

THE UNIVERSITY OF TULSA
THE GRADUATE SCHOOL

EXAMINING SOFTWARE ARCHITECTURE PROPERTY
INTERACTIONS AND THE INFLUENCE OF SYSTEM
REQUIREMENTS

by

Leigh Anne Davis

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

2001

**THE UNIVERSITY OF TULSA
GRADUATE SCHOOL**

**EXAMINING SOFTWARE ARCHITECTURE PROPERTY
INTERACTIONS AND THE INFLUENCE OF SYSTEM
REQUIREMENTS**

by

Leigh Anne Davis

A THESIS

**APPROVED FOR THE DISCIPLINE OF
COMPUTER SCIENCE**

By Thesis Committee

_____, **Chairperson**

ABSTRACT

Davis, Leigh Anne (Master of Computer Science)

Examining Software Property Interactions and the Influence of System Requirements (Chapter I – VIII, pp 1 – 87)

Directed by Dr. Rose Gamble

(134)

With expanding e-business demands ruling the marketplace, and deprecated technologies governing our military installations, the need for secure, quick-to-market solutions has become paramount. Developers have turned to component-based software engineering to fulfill their system requirements in a timely and inexpensive manner. However, the integration of existing components often proves more difficult due to complex interaction problems manifested in implementation.

Emerging research has shown that interoperability problems can be traced to the software architecture of the components and integrated application. Furthermore, the solutions generated for these problems are guided by an implicit understanding of software architecture. This thesis focuses on identifying, classifying, and organizing characteristics that help define an architectural style by using abstraction and semantic nets. Moreover, we illustrate the relationships between the characteristics and their relevance to interoperability via examples of integrated applications.

ACKNOWLEDGEMENTS

“All the means of action-- the shapeless masses, the materials-- lie everywhere about us; what we need is the celestial fire to change the flint into crystal, bright and clear” –
Henry Wadsworth Longfellow

“Science can amuse and fascinate us all, but it is engineering that changes the world.” -
Isaac Asimov

First and foremost, I would like to thank Dr. Rose Gamble. She is the reason this thesis is being read today. Her support, encouragement, and insight have been an inspiration during my journey towards this degree. I would also like to acknowledge my thesis committee, including Dr. Roger Wainwright and Dr. Stephen Rockwell, for taking the time to review and strengthen my work. I would like to thank the members of my research team SEAT: Dan Underwood, Daniel Flagg, Gerður Jónsdóttir, and Jamie Payton. Jamie, in particular, has been my friend and sounding board these past two years and, for that, I am indebted to her. Last but not least, I would like to thank my husband, Bill Jackson, for starting me on the road to my master’s degree. He has had faith in me when I could not muster it, and in the words of Robert Frost, that has made all the difference.

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	III
ACKNOWLEDGEMENTS	IV
TABLE OF CONTENTS	V
LIST OF TABLES	VII
LIST OF FIGURES	VIII
CHAPTER 1	1
1. INTRODUCTION	2
CHAPTER II	6
2. BACKGROUND	7
2.1 Properties Describing a Software Architecture	8
2.2 Pre-Integration Assessment	9
2.3 Connectors	11
2.4 Integration Elements	12
2.5 Post-Integration assessment	13
2.6 Thesis Terminology	13
CHAPTER III	14
3. ARCHITECTURE CHARACTERISTICS FOR INTEROPERABILITY ..	15
3.1 Attempts at Comparison	16
3.2 Uncovering Potential Characteristics for Interoperability	17
3.3 Defining Abstraction Levels	25
3.4 Determining Connectivity	28
3.4.1 <i>Intra-Level Relationships</i>	31
3.4.2 <i>Inter-Level Relationships</i>	33
3.4.3 <i>Transitivity of Semantic Relationships</i>	36
3.4.4 <i>Establishing the Relevant Set</i>	37
CHAPTER IV	39
4. THE INFLUENCE OF APPLICATION REQUIREMENTS	40
4.1 Causes of Incompatibility	41
4.2 The Application – Level Characteristics	42
4.2.1 <i>Control Topology</i>	44
4.2.2 <i>Data Topology</i>	44
4.2.3 <i>Control Structure</i>	45
4.2.4 <i>Synchronization</i>	46

4.3 System-to-Component Interactions	47
CHAPTER V	48
5. THE EVOLUTIONARY CHARACTERISTICS OF ARCHITECTURE ---	49
5.1 Establishing ECOA	50
5.2 The Impact of ECOA	52
5.3 Further Definition of ECOA	54
CHAPTER VI	55
6. THE PRESENCE OF CHARACTERISTICS IN INTEGRATION EFFORTS -----	56
6.1 Examining Component-Level Characteristics	56
6.1.1 <i>Real Estate Locator System</i>	57
6.1.2 <i>A Heterogeneous Computer System for an Academic Department</i>	60
6.1.3 <i>A Software Migration in Telecommunications</i>	61
6.2 The Presence of Application-Level Characteristics	63
6.2.1 <i>Revisiting the Real Estate Locator System</i>	63
6.2.2 <i>The Computer Troubleshooter</i>	64
6.2.3 <i>Comparing the Application Influences</i>	66
CHAPTER VII	71
7. CONCLUSION AND FUTURE WORK	72
CHAPTER VIII	75
8. REFERENCES	76
APPENDIX A	81

LIST OF TABLES

Table 1: System Characteristics	20
Table 2: Control Characteristics - Part 1	21
Table 3: Control Characteristics – Part 2	22
Table 4: Data Characteristics – Part 1	23
Table 5: Data Characteristics – Part 2.....	24
Table 6: Application-Level Characteristics.....	43
Table 7: Characteristics Affecting Interoperability in the Locator Example.....	59
Table 8: Characteristics Affecting Interoperability in the Ffinger Example.....	61
Table 9: Characteristics Affecting Interoperability in the Telecommunication Example	62
Table 10: Application-Level Characteristics Values for the Locator and Troubleshooter Systems.....	64
Table 11: Characteristics Affecting Interoperability in the Troubleshooter Example.....	65

LIST OF FIGURES

Figure 1: Integrated System Terminology	13
Figure 2: Semantic Relationships among Control Characteristics.....	30
Figure 3: Semantic Relationships among Data Characteristics	31
Figure 4: System Requirements interactions with Component Properties.....	47
Figure 5: Package Diagrams for the Locator and Troubleshooter Systems.....	68
Figure 6: Sequence Diagram of Real Estate Locator	69
Figure 7: Sequence Diagram of Computer System Troubleshooting	70

CHAPTER 1
INTRODUCTION

1. Introduction

Due to increasing demand for quick-to-market, cost effective systems, companies have embraced the edicts of software engineering. The precepts of this discipline include the development of software in a managed and organized manner, the use of standard design patterns or styles, and the verification of products through established mathematical techniques. In this way, components (legacy software, commercial-off-the-shelf (COTS) products, and software under design) are constructed or chosen that encompass desired functionality, and they are integrated together to form large, complex systems. Unfortunately, interoperability between components can be a grave problem threatening the integration by grossly delaying deployment and heavily increasing system development cost. Often these problems are not found until after much of the development and implementation of the complex system is complete, limiting resolution choices. Consequently, the composition of the system often requires a large scale-coding effort that wraps needed communication functionality around the closed, inflexible components present. The customized nature of this glue code practically ensures future difficulty as the system and components evolve.

As the need for large, complex software systems has increased, so has the belief that component-based development will ultimately provide the solution to cost and time issues. However, without concrete processes to solve inter-component conflicts, this form of implementation will not, in fact, provide timely, quick-to-market products. Most importantly, some means to predict and/or repair problematic communications between components must be established.

A purely implementation-based answer, middleware, has been developed to alleviate some of the problems faced in heterogeneous component systems. Middleware, a variety of distributed computing services and application development environments that operate between the application logic and the underlying system [CHA99], can be thought of as frameworks for the aforementioned glue code. Unfortunately, middleware has proven to be much less than an easy alternative to custom integration solutions. So far, middleware frameworks have been fixated on interprocess communication, which limits their adaptability in an integration effort. This requires developers with expertise in the product and the respective middleware framework to define a solution. Even with this expertise, customizations and large-scale coding efforts may still be necessary to deploy a product. Due to the infancy of component reusability, this level of skill is difficult to find in-house. Thus, the reliance of companies on consulting from commercial middleware vendors or other third parties leads to a high degree of dependence on a product that may not be well suited to the integration at hand and its future evolution.

A priori detection of interoperability problems has also been attempted. This is done at a course-grained, software architecture level, thus making possible correctness in design of components or -in the case of COTS- correctness in choice of products. Software architecture can be defined as a high-level description of its computational elements (components), the means by which they interact (connectors), and the structural constraints on that interaction [PW92, SG96]. Styles have been delineated which, if used, prescribe exact properties or characteristics of the system [SG96]. These characteristics have been previously distinguished to provide details that further differentiate architectural styles, such as pipe and filter from event-based systems [SC97]. In turn,

subsets based on style constraints have been examined and compared for their role in integration issues [ABD96, AG97, GAO95, SC97, SIT97, BCTW95, GAC97]. Nonetheless, styles do not encompass all possible functionality available in present-day systems, thus limiting the benefit of such analysis.

The focus of this thesis is the identification and organization of fundamental architectural characteristic sets for explicit consideration as culprits of interoperability conflicts current to both present and evolutionary analysis. By examining component-level characteristics, we explore how interactions between characteristic values implicitly influence the integration strategy and how these characteristics might influence the evolution of the integrated system. However, to ensure a thorough analysis, we delineate architectural characteristics that embody a system's structural requirements (so called application-level characteristics). With these, we show how the configuration and coordination properties desired by the developer interact with the properties of the individual components, thus producing a more complete set of interoperability problems. This provides a means for *pre-integration* conflict assessment. With its use of semantic networks and distinct levels of abstraction, it permits a comprehensive treatment of these various published characteristics in an attempt to maintain a set that will be foundational and extensible as new software architectures emerge and grow.

Through the analysis of these characteristics, research is emerging to provide resolutions to interoperability conflicts in the form of course-grained integration elements that can be uncovered at the design-level [PDUG01, DPJGU01]. This should lead to a mapping from an architectural integration solution to a common middleware framework. As this research matures, it will directly tie in to the characteristic set, linking pre-

integration analysis to middleware choices, all before the system is even implemented. These issues, however, are beyond the scope of this thesis and are the subject of future work.

This thesis is organized into the following chapters. Chapter 2 presents research that is directly relevant to this thesis. In Chapter 3, we establish a framework for defining component-level characteristics. In Chapter 4, we discuss the influence of system requirements and their representative architectural characteristics on component-level characteristic interactions. Chapter 5 presents the evolutionary characteristics of architecture (ECO). Case studies relevant to both component-level and application-level characteristics are described in Chapter 6. In closing, Chapter 7 concludes with final results and future work.

CHAPTER II
BACKGROUND

2. Background

Characteristics are one major part of ongoing software architecture research within the area of component interoperability analysis. Without a principled means to justify selection of and connectivity between architecture characteristics, this analysis can only be performed ad hoc, utilizing informal heuristics. 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 pertinent subject areas.

- Properties Describing a Software Architecture - Properties have been defined with respect to architectural styles [ABD96, AG97, GAO95, SC97, SIT97, BCTW95, GAC97].
- Pre-Integration Assessment - This is a type of analysis technique, using abstract architecture information, which can illuminate potential interaction problems in integration [GAC97, ABD96, SIT97, KCBA97, YBB99].
- Connectors - These entities that describe interactions between components can be used to model solutions to interoperability problems [GAR98, KES99]
- Integration Elements - The impetus behind defining and classifying integration elements (architectural connectors specific to resolving interoperability conflicts) is that explicit description of connector types will allow designers the flexibility to choose the correct interaction schemes in an integrated system [GAR98, AG97].

- Post-Integration assessment - Research is being conducted currently to uncover the impetus behind interoperability conflict [IWY00]. Once an integration architecture is in place, analysis can be performed to detect problems.
- Thesis Terminology – The most contentious terms in this thesis are defined.

2.1 Properties Describing a Software Architecture

The *software architecture* of a system is a high-level description of its computational elements, the means by which they interact, and the structural constraints on that interaction [PW92, SG96]. Characteristics have been defined with respect to architectural styles that include the various types of components and connectors, data issues, control issues, and control/data interaction issues [ABD96, AG97, GAO95, SIT97, SC97, BCTW95, GAC97]. For the most part architectural characteristics are defined from different viewpoints; some are restricted to particular application domains; many stress different qualities of the exposed interface, or may encompass mixed levels of abstraction.

Shaw and Clements [SC97] organize and classify particular styles, defining characteristics for this purpose. The goal of their characterization is four-fold: to establish a uniform descriptive standard, provide a systematic organization to support retrieval, discriminate among different styles, and provide organizing advice for selecting a style. They differentiate styles in the following ways:

- The type of components and connectors
- Data communication

- Control communication
- Control/data interaction.

Components and connectors are defined by high-level abstractions such as process, memory, procedure call or threads. Though all values identify the course-grained implementation, they do not define a specific realization of that idea (there are no function or class names defined). Meanwhile, control communication is characterized by topology, synchronicity, and binding time. These all delineate how control moves through a system. Data issues also include topology and binding time, as well as continuity of data and its mode. Relationships between control and data issues include shape and directionality. All of the above characteristics are defined in Appendix A.

