The impact of component architectures on interoperability ☆

L. Davis, R.F. Gamble *, J. Payton

Department of Mathematical and Computer Sciences, University of Tulsa, 600 South College Avenue, Tulsa, OK 74104, USA

Abstract

Component interoperability has become an important concern as companies migrate legacy systems, integrate COTS products, and assemble modules from disparate sources into a single application. While middleware is available for this purpose, it often does not form a complete bridge between components and may be inflexible as the application evolves. What is needed is the explicit design information that will forecast a more accurate, evolvable, and less costly integration solution 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. Current technology does not fully identify what must be made explicit about software architecture to aid in comparison of the architectures and expectations of participating entities within the integrated application. Thus, there can be no relief in the expense or the duration of implementing long-term reliance on middleware. The overall goal of our research is to extract and make explicit the information needed to define and build this technology. This paper focuses on identifying, classifying, and organizing characteristics that help to define an architectural style by using abstraction and semantic nets. We illustrate the relationships between the characteristics and their relevance to interoperability via examples of integrated applications. © 2002 Elsevier Science Inc. All rights reserved.

1. Introduction

With the on-going need for more rapid development schedules and less expensive implementations, innovations in system design are a must. Hypothetically, the composition of existing software components and the migration of legacy components to other environments or applications should cut development cost and time. Unfortunately, components often are not as adaptable to new applications or environments as developers assume they would be. Moreover, addressing interoperability problems is often postponed until implementation, in favor of the use of vendor provided middleware.

Middleware is a variety of distributed computing services and application development environments that operate between the application logic and the underlying system (Charles, 1999). The motivation behind middle-

ware is to provide a generic and reusable solution to communication conflicts. Unfortunately, these solutions have failed to perform as expected. One reason is that they require developers with expertise in the product and the respective middleware framework to define a solution. Due to the infancy of component reusability, this level of skill is hard to find in-house. Customizations and large-scale coding efforts by experts in the field may be necessary to deploy a product. Consequently, project managers recruited to make middleware choices have limited understanding of their domain and services. They have to rely on commercial middleware vendors for consulting, which can lead to dependence on a particular product even if it is not well suited to the integration at hand and its future evolution. These aforementioned factors can make heterogeneous component integration costly and time-consuming.

In order for developers to make prudent middleware decisions, they must first understand the root causes of interoperability problems in the system (Mevidovic et al., 2000). Without this understanding, they cannot determine how a given middleware framework manifests solutions to these conflicts. Problems with obtaining this

---

knowledge are compounded by disagreement on exactly what type of component description is most useful for analysis. Technology is therefore essential to assist the developer in analyzing interoperability problems. It is important that this technology must not be orthogonal to industry demands, but rather complimentary.

The main challenges to developing this technology are:

1. Accumulating and identifying relevant properties of the participating components and application requirements for analysis.
2. Organizing key properties into a uniform representation.
3. Developing algorithms to analyze properties for identification of established interoperability conflict types.
4. Defining resolution techniques that can express potential middleware frameworks.

The approach that we advocate is to use software architecture (Shaw and Garlan, 1996) to reveal system properties which are currently implicit in interoperability conflicts (Davis et al., 2000). Our working hypothesis is that there exists a minimal, usable set of architecture characteristics that can determine the early presence of interoperability conflicts. Once defined, this set can be used further to determine the appropriate architecture connectors to resolve those conflicts (Keshav and Gamble, 1998; Mehta et al., 2000).

Our focus in this paper is the identification and organization of fundamental architectural characteristics for explicit consideration as culprits of interoperability conflicts, described in problems 1 and 2 above. This is the first step toward establishing the technology for assessing interoperability conflicts and formulating integration solutions. Hence, the goal of the paper is to establish a standard set of characteristics using principled methods by which each component can be architecturally identified. We utilize the abstract properties from software architecture because they provide a means for powerful assessment without burdening the developer with unnecessary details.

