

Component Indicators for Architecture Interaction Assessment

J. Payton L. Davis R. Gamble D. Flagg
Department of Mathematical and Computer Sciences
University of Tulsa
600 South College Avenue
Tulsa, OK 74104 USA
+1 918 631 2988
{payton, davisl, gamble, flaggd}@euler.mcs.utulsa.edu
Technical Report UTULSA-MCS-00-15
August 2000

ABSTRACT

Many interoperability problems within an integrated application can be forecasted using design information provided by software architecture. The architectural style, behavior, and non-functional attributes of the overriding application, its components, and the available middleware all contribute to these problems. ADLs and interface definition frameworks have been formulated to describe a software system. However, there are often very specific usage goals, frequently requiring information that is detailed or simply unavailable, as in the case of COTS. We have found that coarser-grained architecture descriptions of style, behavior, etc., provide powerful predictive capabilities. The problems found are significant enough to impact early requirements and design changes, component and middleware choices, and the direction of further analysis, such as ADL usage.

In this paper, we describe architectural indicators for a first pass interoperability assessment. The goal is to form a set that is minimal and realizable, with the potential for implementable forecasting. We encapsulate these indicators into an *architecture interaction conspectus* (AIC) that fronts each software system and is compatible with current interface definition frameworks.

Keywords

Software architecture, component interoperability, interface definition frameworks

1 INTRODUCTION

The progression of component-based software engineering (CBSE) is essential to the rapid, cost effective development of complex software systems.

Much of the research in CBSE targets how to choose among components for the best composite application. Indeed, such reuse requires effective methods of component retrieval. Equally important is the assessment of interactions among components, the composite application requirements, and middleware. This assessment should be direct, making use of readily available properties based on design information. Moreover, the results achieved should forecast potential interoperability problems that can be corrected by modifications to requirements, alternative components, and the use of middleware.

Unfortunately, there is yet no standard methodology to formulate these component comparisons. Qualities for such a methodology are understandability, practicality, and cost-effectiveness, while generating usable results pertaining to the problematic nature of the required interaction. However, a methodology is only as strong as the representative information foundation on which it relies. Therefore, it is imperative that the proper building blocks are in place, meaning that software interoperability properties must be gathered and quantified in a stable manner.

Researchers have delineated common component properties to produce a representative subset for specific analysis. For instance, specification matching involves comparing component interface definitions for component retrieval [33]. Specification matching is directed toward obtaining components that have the appropriate functionality, without focusing on whether control and data can be properly exchanged. Moreover, architecture definition languages (ADLs) have examined properties for interaction problems among components. However, the properties can be very detailed, taking considerable time to determine their values.

Our goal is to summarize component interaction aspects via a set of indicators that reflect the typical information needed to predict interoperability conflicts during system

design. Software architecture offers a view of the design in coarse-grained structural and behavioral terms. Thus, we rely on this view to describe preliminary indicators derived from architectural style properties, protocol information, and non-functional requirements. Together they form an *architecture interaction conspectus* (AIC). The AIC can be thought of as a component résumé in that it can accompany (or be attached to) the component and provide a quick assessment of its fit in an integrated application.

2 RELATED RESEARCH

In this section we review the relevant research in software architecture and component signatures that provides the impetus for our work.

2.1 Software Architecture

The *software architecture* of a system is a coarse-grained description of its computational elements, the means by which they interact, and the structural constraints on that interaction [25,29]. With such abstraction, it is possible to focus upon important conceptual system issues without becoming entangled in implementation and deployment details. One facet of this abstraction is architectural style, which provides information regarding pattern-based configuration and coordination constraints. Often driven by connectors (interaction mechanisms), architectural styles have been both formally and informally defined for systems such as pipe and filter, event-based, and main/subroutine [2,14,21,24,31]. As a result, there are many types of connectors that range from simple to complex, with variants at every level and taxonomies to express and relate them [19,22].