These properties do not necessarily reflect the available information of a component slated for integration in a heterogeneous application. Highly encapsulated components tend to provide little to no detail about their internal functionality, providing a strong impetus to concentrate on broad-based properties.

2.2 Pre-Integration Assessment

A set of initial studies of interoperability has been conducted which focuses on detecting interoperability problems a priori to integration [ABD96, SIT97, KCBA97, YBB99]. Characteristics are defined and utilized to aid in the detection of mismatch (see Appendix A). The phrase *architecture mismatch* is used to describe the underlying reasons for interoperability problems among seemingly “open” software components with available source code [AAG95]. Characterizing software architecture styles (i.e.

event-based or pipe and filter) provides additional details that further differentiate these styles.

Gacek [GAC97] and Abd-Allah [ABD96] define characteristics that typify components, connectors and the constraints therein, as well as, conceptual features such as dynamism, supported data transfer and encapsulation. These are employed by the Architect's Automated Assistant (AAA) to detect potential architectural mismatches. However, the styles considered for analysis are limited, as the tool was unable to distinguish certain styles such as database or blackboard.

Sitaraman [SIT97] also characterizes software architecture styles for the purposes of interoperability problem resolution. Main/subroutine, event-based, and pipe and filter architectural styles are analyzed, and such characteristics as data storage method, data representation, and identity of components are defined from that analysis. These aid in the detection of problems and the formulation of solutions. A decision tree based approach is then used to compare characteristic values, within an intelligent agent infrastructure. The drawback of the system is that it cannot be easily expanded to include other characteristics, styles, and conflicts.

Finally, [KCBA97] taxonomize well-known software architectural elements (component, connectors, styles) using a basic set of characteristic features. The purpose of the classification is to describe allowable combinations of interoperable elements, taking into consideration the interchange of elements to promote communication. The characteristics reflect temporal (over time) and static behaviors of components such as time of control acceptance, forks, data scope, ports, etc.

Nonetheless, characteristics based solely on style properties are not expository enough to detail an integration solution [PKG99]. While, developers may not be able to pinpoint the exact style, they often can describe fundamental control and data characteristics in ways separate from the style, but that are pertinent to interoperability [KG99].

[YBB99] define and compare characteristics to provide an a priori cost assessment of COTS integration. This pre-integration assessment technique uses partial orders of characteristic values to indicate which are hardest to satisfy or are the most restrictive. These values are placed in a vector associated with each component. The vectors are compared and where the values are distinct, the objective is to develop an integration architecture that raises the value of that characteristic in both components to the least common value. A principled method justifying the choice of the defined characteristics is not outlined. Those characteristics that are provided include redundant information coupled with discrepancies in detail among characteristic (high-level versus low-level information). Due to the seemingly ad hoc choice of these architecture characteristics, when attempts are made to delineate several styles per the characteristics, many of them are presented as not relevant to both a style itself and across styles. Thus, the comparison between some styles is impossible due to the lack of defined values in their characteristic sets.

2.3 Connectors

Connectors, entities which describe interactions between components, can be used to model solutions to interoperability problems [GAR98, KES99]. The impetus behind this approach is that explicit description of connector types allows designers the

flexibility to choose the correct interaction schemes in an integrated system [GAR98, AG97]. Connectors, such as procedure calls, pipes, or broadcast events, have been studied in this context. Therefore, it is advantageous to be able to describe many connectors architecturally, and to be able to create unique connectors [GAR98, MMP00]. Architecture definition languages (ADL) such as Wright [AG97], Darwin [MDEK95], and Acme [GMW97] are maturing to accomplish this.

2.4 Integration Elements

Unfortunately, connectors suffer from problems in characterization similar to those of components and architectural styles. There are many types of connectors that range from generic to complex, with variants at every level. However, it is advantageous to be able to describe many connectors architecturally, and to be able to create unique connectors [GAR98]. Architecture definition languages (ADL) such as Wright [ALL97], Darwin [MDEK95], and Acme [GMW97] are maturing to accomplish this.

Integration elements can be defined as architectural connectors for specifically resolving interoperability conflicts. These solutions have been modeled as three basic integration element connectors: *controller*, *translator* and *extender* [KG98]. These integration element connectors supplement the traditional architecture connectors by providing specific functionality to resolve interoperability problems. Because we rely on their definition to discuss types of architectural conflicts, we provide more detail below.

An integration architecture, is defined to be the software architecture description of a solution to interoperability problems between at least two interacting component systems [KG98]. A preliminary taxonomy has been developed to illustrate the relationships between integration architectures and the middleware frameworks and

integration strategies they formulate [KG98].

2.5 Post-Integration assessment

Research is being conducted currently to uncover the impetus behind integration solutions and how they resolve interoperability conflicts [IWY00]. These approaches are concerned with *post-integration assessment*, and, thus, look at the system after an integration solution is in place. [IWY00] employ a formalism called a chemical abstract machine (CHAM) to specify the participating components, including the integration architecture. The results of applying their process to both the intermediate and final solutions were simply compared to illustrate how the deadlock could have been predicted. While this direction aids in the assessment of integration solutions and the conflicts they resolve, it does not provide a means to assess interoperability a priori and pick an appropriate solution.

2.6 Thesis Terminology

For the purposes of this thesis, we use the following terminology (see Figure 1). A module is a component internal to a system that is participating in the integrated application. A component is the participating system. The application is the integrated system of components.

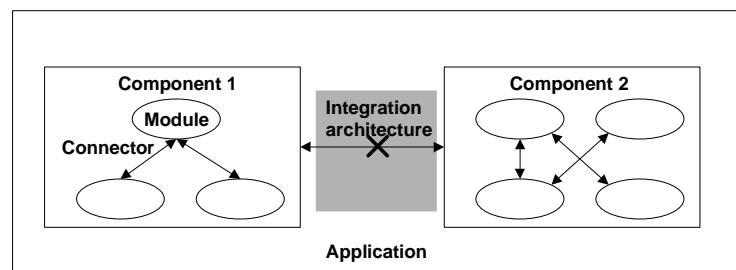


Figure 1: Integrated System Terminology

CHAPTER III
IDENTIFYING AND CATEGORIZING CHARACTERISTICS

3. Architecture characteristics for interoperability

Software architecture characteristics have the potential to deliver clear and early warnings of interoperability problems. To illustrate, suppose you need to put together an investigative task force to examine the reasons why high school girls leave math and science studies. Requirements for the team include timely conclusions, voluntary participation in which only travel is paid, and heterogeneity among team members. Candidate members might be psychologists, mathematicians, computer scientists, and physicists.

You need to ensure this team can perform the analysis and present conclusive findings. For example, will there be problems with meeting scheduling, clashing personalities, lack of experience, or disparate approaches? Vitas can assess qualifications like education, research, and publications. From them, you can determine if any of the people have successfully served on a task force, signifying their collaborative capabilities. You might look for compatible educational institutions or joint publications. Another characteristic might be whether the person has participated in any activities requiring an interdisciplinary group. Geographic distribution might also be of concern, depending on the travel funds available.

You don't have much time to choose the task force, so you specifically look at a set number of characteristics. In fact, you could specify exactly what information the vitae should have and how it should be structured. For a quick assessment, you do not need to know how they function day to day, what kind of computer they have, or what their family life is like. Once you establish your concerns with your potential team, you

may delve deeper before final assignment, possibly calling them to discuss the problems you foresee, calling references, or even holding a preliminary meeting.

By the same principle, the software architecture of a component presents a set of characteristics that pertain to a component's expected data and control exchange. In the following sections we define an extensible framework of characteristics for use in interoperability assessment. The internal details of data structures, function calls, protocols, etc. are not needed for a quick assessment. These may be of concern once early problems are confirmed.

3.1 Attempts at Comparison

Ideally, like comparing vitas, we should be able to directly compare software architecture characteristics. This is the first step in component interoperability assessment [PKG99]. However, there are several obstacles to overcome.

1. Multitudes of architecture characteristics are currently defined. We found 74 characteristics published in various architecture-related reports and articles [ABD96, AG97, GAO95, SIT97, SC97, BCTW95, GAC97].
2. For a given component, the value of every characteristic may not be known.
3. There are abstraction problems - characteristics are defined at different granularities of detail.
4. There are redundancy problems - characteristics overlap in their definitions.
5. There are incompleteness problems – characteristics do not embody full definitions or a full value set.

6. Definition details of similar characteristics are inconsistent.

Therefore, ad hoc grouping of characteristics will never provide useable results. There must be structure to the grouping and understandable associations between groups for comparison to be accurate and informative.

In the following sections, we employ two methods for grouping, associating, and appending characteristics in a principled manner: abstraction levels and semantic networks. There are multiple aspects of a characteristic to examine. We list a few of these below.

- How does it embody architecture expectations?
- How broad or detailed are its definition and values?
- When, during the design phase, is its value known?
- How is it manifested during implementation?
- How does it relate to or depend on other characteristics?
- What is its defined role in component interaction problems?

3.2 *Uncovering Potential Characteristics for Interoperability*

As discussed, there are redundancies among characteristic meanings, definitions, names, and values. The simple union of overlapping definitions does not provide an understanding of a characteristic's relevance, its interactions, and the individual importance of its values. For example, *concurrency* is defined as a system property in [GAC97, SIT97] with enough divergence to warrant further scrutiny. One defines

concurrency saying, “styles often constrain the number of concurrent threads that may execute within a system (examples are single-threaded and multithreaded)” [GAC97]. The other states that *concurrency* is “how and when events are handled” [SIT97]. However, examining the sources in more detail revealed them to be specializations of a single definition (See Table 3).

The characteristic *type of control* – defined as how the system provides control flow to its components [YBB99] – has a similar definition and values to *control structure* (see Table 3). Control structure can be defined as a measure of both the state of control and the possibility of concurrent execution [SIT97]. Its values can be identified as single-thread, multi-thread, and decentralized, whereas type of control has centralized and decentralized values. As a result, type of control was folded into the definition of control structure.

Relevance in component interactions is also used as an additional criterion to fine-tune the characteristic set. For example, by empirical examination we can surmise which characteristics surface most frequently when two components communicate.

As a first pass through the characteristics we perform the following actions.

1. Check definitions of those characteristics with redundant names.
 - a. Collapse closely related definitions under the same name.
 - b. Assign a new name to distinct definitions or definitions from disparate viewpoints and purposes.

2. Determine redundant definitions across characteristics with different names. Where found, we collapse the definitions under the most commonly used name.
3. Maintain characteristics as they are, where viewpoints and usage cause pointed discrepancies among their definitions.

As a result of this consolidation effort, we form an unstructured set of 21 architectural characteristics from the initial set of 74 (see Appendix A).

In line with current practice [SC97], we organize the representative set into three categories: *system*, *data*, and *control*. The *system* category is concerned with the component as a whole and how the characteristics shape the component system. *Data characteristics* deal with how data resides and moves within the component. *Control characteristics*, similar to data characteristics, address control issues. Table 1 defines each characteristic within the system category. The first column gives the characteristic name. The second column gives the values for that characteristic. The third column gives its definition with a brief example of its use. Some of the definitions have been amended or altered from their original published source, which is given in that last column of the table. Tables 2 and 3 describe the control characteristics in a similar fashion. Table 4 and 5 are dedicated to the data characteristics.