Principled methods allow us to establish that the set we seek to define contains properties that qualify as

- highly relevant,
- wide-encompassing, and
- having values that can be obtained with relative ease.

We believe that a vast number of architectural characteristics play a part in interoperability as evidence by published case studies (Garlan et al., 1995; Allen and Garlan, 1997; Abd-Allah, 1996; Sitaraman, 1997; Barret et al., 1996). Past approaches, while providing the foundation for identification, have not been principled (Abd-Allah, 1996; Allen and Garlan, 1997; Garlan et al.,

1995; Shaw and Clements, 1997; Sitaraman, 1997; Barret et al., 1996; Gacek, 1997). As a result, properties have been incompletely and/or redundantly defined. In addition, similar properties have been defined at different levels of abstraction, causing comparison difficulties. In this paper, we provide a comprehensive treatment of these various published characteristics. This treatment involves the use of distinct levels of abstraction and semantic networks to show how properties can be grouped for comparative evaluation. We conclude the paper to show how implicit understanding of the characteristic set aids in discerning interoperability conflicts.

2. Related work

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 (Perry and Wolf, 1992; Shaw and Garlan, 1996). 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 interactions. Many characteristics are used to differentiate among architectural styles, such as pipe and filter from event-based systems (Shaw and Clements, 1997; Chen, 1995; Abowd et al., 1995), but only subsets based on style constraints have been examined for their role in integration issues (Abd-Allah, 1996; Allen and Garlan, 1997; Garlan et al., 1995; Sitaraman, 1997; Barret et al., 1996; Gacek, 1997). For the most part architectural characteristics are defined from different viewpoints. Thus, some are restricted to a particular application domain, some stress different qualities of the exposed interface, and some encompass mixed levels of abstraction.

Connectors, or entities which describe interactions between components, can be used to model solutions to interoperability problems (Garlan, 1998; Keshav, 1999). 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 (Garlan, 1998; Allen and Garlan, 1997). 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 (Garlan, 1998; Mehta et al., 2000). Architecture definition languages (ADL) such as Wright (Allen and Garlan, 1997), Darwin (Magee et al., 1995), and Acme (Garlan et al., 1997) are maturing to accomplish this.

Generally, all integrated systems require new connectors to simply pass data and control among components. When component interoperability problems occur, more complex connector support is needed to

form the middleware framework. We call these supporting connectors *integration elements* and partition them into *translator*, *controller*, and *extender*. A translator converts data and functions between component system formats and performs semantic conversions. A controller coordinates and mediates the movement of information between component systems using predefined decision-making processes. An extender adds new features and functionality to one or more component systems to adapt behavior for integration. An extender is specifically used as part of an integration solution in which the translator and controller cannot accommodate the full functional need, for example, buffering and security checks (Keshav, 1999).

An *integration architecture* is a software architecture description of the overall solution to interoperability problems between at least two interacting component systems (Mularz, 1994). There is an integration architecture (which is a composition of multiple integration elements) that underlies each common middleware framework, such as CORBA and message queuing (Keshav and Gamble, 1998; Keshav, 1999). However, the interoperability problems that form the foundation of a particular integration architecture are still not well understood. Hence, our research is focused on *pre-integration assessment* to discover inherent interoperability problems that lead to specific integration architectures.

Pre-integration assessment is based on the notion that *architecture mismatch* describes some of the underlying reasons for interoperability problems among seemingly "open" software components with available source code (Garlan et al., 1995). Early studies of architecture mismatch stay within the confines of specific architectural style descriptions (Abd-Allah, 1996; Sitaraman, 1997). Additional research shows that characteristics based solely on style properties might not expose conflict with enough detail to aid resolution (Payton et al., 1999).