Architectural characteristics have been researched to further differentiate individual styles, their extensions, and specializations. This set of characteristics includes descriptions of the various types of computational elements, connectors, data issues, control issues, and control/data interaction issues [1,4,5,13,15,28,30]. Architectural characteristics have been viewed with respect to their contribution to component interoperability [1,4,5,13,15,28,30,32]. However, there lacks consistency among the abstraction levels of the characteristics examined, including when they would be known by an application and how they related to each other.

We have found that it is feasible and desirable to limit assessment to abstract characteristics and to separate them according to their involvement in different dimensions of integration. First, *component-level characteristics* contribute to an understanding of the exposed interface of a component system to other external subsystems and are examined for component-component interactions. *Application-level characteristics* formulate the architectural demands on configuration and coordination of the component systems into a single integrated application. By expressing requirements, they impact

application-component interaction [9,10,18].

Although informal diagrams are often used to describe software architecture, they can be ambiguous with respect to component expectations for interaction. This has led to the creation of more formal notations in the form of ADLs [3,20]. These notations describe the structure of software systems in terms of hierarchical configurations of interacting components for system structure, properties, and behavior. Characteristics are modeled differently in each ADL due to differing goals and concerns, e.g., connectors can be modeled as “first-class entities” or in-line as instances depending on the specification emphasis. The ability of an ADL to aid in the design of architectures allows for a preliminary analysis to identify potential conflicts within the system before either an incompatible component is integrated or an incorrect implementation is applied.

Of the common properties represented in an ADL, those that describe services provided and required by each component are very important in architecture interoperability analysis. For instance, Darwin system composition portrays components in terms of the services they offer and their interaction with the surrounding architecture. Defined bindings connect components by linking the ports that *provide* services to the ports of other components that *require* services [20]. In Wright, a component is modeled as having two parts, the interface part and the computation part. The interface part consists of port descriptions from which can be derived provides and requires services. The port protocols govern the component’s interactions with other components within the architecture [3].

Non-functional properties also play a role in determining the architecture of a system [6]. In fact, they often embody the most important and elusive qualities of a software system. However, due to the lack of definitive consensus concerning the qualification of these attributes, their measurement and assessment does not often occur a priori to implementation. When a particular style is chosen for an application, trade-offs often must be made between different non-functional requirements, e.g., the reliability of a main/subroutine styles vs. the performance of an event-based style. While these problems can be reasoned about informally, the properties need qualifiers to provide support to a software architecture analysis method.

Many research efforts have attempted to remedy this issue [8,12,27]. One suggestion is to include in the architecture definition the information that pertains to composing components, the functional assumptions that relate components, and the answers to desired behavior resulting in a “credential” with an attribute/value property list [27]. Another method is to describe components and connectors in terms of both their functional and non-

functional parts. This is achieved by converting detailed requirements into attributes of software such as reliability, performance, accuracy, etc [12]. A third approach endeavors to translate non-functional requirements into goals whose conflicts ultimately help shape the design of the system [8].

It is proven, nonetheless, that software architecture principles provide the best hope of developing a system that fulfills its requirements, functional or otherwise.

2.2 Interface Definition Frameworks

Several researchers seem to agree that fronting components with representative properties can facilitate their comparison. This is analogous to a résumé or patient information attached to medication. An expansive set of such properties, when formulated for a software system, is often referred to as an interface.

While researchers may not concur upon all contents within the interface definition, there are many common elements that cover distinct analysis goals. For example, one interface definition for component retrieval from a library is comprised of a component signature, component specification, and protocol specification [33]. Used in the context of component retrieval, the interface definition serves to eliminate components whose signatures and specifications do not correspond.

With the goal of supporting CBSE, interface definition frameworks can also be used for interface modeling [16]. The component or interface signature within the framework describes the mechanisms for component interaction, such as properties (observable structural elements), operations, and events. Constraints are expressed in terms of pre- and post-conditions, but also as constraints on relationships between elements of the framework. Additionally, the roles of components in the form of services provided and required are specified, along with some non-functional properties, such as performance, reliability, and security.