Characteristic	Values	Definition	Source
<i>Identity of Components</i>	aware, unaware	A component's awareness of the existence or identity of those components with which it communicates. Generally, filters in the pipe and filter architectural style are unaware, whereas object-oriented component names are used for method access.	[SIT97]
<i>Blocking</i>	blocking, non-blocking	Whether a component suspends execution to wait for communication. Most knowledge based systems run to completion without interruption and then wait, once done, for execution to be reinitiated.	[KG99]
<i>Module</i>	Filters, Objects, Layers, Knowledge Sources, Blackboard Data Structures, Control, Interpretation Engine, Memory, Process	Modules (see Figure 1) are loci of computation and state. Each module has an interface specification that defines its properties, which include the signatures and functionality of its resources together with global relationships, performance properties, etc. The specific named entities visible in the interface of the module are its interface points.	[SG96, SDKRYZ95]
<i>Connector</i>	Controller, pipes, procedure calls, shared data, implicit invocation	Connectors are the loci of relations among the modules. Each connector has its protocol specification that defines its properties which include rules about the type of interfaces it is able to mediate for assurances about the properties of the interaction, rules about the order in which the things happen, and the commitments about the interaction.	[AG97, SDKRYZ95, SG96]

Table 1: System Characteristics

Characteristic	Values	Definition	Source
<i>Control Topology</i>	hierarchical, star, arbitrary, linear, fixed	The geometric configuration of components in a system corresponding to potential data exchange. A main/subroutine architectural style has a hierarchical control topology.	[SC97]
<i>Control Flow</i>	no explicit values	The way in which control moves between the modules of a system. It clarifies the control interactions between internal modules and the exit points at which the control is made available. For example, control flow is bi-directional between modules in a hierarchical topology.	[AG97]
<i>Control Scope</i>	restricted, non-restricted	The extent to which the modules internal to the component make their control available to other modules defines a component's control scope. In a main/subroutine style, certain modules are scoped to receive control only from a parent module.	[KCBA97]
<i>Method of Control Communication</i>	point-to-point, broadcast, multicast	Refers to how control is delivered to other modules. The method details whether control will enter a specific module at a specific point, e.g., pipe and filter architectures; if it will be delivered to those who have registered to receive it, e.g., event-based systems; or if it will be sent to all and only those that need it will use it, e.g., message queuing and broker systems,	[BCTW95]
<i>Control Binding Time</i>	write, compile, invocation, run time	The time when a data interaction is established. Unix pipes and filters bind at invocation time.	[SG96, SDKRYZ95]

Table 2: Control Characteristics - Part 1

Characteristic	Values	Definition	Source
<i>Synchronicity</i>	lockstep, asynchronous, synchronous, opportunistic	The level of dependency of a module on another module's control state. It can operate either when no one else has control (synchronous) or during the execution of other components (asynchronous). Decentralized components are most often asynchronous. On the other hand, a main/subroutine style has synchronous control.	[SG96, SDKRYZ95]
<i>Control Structure</i>	single-thread, multi-thread, decentralized	A measure of both the state of control and the possibility of concurrent execution. Control can reside solely with one module (single-thread), it can reside in multiple modules (multithread), and it can reside in multiple modules without any knowledge of other execution states (decentralized). A web-based interface will often have a decentralized control structure, whereas a pipe and filter style will utilize only a single-thread.	[SIT97]
<i>Concurrency</i>	multi-threaded, single-threaded	The possibility that modules of a component can have simultaneous control. The number of threads present in the component denotes the concurrency. Databases support interleaved concurrency in transaction processing to allow multiple users to access a single account.	[GAC97]

Table 3: Control Characteristics – Part 2

Characteristic	Values	Definition	Source
<i>Data Topology</i>	hierarchical, star, arbitrary, linear, fixed	The geometric configuration of modules in a system corresponding to potential data exchange. A main/subroutine architectural style has a hierarchical data topology	[SC97]
<i>Data Flow</i>	no explicit values	The way in which data moves between the modules of a system. It clarifies the data interactions between internal modules and the exit points at which the data is made available. A pipe and filter style enforces a linear data flow.	[AG97]
<i>Data Scope</i>	restricted, unrestricted	The extent to which the modules internal to the component make their data available to other modules defines a component's data scope. In a main/subroutine style a variable is only available for the subroutine in which it is defined and must be explicitly passed if needed by another function.	[KCBA97]
<i>Method of Data Communication</i>	point-to-point, broadcast, multicast	Refers to how data is delivered to other modules. The method details whether data will enter a specific module at a specific point, e.g., pipe and filter architectures; if it will be delivered to those who have registered to receive it, e.g., event-based systems; or if it will be sent to everyone and only those who need it will use it, e.g., message queuing and broker systems.	[BCTW95]

Table 4: Data Characteristics – Part 1

Characteristic	Values	Definition	Source
<i>Data Binding Time</i>	write, compile, invocation, run time	The time when a data interaction is established. A Java process allows run-time binding, making it possible to bind object classes together as they are defined.	[SC97]
<i>Continuity</i>	sporadic, continuous	A general measure of the availability of data flow in the system. A pipeline has fresh data available at all times (continuous).	[SC97]
<i>Supported Data Transfer</i>	explicit, implicit, shared	This delineates the type and format of data communication that a component supports as a precursor to actually choosing a method to communicate. For instance, implicit data transfer denotes an indirect mode of transfer as in an event-based system.	[ABD96]
<i>Data Storage Method</i>	repository, data with events, local data, global source, hidden and distributed	Details such as what type of data and how in the system it will be represented are gleaned from the chosen value of this characteristic. A blackboard architecture pattern utilizes a repository, namely the blackboard. Knowledge sources both store and retrieve data in this common space so that they may share knowledge.	[SIT97]
<i>Data Mode</i>	passed, shared, multi-cast, broadcast	How data is communicated/transferred, in the logical sense, throughout the component. An event-based architecture will often broadcast its data.	[SC97]

Table 5: Data Characteristics – Part 2

In the next section we define the interrelationships among these characteristics within their respective categories.

3.3 Defining Abstraction Levels

There are different views of software architecture that contribute to the description of a component system. [KRU95] describes these in his “4+1 view model of architecture.” Software architects often use these views to delineate the point during the development process when certain system properties can be defined.

Design processes encourage this type of incremental abstraction that begins with a generic level of abstraction, and leads, sometimes through many levels, to a very specific view of the problem [CAN98]. Building a new, custom-designed home serves as a good analogy. A customer will approach an architect with requirements for the structure, often including pictures and drawings that reflect their desired design. From this the architect will construct a mock-up of a building they feel embodies the requirements set forth by the customer, making refinements as per the customer’s reactions and added input. When the model fulfills all the customers’ desires, blueprints are drafted from which the structure will actually be built. In this process, the motivation behind each abstraction is apparent. You cannot begin construction on a home using only a couple of pages ripped out of Architectural Digest. However, the dimensions and scope of the project are reflected in those pictures to a degree where one can envision the home built.

For these same reasons, we use three abstraction levels to distinguish among the characteristics. These levels, *orientation*, *latitude*, and *execution*, represent the point at which the value of an architecture characteristic can be assigned during the design effort

[KG99]. Similarly, our levels of abstraction define where characteristic values house the appropriate degree of information to render them comparable. The levels also aid in distinguishing what architecture information is specified in documentation about COTS products, open source products, and in-house products.

We define each level as follows.

The Orientation Level

The characteristics at this level embody the most coarse-grained information in describing both the application requirements and the participating components. They are related to the high-level architectural style of the component. Orientation characteristics are expressive but lack execution-related detail. Thus, they paint an overall picture of the component. Their values often can be gleaned from developer documentation. Hence, the value of an orientation characteristic should be relatively easy to obtain, even from a COTS product.

From this definition, we find that all of the system characteristics from Table 1 appear at this level. They deal with general configuration and coordination related to the architectural style of the components in the system. *Blocking* describes the general style of communicating data and control information, while *identity of components* provides high-level information concerning how or if components are distinguished in the system. *Control topology* (defined in Table 2) and *control structure* (defined in Table3) are at this level because they are concerned with the general internal organization of control exchange. From Tables 4 and 5, *data topology* and *supported data transfer* are

orientation characteristics due to their ability to describe generic internal data organization.

The Latitude Level

These characteristics delineate a finer-grained description of a component system. The characteristics at this level demarcate where and how communication moves through a system. More insight into the design of the system is likely to be needed to define specific values for these characteristics. These could be acquired from open source software, in which more information is available than from a COTS product.

From this definition, *control flow* and *control scope* in Table 2, and *data flow* and *data scope* are at this level of abstraction. Because *data mode* addresses the reachability and liveness of data in a system, it is also at this level.

The Execution Level

These characteristics are further refined to the extent of providing execution details. Their values define aspects of system implementation, allowing in depth analysis of data structures and communication strategies. Values for these characteristics may be difficult to obtain unless source code is available and well understood, such as with a product current in design. Control characteristics that are execution specific are *binding time*, *method of communication*, and *synchronicity* (seen in Tables 2 and 3). Tables 4 and 5 have similar characteristics like *data binding time*, *method of communication*, and *continuity*. The values of

these characteristics contain information such as possible languages used to implement the component, the manner of communication such as call-and-return or remote procedure call, and the degree of handshaking necessary between components.

Overall these levels represent the details associated with determining a characteristic's value. Generally, the more detailed the later in the development process the value is known. Since characteristics at the same abstraction level are directly comparable, the levels can afford multiple passes at analysis as development progresses.

3.4 Determining Connectivity

Knowing a characteristic's level of abstraction provides the foundation for determining its relationship with other characteristics. For instance, we can separate drinks into categories such as champagne, wine, and beer, and compare their price. We could compare price within a category and across categories, but our motivation for comparison would be different, e.g. which champagne should you choose (intra-category) versus whether or not (due to its higher price) you should buy champagne at all (inter-category).

We have three goals for determining connectivity among characteristics.

1. Identify and define intra-level and inter-level relationship parameters.
2. Ascertain uniform (and comparative) relationships among characteristics.
3. Establish a minimal set of representative characteristics to provide an early analysis, similar to the minimal information required in the vita of a task force candidate.

To achieve these goals, we employ semantic networks, using as a basis the definitions of the characteristics, their abstraction levels, and case studies depicting their significance in architectural understanding [ABD96, AG97, GAO95, SC97, SIT97, BCTW95, GAC97].

Semantic networks have been used in the artificial intelligence domain to formalize associationist theories of knowledge [LS93, TUR95]. In this respect, entities derive meaning in terms of a network of associations with other entities. Graphically, semantic networks depict entities as named nodes with labeled links (edges) to show relationships between them. For example, quality inheritance is often depicted as an “is-a” relationship among entities (e.g. a penguin “is-a” bird), delegating pertinent entities to the highest level of abstraction and reducing the size of the knowledge base used for assessment.

A node on the semantic net is an architectural characteristic, while an edge represents a relationship (semantic connection) from one characteristic to another. Thus, the links connect the characteristics by virtue of their definitions and their purpose in describing the component architecture.

For the semantic nets to be expressive, it is necessary to define uniform relationships among characteristics from which we can infer deeper meaning. For instance, “is-a” is not an informative relationship for the refined set of architectural characteristics in Tables 1-5. Instead, we eliminated or combined characteristics with this relationship because such redundancy does not suit our goals for comparison. The similarity that does exist breaks down when usage, viewpoint, and detail are considered.

For clarity, we define a separate semantic net for the control and data characteristics in Tables 2-5. The semantic nets for control and data characteristics are found in Figures 2 and 3, respectively. In this section, we describe the intra-level relationships among characteristics followed by the inter-level relationships. We conclude the section by discussing the relevance of transitivity across links and the results from the analysis.

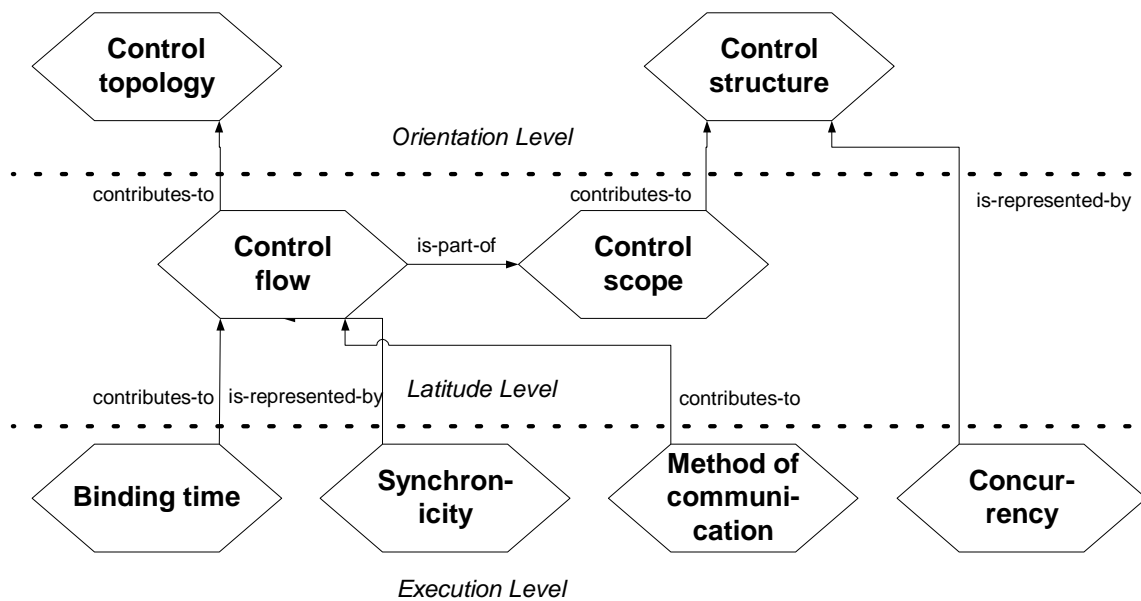


Figure 2: Semantic Relationships among Control Characteristics

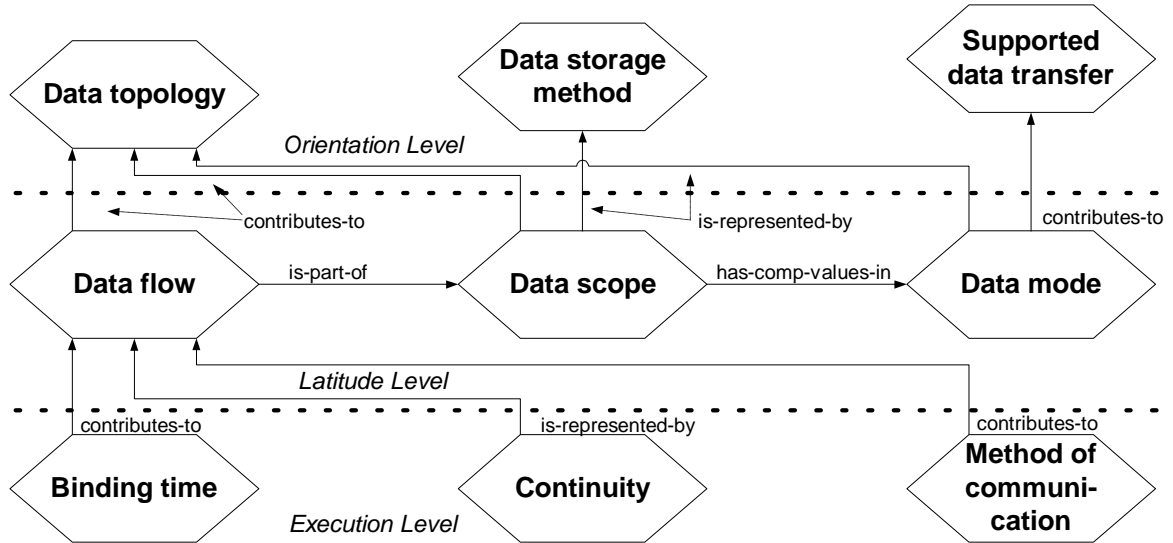


Figure 3: Semantic Relationships among Data Characteristics

3.4.1 Intra-Level Relationships

The intra-level relationships existing between architectural characteristics provide insight into their dependencies at similar abstraction levels. Indeed, we surmise that strongly intra-connected characteristics would be assessed together, suggesting that should one contribute to an interoperability problem, it is likely that the related characteristic poses the same problem [KG99]. The two uniform intra-view labels are *is-part-of* and *has-comparable-values*. We describe each of these in more detail below.

