rightarrow \{8\} \leftarrow CS.Single-Thread(F)$ $CS.Single-Thread(R) \rightarrow \{3\} \leftarrow Bl.Non-Blocking(F)$ $CS.Single-Thread(R) \rightarrow \{8\} \leftarrow Bl.Non-Blocking(F)$

Table 15: Risk – Financial PAIs

Step 1.4 Eliminate those PAIs from the pairwise assessments in which there is no data and/or control exchange. We cannot eliminate any PAIs due to directionality since the connection of components is a complete graph and all connections are bi-directional (refer to Figure 10 in step 1.2).

Step 1.5 Apply the minimization rules. We now apply the minimization rules to compress the PAIs into a more manageable and interdependent set.

Step 1.5.1 Apply the characteristic additivity rule. We apply the characteristic additivity rule to the PAIs identified in Step 1.3. Tables 16, 17, and 18 combine a subset of the original PAIs on the left to form the PAIs on the right from interactions between Market and Risk, Market and Financial, and Risk and Financial, respectively.

Original PAI Set	After Applying Characteristic Additivity
$Bl.Non-Blocking(M) \rightarrow \{3\} \leftarrow CS.Single-Thread(R)$ $Bl.Non-Blocking(M) \rightarrow \{3\} \leftarrow Bl.Non-Blocking(R)$	$Bl.Non-Blocking(M) \rightarrow \{3\} \leftarrow CS.Single-Thread(R), Non-Blocking(R)$
$Bl.Non-Blocking(M) \rightarrow \{8\} \leftarrow CS.Single-Thread(R)$ $Bl.Non-Blocking(M) \rightarrow \{8\} \leftarrow Bl.Non-Blocking(R)$	$Bl.Non-Blocking(M) \rightarrow \{8\} \leftarrow CS.Single-Thread(R), Bl.Non-Blocking(R)$

Table 16: Market – Risk: Characteristic Additivity

Original PAI Set	After Applying Characteristic Additivity
<i>Bl.Non-Blocking(M) → {3} ← CS.Single-Thread(F)</i> <i>Bl.Non-Blocking(M) → {3} ← Bl.Non-Blocking(F)</i>	<i>Bl.Non-Blocking(M) → {3} ← CS.Single-Thread(F), Non-Blocking(F)</i>
<i>Bl.Non-Blocking(M) → {8} ← CS.Single-Thread(F)</i> <i>Bl.Non-Blocking(M) → {8} ← Bl.Non-Blocking(F)</i>	<i>Bl.Non-Blocking(M) → {8} ← CS.Single-Thread(F), Bl.Non-Blocking(F)</i>

Table 17: Market – Financial: Characteristic Additivity

Original PAI Set	After Applying Characteristic Additivity
<i>Bl.Non-Blocking(R) → {3} ← CS.Single-Thread(F)</i> <i>Bl.Non-Blocking(R) → {3} ← Bl.Non-Blocking(F)</i>	<i>Bl.Non-Blocking(R) → {3} ← CS.Single-Thread(F), Non-Blocking(F)</i>
<i>Bl.Non-Blocking(R) → {8} ← CS.Single-Thread(R)</i> <i>Bl.Non-Blocking(R) → {8} ← Bl.Non-Blocking(F)</i>	<i>Bl.Non-Blocking(R) → {8} ← CS.Single-Thread(F), Bl.Non-Blocking(F)</i>

Table 18: Risk - Financial: Characteristic Additivity

Note the similarity of the conflicts across the component characteristics. This is due to the complete graph communication and the autonomous operation aspects of each component.

Step 1.5.2 Apply the conflict union rule. We further combine the above PAIs by applying conflict union. The results can be seen in Tables 19, 20, and 21.

Original PAI Set	After Applying Conflict Union
<i>Bl.Non-Blocking(M) → {3} ← CS.Single-Thread(R), Non-Blocking(R)</i> <i>Bl.Non-Blocking(M) → {8} ← CS.Single-Thread(R), Bl.Non-Blocking(R)</i>	<i>Bl.Non-Blocking(M) → {3,8} ← CS.Single-Thread(R), Non-Blocking(R)</i>

Table 19: Market – Risk: Conflict Union

Original PAI Set	After Applying Conflict Union
<i>Bl.Non-Blocking(M) → {3} ← CS.Single-Thread(F), Non-Blocking(F)</i> <i>Bl.Non-Blocking(M) → {8} ← CS.Single-Thread(F), Bl.Non-Blocking(F)</i>	<i>Bl.Non-Blocking(M) → {3,8} ← CS.Single-Thread(F), Non-Blocking(F)</i>

Table 20: Market – Financial: Conflict Union

Original PAI Set	After Applying Conflict Union
<i>Bl.Non-Blocking(R) → {3} ← CS.Single-Thread(F), Non-Blocking(F)</i> <i>Bl.Non-Blocking(F) → {8} ← CS.Single-Thread(R), Bl.Non-Blocking(R)</i>	<i>Bl.Non-Blocking(R) → {3,8} ← CS.Single-Thread(F), Non-Blocking(F)</i>
<i>CS.Single-Thread(R) → {3} ← Non-Blocking(F)</i> <i>CS.Single-Thread(R) → {8} ← Non-Blocking(F)</i>	<i>CS.Single-Thread(R) → {3,8} ← Non-Blocking(F)</i>

Table 21: Financial – Risk: Conflict Union

Step 1.6 Examine the remaining PAIs in more depth. We now take the PAIs and describe how the problem exists in the context of the TradeMaster example, producing generic integration requirements or concerns for resolution of the PAI.

We again simplify the discussion by grouping the PAIs. The group is across conflicts in this example. Each group has all components represented. This is appealing because it indicates the resolution functionality will be centralized.

Group 1

$$CS.Concurrent(M) \rightarrow \{4\} \leftarrow CS.Single-Thread(R)$$

$$CS.Concurrent(M) \rightarrow \{4\} \leftarrow CS.Single-Thread(F)$$

Description: The market component runs concurrent processes. Multiple, concurrent processes may be simultaneously passed to Risk or Financial. These processes must be sequenced correctly.

Requirements: To solve the sequencing problems, a control mechanism is needed to schedule the processes for acceptance by the single-threaded components. Since they cannot accept multiple processes directly, the control must be sequenced external to the components.

Group 2

$$Bl.Non-Blocking(M) \rightarrow \{3,8\} \leftarrow Bl.Non-Blocking(F), CS.Single-Thread(F)$$

$$Bl.Non-Blocking(F) \rightarrow \{3,8\} \leftarrow Bl.Non-Blocking(R), CS.Single-Thread(R)$$

$$Bl.Non-Blocking(R) \rightarrow \{3,8\} \leftarrow CS.Single-Thread(R), Non-Blocking(F)$$

$$CS.Single-Thread(R) \rightarrow \{3,8\} \leftarrow Bl.Non-Blocking(F)$$

Description: If one component executes independently with no means of interruption and the other expects to pass control to a waiting thread, they will have difficulty communicating control and data. In addition, because a non-blocking component can execute independently, possibly doing unrelated work to the other component, data can be adversely permuted.

Requirements: The integration solution needs to simulate a handshake, and then pass the control to the non-blocking components. In addition, a control mechanism is needed to ensure that no illegal data modifications are performed.

Phase 2: Application-Component Interactions

We now look at PAIs that result from application-component interaction.

Step 2.1 Determine the characteristic/value sets for the application. We assign values to the characteristics of the application that offer a global view of the requirements. Table 22 shows the application characteristics for TradeMaster. The individual components must operate concurrently and asynchronously. Data and control are passed internally without any stylistic restrictions.

CHARACTERISTICS	TRADEMASTER
<i>Control Structure</i>	Concurrent
<i>Control Topology</i>	Arbitrary
<i>Data Topology</i>	Arbitrary
<i>Synchronization</i>	Asynchronous

Table 22: TradeMaster Application Characteristics Values

Step 2.2 Notate the application-component PAIs using bipartite graphs. By using bipartite conflict graphs, we identify the following PAIs,

Application – Component
$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CS.Concurrent(M)$
$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CS.Single-Thread(R)$
$CS.Concurrent(App) \rightarrow \{8\} \leftarrow CS.Single-Thread(R)$
$CT.Arbitrary(App) \rightarrow \{1\} \leftarrow CS.Single-Thread(R)$
$CT.Arbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(R)$
$DT.Arbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(R)$
$DT.Arbitrary(App) \rightarrow \{5\} \leftarrow CS.Single-Thread(R)$
$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CS.Single-Thread(F)$
$CS.Concurrent(App) \rightarrow \{8\} \leftarrow CS.Single-Thread(F)$
$CT.Arbitrary(App) \rightarrow \{1\} \leftarrow CS.Single-Thread(F)$
$CT.Arbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(F)$
$DT.Arbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(F)$
$DT.Arbitrary(App) \rightarrow \{5\} \leftarrow CS.Single-Thread(F)$

Table 23: Application-Component PAIs

Step 2.3 Apply the notation rules. We now apply the application-component rules, consolidating PAIs.

Step 2.3.1 Apply the component union rule. Using the component additivity rule, we combine the PAIs in Table 24's left column into the PAIs in the right column.

Original PAI Set	Applying Component Union Rule
$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CS.Single-Thread(R)$ $CS.Concurrent(App) \rightarrow \{4\} \leftarrow CS.Single-Thread(F)$	$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CS.Single-Thread(R,F)$
$CS.Concurrent(App) \rightarrow \{8\} \leftarrow CS.Single-Thread(R)$ $CS.Concurrent(App) \rightarrow \{8\} \leftarrow CS.Single-Thread(F)$	$CS.Concurrent(App) \rightarrow \{8\} \leftarrow CS.Single-Thread(R,F)$
$DT.Arbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(R)$ $DT.Arbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(F)$	$DT.Arbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(R,F)$
$DT.Arbitrary(App) \rightarrow \{5\} \leftarrow CS.Single-Thread(R)$ $DT.Arbitrary(App) \rightarrow \{5\} \leftarrow CS.Single-Thread(F)$	$DT.Arbitrary(App) \rightarrow \{5\} \leftarrow CS.Single-Thread(R,F)$
$CT.Arbitrary(App) \rightarrow \{1\} \leftarrow CS.Single-Thread(R)$ $CT.Arbitrary(App) \rightarrow \{1\} \leftarrow CS.Single-Thread(F)$	$CT.Arbitrary(App) \rightarrow \{1\} \leftarrow CS.Single-Thread(R,F)$
$CT.Arbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(R)$ $CT.Arbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(F)$	$CT.Arbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(R,F)$

Table 24: Applying Component Union

Step 2.3.2 Apply the component additivity rule. We now apply the component additivity rule as shown in Table 25.