Recently, certain characteristics are used to provide a cost assessment of COTS integration (Yakimovich et al., 1999). Partial orders of characteristic values indicate which are hardest to satisfy or 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 solution is to develop an integration architecture that raises the value of that characteristic in both components to the least common value. There are certain drawbacks to this approach. For example, the choice of characteristics is seemingly ad hoc. There is redundancy among their definitions and they were not well suited to describe even common architectures. In fact, the comparison between simple architecture styles is impossible due to the lack of well-defined values in the characteristic set.

Research has also attempted to uncover the impetus behind performance conflicts once an integration architecture is decided. This *post-integration* analysis is useful to detect problems, such as deadlock, that may be due to refining an abstract architecture toward an implementation. One approach employs a formalism called a chemical abstract machine (CHAM) to specify the participating components and the integration architecture (Inverardi et al., 2000). Similarly, ADLs (defined above), such as Wright, use formal methods to specify integration frameworks and capture protocol information to detect potential problems with a given integration architecture (Allen et al., 1998). While this direction aids in the assessment of integration solutions, it does not provide a means to assess interoperability a priori and direct the selection of an initial solution that is appropriate.

Our direction, in contrast, is toward pre-integration conflict assessment to aid the initial selection of middleware; the first step being the identification of assessment properties. For the purposes of this paper, we use the following terminology (see Fig. 1). A *component* is the participating system or subsystem. A *module* is an internal computational element of a component. A *connector* relates two modules. An *integration architecture* relates two components. The *application* is the integrated system of components.

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
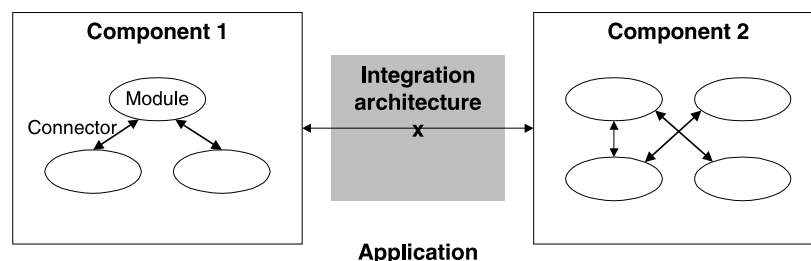

Fig. 1. Integrated system terminology.

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 do not 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 where some of the concerns are 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 components expected data and control exchange. 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 (Payton et al., 1999). 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 (Abd-Allah, 1996; Allen and Garlan, 1997; Garlan et al., 1995; Shaw and Clements, 1997; Sitaraman, 1997; Barret et al., 1996; Gacek, 1997).
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. Simple unioning 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 (Gacek, 1997; Sitaraman, 1997) 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 multi-threaded)" (Gacek, 1997). The other states that *concurrency* is "how and when events are handled" (Sitaraman, 1997). However, examining the sources in more detail revealed them to be specializations of a single definition (see Table 2).

The characteristic *type of control* – defined as how the system provides control flow to its components (Yakimovich et al., 1999) – has a similar definition and values to *control structure*. Control structure can be defined as a measure of both the state of control and the possibility of concurrent execution (Sitaraman, 1997). 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 view points 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.

In line with current practice (Shaw and Clements, 1997), 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. Table 2 describes the control characteristics in a similar fashion. Table 3 is dedicated to the data characteristics.

In the following 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. Kruchten (1995) 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 (Cantor, 1998). 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 an 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 (Kelkar and Gamble, 1999). 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.

3.3.1. 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* and *control structure* defined in Table 2 are at this level because they are concerned with the general internal organization of control exchange. From Table 3, *data topology* and *supported data transfer* are orientation characteristics due to their ability to describe generic internal data organization.

Table 1
System characteristics

Characteristic	Values	Definition	Source reference
Identity of components	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	Sitaraman (1997)
Blocking	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	Kelkar and Gamble (1999)
Module	Filters, objects, layers, knowledge sources, blackboard data structures, control, interpretation engine, memory, process	Modules (see Fig. 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	Shaw et al. (1995) and Shaw and Garlan (1996)
Connector	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	Allen and Garlan (1997), Shaw et al. (1995) and Shaw and Garlan (1996)

3.3.2. 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* from 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.