3 CONSTRUCTING AN ARCHITECTURE INTERACTION CONSPECTUS

Building upon the foundational work described in the background, we compose a minimal set of architectural properties of a system into an AIC, which enables the system's potential interaction capabilities to be assessed. The conspectus forms a major building block for interoperability analysis by highlighting basic, yet relevant, software architecture properties, functional behaviors, and non-functional requirements.

In this section, we describe the contents of our current AIC. We separate the indicators into three main categories: (1) style related characteristics, (2) protocol information, and (3) non-functional properties. For each category, we list and define the indicators, citing their importance for architecture interaction assessment.

3.1 Name and Type

Every entity that participates in the development of an integrated system has a name or identifier that separates it from other systems. We partition these entities into three types. The first type is the *application* design that composes independent subsystems. An application's indicators are in "goal" form, summarizing what is desired for the application. Thus, its indicators can be changed from their initial values as interoperability problems are discovered.

The *components* or participating independent subsystems comprise the second type. For the most part, components are complete, executable systems. Therefore, their indicators are stable. However, some components may be in a design stage. In this case, components can have malleable "goal" indicators. Because applications can themselves be part of a larger complex system, they can also be considered components. With respect to the interoperability assessment that is performed using the AIC, there is only one set of application requirements.

The third system type is middleware, i.e. those subsystems that provide integration services for the application. The manner in which middleware interfaces with the integrated environment (components, application requirements, and other middleware) indicates the need to ascertain distinct properties [21].

Our focus for this paper is on the two types that have similar indicators: applications and components.

3.2 Style Related Characteristics

As discussed in the background, style characteristics have played a role in the type of interoperability problems that can be discovered by understanding the software architecture of a system. We have narrowed the vast number of characteristics to a representative set. Each final characteristic, as defined in Table 1, has multiple semantic links to lower level characteristics defined elsewhere in the literature [9,10,18].

Applications and components differ slightly in how their characteristics impact integration. Table 1 indicates in the second column (using C or A) which characteristics pertain to individual component indicators and which are at the application level. Columns three and four give the definition and value of each characteristic respectively.

Each characteristic in Table 1 is a separate indicator. This allows for interoperability conflicts to be determined from comparing these characteristics with other components and with the application. The resulting generic conflicts lead to design considerations for integration solutions.

3.3 Protocol Information

A protocol describes the transmission of messages between two components. Generally, protocols include the component names and the roles participating in a

CHARACTERISTICS	TYPE	DEFINITION	VALUES
<i>Data Storage Method</i>	C	The details about how data is stored within a system [30]	Repository, Data With Events, Local Data, Global Source, Hidden, Distributed
<i>Supported Data Transfer</i>	C	The method supported by a particular architectural style to achieve data transfer [1]	Explicit, Implicit, Shared
<i>Data Topology</i>	C, A	The geometric form the data flow takes in a system [28]	Hierarchical, Star, Arbitrary, Linear, Fixed
<i>Control Structure</i>	C, A	The structure that governs the execution in the system [30]	Single-Thread, Multi-Thread, Decentralized
<i>Control Topology</i>	C, A	The geometric form the control flow takes in a system [28]	Hierarchical, Star, Arbitrary, Linear, Fixed
<i>Identity of Components</i>	C	Knowledge or awareness of other components in the system [30]	Aware, Unaware
<i>Blocking</i>	C	Whether or not the thread of control is suspended [18]	Blocking, Non-Blocking
<i>Synchronization</i>	A	Whether or not the components need to rendezvous [18,32].	Synchronous, Asynchronous

Table 1: Component-Level Characteristics

communication exchange, the messages exchanged, and the sequence of messages. ADLs often provide the means to specify protocols, since components without compatible protocols are unable to interoperate. However, the ADL specification requires detailed information, beyond that which is needed to accommodate the goals of a first pass interoperability assessment. Additionally, the complete information needed for ADL specification may not always be available, especially with COTS products.