Original PAI Set	Applying Component Additivity
$CS.Concurrent(App) \rightarrow \{4\} \leftarrow$ $CS.Concurrent(M)$ $CS.Concurrent(App) \rightarrow \{4\} \leftarrow CS.Single-Thread(R,F)$	$CS.Concurrent(App) \rightarrow \{4\} \leftarrow$ $CS.Concurrent(M), CS.Single-Thread(R,F)$

Table 25: Applying Component Additivity

Step 2.3.3 Apply the conflict union rule. To compress the set further, we apply the conflict union rule. Refer to Table 26 for more detail.

Original PAI Set	Applying Conflict Union
$DT.Arbbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(R,F)$ $DT.Arbbitrary(App) \rightarrow \{5\} \leftarrow CS.Single-Thread(R,F)$	$DT.Arbbitrary(App) \rightarrow \{3,5\} \leftarrow CS.Single-Thread(R,F)$
$CT.Arbbitrary(App) \rightarrow \{1\} \leftarrow CS.Single-Thread(R,F)$ $CT.Arbbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(R,F)$	$CT.Arbbitrary(App) \rightarrow \{1,3\} \leftarrow CS.Single-Thread(R,F)$

Table 26: Applying Conflict Union

The final PAI set after applying the minimization rules is the following,

$$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CS.Concurrent(M), CS.Single-Thread(R,F)$$

$$DT.Arbbitrary(App) \rightarrow \{3,5\} \leftarrow CS.Single-Thread(R,F)$$

$$CT.Arbbitrary(App) \rightarrow \{1,3\} \leftarrow CS.Single-Thread(R,F)$$

$$CS.Concurrent(App) \rightarrow \{8\} \leftarrow CS.Single-Thread(R,F)$$

Applying the minimization rules compresses the initial set of 13 PAIs down to 4. The benefit of the minimization rules can be fully realized when there is a larger number of components being integrated, resulting in a much larger set of PAIs. For instance, a developer can see clearly the number of components affected by a conflict. This, in turn, can lead to an encompassing integration functionality. If only pairwise conflicts were understood, the larger architecture of the integration solution would be more difficult to visualize, design, and implement. We review these PAIs in more detail in the next step.

Step 2.4 Examine the remaining PAIs in more depth. We examine the final set of PAIs looking for specific conflicts that can help pinpoint a solution. For brevity, we group the PAIs based on their description and requirement similarities.

Group 1

$$DT.Arbitrary(App) \rightarrow \{3,5\} \leftarrow CS.Single-Thread(R,F)$$

$$CT.Arbitrary(App) \rightarrow \{1,3\} \leftarrow CS.Single-Thread(R,F)$$

Description: Financial and Risk have single-threaded control structures and expect control and data to be passed directly to particular interface points for exchanging control. An arbitrary application has no set pattern to where it passes its control, and therefore there is no guarantee that the application can direct control and data directly to Risk's and Financial's interface points. This inhibits the method by which these two components exchange control and data with other components in the application.

Requirements: An integration functionality is needed which gathers control and data from the components and delivers it to the appropriate destination. Moreover, it must simulate a handshake between the components communicating.

Group 2

$$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CS.Concurrent(M), CS.Single-Thread(R,F)$$

$$CS.Concurrent(App) \rightarrow \{8\} \leftarrow CS.Single-Thread(R,F)$$

Description: The concurrency requirement of TradeMaster interferes with the correct sequencing of control transfers to the components because it expects each component to

continue operation autonomously. Side effects of this problem are possible data inconsistencies, e.g. when updates are lost or if dirty reads occur. Market is concurrent and, therefore, sequencing problems can arise between it and the application requirements, as well as the two single-threaded components (Financial and Risk).

Requirement: A central piece providing concurrency control is necessary to facilitate control exchange. The data and control transfer must be sequenced properly, eliminating the chance of inappropriate access, which could cause inconsistent data.

Phase 3: Application-Integration Interactions

We now take a global view of the integration requirements with the goal of creating an integration architecture. The main requirements identified in Phases 1 and 2 are outlined below. We base our integration architecture on these requirements.

- A control mechanism external to the components to schedule the processes for acceptance by the single-threaded components.
- A control mechanism to ensure that no illegal data modifications are performed.
- Simulate a handshake with Risk and Financial, and then pass the control to the non-blocking components.
- An integration functionality, which gathers control and data from the Risk and Financial and delivers it to the appropriate destination.

Step 3.1 Assessing the integration requirements as a whole. Examining the generic integration requirements gathered from Phases 1 and 2, it is apparent that a central piece providing concurrency control is needed. This integration functionality is responsible for sequencing for the control transfers, while at the same time routing control and data to the appropriate receivers. For TradeMaster, we call this the Transaction Manager, which is placed between the user and the components. The Transaction Manager handles both concurrency and routing (using information available from its decision and data components).

A Mediator is instantiated at both Risk and Financials allowing them to communicate with each other and Market. The Mediators gather control and data from their associated components to place on a Message Bus. The Message Bus is responsible for delivering or retrieving data and control to the Transaction Manager. The architecture of the final integrated application can be seen in Figure 12.

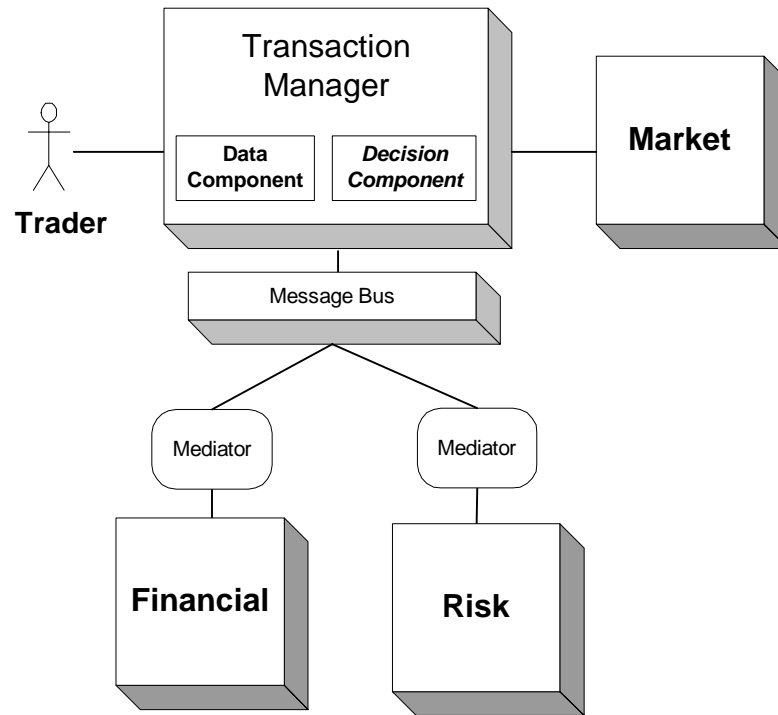


Figure 12: TradeMaster Integration Solution

Step 3.2 Re-examining the application requirements. In this study, the application requirements and integration solution are not in conflict. This is primarily due to the flexibility embodied in the application characteristics. The integration architecture handles any issues a rigid component (e.g. one with a single-threaded control structure) has. The integration solution allows the integrated application to maintain its initial characteristic values while at the same time solving all the identified PAIs.

CHAPTER VI

CONCLUSION AND FUTURE WORK

The practice of component-based software engineering is gathering momentum and needs techniques and tools to constrain and manage the discovery and analysis of interoperability conflicts. The causes of interoperability problems must be thoroughly addressed to make this form of system design reliable. The most manageable way to achieve this is to assess interoperability early in the design using architectural information as the basis. This allows interaction problems and their causes to be identified. They can then be used to facilitate the formation of the appropriate integration architectures.

In this thesis, we define and notate problematic architecture interactions, showing the architectural characteristics of the components involved in the conflict and the type of conflict they are causing. In addition, we define four information-preserving, minimization rules that, when applied to the PAIs, compresses the set. This leads to the formulation of the initial requirements for an integration architecture solution.

The assessment of conflicts between the application and integration solution requires further research, specifically in the characterization of the integration functionality [DG01]. A more formal approach is essential to ensure correctness when

complex applications are integrated. Furthermore, we are developing an assessment tool to detect PAIs and to maintain a history of their occurrence to allow for experimentation of new component insertion into an existing application [DG02a, JFDG01].

It is important that the initial integration is performed correctly to ensure the evolvability of the application. In cases where there are a vast number of components, many companies choose to integrate a subset first (usually those from a single business unit). The composite system is then deployed and remaining software components are later inserted incrementally. The objective of incremental integration of components is to make only minimal changes to the integration solution. By doing so, we maximize the possibility that more components can be easily inserted later. This approach to integrated application development stresses the importance of constructing a well understood and extendable integration solution. If the integration is poorly constructed - ignoring interoperability problems or solving them in an ad hoc manner - it can have a cascading effect, resulting in the dissolution of the current integration solution.

Having a high-level assessment that lays the foundation for design decisions proves valuable to companies that are striving to incorporate incremental development into their infrastructure. They can use the design history provided in the form of PAIs to evaluate the fit of a new component into the integrated application [JFDG01]. The results of a pre-integration assessment can also exist as one of the criteria used to evaluate a product when choosing between different components from disparate vendors, where the number of PAIs or their severity can be an indicator of its fit.

CHAPTER VII

REFERENCES

- [A96] Abd-Allah, A. Composing Heterogeneous Software Architectures. Ph. D. Dissertation, Computer Science, University of Southern California, 1996.
- [A97] Allen, R. A Formal Approach to Software Architecture. Ph. D. Dissertation, Computer Science, Carnegie Mellon University, 1996.
- [AG97] Allen, R., and Garlan, D. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodologies* (1997), 6(3): 213-249.
- [AAG95] Abowd, G., Allen, and R., Garlan, D. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodologies* (1995), 4(4): 319-364.
- [BCTW96] Barret, D., Clarke, L., Tarr, P., and Wise, A. A Framework for Event-Based Software Integration. *ACM Transactions on Software Engineering and Methodology* (1996), 5(4): 378--421.
- [BMRSS96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*: John Wiley & Sons. 1996.
- [C99] Charles, J. Middleware Moves to the Forefront. *IEEE Computer* (1999). 32(5): 17-19.
- [CIW99] Compare, D., Inverardi, P., and Wolf, A. Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming* (1999), 33(2): 101-131.
- [D01] Davis, L. Examining Software Architecture Property Interaction and the Influence of System Requirements. M.S. Thesis, Department of Mathematical and Computer Sciences, University of Tulsa, 2001.