3.3.3. 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*. Table 2 has 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 processes.

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

Table 2
Control characteristics

Characteristic	Values	Definition	Source reference
Control topology	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	Shaw and Clements (1997)
Control flow	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	Allen and Garlan (1997)
Control scope	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	Kazman et al. (1997)
Method of control communication	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	Barret et al. (1996)
Control binding time	Write, compile, invocation, run time	The time when a data interaction is established. Unix pipes and filters bind at invocation time	Shaw and Clements (1997) and Shaw et al. (1995)
Synchronicity	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	Shaw and Clements (1997) and Shaw et al. (1995)
Control structure	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	Sitaraman (1997)
Concurrency	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	Gacek (1997)

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.

Table 3
Data characteristics

Characteristic	Values	Definition	Source reference
Data topology	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	Shaw and Clements (1997)
Data flow	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	Allen and Garlan (1997)
Data scope	Restricted, unrestricted	The extent to which the modules internal to the component makes 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	Kazman et al. (1997)
Method of data communication	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	Barret et al. (1996)
Data binding time	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	Shaw and Clements (1997)
Continuity	Sporadic, continuous	A general measure of the availability of data flow in the system. A pipeline has fresh data available at all times (continuous)	Shaw and Clements (1997)
Supported data transfer	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	Abd-Allah (1996)
Data storage method	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	Sitaraman (1997)
Data mode	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	Shaw and Clements (1997)

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 (Abd-Allah, 1996; Allen and Garlan, 1997; Garlan et al., 1995; Shaw and Clements, 1997; Sitaraman, 1997; Barret et al., 1996; Gacek, 1997).

Semantic networks have been used in the artificial intelligence domain to formalize associationist theories of knowledge (Luger and Stubblefield, 1993; Turban, 1995). 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 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 1–3. 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 and 3. The semantic nets for control and data characteristics are found in Figs. 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.
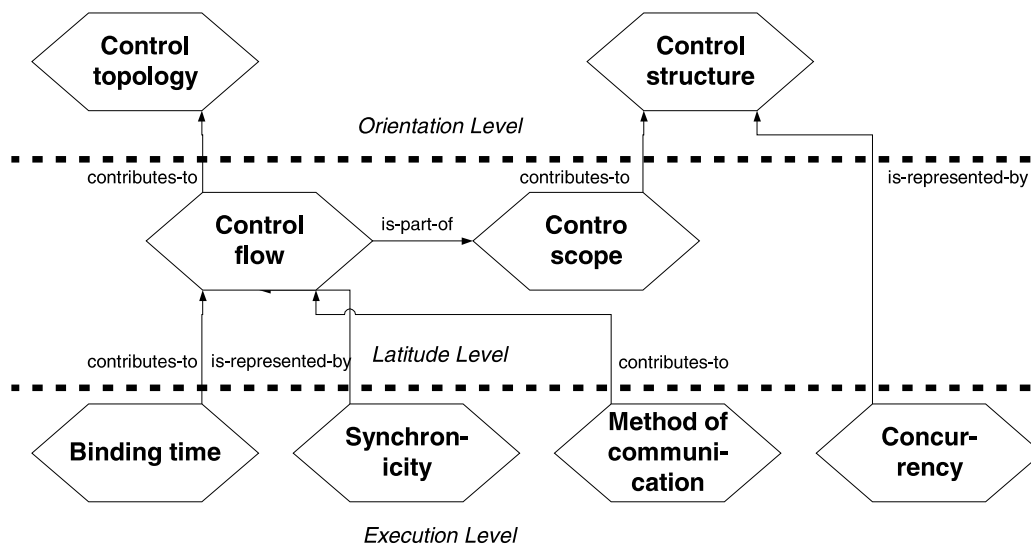