3.4.1.1 The *is-a-part-of* Label

The label *is-part-of* indicates containment. In our usage, it also refers to some existing overlap between characteristics at the same level of abstraction. This overlap is apparent by definition of the characteristics, their purpose, and their values. However, the subordinate characteristics cannot be omitted because there is not a complete semantic overlap.

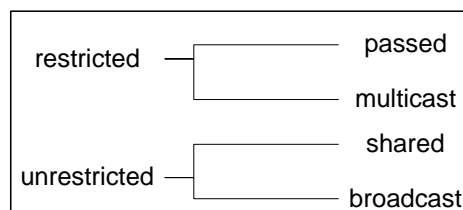
For both control data and exchange, flow *is-part-of* scope. Flow has semantics based on the detailed movement of control or data, while scope indicates where that movement can occur. There are restrictions as to where control or data can flow, and these are also embodied in the scope, indicating connectivity. Hence, flow has attributes that are contained within scope. A more precise definition follows.

Given that X and Y are architecture characteristics, X is-part-of Y if and only if X and Y are at the same abstraction level and either X has attributes embodied in Y or X performs functions also used by Y.

3.4.1.2 The has-comparable-values-in Label

The label *has-comparable-values-in* means that while the characteristic may have different usage, functionality, and/or purpose in architecture description, there exists a definitive or predictive mapping between their values. This label is distinct from *is-part-of* because in the case of *has-comparable-values-in*, the intersection of the values is likely to be empty.

For example, data scope *has-comparable-values-in* data mode (Figure 3). Therefore, there is a mapping between the values of data scope (restricted, unrestricted) and data mode (passed, shared, multicast, broadcast) (see Tables 4 and 5). Specifically, the mapping is seen below.



The definition of *has-comparable-values-in* follows.

*Given X and Y are architectural characteristics, X **has-comparable-values-in** Y if and only if X is at the same abstraction level as Y and there exists at least one value in X that can be mapped onto at least one value in Y.*

The relevance of *has-comparable-values-in* is that if there is a known value for Y, then X, when known, will not conflict with that value - either it supports it or it offers no new information.

3.4.2 Inter-Level Relationships

The inter-level relationships connect characteristics across the previously defined levels to indicate a tie-in between detail and generality. These relationships have great bearing on determining the relevant characteristic set. Without them, we would not be able to establish an encompassing set whose values assured the consideration of lower-level properties. The two uniform inter-level labels are *contributes-to* and *restricts*.

3.4.2.1 The *contributes-to* Label

The *contributes-to* label provides a type of specificity link between characteristics at different abstraction levels. Lower level characteristics connected with this label indicate the direct refinement of architecture descriptions from different viewpoints and levels of understanding during the development process.

Control structure contains certain assumed scoping information. Method of communication refines control flow by extending its definition. For instance, assume that control flow is point-to-point. If explicit communication exists as the method of

communication, then being directed to a known point thus refines control flow. Control flow is also refined by control binding time. Because binding time designates the time of connection among components, it gives shape to the flow of control. In turn, control flow proffers a pattern by which the geometry of the control flow can be established, thus contributing to control topology.

Similarly, the *contributes-to* relation appears frequently in the data semantic net (Figure 3). Data flow *contributes-to* data topology by more accurately specifying the flow protocol. Data accessibility information can be added to data topology given the values of data scope. Data mode also *contributes-to* supported data transfer by providing how data is obtained prior to transfer. The supported data transfer value governs data availability in the integrated system, making it a more overriding characteristic. Method of communication *contributes-to* data flow much in the same way it does control flow. Furthermore, data binding time *contributes-to* data flow by adding information as to where data will flow in relation to the component's data connection time. The detailed definition of *contributes-to* appears below.

*Given X and Y are architectural characteristics. X **contributes-to** Y
if and only if X is at a lower level of abstraction than Y and X
extends or refines some part of Y.*

The values in Y are more general than those in X due to the different abstraction levels. Therefore, the *contributes-to* relation allows information from X to validate what is known about Y.

3.4.2.2 The *is-represented-by* Label

Is-represented-by is similar to the intra-level label *has-comparable-values-in*. Basically, the value of a high-level characteristic is further supported when the value of a low-level characteristic (related by a *represented-by*) becomes known.

Referring to Figure 2, concurrency (at the execution level) *is-represented-by* control structure because concurrency is a type of control execution while control structure governs that execution. For instance, a single-threaded system will not manifest concurrency, but a multi-threaded structure may. Hence, knowledge of concurrency eliminates ambiguity about a multi-threaded structure. Synchronicity, on the other hand, *is-represented-by* control flow, due to the system's need to handshake during communication. Should a system require that control or data be passed in a rendezvous between components, for example, the control flow of the system is represented by this direct flow.

In the data semantic net (Figure 3), data scope *is-represented-by* data storage method, since scoping can limit the usability of certain types of storage. How data is made available in the system details further the geometric form the data takes, hence data mode *is-represented-by* data topology. Furthermore, because continuity defines the type of data flow, it *is-represented-by* data flow. The specific definition follows.

Given that X and Y are architectural characteristics. X is-represented-by Y if and only if X is at a lower level of abstraction than Y and the functionality of the value of X is reflected in some way by the value of Y.

Thus, the values that X provides offer more low-level architectural details when compared to those values of Y.

3.4.3 Transitivity of Semantic Relationships

To further our understanding of the characteristics and their relationships to each other, we take advantage of the transitive relationships appearing in the semantic nets. Because there is both value overlap and mapping inherent in the intra-level links (Section 3.4.1), they have stronger associations than the inter-level links (Section 3.4.2). This is to be expected given similar abstraction levels. Therefore, the meaning of the inter-level relationships is maintained across a transitive link that includes an intra-level connection.

For example, in Figure 3, continuity *is-represented-by* data flow, which *is-part-of* data scope. Because *is-part-of* is a stricter relationship, it can at least be stated that continuity *is-represented-by* data scope. In fact, this is the case, because the continuity of the data flow will inevitably dictate the extent to which data will be made available. Thus, should data be sporadic, it will be restricted instead of freely available in the system.

In the same manner, synchronicity *is-represented-by* control scope through transitivity. This is understandable because a synchronous component must in some way handshake with its partner to transfer control, thus restricting its reach until rendezvous time. Hence, the dependency of one component on another's control state can be reflected in the extent of its reach.

The inter-level links are themselves transitive. Thus, *contributes-to* maintains its meaning across multiple *contributes-to* links. The same relationship applies to *is-*

represented-by links. For example, we can infer that method of communication *contributes-to* control topology. Indeed, method of communication can directly substantiate the underlying topology's value. For instance, point-to-point communication supports a hierarchical topology.

Given that *contributes-to* and *is-represented-by* maintain their meanings across intra-level links, transitivity can be applied from the execution level to the orientation level across *is-part-of* and *has-comparable-values* links. Synchronicity *contributes-to* control structure transitively in that the dependency of a module on another module's control state has a direct effect on the measure of both the state of control and the possibility of concurrent execution. For instance, if a module transmits control synchronously (the modules must handshake with no concurrent execution), the directness of that communication warrants a single-thread of communication.

3.4.4 Establishing the Relevant Set

Considering the different levels of abstraction discussed in the previous section, we hypothesize that the orientation characteristics would provide a representative set that is foundational for interoperability analysis. However, to validate this hypothesis, the set must encompass the information from lower-level characteristics. The semantic nets provide enough information through the direct and transitive links across levels to justify the orientation-level characteristics' use in a preliminary interoperability analysis. In addition, the semantic networks provide a framework for these characteristics which is scalable and extensible by definition.

It is important to note that we do not use a semantic net for the system characteristics in Table 1, because all of these characteristics reside at the orientation level. Future research could reveal a need for intra-level relationships at the orientation level, however this is beyond the scope of this thesis. These system characteristics embody the most breadth of detail; their descriptions are the most readily available. Consequently, these characteristics are necessary to conduct a thorough analysis. Due to this fact, we do not further detail any connectivity present at this level.

CHAPTER IV
THE INFLUENCE OF APPLICATION REQUIREMENTS

4. The Influence of Application Requirements

Different types of interactions can take place between architecture descriptions of components, middleware and systems. An interoperability problem occurs when behavior expectations among these entities are not met. There can be component-to-component interactions, application-to-component interactions, middleware-to-requirements interaction, and so on. The focus of this chapter is defining the properties that principally reveal problematic architecture interactions between application requirements and component properties.

The positive and negative influences of application requirements on interoperability problems directly build on the findings presented in the previous chapter. Through the use of comparable software architecture properties, a meaningful set of interoperability problems can be established for the integration prior to its implementation. In fact, those properties are implicitly considered when resolving interoperability issues. By making them and their use explicit, both apparent and imperceptible interoperability problems are brought to light.

Those interactions found at the component level, however, should be augmented or transformed by the integrated system requirements to reflect a broader range of issues. This process is also done implicitly and often concurrently at implementation time. By considering application requirements as a separate interoperability analysis step, we obtain a more reliable and complete conflict set. This provides insurance that the formulated integration solution will endure.

4.1 Causes of Incompatibility

The previous chapter focused on architectural attributes that functionally characterize the components found in an application. However, the entire system can be functionally characterized as well. Specifically, *application-level architecture characteristics*, which dominate this chapter, are those architecture properties that are synthesized from the integrated application and environment requirements.

An important justification for the choice of the application-level characteristics, defined in Table 6, lies in the already defined abstraction of component-level characteristics. Any comparison between these component-level characteristics and characterized application requirements must occur at parallel levels of abstraction for the assessment to be valuable. Those chosen, then, should encompass a broad range of lower-level functionality.

For example, control topology can be defined as the geometric formation of the control flow in an integrated system. This characteristic identifies the control flow across the participating component systems, hinting at the method of control communication and the restriction of that communication therein.

Our approach is to maintain a small set of characteristics, all of which are at a high level of abstraction. Furthermore, a long-term goal of this research is the automation of this analysis within a tool. A small, broadly defined set serves as low overhead for the developer to determine the characteristic value inputs.

4.2 The Application – Level Characteristics

These characteristics formulate the architectural demands on configuration and coordination of the participating components in the application. Configuration defines the way in which the structure of the system is established, including which components the system contains, where they are located, and how they are interconnected [RE98]. Coordination is concerned with the interaction of the various components, including when an interaction takes place, which parties are involved and what protocols to follow [RE98].

The configuration of the system, that is, which components communicate, influences the component-level characteristic comparisons in two ways.

- It identifies which component are going to communicate, making pertinent the data formats of the communicating components and a means of communicating data/control
- It identifies which component are not going to communicate, making irrelevant those potential problems between unassociated components

Coordination affects the set of component interactions by specifying the directionality of control and data flow. As component interactions at the component-level are assumed to be symmetric, given component A and B, the problem exists between A and B and B and A. This indicates the possibility of multiple resolution strategies. Therefore, unidirectional flow, bi-directional flow or dynamic flow of control/data impact the number of interoperability problems in the following ways.

- Reducing the problem set
- Maintaining the problem set
- Increasing the problem set

In this way, the interoperability problems can gain complexity or become more manageable based on the requirements of the integrated system.

The characteristics of control and data topology address how the control and data flow serve to connect the entire system in a configuration. On the other hand, control structure and synchronization define the coordination of component interactions to form the integrated system.

Table 6 below summarizes the application-level characteristics. The relationship of these characteristics to configuration and coordination requirements is defined in more detail below.