- [DG01] Davis, L., and Gamble, R. Conflict Patterns: Toward Identifying Suitable Middleware. In, *3rd Int'l Conference on Information Reuse and Integration*, (2001). Las Vegas, NV.
- [DG02a] Davis, L., and Gamble, R. Identifying Evolvability for Integration. In, *International Conference on COTS-Based Software Systems*, (2002). Orlando, Florida.
- [DG02b] Davis, L., and Gamble, R. Ensuring QoS in Integrated Systems. SEAT-UTULSA-2002-1. Department of Mathematical and Computer Sciences, The University of Tulsa, 2001.
- [DPG00a] Davis, L., Payton, J., and Gamble, R. How System Architectures Impede Interoperability. In, *2nd International Workshop On Software and Performance*, (2000). Ottawa, Canada.
- [DPG00b] Davis, L., Payton, J., and Gamble, R. Toward Identifying The Impact Of COTS Evolution On Integrated Systems. In, *2nd Int'l Workshop on the Successful Development of COTS*, (2000). Limerick, Ireland.
- [DPG02] Davis, L., Payton, J., and Gamble, R. The Impact of Component Architectures on Interoperability. *Journal of Systems and Software* (2002), 61(1): 31-45.
- [DGPJU01] Davis, L., Gamble, R., Payton, J., Jónsdóttir, G., and Underwood, D. A Notation for Problematic Architecture Interactions. In, *3rd joint meeting of the European Software Engineering Conference, and ACM SIGSOFT's Symposium on the Foundations of Software Engineering*, (2001). Vienna, Austria.
- [G97] Gacek, C. Detecting Architectural Mismatches During Systems Composition. USC/CSE-97-TR-506. Ph. D. Dissertation, Computer Science, University of Southern California, Los Angeles, CA, 1997.
- [G98] Garlan, D. Higher-Order Connectors. In, *Workshop on Compositional Software Architectures*, (1998). Monterey, CA.
- [GAO95] Garlan, D., Allen, and A., Ockerbloom, J. Architectural Mismatch, or Why it is Hard to Build Systems out of Existing Parts. In, *17th International Conference on Software Engineering*, (1995). Seattle, WA.
- [GMW97] Garlan, D., Monroe, R., and Wile, D. ACME: An Architectural Description Language. In, *CASCON*, (1997). Toronto, Ontario.

- [GHJV95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns Elements of Reusable Object-Oriented Software: Addison-Wesley. 1995.
- [IWY00] Inverardi, P., Wolf, A., and Yankelevich, D. Static Checking of System Behaviors Using Derived Component Assumptions. *ACM Transactions on Software Engineering and Methodologies* (2000), 9(3): 239-272.
- [JG02] Jónsdóttir, G., and Gamble, R. Designing Secure Integration Architectures. SEAT-UTULSA-2002-2. Department of Mathematical and Computer Sciences, The University of Tulsa, 2002.
- [JFDG01] Jónsdóttir, G., Flagg, D., Davis, L., and Gamble, R. Integrating Components Incrementally for Composite Application Development. SEAT-UTULSA-2001-19. Department of Mathematical and Computer Sciences, The University of Tulsa, 2001.
- [JDFG02] Jónsdóttir, G., Davis, Flagg, D., and Gamble, R. Problematic Architecture Interactions. SEAT-UTULSA-2001-20. Department of Mathematical and Computer Sciences, The University of Tulsa, 2002.
- [KCBA97] Kazman, R., Clements, P., Bass, L., and Abowd, G. Classifying Architectural Elements as a Foundation for Mechanism Matching. In, *1st International Computer Software and Applications Conference*. (1997). Washington, D.C.
- [K99] Keshav, R. Architecture Integration Elements: Connectors that Form Middleware. M.S. Thesis, Department of Mathematical and Computer Sciences, University of Tulsa, 1999.
- [KG98] Keshav, R., Gamble, R. Towards a Taxonomy of Architecture Integration Strategies. In, *3rd International Software Architecture Workshop*, (1998). Orlando, Florida.
- [KG99] Kelkar, A., and Gamble, R. Understanding the Architectural Characteristics behind Middleware Choices. In, *1st International Conference in Information Reuse and Integration*, (1999). Atlanta, Georgia.
- [LV95] Luckham, D., and Vera, J. An Event-Based Architectural Definition Language. *IEEE Transactions on Software Engineering* (1995), 21(9): 717-734.
- [M94] Mularz, D. Pattern-Based Integration Architectures. In, *Pattern Languages of Programming (Plop)*, (1994). Monticello, Illinois.

- [MMP00] Mehta, N., Medvidovic, and N., Phadke, S. Towards a Taxonomy of Software Connectors. In, *22nd International Conference on Software Engineering*, (2000). Limerick, Ireland.
- [MRT99] Medvidovic, N., Rosenblum, and D., Taylor, R. A Language and Environment for Architecture-Based Software Development and Evolution. In, *21st International Conference on Software Engineering*, (1999). Los Angeles, CA.
- [MGR00] Medvidovic, N., Gamble, and R., Rosenblum, D. Towards Software Multioperability: Bridging Heterogeneous Software Interoperability Platforms. In, *4th International Software Architecture Workshop*, (2000). Limerick, Ireland.
- [MDEK95] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying Distributed Software Architectures. In, *5th European Software Engineering Conference*, (1995). Barcelona, Spain.
- [PW92] Perry, D., and Wolf, A. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* (1992), 17(4): 40-52.
- [PDUG01] Payton, J., Davis, L., Underwood, D., and Gamble, R. Using XML for an Architecture Interaction Conspectus. In, *23rd International Conference on Software Engineering, Workshops of XML Technologies and Software Engineering* (2001). Toronto, Canada.
- [PGKD00] Payton, J., Gamble, R., Kimsen, and S., Davis, L. The Opportunity for Formal Models of Integration. In, *2nd Int'l Conference on Information Reuse and Integration*, (2000). Honolulu, Hawaii.
- [PJFG02] Payton, J., Jónsdóttir, G., Flagg, D., and Gamble, R.F. Merging Integration Solutions for Architecture and Security Mismatch. In, *International Conference on COTS-Based Software Systems*, (2002). Orlando, Florida.
- [SC97] Shaw, M., and Clements, P. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In, *1st International Computer Software and Applications Conference*, (1997). Washington, D.C.
- [SG96] Shaw, M., and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall. 1996.
- [SIT97] Sitaraman, R. Integration of Software Systems at an Abstract Architectural Level. M.S. Thesis, Department of Mathematical and Computer Sciences, University of Tulsa, 1997.

- [SG01] Spitznagel, B., and Garlan, D. A Compositional Approach for Constructing Connectors. In, *Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, (2001). Amsterdam , The Netherlands.
- [STI97] Stiger, P. An Assessment of Architectural Styles and Integration Components. M.S. Thesis, Department of Mathematical and Computer Sciences, University of Tulsa, 1997.
- [YBB99] Yakimovich, D., Bieman, and J.,Basili, V. Software Architecture Classification for Estimating The Cost Of COTS Integration. In, *21st International Conference on Software Engineering*, (1999). Los Angeles, CA.

APPENDIX A

In this appendix, we list all of PAIs used for a pre-integration assessment and their explanation. We are repeating the conflict categories from Table 1 for quick references. The PAIs are ordered in separate tables according to the characteristics causing the problems. The definition of each characteristic is included above each table. Please refer to [D01] for more detailed explanation.

#	CONFLICT CATEGORY	CONFLICT 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 is not processed in the expected manner.
9	Invalid data	Data formats are incompatible.
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.

Table 27: Conflict Categories

Component-Component PAIs

Blocking (Bl): Whether or not the thread of control is suspended.

PAIs	PAI Description
$Bl.Blocking(C_1) \rightarrow \{3\} \leftarrow Bl.Blocking(C_2)$	If both components are waiting for the other to initiate execution, they will not be able to communicate.
$Bl.Blocking(C_1) \rightarrow \{3\} \leftarrow Bl.Non-Blocking(C_2)$ $Bl.Blocking(C_1) \rightarrow \{8\} \leftarrow Bl.Non-Blocking(C_2)$	If a component is blocked and waiting for a non-blocking component that is executing to initiate execution these two components might be unable to exchange control. There is no guarantee that the non-blocking component has knowledge that it should communicate with the blocking component. In addition, the non-blocking component might modify the blocked component's data causing the data to become inconsistent.
$Bl.Blocking(C_1) \rightarrow \{3\} \leftarrow CS.Concurrent(C_2)$ $Bl.Blocking(C_1) \rightarrow \{4\} \leftarrow CS.Concurrent(C_2)$	If one component blocks, waiting for the other component to initiate execution, and the other component has multiple, concurrent threads running with no focus of control, the concurrent component will likely have difficulty knowing where to pass control, inhibiting control transfer. Moreover many threads of the concurrent component may want to communicate at once causing sequencing problems.
$Bl.Blocking(C_1) \rightarrow \{2\} \leftarrow CT.Arbitrary(C_2)$ $Bl.Blocking(C_1) \rightarrow \{3\} \leftarrow CT.Arbitrary(C_2)$	A component with an arbitrary control topology has no means of passing control to a specific destination. If a component blocks, waiting for another component's communication to initiate its execution, there is no guarantee that these components will be able to exchange control.
$Bl.Blocking(C_1) \rightarrow \{12\} \leftarrow IC.Unaware(C_2)$ $Bl.Blocking(C_1) \rightarrow \{13\} \leftarrow IC.Unaware(C_2)$	If a component blocks, waiting for the other component to initiate execute, but the other component is unaware of any other components in the system, neither component will be able to commence control and data exchange.
$Bl.Blocking(C_1) \rightarrow \{3\} \leftarrow SCT.Implicit(C_2)$	A component with implicit control transfer will simply broadcast its control. A blocked component will wait for another component to handshake. Implicit control transfer does not involve handshaking, inhibiting control transfer.
$Bl.Blocking(C_1) \rightarrow \{3\} \leftarrow SCT.None(C_2)$ $Bl.Blocking(C_1) \rightarrow \{12\} \leftarrow SCT.None(C_2)$	A blocked component will wait for another component to handshake. A component that cannot transfer any control cannot initialize a control transfer and is therefore not capable of rendezvousing with the blocked component.
$Bl.Non-Blocking(C_1) \rightarrow \{3\} \leftarrow Bl.Non-Blocking(C_2)$ $Bl.Non-Blocking(C_1) \rightarrow \{8\} \leftarrow Bl.Non-Blocking(C_2)$	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. Moreover, because a non-blocking component can execute independently, possibly doing unrelated work to the other

	component, data can be adversely permuted.
<i>Bl.Non-Blocking(C₁) → {3} ← CS.Single-Thread(C₂)</i> <i>Bl.Non-Blocking(C₁) → {8} ← CS.Single-Thread(C₂)</i>	<p>If one component executes independently with no means of interruption, and the other expects to pass control to a waiting thread, they will have difficulty communicating control and data. In addition, because a non-blocking component can execute independently, possibly doing unrelated work to the other component, data can be adversely permuted or a gap can result.</p>

Table 28: Blocking PAIs

Control Structure (CS): The structure that governs the execution in the system.