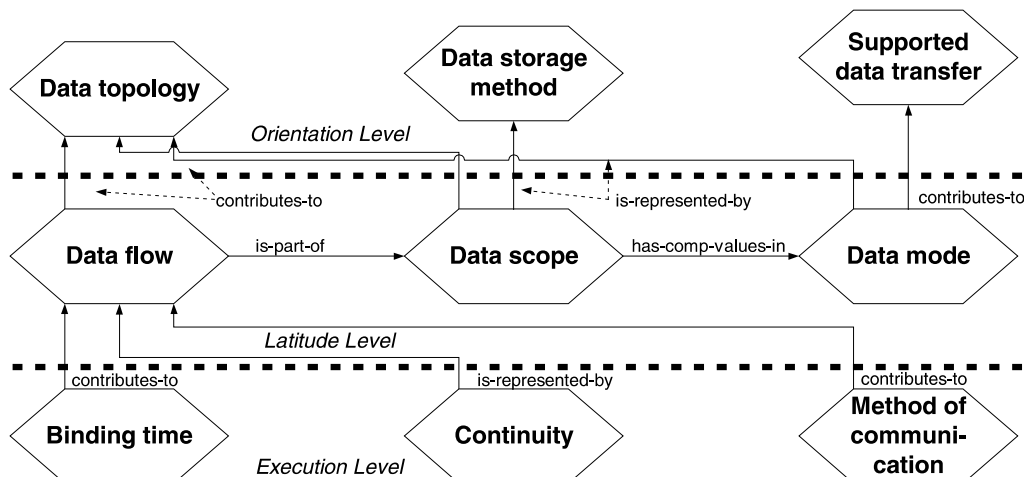Fig. 2. Semantic relationships among control characteristics.

Fig. 3. Semantic relationships among data characteristics.

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 (Kelkar and Gamble, 1999). The two uniform intra-view labels are *is-part-of* and *has-comparable-values*. We describe each of these in more detail below.

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*.

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 (Fig. 3). Therefore, there is a mapping between the values of data scope (restricted, unrestricted) and data mode (passed, shared, multicast, broadcast) (see Table 3). Specifically, the mapping is seen below.
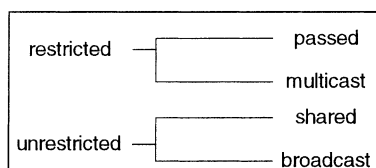
The definition of *has-comparable-values-in* follows:

> 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.

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*.

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 (Fig. 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 *contrib-*

utes-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.

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 Fig. 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 (Fig. 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.

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. By definition, the intra-level links (Section 4.1) have stronger associations than the inter-level links (Section 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 Fig. 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.

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 the representative set we are seeking. 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.

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. 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.

5. The presence of characteristics

The objective of the previous sections was to identify, through principled means, a representative set of architectural characteristics as a foundation for the interoperability analysis of component architectures. In this section, 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 (Lutz, 2000). 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, as seen in the above post-integration assessment approaches (Inverardi et al., 2000; Allen et al., 1998). By comparing the intermediate and final solutions with CHAM, Inverardi et al. (2000) illustrate how the deadlock could have been predicted. We take a similar approach in this section.

The first system we examine was developed in-house, making it possible for us to track the progress toward an integrated solution. The latter two systems are from published reports. While we examine these "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.

5.1. Embedding a knowledge-based systems

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 profile is developed by the KBS. 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, such as:

- 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

Table 4
Characteristics affecting interoperability in the locator example

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 5
Characteristics affecting interoperability in the ffinger example

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	Topology is representative of the different binding times because of the hindrance to data and control flow
Control topology	
Identity of components	Inconsistencies in the naming of components arise because of the awareness among the components

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 4.

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.

5.2. A heterogeneous computer system for an academic department

The first published integrated application that we discuss examines the implementation of a Face-Finger distributed service (ffinger) across heterogeneous computer systems (Notkins et al., 1988). 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 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 5.