Characteristic	Definition	Values
<i>Control Topology</i>	The geometric formation of the control flow across the integrated system	Hierarchical, Linear, Star, Arbitrary, Fixed
<i>Data Topology</i>	The geometric formation of the data flow across the integrated system.	Hierarchical, Linear, Star, Arbitrary, Fixed
<i>Control Structure</i>	A measure of both the state of control and the possibility of concurrent execution in an application.	Single-Threaded, Multi-Threaded, Decentralized
<i>Synchronization</i>	Whether or not the components need to rendezvous	Synchronous, Asynchronous

Table 6: Application-Level Characteristics

4.2.1 Control Topology

In depicting the desired control interactions among components in an integrated system, this characteristic impacts the pairwise control exchanges between components therein. Different system-wide component arrangements (based on required control interchanges) can alter the complexity of the pair-wise interactions, leading to interoperability problems. For example, it is expected that a root node initiates system execution in a hierarchical control topology, such as a main/subroutine system. In addition, the root node anticipates control to return along the path through which it is passed [SC97]. This places constraints on the control interaction (how and where control flows) between linked components. Depending on the control characteristics of the components, such as their individual topologies and control flow, the call-and-return expectation may not be satisfiable, producing an interoperability problem. For instance, assume that cooperating components internally expect to relinquish control to the operating system, e.g. a Unix filter. It follows that the system's expectations of direct control exchange cannot be met as its participating components are passive and expect to be acted upon.

4.2.2 Data Topology

Like control topology, the chosen value of data topology depicts the arrangement of the components according to their required data interactions. It defines the pairwise comparisons that actually represent data exchanges. The data topology of the system can have a two-fold affect on the component interactions of the integrated system.

It is obvious that when attempting a heterogeneous integration, the data formats of each component will be dissimilar and require translation upon communication.

Therefore, the value of a system's data topology defines the communications that must be converted, e.g., by detailing which components transfer data between them. In this way, the required data flow graph of the application substantiates (if there is data exchange) or shrinks (if there is no data exchange) the number of potential interoperability problems based on this exchange.

Another issue related to the value of the data topology characteristic is how the system expects data to be transferred across the link between components. For instance, assume that the value of data topology for the integrated system is hierarchical. The requirements are that data should be transferred from a node through a single connection to its child or each of its children. A child node, then, must also know to return results to its parent [SC97]. Now consider data transfer properties of particular components in this application. Assume the components encapsulate their data in an event and broadcast it onto the system bus, taking no active part to ensure direct data communication to the required recipient. In this case, the components make their data available to export, but cannot direct it appropriately. Because a hierarchical application topology makes necessary the explicit passing of data, component communication will not comply with system expectations [ABD96].

4.2.3 Control Structure

How the state of control is kept in a system and its degree of concurrency dictates which component governs control at any time during execution, and how that control is governed. This will constrain the individual control structures of the components, impacting the interoperability problems brought about by control interactions. For instance, a single-threaded control structure constrains the system, focusing the locus of

control in one component at a time. Additionally, the individual components will be responsible for its transfer [SIT97]. Assume, then, that two interacting components each have decentralized control structures. Control can reside in any component or in multiple components simultaneously, and there are few constraints on when control must be passed. Given a single-threaded system requirement, the components will not abide a centralized control state which will cease to exist upon a directed transfer of control. Where there were likely no interoperability problems concerning control state at the component level, they are now manifested at the application level

4.2.4 Synchronization

The synchronous or asynchronous values of this characteristic prompt expectations concerning the wait state of components. This can, in turn, affect component interaction [YBB99, DPG00]. As an example, synchronous communication between two components requires their connection to be a rendezvous. They must “meet” in some manner to pass data and/or control as in a procedure call. Now assume that one or both of the components do not discontinue execution subsequent to data and control exchange. This epitomizes passivity on the part of these components. They must be acted upon, interrupted, if you will. Given they do not have the functionality to initialize a rendezvous, they will not know or care where to meet, making simple connection impossible.

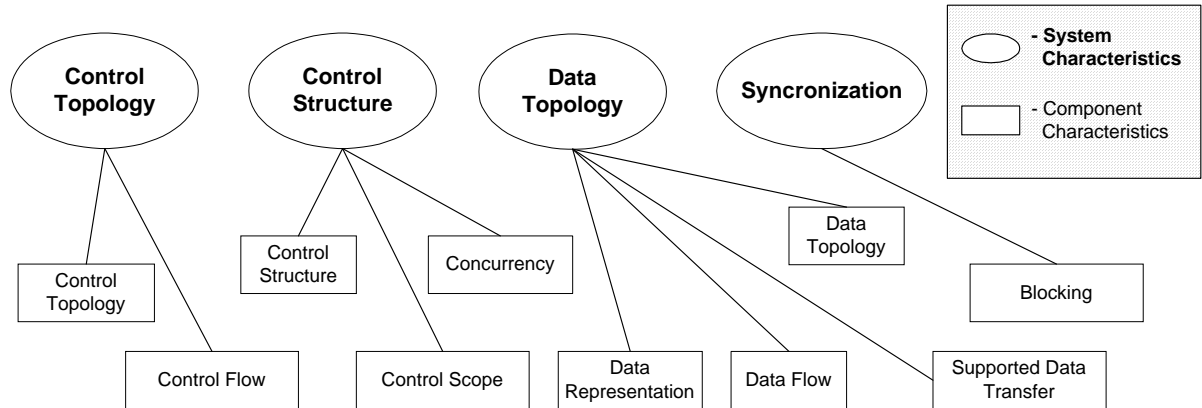


Figure 4: System Requirements interactions with Component Properties

4.3 System-to-Component Interactions

Often requirements are established without concern for the properties of the participating components. Therefore, great effort can be expended in making the components conform to the requirements when their expectations clearly diverge. Figure 4 summarizes the interaction potential across the component and application levels. Complicating these system-to-component conflicts is the degree of detail provided by the components themselves.

It is important to note that the application requirements can have a benign influence on component interactions. In these cases, they simply serve to substantiate the conflicts found at the component level. For example, reconsider the application that requires a decentralized control structure. If its components were to have the same control structure value (decentralized) they are more in line with the overriding system requirement. It is likely that the influence of the system will manifest those conflicts that are already present at the component level. Hence, multiple components trying to access another component concurrently is already a concern.

CHAPTER V
EVOLUTIONARY CHARACTERISTICS OF ARCHITECTURE

5. The Evolutionary Characteristics Of Architecture

Evolution is inevitable when dealing with current software systems. Often it stems from user requests, developer needs, advancements in technology, and component upgrades. COTS products are especially susceptible to evolution, including radical changes, based on attaining and keeping a broad customer base. Sometimes this evolution is embraced as customers see added functionality or performance in the upgraded product. Other times, the product has changed so drastically that customers may be wary of adopting it because of unknown bugs, missing functionality, or lack of backward compatibility.

Problems created by component evolution are magnified when a COTS product is part of a system built from independent, heterogeneous components. Often an integration solution or middleware is used to bridge the interaction among components. When one component is modified or upgraded in a manner that alters its interaction properties, it can affect the way the middleware performs. Indeed, this is a weighty issue for developers because these changes can require that an expensive integration effort be redone. The reason being that evolution can generate additional component integration issues, while, at the same time, making others obsolete.

The impact of a component upgrade on interoperability determines the complexity, and ultimately the cost, of changes to the existing middleware. Simple reinsertion of the component may, in fact, work if the evolution is not drastic. However, trial and error is not the best approach, because it will not provide any clues as to what integration solution changes to make when they *are* needed. In contrast, we advocate

explicit understanding of architectural system properties that might evolve to predict the severity of new integration problems resulting from a component's evolution. This allows the developer to assess tradeoffs in replacing the component with its more recent self. Furthermore, it suggests that in some cases, perhaps the best cause of action is no action at all.

To assess the gravity of evolution the developer must first identify those component properties that reflect evolutionary changes. These properties must be such that they can be expressible for COTS products due to the extent to which the products are self-contained. Our approach uses software architecture characteristics of the component as a basis for these properties. When assembling the integrated system initially, these component characteristics provide a meaningful aid to predict interoperability conflict. Consequently, we hypothesize that the impact on interoperability (both in terms of new and obsolete conflicts) can likewise be predicted, should the values of these characteristics change.

5.1 Establishing ECOA

Characteristics from three recognized aspects of software description (control, data, system) comprise a representative set regarded for evolution. This set, *evolutionary characteristics of architecture* (ECOA), is defined by software properties readily available without benefit of source code. Particularly with COTS, which are more often than not black box, these properties reside at a level of abstraction that makes their qualification a possibility. Knowing the style in which the product was developed, or the language used to implement it, allows some assumptions about its properties, data, control, and system [AAG95]. Indeed, the consumer-driven description of software

provides useful information about its interface. Operating systems, which need to fulfill consumer profiles of personal user, network user, server, etc, define the operations which it must contain to fulfill its promise to that particular customer. Moreover, guidelines for determining values for characteristics such as control structure, supported data transfer, and data storage method often coincide with a need for parallel computing or a decentralized system.

Due to the nature of heterogeneous components it is not always possible to delineate values for every characteristic in the set. However, maintaining a high level of abstraction when trying to qualify characteristics aims for the most open and available information regarding the component systems and their interaction. ECOA are simply representative of what will be most easily described to make an initial assessment.

ECOA were established by first examining characteristic framework developed in Chapter 3 [KG99]. It was a natural extension to consider these characteristics given previous research about their explicit involvement in interoperability problems [DPG00]. As was outlined in Section 3.4, the characteristics influencing interoperability were further stratified through the use of semantic relationships [DGP02]. Due to the stratification and connectivity identified in the semantic networks, they lend themselves as models for change propagation in interoperability analysis. We feel the characteristics which crown the semantic networks (see Figures 2 and 3) serve as both a means for current interoperability analysis, but also identify those characteristics which will likely change when the integrated system evolves. Hence, the set, ECOA, is defined below.

ECOA = {*Identity of Components, Blocking, Supported Data Transfer, Data Topology, Data Storage Method, Control Structure, Control Topology*}

Their choice can be justified in two fundamental ways. First, the top-most characteristics in the semantic networks are encompassing, and stand the best chance to be qualified using available component descriptions. Second, any changes made at the lower levels of architectural description will propagate up to top, insuring that the brunt of predictive evolution analysis can fall there.

5.2 The Impact of ECOA

Within the set, each of the characteristics represents some aspect of a component that, when modified, affects the set of potential interoperability problems. As presented in Section 3.4.3, the transitivity of the inter-level semantic relationships (see Figures 2 and 3) holds. Thus, the evolution of any lower level characteristic value will influence the highest-level of abstraction. Below we present examples of this influence.

- *Identity of components.* The integration will be affected by a component migration to an object-oriented language should the original system employ explicit call and return policies. No longer will each component have the addresses of the components with which it interacts. However, it is possible that each component system relies on these addresses for communication. Therefore, the existing integration will likely not support a specific intermediary that houses these addresses and provides the appropriate direction for the calls.

- *Blocking.* Should a component that blocks be present in a distributed, concurrently executing system, it requires that there be a mechanism in place that will arrest it from its wait state. If, in turn, the component receives an upgrade changing its blocking status to non-blocking, the component now need not be activated but once when the process is created.
- *Supported data transfer.* The interoperability conflict set of a system would be altered should the supported data transfer of a component reflect an explicit form as in the main subroutine style, and then change to accommodate an event system style which implicitly transfers data by broadcast. By making communication of data implicit it is likely the component will no longer know component addresses to which it used to send pointed communication. It will also not functionally be able to direct that communication.
- *Data storage method.* Say an integrated application mandates the direct passing of data among components. When a component is modified to use a shared repository, this repository may need to be available to other components. Thus, there is now a question as to whether the other components can communicate with this repository. Additionally, read/write access issues must also be considered due to the number of new components who might access it.

- *Control structure.* An integrated system developed for a single-user on one computer, which employs a single-thread of control, is to be evolved to a distributed system, making decentralization imperative. New issues such as data accessibility, data integrity, data location, component location, and wait state become paramount, permuting the current set of interoperability issues to accommodate them.
- *Control and data topologies.* A COTS product allows restricted access to its database by a certain set of components in the integrated system. Components that are restricted may obtain access via a built-in unrestricted third party. Eventually the COTS vendor upgrades the product to allow direct access to more components, consequently changing both the control and data topologies if better performance can be achieved. With widespread access, a lessening or pairing down of the interaction set can occur because those previously restricted components have direct database entry now.

5.3 Further Definition of ECOA

Requirements and technical advancements will further define ECOA. Such things as migrating an application to a higher-level language, upgrading to a multi-processor unit, becoming part of an inter-office network, implementing parallel computing, etc. are all evolutionary influences that impact the systems involved. Thus characterization and abstraction of these types of implementation concepts could aid in augmenting ECOA. In this way, the resulting set would continue to depict properties that may impact the evolution of integrated systems.

CHAPTER VI
INTEROPERABILITY ANALYSIS OF CHARACTERISTICS

6. The Presence of Characteristics in Integration Efforts

In Chapter 3, we identify, through principled means, a representative set of architectural characteristics as a foundation for the interoperability analysis of component architectures. In the first section of this chapter, we discuss the emergent characteristic set with respect to three integrated applications. In fact, we illustrate that these characteristics are implicitly considered, rather than explicitly used for analysis of interoperability; the point being that their influence is present but is not utilized for identifying problems. In order for developers to perform interoperability analysis, to make integration decisions, and to trace and reuse those decisions, such architectural information must be made explicit for their use [LUT00]. The importance of facilitating this process increases dramatically when a large number of components are considered.