PAIs	PAI Description
$CS.Single-Thread(C_1) \rightarrow \{3\} \leftarrow Bl.Non-Blocking(C_2)$ $CS.Single-Thread(C_1) \rightarrow \{8\} \leftarrow Bl.Non-Blocking(C_2)$	See Table 28.
$CS.Single-Thread(C_1) \rightarrow \{4\} \leftarrow CS.Concurrent(C_2)$	A single-threaded component, which must receive discrete transfers of control, could be incorrectly or pre-maturely initiated by the multiple and simultaneously control transfers of a concurrent component. This is due to multiple, concurrent threads, broadcasting control in no particular sequence.
$CS.Single-Thread(C_1) \rightarrow \{1\} \leftarrow CT.Arbitrary(C_2)$ $CS.Single-Thread(C_1) \rightarrow \{2\} \leftarrow CT.Arbitrary(C_2)$	A single threaded component expects to be passed control directly to its entry point and it passes control in the same manner. A component with arbitrary control topology has no such interface to which control can be passed. In addition, it does not have the ability to direct its control transfers.
$CS.Single-Thread(C_1) \rightarrow \{5\} \leftarrow DT.Arbitrary(C_2)$ $CS.Single-Thread(C_1) \rightarrow \{6\} \leftarrow DT.Arbitrary(C_2)$	A single threaded component expects to be passed and to pass its data directly to a specific interface point. A component with arbitrary data topology has no fixed point to which data can be passed. In addition, the arbitrary component would be unable send data directly to the single-threaded component's interface point.
$CS.Multi-Thread(C_1) \rightarrow \{4\} \leftarrow CS.Concurrent(C_2)$	A sequencing problem can occur if multiple, concurrent threads try to simultaneously initiate execution with a multi-threaded component. The difficulty lies in ordering the accesses to the multi-threaded component according to its specific needs.
$CS.Concurrent(C_1) \rightarrow \{3\} \leftarrow Bl.Blocking(C_2)$ $CS.Concurrent(C_1) \rightarrow \{4\} \leftarrow Bl.Blocking(C_2)$	See Table 28.
$CS.Concurrent(C_1) \rightarrow \{4\} \leftarrow CS.Concurrent(C_2)$	A sequencing problem can occur if multiple, concurrent threads try to initiate execution with another concurrent component. Sequencing the accesses appropriately becomes problematic.
$CS.Concurrent(C_1) \rightarrow \{4\} \leftarrow CT.Hierarchical(C_2)$	A component with a hierarchical control topology has a single entry point. A concurrent component might try to initiate the component with multiple threads, transferring control out of the expected sequence, causing inconsistencies in processing.
$CS.Concurrent(C_1) \rightarrow \{4\} \leftarrow CT.Star(C_2)$	A component with star control topology has a single entry point and is, in fact, a one-tier hierarchy. Thus, if a concurrent component tries to initiate the component with multiple threads of control the identical problem will occur as described immediately above.
$CS.Concurrent(C_1) \rightarrow \{4\} \leftarrow CT.Linear(C_2)$	A concurrent control structure executes multiple and sometimes simultaneous, threads of control. Should it attempt to communicate with a

	component that has a linear control topology, the broadcasting control transfers might bombard the component or arrive unsequenced. This could cause anomalies in the component's subsequent processing.
$CS.Concurrent(C_1) \rightarrow \{8\} \leftarrow DSM.Local(C_2)$ $CS.Concurrent(C_1) \rightarrow \{10\} \leftarrow DSM.Local(C_2)$	A component running multiple, concurrent threads can perform illegal operations on another component's local data. In addition, the data transfers might not be sequenced properly.
$CS.Concurrent(C_1) \rightarrow \{10\} \leftarrow DT.Hierarchical(C_2)$	A component with a hierarchical data topology expects a single, pointed data communication. A concurrent component might try to initiate multiple data transfers to the other component causing it to be overwhelmed with unsequenced data transfers which it will not be able to properly process.
$CS.Concurrent(C_1) \rightarrow \{10\} \leftarrow DT.Star(C_2)$	Data transferred to a component with a star data topology enters the center hub or root module in a discrete chunk. If a concurrent component tries to transfer data in multiple or continuous pieces to the hub their order can be incorrect.
$CS.Concurrent(C_1) \rightarrow \{10\} \leftarrow DT.Linear(C_2)$	A component with a linear data topology has a single point of entry and if a concurrent component tries to initiate multiple data transfers they might arrive out of the expected sequence.
$CS.Concurrent(C_1) \rightarrow \{4\} \leftarrow SCT.Explicit(C_2)$	A concurrent component might initiate multiple control transfers at the same time, causing sequencing problems, because a component that transfers control explicitly expects pointed transfer of control.
$CS.Concurrent(C_1) \rightarrow \{10\} \leftarrow SDT.Explicit-Discrete(C_2)$	A concurrent component might initiate multiple, simultaneous data transfers, causing data sequencing problems, because a component that transfers data explicitly and discretely expects pointed transfer of data.

Table 29: Control Structure PAIs

Control Topology (CT): The geometric form control flow takes in a system.

PAIs	PAI Description
$CT.Hierarchical(C_1) \rightarrow \{4\} \leftarrow CS.Concurrent(C_2)$	See Table 29.
$CT.Hierarchical(C_1) \rightarrow \{1\} \leftarrow CT.Arbitrary(C_2)$ $CT.Hierarchical(C_1) \rightarrow \{2\} \leftarrow CT.Arbitrary(C_2)$	A component with a hierarchical control topology has a fixed entry point and expects to pass control to a specific entry point of another component. A component with arbitrary control topology has no such entry point. In addition, it does not have the ability to direct its control transfers.
$CT.Hierarchical(C_1) \rightarrow \{1\} \leftarrow IC.Unaware(C_2)$ $CT.Hierarchical(C_1) \rightarrow \{12\} \leftarrow IC.Unaware(C_2)$	A hierarchical component expects pointed transfer of control. The unaware component, however, has no knowledge of other components in the system and is therefore unable to initialize a control transfer, and hence, unable to pass to a specific interface point.
$CT.Hierarchical(C_1) \rightarrow \{1\} \leftarrow SCT.Implicit(C_2)$ $CT.Hierarchical(C_1) \rightarrow \{2\} \leftarrow SCT.Implicit(C_2)$	A hierarchical component expects pointed transfer of control. A component, which only supports implicit control transfer, expects to be transferred control implicitly. Neither component will be able to transfer control to the other component in the appropriate manner.
$CT.Hierarchical(C_1) \rightarrow \{1\} \leftarrow SCT.None(C_2)$ $CT.Hierarchical(C_1) \rightarrow \{12\} \leftarrow SCT.None(C_2)$	A hierarchical component expects pointed transfer of control. A component, which does not support any control transfer, will be unable transfer any control to other component, and hence, will be unable to direct control to the interface point of the hierarchical component.
$CT.Star(C_1) \rightarrow \{4\} \leftarrow CS.Concurrent(C_2)$	See Table 29.
$CT.Star(C_1) \rightarrow \{1\} \leftarrow CT.Arbitrary(C_2)$ $CT.Star(C_1) \rightarrow \{2\} \leftarrow CT.Arbitrary(C_2)$	A component with star control topology has a single entry point and is, in fact, only a one level hierarchy. In addition, it expects to pass control to a specific entry point. A component with arbitrary control topology does not have a single-entry point and is not capable of passing to such an entry point.
$CT.Star(C_1) \rightarrow \{1\} \leftarrow IC.Unaware(C_2)$ $CT.Star(C_1) \rightarrow \{12\} \leftarrow IC.Unaware(C_2)$	A component with a star topology expects pointed transfer of control. The unaware component, however, has no knowledge of other components in the system. It will, therefore be unable to initialize a control transfer to the other component, and hence unable to perform a pointed transfer of control.
$CT.Star(C_1) \rightarrow \{1\} \leftarrow SCT.Implicit(C_2)$ $CT.Star(C_1) \rightarrow \{2\} \leftarrow SCT.Implicit(C_2)$	A component with a star topology expects pointed transfer of control. A component, which only supports implicit control transfer, expects to be transferred control implicitly. Neither component will be able to transfer control to the other component in the appropriate manner.
$CT.Star(C_1) \rightarrow \{1\} \leftarrow SCT.None(C_2)$ $CT.Star(C_1) \rightarrow \{12\} \leftarrow SCT.None(C_2)$	A component with star topology expects pointed transfer of control. A component, which does not support any control transfer, will be unable to transfer any control to other component, and hence, unable to perform a pointed transfer of control.
$CT.Arbitrary(C_1) \rightarrow \{2\} \leftarrow Bl.Blocking(C_2)$ $CT.Arbitrary(C_1) \rightarrow \{3\} \leftarrow Bl.Blocking(C_2)$	See Table 28.

<i>CT.Arbitrary(C₁)</i> → {1} ← <i>CS.Single-Thread(C₂)</i> <i>CT.Arbitrary(C₁)</i> → {2} ← <i>CS.Single-Thread(C₂)</i>	See Table 29.
<i>CT.Arbitrary(C₁)</i> → {1} ← <i>CT.Linear(C₂)</i> <i>CT.Arbitrary(C₁)</i> → {2} ← <i>CT.Linear(C₂)</i>	Unlike a component with arbitrary topology a component with linear control topology has a fixed point of entry. They will therefore be unable to exchange control since neither is able to abide to the other component's topology.
<i>CT.Arbitrary(C₁)</i> → {2} ← <i>CT.Arbitrary(C₂)</i>	Neither component has fixed point of entry. Therefore, they are therefore unable to exchange control since they have specific interface point to which they can pass their control.
<i>CT.Arbitrary(C₁)</i> → {1} ← <i>SCT.Explicit(C₂)</i> <i>CT.Arbitrary(C₁)</i> → {2} ← <i>SCT.Explicit(C₂)</i>	A component with arbitrary control topology has no fixed point of entry and is unable to pass control to such an entry point. A component which supports explicit control transfer expects to be passed control explicitly and can only transfer control in such a manner.
<i>CT.Linear(C₁)</i> → {4} ← <i>CS.Concurrent(C₂)</i>	See Table 29.
<i>CT.Linear(C₁)</i> → {1} ← <i>IC.Unaware(C₂)</i>	A component with a linear topology expects pointed transfer of control. The unaware component, however, has no knowledge of other components in the system and will be unable to direct its transfer to a particular interface point.
<i>CT.Linear(C₁)</i> → {1} ← <i>SCT.Implicit(C₂)</i>	A component with linear topology expects pointed transfer of control. A component, which only supports implicit control transfer, cannot direct its control to a specific interface point.
<i>CT.Linear(C₁)</i> → {1} ← <i>SCT.None(C₂)</i> <i>CT.Linear(C₁)</i> → {12} ← <i>SCT.None(C₂)</i>	A component with linear topology expects pointed transfer of control. A component, which does not support any control transfer, will be unable to transfer any control to the other component and therefore unable to perform pointed transfer of control.