The heterogeneous RPC (HRPC) and the heterogeneous name service (HNS) are created as a solution to the above conflict (Notkins et al., 1988). 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.

5.3. A software migration in telecommunications

Architecture properties form the basis for a software migration of telecommunication components (Gruhn and Wellen, 1997). 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, such as

- 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.

Table 6
Characteristics affecting interoperability in the telecommunication example

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

- 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 6.

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 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 in the problematic interactions. We show that orientation-level architectural characteristics explicitly play a role in the conflicts.

6. Conclusion

Development can be paralyzed if misunderstood interoperability issues are addressed well after the application requirements are established. This is made more complex by poorly detailed or unorganized information concerning a system's data and control communication and interoperation. In this paper, we isolate architectural characteristics that are relevant to interoperability problems in distributed component architectures. Through the reduction, abstraction, and linking of the original set of 74 characteristics, a manageable set emerges. Those characteristics in the set that are at the highest level of abstraction, the *orientation level*, are also highly connected to the remaining characteristics.

Furthermore, the evidence of their presence in three separate case studies serves as validation of their representation.

We are not claiming that this is a complete set, as further research may indicate additional characteristics. Depending on particular applications, there may be other specific characteristics needed. However, as additional characteristics are defined, the process can be reused to establish the abstraction level and the connectivity of the characteristics. Our approach is geared toward *pre-integration* conflict assessment that can aid in the initial selection of middleware. With its use of distinct levels of abstraction and semantic networks, it provides a comprehensive treatment of the various published characteristics in an attempt to maintain a foundational set, even as new software architectures emerge and grow.

Research is on-going to establish a theory of comparison for this purpose. Our future work will include this foundational set 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.

Acknowledgements

References

Abd-Allah, A., 1996. Composing heterogeneous software architectures. Ph.D. dissertation, University of Southern California, Los Angeles, CA.

Abowd, G., Allen, R., Garlan, D., 1995. Formalizing style to understand description of software architecture. ACM TOSEM 4 (4), 319–364.

Allen, R., Garlan, D., 1997. A formal basis for architectural connection. ACM TOSEM.

Allen, R., Garlan, D., Ivers, J., 1998. Formal modeling and analysis of the HLA component integration standard. FSE-6.

Barret, D., Clarke, L., Tar, P., Wize, A., 1996. An event-based software integration framework. Technical report 95-048 (revised 1/96), University of Massachusetts.

Cantor, M.R., 1998. Object-Oriented Project Management with UML. Wiley, New York.

Charles, J., 1999. Middleware moves to the forefront. Computer 22 (5), 52 (J.H. Aylor, and D. Carver (eds.)).

Chen, C., 1995. Integrating Existing Event-based Distributed Applications, Xerox Corporation.

Davis, L., Kelkar, A., Gamble, R., 2000. How system architectures impede interoperability. In: Second International Workshop on Software and Performance.

Garlan, D., Allen, R., Ockerbloom, J., 1995. Architectural mismatch or why it is so hard to build systems out of existing parts. In: Proceedings of the 17th International Conference on Software Engineering, pp. 13–22.

Garlan, D., Monroe, R., Wile, D., 1997. ACME: An architecture description interchange language. In: CASCON97.

Garlan, D., 1998. Higher-order connectors. In: Workshop on Compositional Software Architectures.

Gacek, C., 1997. Detecting architectural mismatches during systems composition, Technical report USC/CSE-97-TR-506, University of Southern California, Los Angeles, CA.

Gruhn, V., Wellen, U., 1997. Integration of heterogeneous software architectures – an experience report. In: First Workshop IFIP Conference on Software Architecture.

Inverardi, P., Wolf, A.L., Yankelevich, D., 2000. Static checking of system behaviors using derived component assumptions. ACM TOSEM 9 (3), 239–272.

Kazman, R., Clements, P., Bass, L., Abowd, G., 1997. Classifying architectural elements as foundation for mechanism mismatching. In: Proceedings of COMPSAC, pp. 14–17.