A common method for understanding integration issues is to examine application results [IWY00, AGI98]. For example, comparing the intermediate and final solutions with CHAM, Invarardi, et al (2000) illustrates how the deadlock could have been predicted. We take a similar approach in this chapter.

6.1 *Examining Component-Level Characteristics*

We discuss three distinct integrated systems. Because the first system was developed as part of a student project at the University of Tulsa, we have detailed knowledge of the integration analysis performed. The latter two systems are from published reports. While we examine these two systems “after the fact,” the inherent use of architectural characteristics is present in the implementations. By making the consideration of these characteristics a best practices approach, we come closer to

attaining the goal of traceability and reusability of integration decisions, aiding both pre-integration and post-integration assessment.

6.1.1 Real Estate Locator System

The in-house application, called the Real Estate Locator system, integrates three independent components: an embedded knowledge based system (KBS), a graphical user interface (GUI), and an external knowledge or data resource (KB or DB). The KBSs were already in use and had not been developed to communicate with an external resource. Only a command line interface was available to interact with the KBS. To construct the application, neither the source code for the KBS nor the external resource was to be altered.

Due to the heterogeneous properties of the components, it is unlikely these components could communicate out of the box. What does a developer implicitly consider to facilitate their interaction? Some generic factors are where and how data and control are made available and exchanged. Because the KBS and external resource are considered closed, qualifying these factors can be difficult. However, understanding their intent and the style by which they communicate data and control leads to assigning orientation characteristic values.

The Real Estate Locator System provides recommendations on the type, price, and location of homes for sale in the Tulsa area. The system is designed for use by prospective homebuyers in the search for affordable housing given certain constraints on size, layout, neighborhood, and price. The system calculates its recommendations by first

obtaining a user profile through a series of interactive questions. Given the user profile, it searches a database of listings to provide the results.

The KBS constructs the user profile. The external resource is a database of real estate listings. The integrated system is designed around a custom interface for concurrent, asynchronous access to multiple, plug and play KBS's by multiple users. These other KBSs might be for home tax assessment, mortgage calculations, etc. This intent translates to a requirement that the system components be highly decoupled. Consequently, the components must work independently, unaware of the executing presence of the other components. Furthermore, the control structure is decentralized to allow the GUI to execute independently from the running KBSs. To accommodate future concurrency, the system components must communicate asynchronously.

For its basic architecture, the Locator system has an arbitrary control topology, with multiple, non-blocking threads. Though the GUI can be designed accordingly, the DB and the KBS do not accommodate this functionality. Thus, conflicts arise among component expectations that impact integration.

- The data formats are different between the GUI and the DB, the GUI and the KBS, and the KBS and the DB, resulting in data translation conflicts that are corrected by intervening translators.
- The KBS and DB each have a single point of entry and exit for control and data exchange, while the GUI has an arbitrary number to allow for multiple users, KBSs, and database listings. This causes a conflict among the components as to what processes perform the exchange and to where it is directed.

- The KBS and the DB must synchronize to communicate with other components, while the GUI expects asynchronous communication. This causes a conflict with respect to how the control and data are transferred.

The orientation characteristics that are apparent in these conflicts are shown in Table 7.

Characteristic	Involvement
Data Topology	Determines for each component the type of entry points for data exchange.
Supported Data Transfer	Indicates the direct or indirect style of the data transmission along with its representation.
Blocking	Indicates the synchronous or asynchronous communication style.
Control Topology	Defines the entry points for control exchange.
Control Structure	Indicates the potential for multiple threads of control and their organization.

Table 7: Characteristics Affecting Interoperability in the Locator Example

The integration solution for the above conflicts implements multiple translators. For instance, one translator resolves data representation problems between the KBS profile information and SQL statements to the database. User commands are also translated. Mediating processes are used to synchronize with the KBS and database to exchange control and data, which is then stored in the newly implemented buffers. These same processes intercept the broadcast data from the GUI and route it along with control to the correct component's entry point. In conjunction with the buffers, polling mechanisms are used by the GUI to gather the data and initiate control exchange asynchronously.

6.1.2 A Heterogeneous Computer System for an Academic Department

This section examines the implementation of a Face-Finger distributed service (ffinger) across heterogeneous computer systems [NBLLSZ88]. The intent of the system was to allow clients residing on different machines to use the ffinger system on the server. The server for ffinger is implemented on a UNIX operating system using remote procedure calls (RPC). The clients that call the server reside on workstations including VAX, XEROX, and SUN machines.

Though basic in its architecture, interoperability problems still are evident.

- Different protocols cause the RPC facilities of the clients to be in conflict.
- There are naming conflicts because the individual clients are independently named and aware of the other components in the application
- Conflicts arise due to the time at which the components become aware of those components with which they communicate. For some, this occurs at run time, while others are bound at compile time.
- The clients and the server use different data formats, so conflicts occur during data transfer.

The orientation characteristics that participate in these conflicts are in Table 8.

Characteristic	Involvement
Connector	The distinct RPC problem is due to different connector protocols.
Supported Data Transfer	Without like data transfer methods, transparent communication is lost.
Data Topology Control Topology	Topology is representative of the different binding times because of the hindrance to data and control flow.
Identity of Components	Inconsistencies in the naming of components arise because of the awareness among the components.

Table 8: Characteristics Affecting Interoperability in the Ffinger Example

The Heterogeneous RPC (HRPC) and the Heterogeneous Name Service (HNS) are created as a solution to the above conflict [NBLLSZ88]. The HRPC system is used as the underlying communication facility provided for all the clients. It implements a single intermediate connector type to facilitate communication across the different protocols and data transfer methods. The HRPC system solves the topology conflicts by delaying the binding of the clients and server to run time, thus making the system more dynamic. The HNS creates a global name space to resolve naming conflicts, providing additional mappings to attain consistency.

6.1.3 A Software Migration in Telecommunications

Architecture properties form the basis for a software migration of telecommunication components [GW99]. Therefore, this published application study makes the values of architecture properties more explicit. The project involves developing the migration path towards an integrated software architecture starting from independent, heterogeneous software components. These software components are responsible for finance and control operations, administrating customer master data, data

access, external sales partners, reporting and statistics, and billing. All components use their own data repository.

The effort to integrate components with such diverse roles and responsibilities presents conflicts.

- Each component has shared data, but there are problematic differences in their data transfer methods.
- There are different data formats and data flows across the components.
- Different control flows make it difficult to establish a single control path through the distributed system. This is a major conflict because there are many components.
- Components have distinct methods of communication.

The encompassing characteristics evident in the above conflicts appear in Table 9.

Characteristic	Involvement
Supported data transfer	Because the modules have different styles to support data transfer and format, there are conflicts in expectations of how transfer takes place.
Data topology	Data topology represents and coincides with the data flow conflicts, causing data sharing and integration conflicts.
Control topology	During control integration, problems can occur due to differences in the form the flow of control takes throughout the system.
Control structure	Inconsistencies in the component's points of entry and exit, combined with their differences in execution, impair its ability to integrate fluidly.

Table 9: Characteristics Affecting Interoperability in the Telecommunication Example

In the actual implementation of the integrated system, the problems mentioned above are resolved through progressive migration phases that involve data and control integration. These phases begin with data exchange support by making customer master data global and translating between formats. Data integration follows among common components. A separate, independent component is used to control the overall functionality across components to establish a single control flow. The final step in the integration is implementation of the underlying transport mechanism (i.e., CORBA) to allow the distributed systems to communicate using different control structures.

When considering the importance of characteristic conflicts to interoperability analysis, sample applications illustrate the way in which component properties are evident within interoperability problem occurrences. We show that orientation-level architectural characteristics explicitly play a role in the conflicts.

6.2 *The Presence of Application-Level Characteristics*

In this section, we revisit the Real Estate Locator from Section 6.1.1 and introduce another system, the Computer Troubleshooting system. Both systems were developed in-house using similar architectures. We employ them to illustrate how the application-level characteristics are implicitly considered as to they impact interoperability conflicts leading to different integration strategies.

6.2.1 *Revisiting the Real Estate Locator System*

As stated earlier, the design intent for the Real Estate Locator is for multi-user access to multiple, interchangeable KBSs. For instance, additional systems may be inserted to form or review the profile. The GUI and real estate database also should be

interchangeable. This intent translates to a highly decoupled system of components. Consequently, the components must work independently, unaware of the executing presence of others.

To satisfy these application requirements, control and data topologies across the application are *arbitrary*. Therefore, control and data flows are not restricted to a particular geometry. Furthermore, the application control structure is *decentralized* to allow the GUI to execute independently from the running KBSs or updated listings. To accommodate the decoupling and concurrency, the system components communicate *asynchronously* (representing the value of the synchronization characteristic). Table 10 delineates these application-level characteristics.

Application Characteristic	Locator Values	Troubleshooter Values
Control Topology	Arbitrary	Hierarchical
Data Topology	Arbitrary	Hierarchical
Control Structure	Decentralized	Single threaded
Synchronization	Asynchronous	Synchronous

Table 10: Application-Level Characteristics Values for the Locator and Troubleshooter Systems

6.2.2 The Computer Troubleshooter

Prior to introducing the application-level characteristics identified in the Troubleshooter, it is important to make clear the component-level characteristics present. In the following subsections, first the interoperability problems engendered by component-level characteristics are addressed, followed by the influence of the application-level characteristics.

6.2.2.1 Interoperability Problems at the Component-Level

As with the Locator system, the Computer Troubleshooter integrates a KBS, an external knowledge source with user defined GUI. The purpose of the Troubleshooter,

however, is much different, as it provides directed and informative troubleshooting of PC components by tying together expertise from available web sites.

The KBS component interacts through a separate GUI component performing a type of triage for one user which narrows the problem to select issues like hardware, OS, etc. With every interaction, the questions become more pointed until a specific area is identified. Throughout the interaction, the KBS seeks specific web pages, such as a hardware manufacturer, to augment its reasoning toward a solution.

The Troubleshooter is meant for single-user access, with synchronous, point-to-point communication. Being developer defined, the GUI embodies the same properties as both the KBS and the DB, who, in turn, accommodate the expectations of the system. Therefore, one can conceive there are less interoperability problems. In fact, there is only one main problem.

- The data formats of the participating components are different. Restricted ontological information is needed to communicate with and add resources to the collection of web pages used.

Characteristic	Involvement
Supported Data Transfer	Indicates the method and type of data transferred.

Table 11: Characteristics Affecting Interoperability in the Troubleshooter Example

Table 11 shows that only one characteristic is the main cause of interoperability problems. This is because the GUI is compatible with the KBS. In addition, the way that the KBS communicates with the Web requires only translation and an ontological component. Hence, for the integration solution translators are used between the GUI and KBS, and again between the KBS and Web.

6.2.2.2 The Influence of the Application

Because the design intent for the Computer Troubleshooting system is that of a single-user system, that is interaction intensive, direct communication must occur between the three distinct components of the system to enhance performance. The volleying nature between the user and KBS, via the GUI, as well as the KBS and the external web pages, bring clarity to values of the architecture characteristics. Both the control and data topologies of the system are *hierarchical* in nature (the GUI serves as root) to accommodate the direct exchange. The KBS, in turn, must have a similar exchange with the web pages. The control structure is *single-threaded* as a single-user, stand alone system. Accordingly, the GUI blocks after passing control to the KBS, requiring the components communicate *synchronously*.

As displayed in Table 10, even with similar components, the application requirements assign different values to the application-level characteristics. These divergent values do not in and of themselves indicate the scale of interoperability problems, or even, that the integration solutions will be different. It is their incompatibility with component properties and expectations. In the next section, we pin the values in Table 10 against the particular component-level characteristics.

6.2.3 Comparing the Application Influences

Using each application-level characteristic, we explain what incompatibilities arise. The discussion is simplified by the fact that the KBS and each of the external data/knowledge sources maintain the same architectural properties across both systems. Because the GUI, though a distinct component, was built as part of the application its communication style is more reflective of the system requirements. To complete the discussion, integration solutions are given for those interoperability conflicts that result

from the interplay between requirements and component properties.

The remainder of this section addresses each application-level characteristic and its influence on the interoperability aspects within each application.

Control topology. In the Locator, arbitrary control topologies do not give distinct expectations as to how and where control flows. In contrast, the KBS and database anticipate direct, reciprocal control exchange. Therefore, it necessitates intervention to make possible the appropriate control interchange. Consequently, a controller is needed to coordinate the polling of the KBS and the database by the GUI to route control flow and an extender is needed to store the information for polling.

The Troubleshooter, on the other hand, follows a hierarchical control topology. Therefore, the components' topological properties (also hierarchical) are compatible with the system requirements. Consequently, this system property as defined for the Locator, does not lead to interoperability conflicts, and requires only traditional connectors.

Data topology. The Locator has an arbitrary data flow in response to the requirement for decoupling. As with control topology, it provokes a similar type of data exchange conflict. Added to this are the incongruous data representations across component. By requiring this data topology the placement of multiple translators between each of the components is necessary to intercept data that is held by extenders and provided to the controller for re-direction.

The Troubleshooter system, likewise, needs data translation as its configuration requires that all components communicate. However, the hierarchical nature of its data exchange does not induce any further conflicts, because the data is passed directly, which is compatible with the component properties.