Table 30: Control Topology PAIs

Data Storage Method (DSM): How data is stored within a system.

PAIs	PAI Descriptions
<i>DSM.Local(C₁)</i> → {8} ← <i>CS.Concurrent(C₂)</i> <i>DSM.Local(C₁)</i> → {10} ← <i>CS.Concurrent(C₂)</i>	See Table 29.
<i>DSM.Local(C₁)</i> → {7} ← <i>IC.Unaware(C₂)</i>	A component storing its data locally expects to be passed data directly. However, the unaware component has no knowledge of any other components in the system, and hence, no knowledge of the component's data storage location.

Table 31: Data Storage Method PAIs

Data Topology (DT): The geometric form data flow takes in a system.

PAIs	PAI Description
$DT.Hierarchical(C_1) \rightarrow \{10\} \leftarrow CS.Concurrent(C_2)$	See Table 29.
$DT.Hierarchical(C_1) \rightarrow \{5\} \leftarrow DT.Arbitrary(C_2)$ $DT.Hierarchical(C_1) \rightarrow \{6\} \leftarrow DT.Arbitrary(C_2)$	A component with hierarchical data topology has a fixed entry point and expects to pass data to a specific entry point of another component. A component with arbitrary data topology has no such entry point. In, addition it does not have the ability to direct its data transfers.
$DT.Hierarchical(C_1) \rightarrow \{5\} \leftarrow IC.Unaware(C_2)$ $DT.Hierarchical(C_1) \rightarrow \{13\} \leftarrow IC.Unaware(C_2)$	A hierarchical component expects pointed transfer of data. The unaware component, however, has no knowledge of other components in the system and is therefore unable to initialize a data transfer, and hence, unable to pass it to a specific interface point.
$DT.Hierarchical(C_1) \rightarrow \{5\} \leftarrow SDT.Implicit-Discrete(C_2)$ $DT.Hierarchical(C_1) \rightarrow \{6\} \leftarrow SDT.Implicit-Discrete(C_2)$	A hierarchical component expects pointed transfer of data. A component, which only supports implicit and discrete data transfer, expects to be transferred data implicitly. Neither component will be able to transfer data to the other component in the appropriate manner.
$DT.Hierarchical(C_1) \rightarrow \{5\} \leftarrow SDT.Implicit-Continuous(C_2)$ $DT.Hierarchical(C_1) \rightarrow \{6\} \leftarrow SDT.Implicit-Continuous(C_2)$	A hierarchical component expects pointed transfer of data. A component, which only supports implicit and continuous data transfer, expects to be transferred data implicitly. Neither component will be able to transfer data to the other component in the appropriate manner.
$DT.Hierarchical(C_1) \rightarrow \{5\} \leftarrow SDT.Shared(C_2)$ $DT.Hierarchical(C_1) \rightarrow \{13\} \leftarrow SDT.Shared(C_2)$	A hierarchical component expects pointed transfer of data. A component, which only shares its data and has no means to pass data to another component, cannot initialize a data transfer. Thus, it will be unable to perform a pointed transfer of data.
$DT.Hierarchical(C_1) \rightarrow \{5\} \leftarrow SDT.None(C_2)$ $DT.Hierarchical(C_1) \rightarrow \{13\} \leftarrow SDT.None(C_2)$	A hierarchical component expects pointed transfer of data. A component, which does not support any data transfer, will be unable transfer any data to other components, and hence, will be unable to direct data to the interface point of the hierarchical component.
$DT.Star(C_1) \rightarrow \{10\} \leftarrow CS.Concurrent(C_2)$	See Table 29.
$DT.Star(C_1) \rightarrow \{5\} \leftarrow DT.Arbitrary(C_2)$ $DT.Star(C_1) \rightarrow \{6\} \leftarrow DT.Arbitrary(C_2)$	A component with star data topology has a single entry point and is, in fact, only a one level hierarchy. In addition, it expects to pass data to a specific entry point. A component with arbitrary data topology does not have a single-entry point and is not capable of passing to such an entry point.
$DT.Star(C_1) \rightarrow \{5\} \leftarrow IC.Unaware(C_2)$ $DT.Star(C_1) \rightarrow \{13\} \leftarrow IC.Unaware(C_2)$	A component with a star topology expects pointed transfer of data. The unaware component, however, has no knowledge of other components in the system. It will, therefore be unable to initialize a data transfer to the other component, and hence unable to perform a pointed transfer of data.
$DT.Star(C_1) \rightarrow \{5\} \leftarrow SDT.Implicit-Discrete(C_2)$ $DT.Star(C_1) \rightarrow \{6\} \leftarrow SDT.Implicit-Discrete(C_2)$	A component with a star topology expects pointed transfer of data. A component, which only supports implicit, discrete data transfer, expects to be

	transferred data implicitly. Neither component will be able to transfer data to the other component in the appropriate manner.
$DT.Star(C_1) \rightarrow \{5\} \leftarrow SDT.Implicit-Continuous(C_2)$ $DT.Star(C_1) \rightarrow \{6\} \leftarrow SDT.Implicit-Continuous(C_2)$	A component with a star topology expects pointed transfer of data. A component, which only supports implicit, continuous data transfer, expects to be transferred data implicitly. Neither component will be able to transfer data to the other component in the appropriate manner.
$DT.Star(C_1) \rightarrow \{5\} \leftarrow SDT.Shared(C_2)$ $DT.Star(C_1) \rightarrow \{13\} \leftarrow SDT.Shared(C_2)$	A component with a star topology has a fixed entry point and expects to transfer its data to a specific interface point. A component, which only supports sharing of its data, will be unable to initialize any transfer of data, and hence direct a data transfer to a specific interface point.
$DT.Star(C_1) \rightarrow \{5\} \leftarrow SDT.None(C_2)$ $DT.Star(C_1) \rightarrow \{13\} \leftarrow SDT.None(C_2)$	A component with star topology expects pointed transfer of data. A component, which does not support any data transfer, will be unable to transfer any data to other component, and hence, unable to perform a pointed transfer of data.
$DT.Arbitrary(C_1) \rightarrow \{5\} \leftarrow CS.Single-Thread(C_2)$ $DT.Arbitrary(C_1) \rightarrow \{6\} \leftarrow CS.Single-Thread(C_2)$	See Table 29.
$DT.Arbitrary(C_1) \rightarrow \{5,6\} \leftarrow CT.Linear(C_2)$	Unlike a component with arbitrary topology a component with linear data topology has a fixed point of entry. They will therefore be unable to exchange data since neither is able to abide to the other component's topology.
$DT.Arbitrary(C_1) \rightarrow \{6\} \leftarrow DT.Arbitrary(C_2)$	Neither component has fixed point of entry. Therefore, they are therefore unable to exchange data since they have no specific interface points to which they can pass their data.
$DT.Arbitrary(C_1) \rightarrow \{5\} \leftarrow SDT.Explicit-Discrete(C_2)$	A component with arbitrary data topology has no fixed point of entry and is unable to pass data to such an entry point. A component, which supports explicit data transfer expects to be passed data explicitly and can only transfer data in such a manner.
$DT.Linear(C_1) \rightarrow \{10\} \leftarrow CS.Concurrent(C_2)$	See Table 27.
$DT.Linear(C_1) \rightarrow \{5\} \leftarrow IC.Unaware(C_2)$	A component with a linear topology expects pointed transfer of data. The unaware component, however, has no knowledge of other components in the system and will be unable to direct its transfer to a particular interface point.
$DT.Linear(C_1) \rightarrow \{5\} \leftarrow SDT.Implicit-Discrete(C_2)$	A component with linear topology expects pointed transfer of data. A component, which only supports implicit, discrete data transfer, cannot direct its data to a specific interface point.
$DT.Linear(C_1) \rightarrow \{5\} \leftarrow SDT.Implicit-Continuous(C_2)$	A component with linear topology expects pointed transfer of data. A component, which only supports implicit, continuous data transfer, cannot direct its data to a specific interface point.
$DT.Linear(C_1) \rightarrow \{5\} \leftarrow SDT.Shared(C_2)$ $DT.Linear(C_1) \rightarrow \{13\} \leftarrow SDT.Shared(C_2)$	A component with linear data topology requires data to be passed to a particular interface point. A component only sharing its data cannot initialize data transfer, and hence, cannot perform a pointed

	transfer of data.
$DT.Linear(C_1) \rightarrow \{5\} \leftarrow SDT.None(C_2)$ $DT.Linear(C_1) \rightarrow \{13\} \leftarrow SDT.None(C_2)$	<p>A component with linear topology expects pointed transfer of data. A component, which does not support any data transfer, will be unable to transfer any data to the other component and therefore unable to perform pointed transfer of data.</p>

Table 32: Data Topology PAIs

Identity of Components (IC): Awareness of other components in the system.

PAIs	PAI Descriptions
$IC.Unaware(C_1) \rightarrow \{12\} \leftarrow Bl.Blocking(C_2)$ $IC.Unaware(C_1) \rightarrow \{13\} \leftarrow Bl.Blocking(C_2)$	See Table 28
$IC.Unaware(C_1) \rightarrow \{1\} \leftarrow CT.Hierarchical(C_2)$ $IC.Unaware(C_1) \rightarrow \{12\} \leftarrow CT.Hierarchical(C_2)$	See Table 30.
$IC.Unaware(C_1) \rightarrow \{1\} \leftarrow CT.Star(C_2)$ $IC.Unaware(C_1) \rightarrow \{12\} \leftarrow CT.Star(C_2)$	See Table 30.
$IC.Unaware(C_1) \rightarrow \{1\} \leftarrow CT.Linear(C_2)$	See Table 30.
$IC.Unaware(C_1) \rightarrow \{7\} \leftarrow DSM.Local(C_2)$	See Table 31.
$IC.Unaware(C_1) \rightarrow \{5\} \leftarrow DT.Hierarchical(C_2)$ $IC.Unaware(C_1) \rightarrow \{13\} \leftarrow DT.Hierarchical(C_2)$	See Table 32.
$IC.Unaware(C_1) \rightarrow \{5\} \leftarrow DT.Star(C_2)$ $IC.Unaware(C_1) \rightarrow \{13\} \leftarrow DT.Star(C_2)$	See Table 32.
$IC.Unaware(C_1) \rightarrow \{5\} \leftarrow DT.Linear(C_2)$	See Table 32.
$IC.Unaware(C_1) \rightarrow \{12\} \leftarrow IC.Unware(C_2)$ $IC.Unaware(C_1) \rightarrow \{13\} \leftarrow IC.Unware(C_2)$	Unaware components have no knowledge that other components in the application exist. This makes two unaware components unable to initialize any control or data exchange.
$IC.Unaware(C_1) \rightarrow \{1\} \leftarrow SCT.Explicit(C_2)$ $IC.Unaware(C_1) \rightarrow \{12\} \leftarrow SCT.Explicit(C_2)$	A component with explicit supported control transfer expects to be passed control directly. An unaware component has no knowledge of other components in the system. It is therefore unable to initialize a control transfer to other components and hence unable to perform a pointed transfer of control.
$IC.Unaware(C_1) \rightarrow \{5\} \leftarrow SDT.Explicit(C_2)$ $IC.Unaware(C_1) \rightarrow \{13\} \leftarrow SDT.Explicit(C_2)$	A component with explicit supported data transfer expects to be passed data explicitly. An unaware component has no knowledge of other components in the system. It can neither initialize a control transfer nor pass data explicitly to other components in the system.