Kelkar, A., Gamble, R., 1999. Understanding the architectural characteristics behind middleware choices. In: Proceedings of the 1st Conference on Information Reuse and Integration.

Keshav, R., Gamble, R., 1998. Towards a taxonomy of architecture integration strategies. In: 3rd International Software Architecture Workshop, pp. 49–51.

Keshav, R., 1999. Architecture integration elements: connectors that form middleware, M.S. thesis, University of Tulsa, Tulsa, OK.

Kruchten, P., 1995. The 4 + 1 View Model of Architecture. IEEE Computer Soc. Press, Silver Spring, MD.

Luger, G., Stubblefield, W., 1993. Artifical Intelligence, second ed. Benjamin/Cummings, Menlo Park, CA.

Lutz, J.C., 2000. EAI architecture patterns. EAI Journal, 64–73.

Magee, J., Dulay, N., Eisenbach, S., Kramer, J., 1995. Specifying distributed software architectures. In: ESES – 5, Barcelona, Spain.

Mehta, N.R., Medvidovic, N., Phadke, S., 2000. Towards a taxonomy of software connectors. In: 22nd International Conference on Software Engineering.

Mevidovic, N., Gamble, R., Rosenblum, D., 2000. Bridging heterogeneous software interoperability platforms. In: 4th International Software Architecture Workshop.

Mularz, D.E., 1994. Pattern-based integration architectures. PLOP.

Notkins, D., Black, A., Lazowska, E., Levy, H., Sanislo, J., Zahorjan, J., 1988. Interconnecting heterogeneous computer systems. Communications of ACM 31 (3).

Payton, J., Keshav, R., Gamble, R., 1999. System development using the integrating component architectures process. In: Proceedings of the First Workshop on Ensuring Successful COTS Development, pp. 49–51.

Perry, D., Wolf, A., 1992. Foundations for the study of software architecture. SIGSOFT Software Engineering Notes 17 (4), 40–52.

Shaw, M., Clements, P., 1997. A field guide to boxology: preliminary classification of architectural styles for software systems. In: Proceedings of COMPSAC97, 1st International Computer Software and Applications Conference, pp. 6–13.

Shaw, M., Garlan, D., 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs, NJ.

Shaw, M., Deline, R., Klien, D., Ross, T., Young, D., Zelesnik, G., 1995. Abstraction for software architecture and tools to support them. IEEE Transactions on Software Engineering 21 (4).

Sitaraman, R., 1997. Integration of software systems at an abstract architectural level. M.S. thesis, University of Tulsa, Tulsa, Oklahoma.

Turban, E., 1995. Decision Support and Expert Systems. Prentice-Hall, Englewood Cliffs, NJ.

Yakimovich, D., Bieman, J., Basili, V., 1999. Software architecture classification for estimating the cost of COTS integration. In: Proceedings from the 21st International Conference on Software Engineering, pp. 296–302.

Leigh A. Davis is a member of SEAT at the University of Tulsa. She attended New York University where she received a B.F.A. in Film Production. Being a great lover of film, but not a financially successful filmmaker, she began her graduate studies in Computer Science at the University of Tulsa, where she will receive an M.S. in May 2001. She intends to continue for a Ph.D. Besides film, Davis is a devotee of pillates, as well as a writer and performer.

Rose F. Gamble is an Associate Professor of Computer Science and the Director of the Software Engineering and Architecture Team (SEAT) at the University of Tulsa. Gamble received her Ph.D. from Washington University in St. Louis in Computer Science. Her research interests are in software engineering and knowledge based systems.

Jamie Payton attends the University of Tulsa, where she has participated in the university's undergraduate research program for the past four years. In May of 2001, she will complete her Bachelor of Science degree in Computer Science. After graduation, Payton plans to attend graduate school to obtain a doctoral degree in Computer Science.