In Figure 5 the configurations of both the Locator (on the left) and Troubleshooter (on the right) are depicted. The package diagrams indicate what components communicate, as well as the fundamental nature of their communication. In this way, the data and control dependencies can be visually identified

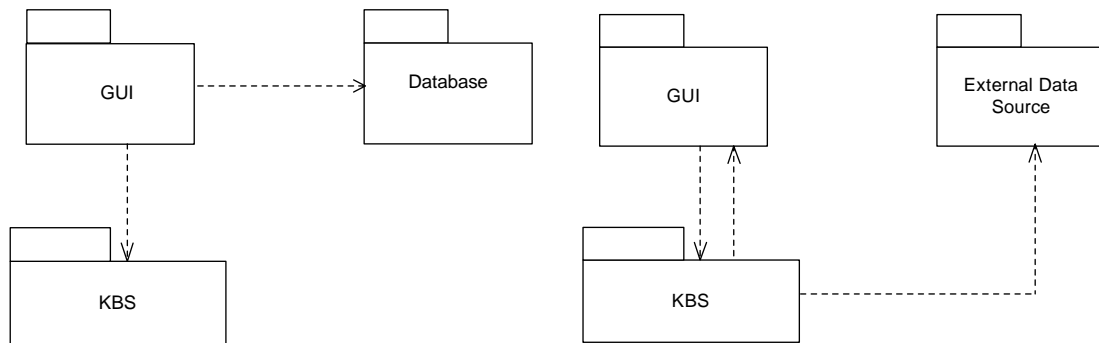


Figure 5: Package Diagrams for the Locator and Troubleshooter Systems

Control structure. In the Locator, a decentralized control structure enables concurrent execution of the components. However, this means that control may be passed to a component from different components (possibly simultaneously). For components like the KBS and database, control would be returned to the component that passed it in (i.e., in a hierarchical fashion). But in this instance, the KBS cannot retain the identity of the component initiating execution, causing a coordination problem. Therefore, the system needs a controller to decide, (e.g. based on the response of the KBS or database) where to direct control for continued execution. In contrast, the Troubleshooter system maintains a single-threaded control structure that is complimentary to its components. Thus, the KBS and external web pages can return control directly to the component that initiated execution.

Synchronization. The Locator system requires asynchronous communication that is an advantage to plug-and-play systems. However, the KBS and database communicate

synchronously - inducing a coordination problem. An extender is needed to buffer the information (control and data) for exchange, while a controller communicates to the component with the appropriate communication method.

The Troubleshooter system uses only synchronized communication, both at the system level and the component level. As a result, these requirements are compatible with those components that require handshaking, like the KBS and web pages use. Therefore, no interoperability conflicts result.

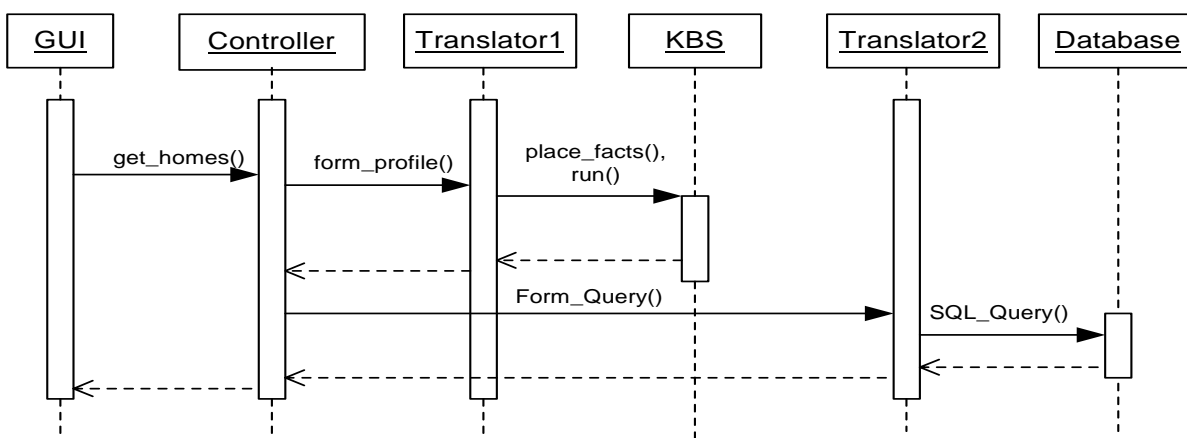


Figure 6: Sequence Diagram of Real Estate Locator

In plotting the control and data flow of the two systems, coordination problems can be charted. Figure 5 gives the base communication paths for both Locator and the Troubleshooter. Notice that the Locator package diagram at the left shows control never being returned to the GUI – indicating that the user never sees the results of their inquiry. Meanwhile, the Troubleshooter expects data and control to be passed between the user (GUI) and the diagnostic tool (KBS). In Figures 6 and 7 a trace of the control and data flow is depicted through a sequence diagram documenting the ways in which the

integration solutions coordinate communication.

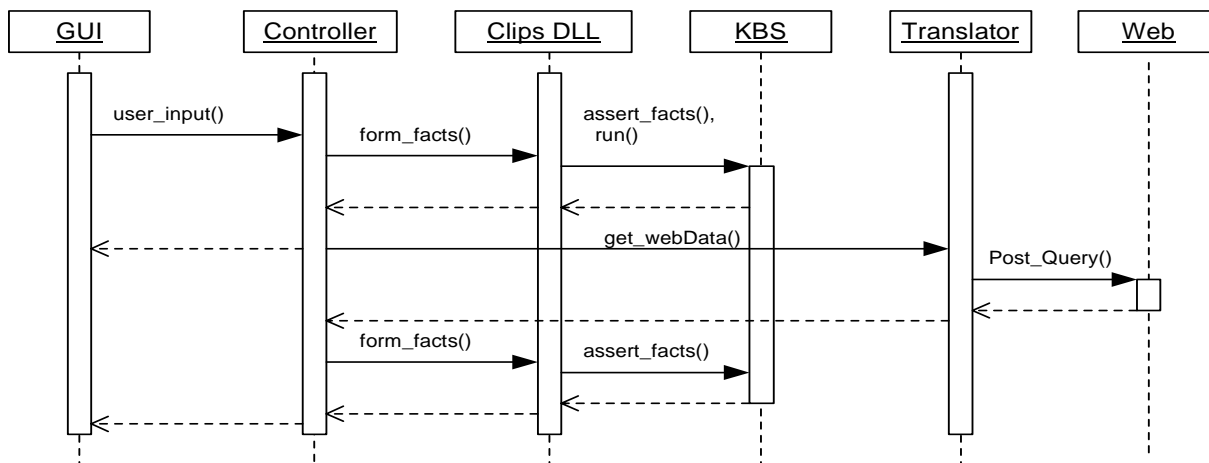


Figure 7: Sequence Diagram of Computer System Troubleshooting

In all, the Locator system requires a single controlling process that routes data and control among the components, multiple buffers to store data, and translators between the communicating components. The Troubleshooter requires only the use of a translator for data representation problems.

CHAPTER VII
CONCLUSION AND FUTURE WORK

7. Conclusion and Future Work

Development can be derailed if misunderstood interoperability issues are addressed well after the application requirements are established and the components are in place. Tackling interoperability issues is made more complex by poorly detailed or unorganized information concerning a system's data and control communication and interoperation. In this thesis, we use principled methods to isolate architectural characteristics that are relevant to interoperability problems in distributed component architectures.

We do not presume that this is a complete set, as further research will likely identify additional characteristics. Particular applications may require more specific characteristics. However, as additional characteristics are defined, the framework we have developed can be reused to establish the abstraction level and the connectivity of the characteristics. Further analysis can assess if new characteristics exist at the granularity of the application and are sufficiently connected to configuration and coordination to warrant application-level status. Likewise, the characteristics can be assessed with respect to their evolution potential.

Currently, we focus our attention on *pre-integration* conflict assessment that can aid in the initial selection of middleware. Research is ongoing to establish a theory of comparison for this purpose. Our continuing work includes these foundational sets of characteristics in an effort to better classify components and their placement in an integrated system. Our goal is to advance the architecture interoperability study further

by formulating a methodology with which we may link characteristics directly to conflicts, and, ultimately, to interoperability problem solutions.

Ultimately, it is necessary to know how and why an existing integration solution is impacted by evolution in order to design it better to withstand changes. This requires an understanding of patterns of change and their resultant effect, thereby offering some insight into flexible middleware designs. A first step toward this goal is identifying the properties of distinct components in an integrated system that, when changed, affect the functioning of a middleware solution.

As a set of software properties provides a good device to explicitly predict interoperability problems, its discrete nature suggests a chance at automating that analysis. A long-term goal for this research is the establishment of an automated interoperability analysis tool. Nonetheless, elaboration on these characteristics is needed to prepare them for participation in a tool, such as discovering a possible hierarchy among the characteristics of all sets, ECOA included. A hierarchy of this type might establish a weight associated with each characteristic and/or value as it relates to the severity and/or frequency of impact on integration when change occurs.

Utilizing broad, but descriptive, architectural characteristics of the component systems provides dual benefits to discover initial interoperability problems, and to assess inevitable future conflicts brought on by evolution. Through our research we hope to pinpoint advantageous designs for consistent and flexible interactions of component systems. This approach discourages anachronistic design by allowing developers insight

into evolutionary transformations, making it easier to implement software that can withstand change.

CHAPTER VIII
REFERENCES

8. References

- [ABD96] Abd-Allah, A. Composing Heterogeneous Software Architectures. Ph. D. Dissertation, Computer Science, University of Southern California, 1996.
- [AAG95] Abowd, G., Allen, A., Garlan, D. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodologies*,4(4): 319-64, 1995.
- [ALL97] Allen, R. A Formal Approach to Software Architecture. Ph. D. Dissertation, Department of Computer Science, Carnegie Mellon University, 1997.
- [AG97] Allen, R., Garlan, D. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodologies*,6(3): 213-49, 1997.
- [AGI98] Allen, R., Garlan, D., Ivers, J. Formal Modeling and Analysis of the HLA Component Integration Standard. In, *FSE-6*, 1998.
- [BCTW95] Barret, D., Clarke, L., Tarr, P., Wise, A. An Event-Based Software Integration Framework 95-048. Laboratory for Advances Software Engineering Research, Computer Science Dept., Univ. of Massachusetts, 1995, revised (1/96).
- [CAN98] Cantor, M.. *Object-Oriented Project Management with UML*. New York, NY: Wiley Computer Publishing, 1998.
- [CHA99] Charles, J. Middleware Moves to the Forefront. *Computer*,22(5), 1999.
- [CHE95] Chen, C. Integrating Existing Event-based Distributed Applications Xerox Corporation, 1995.
- [DGP02] Davis, L., Gamble, R., Payton, J. The Impact of Component Architectures on Interoperability. *to be published in Journal of Systems and Software*, 2002.
- [DPG00] Davis, L., Payton, J., Gamble, R. Toward Identifying The Impact Of COTS Evolution On Integrated Systems. In, *2nd Workshop on Successful Development of COTS*, 2000.
- [DPJGU01] Davis, L., Payton, J., Jonsdottir, G., Gamble, R., Underwood, D. A Notation for Problematic Architecture Interactions. submitted to FSE01, 2001.

- [GAC97] Gacek, C. Detecting Architectural Mismatches During Systems Composition USC/CSE-97-TR-506. Center for Software Engineering, University of Southern California, Los Angeles, CA, 1997.
- [GAR98] Garlan, D. Higher-Order Connectors, *Workshop on Compositional Software Architectures*, Monterey, CA, January 6-7, 1998.
- [GAO95] Garlan, D., Allen, A., Ockerbloom, J. Architectural Mismatch, or Why it is hard to build systems out of existing parts. In, *17th International Conference on Software Engineering*. Seattle, WA, 1995.
- [GMW97] Garlan, D., Monroe, R., Wile, D. ACME: An Architectural Description Language. In, *CASCON*, 1997.
- [GW99] Gruhn, V., Wellen, U. Integration of Heterogeneous Software Architectures- An Experience Report. In, *First Workshop IFIP Conference on Software Architecture*, 1999.
- [IWY00] Inverardi, P., Wolf, A., Yankelevich, D. Static Checking of System Behaviors Using Derived Component Assumptions. *ACM TOSEM*,9(3): 239-72, 2000.
- [KA98] Kazman, R., Carriere, S. View Extraction and View Fusion in Architectural Understanding. In, *ICSR5*. Victoria, B.C., 1998.
- [KCBA97] Kazman, R., Clements, P., Bass, L., Abowd, G. Classifying Architectural Elements as a Foundation for Mechanism Matching. In, *1st International Computer Software and Applications Conference*. Washington, D.C., 14-17, 1997.
- [KG99] Kelkar, A., Gamble, R. Understanding the Architectural Characteristics behind Middleware Choices. In, *1st International Conference in Information Reuse and Integration*, 1999.
- [KES99] 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, *3rd International Software Architecture Workshop*, 1-2, November, 1998.
- [KRU95] Kruchten, P. The 4+1 View Model of Architecture. *IEEE Computer*: 42-50, 1995.