Table 33: Identity of Components PAIs

Supported Control Transfer (SCT): The method supported to achieve control transfer.

PAIs	PAI Descriptions
$SCT.Explicit(C_1) \rightarrow \{4\} \leftarrow CS.Concurrent(C_2)$	See Table 29.
$SCT.Explicit(C_1) \rightarrow \{1\} \leftarrow CT.Arbitrary(C_2)$ $SCT.Explicit(C_1) \rightarrow \{2\} \leftarrow CT.Arbitrary(C_2)$	See Table 30.
$SCT.Explicit(C_1) \rightarrow \{1\} \leftarrow IC.Unaware(C_2)$ $SCT.Explicit(C_1) \rightarrow \{12\} \leftarrow IC.Unaware(C_2)$	See Table 33.
$SCT.Explicit(C_1) \rightarrow \{1\} \leftarrow SCT.Implicit(C_2)$ $SCT.Explicit(C_1) \rightarrow \{2\} \leftarrow SCT.Implicit(C_2)$	A component supporting explicit control transfers will make direct calls and expect to be transferred control in the same manner. However a component with implicit supported control transfer will simply broadcast its control and not direct it to the other component.
$SCT.Explicit(C_1) \rightarrow \{1\} \leftarrow SCT.None(C_2)$ $SCT.Explicit(C_1) \rightarrow \{12\} \leftarrow SCT.None(C_2)$	A component, which transfers and expects control to be passed explicitly, will be unable to communicate with a component, which has no means to transfer its control.
$SCT.Implicit(C_1) \rightarrow \{3\} \leftarrow Bl.Blocking(C_2)$	See Table 28.
$SCT.Implicit(C_1) \rightarrow \{1,2\} \leftarrow CT.Hierarchical(C_2)$	See Table 30.
$SCT.Implicit(C_1) \rightarrow \{1,2\} \leftarrow CT.Star(C_2)$	See Table 30.
$SCT.Implicit(C_1) \rightarrow \{1,2\} \leftarrow CT.Linear(C_2)$	See Table 30.
$SCT.Implicit(C_1) \rightarrow \{12\} \leftarrow SCT.None(C_2)$	A component with implicit supported control transfer will make implicit calls and expect to be transferred control implicitly. A component with no means to transfer its control will be unable to communicate.
$SCT.None(C_1) \rightarrow \{3,12\} \leftarrow Bl.Blocking(C_2)$	See Table 28.
$SCT.None(C_1) \rightarrow \{1,12\} \leftarrow CT.Hierarchical(C_2)$	See Table 30.
$SCT.None(C_1) \rightarrow \{1,12\} \leftarrow CT.Star(C_2)$	See Table 30.
$SCT.None(C_1) \rightarrow \{1,12\} \leftarrow CT.Linear(C_2)$	See Table 30.

Table 34: Supported Control Transfer PAIs

Supported Data Transfer (SDT): The method supported to achieve data transfer.

PAIs	PAI Descriptions
$SDT.Explicit(C_1) \rightarrow \{10\} \leftarrow CS.Concurrent(C_2)$	See Table 29.
$SDT.Explicit(C_1) \rightarrow \{5\} \leftarrow DT.Arbitrary(C_2)$ $SDT.Explicit(C_1) \rightarrow \{6\} \leftarrow DT.Arbitrary(C_2)$	See Table 32.
$SDT.Explicit(C_1) \rightarrow \{5\} \leftarrow IC.Unaware(C_2)$ $SDT.Explicit(C_1) \rightarrow \{13\} \leftarrow IC.Unaware(C_2)$	See Table 33.
$SDT.Explicit(C_1) \rightarrow \{5\} \leftarrow SDT.Implicit-Discrete(C_2)$ $SDT.Explicit(C_1) \rightarrow \{6\} \leftarrow SDT.Implicit-Discrete(C_2)$	A component with explicit supported data transfer expects to be passed data in the same way it passes data: directly and discretely. Implicit-discrete data transfer implies that data is discretely broadcasted and that it expects to be passed data in the same manner. There neither component can pass data in the manner expected by the other component.
$SDT.Explicit(C_1) \rightarrow \{5\} \leftarrow SDT.Implicit-Continuous(C_2)$ $SDT.Explicit(C_1) \rightarrow \{6\} \leftarrow SDT.Implicit-Continuous(C_2)$ $SDT.Explicit(C_1) \rightarrow \{11\} \leftarrow SDT.Implicit-Continuous(C_2)$	A component with explicit-discrete supported data transfer will expect to be passed data directly and discretely and it passes its data in such a manner. Implicit-continuous implies that data is simply broadcasted in a stream and that the component expects to be passed data implicitly in a stream. In addition, those data transfer assumptions are incompatible. There neither component can pass data in the manner expected by the other component.
$SDT.Explicit(C_1) \rightarrow \{5\} \leftarrow SDT.Shared(C_2)$ $SDT.Explicit(C_1) \rightarrow \{13\} \leftarrow SDT.Shared(C_2)$	A component that only supports shared access to its data will not be able to initialize a data transfer. It will therefore be unable to perform a pointed data transfer to a component that requires data to be passed explicitly to it.
$SDT.Explicit(C_1) \rightarrow \{5\} \leftarrow SDT.None(C_2)$ $SDT.Explicit(C_1) \rightarrow \{13\} \leftarrow SDT.None(C_2)$	A component unable to transfer any data will be unable to initialize a control transfer. It will therefore not be able to perform an explicit-discrete data transfer to a component expecting to be passed data in such a manner.
$SDT.Implicit-Discrete(C_1) \rightarrow \{5\} \leftarrow DT.Hierarchical(C_2)$ $SDT.Implicit-Discrete(C_1) \rightarrow \{6\} \leftarrow DT.Hierarchical(C_2)$	See Table 32.
$SDT.Implicit-Discrete(C_1) \rightarrow \{5\} \leftarrow DT.Star(C_2)$ $SDT.Implicit-Discrete(C_1) \rightarrow \{6\} \leftarrow DT.Star(C_2)$	See Table 32.
$SDT.Implicit-Discrete(C_1) \rightarrow \{5\} \leftarrow DT.Linear(C_2)$	See Table 32.
$SDT.Implicit-Discrete(C_1) \rightarrow \{13\} \leftarrow SDT.Shared(C_2)$	A component supporting implicit-discrete data transfer expects to be passed data in the same manner. A component only sharing its data has no means to initialize a control transfer, and hence no means to explicitly transfer data to the other component.
$SDT.Implicit-Discrete(C_1) \rightarrow \{13\} \leftarrow SDT.None(C_2)$	A component, which supports no data transfers cannot transfer data to a component requiring an explicit data transfer.
$SDT.Implicit-Continuous(C_1) \rightarrow \{5\} \leftarrow DT.Hierarchical(C_2)$	See Table 32.

<i>SDT.Implicit-Continuous</i> (C_1) \rightarrow {6} \leftarrow <i>DT.Hierarchical</i> (C_2)	
<i>SDT.Implicit-Continuous</i> (C_1) \rightarrow {5} \leftarrow <i>DT.Star</i> (C_2) <i>SDT.Implicit-Continuous</i> (C_1) \rightarrow {6} \leftarrow <i>DT.Star</i> (C_2)	See Table 32.
<i>SDT.Implicit-Continuous</i> (C_1) \rightarrow {5} \leftarrow <i>DT.Linear</i> (C_2)	See Table 32.
<i>SDT.Implicit-Continuous</i> (C_1) \rightarrow {13} \leftarrow <i>SDT.None</i> (C_2)	A component with no means of transferring its data cannot initialize a data transfer.
<i>SDT.Shared</i> (C_1) \rightarrow {5} \leftarrow <i>DT.Hierarchical</i> (C_2) <i>SDT.Shared</i> (C_1) \rightarrow {13} \leftarrow <i>DT.Hierarchical</i> (C_2)	See Table 32.
<i>SDT.Shared</i> (C_1) \rightarrow {5} \leftarrow <i>DT.Star</i> (C_2) <i>SDT.Shared</i> (C_1) \rightarrow {13} \leftarrow <i>DT.Star</i> (C_2)	See Table 32.
<i>SDT.Shared</i> (C_1) \rightarrow {5} \leftarrow <i>DT.Linear</i> (C_2) <i>SDT.Shared</i> (C_1) \rightarrow {13} \leftarrow <i>SDT.Shared</i> (C_2)	See Table 32. Two components, both only sharing their data will be unable to initialize a data transfer. Thus they will be unable to exchange data.
<i>SDT.Shared</i> (C_1) \rightarrow {13} \leftarrow <i>SDT.None</i> (C_2)	A component supporting no data transfers and another only supporting shared data transfer will be unable to exchange data since neither can initialize data transfer.
<i>SDT.None</i> (C_1) \rightarrow {5} \leftarrow <i>DT.Hierarchical</i> (C_2) <i>SDT.None</i> (C_1) \rightarrow {13} \leftarrow <i>DT.Hierarchical</i> (C_2)	See Table 32.
<i>SDT.None</i> (C_1) \rightarrow {5} \leftarrow <i>DT.Star</i> (C_2) <i>SDT.None</i> (C_1) \rightarrow {13} \leftarrow <i>DT.Star</i> (C_2)	See Table 32.
<i>SDT.None</i> (C_1) \rightarrow {5} \leftarrow <i>DT.Star</i> (C_2) <i>SDT.None</i> (C_1) \rightarrow {13} \leftarrow <i>DT.Star</i> (C_2)	See Table 32.
<i>SDT.None</i> (C_1) \rightarrow {13} \leftarrow <i>SDT.None</i> (C_2)	Components unable to initialize any data transfers will never be able to exchange data.

Table 35: Supported Data Transfer PAIs

Application-Component PAIs

Control Structure (CS): The structure that governs the execution in the system.

PAIs	PAI Descriptions
<p>$CS.Single-Thread(App) \rightarrow \{3\} \leftarrow Bl.Non-Blocking(C_1)$</p> <p>$CS.Single-Thread(App) \rightarrow \{8\} \leftarrow Bl.Non-Blocking(C_1)$</p>	<p>If one component executes independently with no means of interruption, but the application only has a single-thread of control, the component might execute infinitely, not allowing other components in the application to execute. In addition, because a non-blocking component can execute independently, possibly doing unrelated work to the other component, data can be adversely permuted.</p>
<p>$CS.Single-Thread(App) \rightarrow \{4\} \leftarrow CS.Concurrent(C_1)$</p>	<p>A sequencing problem can occur if a component tries to initiate execution with multiple, concurrent threads when the application can only support a single-thread.</p>
<p>$CS.Single-Thread(App) \rightarrow \{2\} \leftarrow CT.Arbitrary(C_1)$</p>	<p>An application with single threaded control structure expects to pass its control directly to a specific interface of a component. A component with an arbitrary control topology has no fixed point to which control can be passed.</p>
<p>$CS.Single-Thread(App) \rightarrow \{6\} \leftarrow DT.Arbitrary(C_1)$</p>	<p>An application with single threaded control structure expects to pass its data directly to a specific interface of a component. A component with an arbitrary data topology has no fixed point to which data can be passed.</p>
<p>$CS.Multi-Thread(App) \rightarrow \{4\} \leftarrow CS.Concurrent(C_1)$</p> <p>$CS.Multi-Thread(App) \rightarrow \{8\} \leftarrow CS.Concurrent(C_1)$</p>	<p>If the application has multi-threaded control structures it works under a timesharing principle. A concurrent component might try to initiate multiple threads of at the same time causing sequencing problems.</p>
<p>$CS.Concurrent(App) \rightarrow \{2\} \leftarrow Bl.Blocking(C_1)$</p> <p>$CS.Concurrent(App) \rightarrow \{4\} \leftarrow Bl.Blocking(C_1)$</p>	<p>If an application has a concurrent control structure, difficulties might arise when a blocking component is waiting for a specific component to handshake with it. The concurrent application does not know where to transfer its control. In addition, the requests might not be sequenced appropriately.</p>
<p>$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CS.Single-Thread(C_1)$</p> <p>$CS.Concurrent(App) \rightarrow \{8\} \leftarrow CS.Single-Thread(C_1)$</p>	<p>A concurrent application expects to be able to have multiple threads of control initialize a component. If the component is single-threaded the control and/or control transfers must be sequenced according to some predefined algorithm.</p>
<p>$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CS.Concurrent(C_1)$</p>	<p>A concurrent application might try to initialize the same component multiple times and the initializations might be out of order.</p>
<p>$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CT.Hierarchical(C_1)$</p>	<p>A component with hierarchical control topology has restricted points to which control must be passed and a concurrent application might not be able to sequence those control transfers correctly.</p>
<p>$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CT.Star(C_1)$</p>	<p>An application with concurrent control structure might have problems in sequencing control transfers to a component with star control topology.</p>

$CS.Concurrent(App) \rightarrow \{4\} \leftarrow CT.Linear(C_1)$	A linear component has a fixed point of entry but a concurrent application might try to initialize it with multiple concurrent threads. This could cause sequencing problems.
$CS.Concurrent(App) \rightarrow \{10\} \leftarrow DT.Hierarchical(C_1)$	A component with hierarchical data topology has restricted points to which data must be passed and a concurrent application might not be able to sequence those data transfers correctly.
$CS.Concurrent(App) \rightarrow \{10\} \leftarrow DT.Star(C_1)$	An application with concurrent control structure might have problems in sequencing data transfers to fixed entry points. A component with star data topology has such an entry point.
$CS.Concurrent(App) \rightarrow \{10\} \leftarrow DT.Linear(C_1)$	A linear component has a fixed point of entry but a concurrent application might try to initialize it with multiple concurrent threads causing sequencing problems with data transfers.

Table 36: Application Control Structure PAIs

Control Topology (CT): The geometric form control flow takes in a system.

PAIs	PAI Descriptions
<i>CT.Hierarchical(App) → {4} ← CS.Concurrent(C₁)</i>	A hierarchical configuration of components forces the participating components to execute a call and return communication hierarchically. A component with concurrent control structure might run into difficulty sequencing the control transfers in such a way that the hierarchical nature of the application is maintained.
<i>CT.Hierarchical(App) → {1} ← CT.Arbitrary(C₁)</i>	A hierarchical configuration of an application forces the participating components to communicate using hierarchical call and return. A component with arbitrary control topology has no set topology, making call and return difficult.
<i>CT.Hierarchical(App) → {1} ← IC.Unaware(C₁)</i> <i>CT.Hierarchical(App) → {12} ← IC.Unaware(C₁)</i>	A component that is unaware does not know whom it receives control from or where to return control, making direct call and return difficult. An application with hierarchical control topology, however, requires its components to operate in this a fashion.
<i>CT.Hierarchical(App) → {1} ← SCT.Implicit(C₁)</i>	A hierarchical configuration of components forces the participating components to communicate hierarchically. A component that only supports implicit control transfers will be unable to make direct calls, inhibiting direct call and return, required by the application type.
<i>CT.Hierarchical(App) → {1} ← SCT.None(C₁)</i> <i>CT.Hierarchical(App) → {12} ← SCT.None(C₁)</i>	A component that supports no control transfers will be unable to initialize a control transfer. Hence, it is unable to make direct calls. However an application operating under a hierarchical topology requires its participating components to communicate in a call and return fashion which cannot be fulfilled by this component.
<i>CT.Star(App) → {4} ← CS.Concurrent(C₁)</i>	A component with concurrent control structure might run into difficulty sequencing the control transfers so that the star topology of the application is maintained.
<i>CT.Star(App) → {1} ← CT.Arbitrary(C₁)</i>	A star configuration of components is really a one-level hierarchical configuration. It requires the components to make direct calls while communicating. A component with arbitrary control topology has no set topology, making direct calls to a specific point difficult.
<i>CT.Star(App) → {1} ← IC.Unaware(C₁)</i> <i>CT.Star(App) → {12} ← IC.Unaware(C₁)</i>	A star configuration of an application forces the participating components to communicate using direct call and return. An unaware component has no knowledge of other components in the application and will be unable to initialize a control transfer. More specifically, it will be unable to make direct calls.
<i>CT.Star(App) → {1} ← SCT.Implicit(C₁)</i>	If the application has a star control topology it requires its components to make direct calls and returns. If a component passes its control implicitly

	it will simply broadcast its control and not direct the calls to a particular interface point.
<i>CT.Star(App) → {1} ← SCT.None(C₁)</i> <i>CT.Star(App) → {12} ← SCT.None(C₁)</i>	A component that supports no control transfers will be unable to make direct calls, since it has no way of initializing a control transfer. An application with star topology requires its components to make direct calls and returns.
<i>CT.Arbitrary(App) → {1} ← Bl.Blocking(C₁)</i> <i>CT.Arbitrary(App) → {3} ← Bl.Blocking(C₁)</i>	If an application has an arbitrary control topology it communicates control randomly and has no set structure. This might possible leave a blocked component, waiting to handshake with another component blocked indefinitely, since the application is unable to specifically communicate control to it.
<i>CT.Arbitrary(App) → {1} ← CS.Single-Thread(C₁)</i> <i>CT.Arbitrary(App) → {3} ← CS.Single-Thread(C₁)</i>	A component with a single-threaded control structure expects control and data to be passed directly to a particular interface point. An arbitrary application will simply broadcast control to the components, making communication and handshaking with a specific component difficult.
<i>CT.Arbitrary(App) → {1} ← CT.Hierarchical(C₁)</i>	A hierarchical component expects control and data to be passed directly to a particular interface point. An arbitrary application will simply broadcast control without directing it to a particular interface point.
<i>CT.Arbitrary(App) → {1} ← CT.Star(C₁)</i>	An arbitrary application will simply broadcast control without directing it to a particular interface point. However, a component with star control topology expects pointed transfers for control.
<i>CT.Arbitrary(App) → {1} ← CT.Linear(C₁)</i>	An arbitrary application will be unable to pass data directly to the fixed entry point of a component with linear control topology.
<i>CT.Arbitrary(App) → {1} ← SCT.Explicit(C₁)</i>	A component supporting explicit control transfers will require the governing application to pass control to it explicitly. However, an arbitrary application usually broadcasts control and is therefore unable to direct its control transfers.
<i>CT.Linear(App) → {4} ← CS.Concurrent(C₁)</i>	If an application has a linear control topology it expects that control flows in linear fashion. When a component has a concurrent control structure it can pass control to multiple component at the same time, making the enforcement of linear control topology impossible.
<i>CT.Linear(App) → {1} ← CT.Arbitrary(C₁)</i>	A linear configuration of components forces the participating components to communicate in linear fashion. A component with arbitrary control topology has no set topology, making it unable to guarantee that it will pass its control to the appropriate component.
<i>CT.Linear(App) → {1} ← IC.Unaware(C₁)</i> <i>CT.Linear(App) → {12} ← IC.Unaware(C₁)</i>	A component that is unaware has no knowledge of other components in the system. It is unable to initialize a control transfer and is therefore not able to make a direct call. A component with linear control topology requires pointed transfer of control.

$CT.Linear(App) \rightarrow \{1\} \leftarrow SCT.Implicit(C_1)$	A linear application requires control to be passed in a linear fashion to specific interface points. A component, which supports an implicit control transfer, will simply broadcast its control without directing it to a particular point.
$CT.Linear(App) \rightarrow \{1\} \leftarrow SCT.None(C_1)$ $CT.Linear(App) \rightarrow \{12\} \leftarrow SCT.None(C_1)$	A component not supporting any type of control transfers will be unable to initialize a control transfer, and hence direct it to a particular interface point. A linear configuration of an application requires the participating components to communicate using direct calls.

Table 37: Application Control Topology PAIs

Data Topology (DT): The geometric form data flow takes in a system.