- [LS93] Luger, G., Stubblefield, W. *Artificial Intelligence 2nd Edition*. CA: Benjamin/Cummings, 1993.
- [LUT00] Lutz, J. EAI Architecture Patterns. *EAI Journal*: 64-73, 2000.
- [MDEK95] Magee, J., Dulay, N., Eisenbach, S., Kramer, J. Specifying Distributed Software Architectures. In, *The 5th European Software Engineering Conference*. Barcelona, Spain, 1995.
- [MMP00] Mehta, N., Medvidovic, N., Phadke, S. Towards a Taxonomy of Software Connectors. In, *22nd International Conference on Software Engineering*, 2000.
- [NBLLSZ88] Notkins, D., Black, A., Lazowska, E., Levy, H., Sanislo, J., Zahorjan, J. Interconnecting Heterogeneous Computer Systems. *Communications of the ACM*,31(3), 1988.
- [PDUG01] Payton, J., Davis, L., Underwood, D., Gamble, R.. Using XML for an Architecture Interaction Conspectus. *to appear in XSE2001*, 2001.
- [PKG99] Payton, J., Keshav, R., Gamble, R. System Development Using the Integrating Component Architectures Process. In, *The ICSE '99 Workshop on Ensuring Successful COTS Development*, 49-51, 1999.
- [PW92] Perry, D., Wolf, A. Foundations for the Study of Software Architecture. *ACM SIGSOFT*,17(4): 40-52, 1992.
- [RE98] Radestock, M., Eisenbach, S. Component Coordination on Middleware Systems. In, *IFIP International Conference on Distributed Systems Platforms*, 1998.
- [SC97] Shaw, M., Clements, P. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In, *1st International Computer Software and Applications Conference*. Washington, D.C., 6-17, 1997.
- [SDKRYZ95] Shaw, M., Deline, R., Klien, D., Ross, T., Young, D., Zelesnik, G. Abstraction for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*,21(4), 1995.
- [SG96] Shaw, M., 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.

- [TUR95] Turban, E. *Decision Support and Expert Systems*. Englewood Cliffs, New Jersey: Prentice Hall, 1995.
- [YBB99] Yakimovich, D., Bieman, J., Basili, V. Software Architecture Classification for Estimating The Cost Of COTS Integration. In, *21st International Conference on Software Engineering*. Los Angeles, CA, 296-302, 1999.

APPENDIX A

Appendix A

In this appendix, we list all of the 74 characteristics, their original sources and their definitions.

Appendix – Published Architectural Characteristics		
CHARACTERISTIC	REFERENCE	DEFINITION
<i>Access control</i>	[SIT97]	Indicates the presence of restrictions on a module's willingness to receive messages
<i>Architectural Component</i>	[SG96, SDKRYZ95]	Components are loci of computation and state. Each component has an interface specification that defines its properties, which include the signatures and functionality of its resources together with global relationships, performance properties, etc. The specific named entities visible in the interface of the component are its interface points.
<i>Architectural Connector</i>	[SG96, SDKRYZ95, AG97]	Connectors are the loci of relations among the modules. Each connector has its protocol specification that defines its properties which include rules about the type of interfaces it is able to mediate for, assurances about the properties of the interaction, rules about the order in which the things happen, and the commitments about the interaction.
<i>Blocking</i>	[KG99]	Whether a component suspends execution to wait for communication. Most knowledge based systems run to completion without interruption and then wait, once done, for execution to be reinitiated.
<i>Component packaging</i>	[YBB99]	Whether a component is independent or must be linked with other components, i.e. a DLL versus a class.
<i>Components</i>	[AG97]	Computational units with well defined interfaces
<i>Concurrency</i>	[GAC97]	The possibility that modules of a component can have simultaneous control. The number of threads present in the component denotes the concurrency. Databases support interleaved concurrency in transaction processing to allow multiple

		users to access a single account.
<i>Concurrency</i>	[SIT97]	How and when events are handled
<i>Configuration</i>	[AG97]	This consists of the topology of the control and data, as well as, the decomposition, and distribution of components
<i>Connecting elements</i>	[KCBA97]	Elements that connect modules together
<i>Connectors</i>	[AG97]	Compositional mechanisms for gluing the components together
<i>Continuity</i>	[SC97]	A general measure of the availability of data flow in the system. A pipeline has fresh data available at all times (continuous).
<i>Control binding time</i>	[SC97, SDKRYZ95]	When the identity of a partner in transfer of control operation is established
<i>Control component</i>	[ABD96]	A control component models control that is executed by the underlying machine and which can initiate data and control transfer. It is assumed to have single thread of control
<i>Control connector</i>	[ABD96]	A control connector models the potential for two or more control components to engage in control transfer
<i>Control flow</i>	[AG97]	The way in which control moves between the modules of a system. It clarifies the control interactions between internal modules and the exit points at which the control is made available. For example, control flow is bi-directional between modules in a hierarchical topology.
<i>Control scope</i>	[KCBA97]	The extent to which the modules internal to the component make their control available to other modules defines a component's control scope. In a main/subroutine style, certain modules are scoped to receive control only from a parent module.
<i>Control structure</i>	[SIT97]	A measure of both the state of control and the possibility of concurrent execution. Control can reside solely with one module (single-thread), it can reside in multiple modules (multithread), and it can reside in multiple modules without any knowledge of other execution states (decentralized). A web-based interface will often have a decentralized control structure, whereas a pipe and filter style will utilize only a single-thread.

<i>Control topology</i>	[SC97]	The geometric configuration of components in a system corresponding to potential data exchange. A main/subroutine architectural style has a hierarchical control topology.
<i>Data binding time</i>	[SC97, SDKRYZ95]	The time when a data interaction is established. A Java process allows run-time binding, making it possible to bind object classes together as they are defined.
<i>Data component</i>	[ABD96]	A data component models data that is used to store state or is transferred across data connector
<i>Data connector</i>	[ABD96]	A data connector models the potential for two or more control components or objects to engage in data transfer among themselves
<i>Data flow</i>	[AG97]	The way in which data moves between the modules of a system. It clarifies the data interactions between internal modules and the exit points at which the data is made available. A pipe and filter style enforces a linear data flow.
<i>Data mode</i>	[SC97]	How data is communicated/transferred, in the logical sense, throughout the component. An event-based architecture will often broadcast its data.
<i>Data representation</i>	[SDKRYZ95 CHE95]	The format of data representation (i.e., the data structure and the ontology the systems use).
<i>Data scope</i>	[KCBA 97]	The extent to which the modules internal to the component make their data available to other modules defines a component's data scope. In a main/subroutine style a variable is only available for the subroutine in which it is defined and must be explicitly passed if needed by another function.
<i>Data storage method</i>	[SIT97]	Details such as what type of data and how in the system it will be represented are gleaned from the chosen value of this characteristic. A blackboard architecture pattern utilizes a repository, namely the blackboard. Knowledge sources both store and retrieve data in this common space so that they may share knowledge.
<i>Data topology</i>	[SC97]	The geometric configuration of modules in a system corresponding to potential data

		exchange. A main/subroutine architectural style has a hierarchical data topology
<i>Directionality</i>	[SC97]	If the data and control topologies are the same, this measures whether the control flows in the same direction as the data or the opposite direction
<i>Distribution</i>	[ABD96]	A system may or may not constrain the mapping of system entities to nodes. If the mapping is more than one node, then the style's systems are naturally distributed.
<i>Does a fork occur</i>	[KCBA97]	Indicates whether a process creates other process
<i>Dynamism</i>	[ABD96]	Some styles constrain the topology allowing only static data and control flow. A style is dynamic if and only if it allows non-blocking control connectors (spawns)
<i>Encapsulation</i>	[ABD96]	The presence of embedded components in the system that can only be accessed by a well-defined interface.
<i>Event access protocol</i>	[SIT97]	Defined restrictions on sending a messages
<i>Event definition</i>	[SIT97]	How and when events are defined.
<i>Event representation</i>	[SIT97]	The method of description of an event
<i>Glue</i>	[AG97]	Glue specification describes how the activities of the client and the server are coordinated
<i>Identity of components</i>	[SIT97]	A component's awareness of the existence or identity of those components with which it communicates. Generally, filters in the pipe and filter architectural style are unaware, whereas object-oriented component names are used for method access.
<i>If data transfer</i>	[KCBA97]	Defines if data is transferred under specific conditions
<i>Implementation relationships</i>	[AG97]	Explains how a connector is implemented, i.e. a streaming pipe, a remote procedure call
<i>Interaction relationship</i>	[AG97]	Explains the protocol used to facilitate communication between specific participants
<i>Invocation</i>	[BCTW95]	This is how the components in the system recognize whether or not they should execute.
<i>Isomorphic control and data shapes</i>	[SC97]	Are the control and the data topologies substantially identical to each other
<i>Late binding</i>	[KA98]	Late binding refers to important

		architectural relations that are not bound at compile time. Late binding is found in many complex software systems such as polymorphism, function pointers, parameters provided by the user at run time, etc.
<i>Layering</i>	[ABD96]	Whether or not styles impose layering constraints on control components in the system
<i>Message delivery</i>	[SIT97]	How a module is invoked when an event is announced.
<i>Method of control communication</i>	[BCTW95]	Refers to how control is delivered to other modules. The method details whether control will enter a specific module at a specific point, e.g., pipe and filter architectures; if it will be delivered to those who have registered to receive it, e.g., event-based systems; or if it will be sent to all and only those that need it will use it, e.g., message queuing and broker systems
<i>Method of data communication</i>	[BCTW95]	Refers to how data is delivered to other modules. The method details whether data will enter a specific module at a specific point, e.g., pipe and filter architectures; if it will be delivered to those who have registered to receive it, e.g., event-based systems; or if it will be sent to everyone and only those who need it will use it, e.g., message queuing and broker systems
<i>Module</i>	[SC97, SDKRYZ95]	Modules are loci of computation and state. Each module has an interface specification that defines its properties, which include the signatures and functionality of its resources together with global relationships, performance properties, etc. The specific named entities visible in the interface of the module are its interface points.
<i>Name resolution</i>	[AG97]	Naming resolution is an integral part of the executable system. Naming in the system is essential at the module level and lower level.
<i>Naming Issue</i>	[BCTW95]	In order for the model to receive messages, it must be identifiable. A message server that locates each module that should receive a message typically handles

		naming issues.
<i>Naming issues of components</i>	[SIT97]	Whether a component's style requires that it be named in order for it to be able to communicate.
<i>Object</i>	[ABD96]	It is an encapsulation of a set of data components with the set of control components
<i>Parameter passing</i>	[SIT97]	How parameters to events are transferred to parameters of modules
<i>Port</i>	[ABD96]	It is typically associated with a control component, and is the latter's entry and exit points for data during data transfer
<i>Ports</i>	[AG97]	Defines the logical point of interaction between the component and the environment (basically the entry point or interface of a component).
<i>Ports in or out</i>	[KCBA97]	Associated with control components
<i>Production of output</i>	[SIT97]	The nature of how output is produced
<i>Protocol</i>	[SDKRYZ95]	Indicates the allowable interactions among a collection of components, and provides guarantees about those interactions.
<i>Roles</i>	[AG97]	Describes the expected local behavior of each of the interacting modules or components
<i>Structure</i>	[ABD96]	Attributes of components and connectors, plus approaches to component construction (COTS, manual, project specific reuse, etc)
<i>Supported data transfer</i>	[ABD96]	This delineates the type and format of data communication that a component supports as a precursor to actually choosing a method to communicate. For instance, implicit data transfer denotes an indirect mode of transfer as in an event-based system.
<i>Synchronicity</i>	[SC97, SDKRYZ95]	The level of dependency of a module on another module's control state. It can operate either when no one else has control (synchronous) or during the execution of other components (asynchronous). Decentralized components are most often asynchronous. On the other hand, a main/subroutine style has synchronous control.
<i>System</i>	[ABD96]	It is the non-empty set of interconnected control components or objects satisfying some unique purpose

<i>System Component</i>	[SG96, SDKRYZ95]	Components are loci of computation and state. Each component has an interface specification that defines its properties, which include the signatures and functionality of its resources together with global relationships, performance properties, etc. The specific named entities visible in the interface of the component are its interface points.
<i>System Connector</i>	[SG96, SDKRYZ95], AG97]	Connectors are the loci of relations among the components. Each connector has its protocol specification that defines its properties which include rules about the type of interfaces it is able to mediate for, assurances about the properties of the interaction, rules about the order in which the things happen, and the commitments about the interaction.
<i>Time of control data transmission</i>	[KCBA97]	When data/control is passed? Also might indicate the binding between two modules or separately
<i>Time of control acceptance</i>	[KCBA97]	In case of multiple modules, do they accept control simultaneously
<i>Time of data acceptance</i>	[KCBA97]	In case of multiple modules, do they accept data simultaneously
<i>Topological information</i>	[KA98]	It includes the relations such as allocation of software processes and processors. The relations may be specific when the system is built or when it is running. There may be two ways: static, dynamic
<i>Topology</i>	[ABD96]	Component and connector topology
<i>Trigger</i>	[ABD96]	It associates an action with the reception of data component by control component or object
<i>Triggering capability</i>	[ABD96]	Some styles allow transfer of data along explicit data connectors or global network to cause certain actions, e.g. control transfer or additional data transfers
<i>Type of control</i>	[YBB99]	How the system provides control flow to its components.