PAIs	PAI Descriptions
$DT.Hierarchical(App) \rightarrow \{10\} \leftarrow CS.Concurrent(C_1)$	A hierarchical configuration of components forces the participating components to pass data hierarchically. A component with concurrent data structure might run into difficulty sequencing the data transfers in such a way that the hierarchical nature of the application is maintained.
$DT.Hierarchical(App) \rightarrow \{5\} \leftarrow DT.Arbitrary(C_1)$	A hierarchical configuration of an application forces the participating components to exchange data using hierarchical call and return. A component with arbitrary data topology has no set topology, making call and return difficult.
$DT.Hierarchical(App) \rightarrow \{5\} \leftarrow IC.Unaware(C_1)$ $DT.Hierarchical(App) \rightarrow \{13\} \leftarrow IC.Unaware(C_1)$	A component that is unaware does not know whom it receives data from or where to return data, making direct calls to pass data difficult. An application with hierarchical data topology, however, requires its components to operate in this a fashion.
$DT.Hierarchical(App) \rightarrow \{5\} \leftarrow SDT.Implicit-Discrete(C_1)$	A hierarchical configuration of components forces the participating components to communicate data hierarchically. A component that only supports implicit, discrete data transfers will be unable to make direct calls, inhibiting direct call for passing data, which are required by the application type.
$DT.Hierarchical(App) \rightarrow \{5\} \leftarrow SDT.Implicit-Continuous(C_1)$	A hierarchical configuration of components forces the participating components to communicate data hierarchically. A component that only supports implicit, continuous data transfers will be unable to make direct calls, inhibiting direct call for passing data, which are required by the application type.
$DT.Hierarchical(App) \rightarrow \{5\} \leftarrow SDT.Shared(C_1)$ $DT.Hierarchical(App) \rightarrow \{5\} \leftarrow SDT.Shared(C_1)$	A hierarchical configuration of components forces the participating components to communicate hierarchically. A component that only shares its data will be unable to make direct calls with data.
$DT.Hierarchical(App) \rightarrow \{5\} \leftarrow SDT.None(C_1)$ $DT.Hierarchical(App) \rightarrow \{13\} \leftarrow SDT.None(C_1)$	A component that supports no data transfers will be unable to initialize a data transfer. Hence, it is unable to make direct calls to pass data. However an application operating under a hierarchical topology requires its participating components to communicate data in a call and return fashion, which cannot be fulfilled by this component.
$DT.Star(App) \rightarrow \{10\} \leftarrow CS.Concurrent(C_1)$	A component with concurrent data structure might run into difficulty sequencing the data transfers so that the star topology of the application is maintained.
$DT.Star(App) \rightarrow \{5\} \leftarrow DT.Arbitrary(C_1)$	A star configuration of components is really a one-level hierarchical configuration. It requires the components to make direct calls while communicating data. A component with arbitrary data topology has no set topology, making direct calls to pass data to a specific point difficult.
$DT.Star(App) \rightarrow \{5\} \leftarrow IC.Unaware(C_1)$	

$DT.Star(App) \rightarrow \{13\} \leftarrow IC.Unaware(C_1)$	A star configuration of an application forces the participating components to communicate data using direct calls. An unaware component has no knowledge of other components in the application and will be unable to initialize a data transfer. More specifically, it will be unable to make direct calls to pass data.
$DT.Star(App) \rightarrow \{5\} \leftarrow SDT.Implicit-Discrete(C_1)$	If the application has a star data topology it requires its components to make direct calls to pass data. If a component passes its data implicitly and discretely it will simply broadcast its data and not direct it to a particular interface point.
$DT.Star(App) \rightarrow \{5\} \leftarrow SDT.Implicit-Continuous(C_1)$	If the application has a star data topology it requires its components to make direct calls to pass data. If a component passes its data implicitly and continuously it will simply broadcast its data and not direct it to a particular interface point.
$DT.Hierarchical(App) \rightarrow \{5\} \leftarrow SDT.Shared(C_1)$ $DT.Hierarchical(App) \rightarrow \{13\} \leftarrow SDT.Shared(C_1)$	A hierarchical configuration of components forces the participating components to communicate data hierarchically. A component that only shares its data will be unable to initialize a data transfer, and hence make direct calls to pass data.
$DT.Star(App) \rightarrow \{5\} \leftarrow SDT.None(C_1)$ $DT.Star(App) \rightarrow \{13\} \leftarrow SDT.None(C_1)$	A component that supports no data transfers will be unable to make direct calls to pass data, since it has no way of initializing a data transfer. An application with star topology requires its components to make direct calls to pass data.
$DT.Arbitrary(App) \rightarrow \{3\} \leftarrow Bl.Blocking(C_1)$ $DT.Arbitrary(App) \rightarrow \{5\} \leftarrow Bl.Blocking(C_1)$	If an application has an arbitrary data topology it communicates data randomly and has no set structure. This might possible leave a blocked component, waiting to handshake with another component blocked indefinitely, since the application is unable to specifically communicate data to it.
$DT.Arbitrary(App) \rightarrow \{3\} \leftarrow CS.Single-Thread(C_1)$ $DT.Arbitrary(App) \rightarrow \{5\} \leftarrow CS.Single-Thread(C_1)$	A component with a single-threaded data structure expects data to be passed directly to a particular interface point. An arbitrary application will simply broadcast data to the components, making communication and handshaking with a specific component difficult.
$DT.Arbitrary(App) \rightarrow \{5\} \leftarrow DT.Hierarchical(C_1)$	A hierarchical component expects data to be passed directly to a particular interface point. An arbitrary application will simply broadcast data without directing it to a particular interface point.
$DT.Arbitrary(App) \rightarrow \{5\} \leftarrow DT.Star(C_1)$	An arbitrary application will simply broadcast data without directing it to a particular interface point. However, a component with star data topology expects pointed transfers for data.
$DT.Arbitrary(App) \rightarrow \{5\} \leftarrow DT.Linear(C_1)$	An arbitrary application will be unable to pass data directly to the fixed entry point of a component with linear data topology.
$DT.Arbitrary(App) \rightarrow \{5\} \leftarrow SDT.Explicit-Discrete(C_1)$	A component supporting explicit data transfers will require the governing application to pass data to it explicitly. However, an arbitrary application usually broadcasts data and is therefore unable to direct its

	data transfers.
$DT.Linear(App) \rightarrow \{10\} \leftarrow CS.Concurrent(C_1)$	If an application has a linear data topology it expects that data flows in linear fashion. When a component has a concurrent data structure it can pass data to multiple component at the same time, making the enforcement of linear data topology impossible.
$DT.Linear(App) \rightarrow \{5\} \leftarrow DT.Arbitrary(C_1)$	A linear configuration of components forces the participating components to communicate in linear fashion. A component with arbitrary data topology has no set topology, making it unable to guarantee that it will pass its data to the appropriate component.
$DT.Linear(App) \rightarrow \{5\} \leftarrow IC.Unaware(C_1)$ $DT.Linear(App) \rightarrow \{13\} \leftarrow IC.Unaware(C_1)$	A component that is unaware has no knowledge of other components in the system. It is unable to initialize a data transfer and is therefore not able to make a direct call to pass data. A component with linear data topology requires pointed transfer of data.
$DT.Linear(App) \rightarrow \{5\} \leftarrow SDT.Implicit-Discrete(C_1)$	A linear application requires data to be passed in a linear fashion to specific interface points. A component, which supports an implicit, discrete data transfer, will simply broadcast its data without directing it to a particular point.
$DT.Linear(App) \rightarrow \{5\} \leftarrow SDT.Implicit-Continuous(C_1)$	A linear application requires data to be passed in a linear fashion to specific interface points. A component, which supports an implicit, continuous data transfer, will simply broadcast its data without directing it to a particular point.
$DT.Linear(App) \rightarrow \{5\} \leftarrow SDT.Shared(C_1)$ $DT.Linear(App) \rightarrow \{5\} \leftarrow SDT.Shared(C_1)$	A hierarchical configuration of components forces the participating components to communicate hierarchically. A component that only shares its data will be unable to make direct calls with data.
$DT.Linear(App) \rightarrow \{5\} \leftarrow SDT.None(C_1)$ $DT.Linear(App) \rightarrow \{13\} \leftarrow SDT.None(C_1)$	A component not supporting any type of data transfers will be unable to initialize a data transfer, and hence direct it to a particular interface point. A linear configuration of an application requires the participating components to communicate data using direct calls.

Table 38: Application Data Topology PAIs

Synchronization (Sn): Whether or not the components need to rendezvous.

PAIs	PAI Descriptions
<p>$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow Bk.Non-blocking(C_1)$ $Sn.Synchronous(App) \rightarrow \{8\} \leftarrow Bk.Non-blocking(C_1)$</p>	<p>A synchronous application requires its components to handshake. A non-blocking component can continue processing infinitely making a handshake impossible. Moreover, a component that needs to receive data from other components might incorrectly process its data.</p>
<p>$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow CS.Concurrent(C_1)$ $Sn.Synchronous(App) \rightarrow \{4\} \leftarrow CS.Concurrent(C_1)$ $Sn.Synchronous(App) \rightarrow \{10\} \leftarrow CS.Concurrent(C_1)$</p>	<p>A synchronous application requires its components to handshake. In a concurrent application, control and data transfers might not be sequenced appropriately making the handshake impossible.</p>
<p>$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow CT.Arbitrary(C_1)$</p>	<p>A component with arbitrary control topology might not be able to fulfill the handshaking requirement set forth by the application because there is no set structure on its control topology.</p>
<p>$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow DT.Arbitrary(C_1)$</p>	<p>A synchronous application requires its components to handshake. A component with arbitrary data topology might not be able to fulfill this requirement because there is no set structure on its data topology.</p>
<p>$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow IC.UnAware(C_1)$</p>	<p>An unaware component in a synchronous application will be unable to perform the handshaking required by the governing application. It has no knowledge of other components in the application and therefore does not have the ability to initialize a control transfer.</p>
<p>$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow SCT.Implicit(C_1)$</p>	<p>A synchronous application requires its components to handshake. A component that transfers its control implicitly will simply broadcast its control and not direct it to a specific point. Thus, it will be unable to specifically handshake with another component.</p>
<p>$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow SCT.None(C_1)$ $Sn.Synchronous(App) \rightarrow \{12\} \leftarrow SCT.None(C_1)$</p>	<p>A component that has no supported control transfer is unable to initialize a control transfer and is therefore unable to handshake with other components as required by a synchronous application.</p>
<p>$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow SDT.Implicit-Discrete(C_1)$</p>	<p>A component supporting implicit and discrete data transfers will be unable to specifically handshake with other components. It simply broadcasts its data without directing it to a specific point. However, a synchronous application requires its components to handshake when they exchange data.</p>
<p>$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow SDT.Implicit-Continuous(C_1)$</p>	<p>A component supporting implicit and continuous data transfers will be unable to specifically handshake with other components. The causes are the same as seen in the PAI immediately above.</p>
<p>$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow SDT.Shared(C_1)$ $Sn.Synchronous(App) \rightarrow \{13\} \leftarrow SDT.Shared(C_1)$</p>	<p>A component that has does not support any type of data transfers, and simply shares its data, will be unable to initialize any data transfers. In addition, it will be unable to handshake with other</p>

	components as required by a synchronous application.
$Sn.Synchronous(App) \rightarrow \{3\} \leftarrow SDT.None(C_1)$ $Sn.Synchronous(App) \rightarrow \{13\} \leftarrow SDT.None(C_1)$	A synchronous application requires its components to handshake. A component that has no ability to transfer its data will be unable to handshake with other components because it is unable to initialize a data transfer.
$Sn.Asynchronous(App) \rightarrow \{3\} \leftarrow Bk.Blocking(C_1)$	An asynchronous application does not require its components to handshake before exchanging control or data. This can leave a component that expects other components to handshake infinitely blocked. The application will not enforce the handshake.

Table 39: Application Synchronization PAIs