In the AIC, protocol information is provided in terms of the roles that a component may play in an application, and the protocols in which roles participate. We divide protocol information into that which is available only by considering either component information or application information, forming two distinct indicators in the AIC. We do not detail the specific sequencing of messages, since this is beyond the scope of the AIC. The protocol indicators are as follows.

Component Protocol Information. This indicator specifies a list of the probable roles played by a particular component, and the names of the protocols in which they participate. Although protocol names are present, there is no indication of how the roles participate in the protocol.

Application Protocol Information. This indicator specifies the names of all protocols needed within an application. It also identifies the names of roles that should participate in each protocol. However, this role information is general, and is not specific to any particular component.

These two indicators can provide an immediate view of expected behavior. With this information, it is possible to determine if the components that are present fulfill the behavioral expectations of the application. Furthermore, these indicators can be used to decide if the addition of a component or use of an alternative component is feasible in terms of satisfying functional requirements. Hence, even a small amount of protocol information can be used to predict interoperability problems.

3.4 Non-Functional Property Expression

Achieving non-functional requirements goals by looking at only individual components is a necessary first step to incorporate them into design decisions. However, it is not well suited for analysis of heterogeneous component-based software systems, where the components must be viewed as a group with respect to application requirements. Consequently, our research focuses on delineating course-grained, non-functional properties which are the most pertinent when attempting an initial assessment or choosing components that better satisfy requirements. Some of the more important extra-functional properties at the software architecture level are as follows [6,8,27].

- *Performance* refers to issues of time usage versus space needed by a system. Hence, the values **time** and **space** classify this property. These values can be ranked as either high or low.
- *Security* entails encryption strength, correctness, policies and protocols. Some quantifiable aspects

of security are **encryption, authentication, mediation, and audit**. Due to the varying strengths, expenses and complexities of encryption available, it is ranked high, medium, low and none. All other values can simply be ranked high, low, none.

- *Modifiability* allows for evolution of **data constructs, functions, and objects** in a product. Due to the either/or nature of modifiability, a ranking of yes or no is assigned.
- *Reliability* delineates the soundness of **communications**, and the stability of **data** in an

implementation. Assumptions about communications can be made according to the strategies of their transmissions. Thus, it is quantified by the presence of a direct or indirect scheme. Furthermore, data can be ranked as either volatile or persistent to denote its soundness in the application.

There are many other quantifiable aspects of each attribute. For example, access control mechanisms of the application could also be assessed with regard to security. Yet, experience allows the assumption that most software has some type of access control. Encryption, however, is

Figure 1: A View of the Architecture Interaction Conspectus

not typically present and, therefore, should be examined. Consequently, the values outlined in each category are those necessary for a fundamental analysis.

It is important to note that the rankings of each value can be termed “fuzzy.” For instance, should the time performance (with ranks low and high) of an application be medium to high, a high ranking will be chosen.

Figure 1 provides a summary view of the AIC.

4 USING THE ARCHITECTURE INTERACTION CONSPECTUS

The goal of this section is to show the fundamental, yet expressive, assessment capabilities that are available using the AIC presented in Section 3.

4.1 Characteristic Interaction Theory

Each component will have a set of values assigned to all known characteristic indicators (Table 1). For each characteristic, there is at most one value. Analysis is first performed on a component-component basis by examining the following:

- Similar characteristics with like values
- Similar characteristics with mismatched values
- Different characteristics

By distinguishing these different assessments, we call attention to the fact that conflicts are not solely produced by characteristics with mismatched values [1,32].

We define problematic architecture interactions as in [11].

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

The notation

$$\rightarrow T \leftarrow$$

means "problematically interacts causing conflict set T "

where T is a subset of conflicts defined in [11]. For illustration purposes, we present the two conflicts.

F1: Restricted points of control transfer

F2: Sequencing multiple control transfers

As an example, consider Figure 2 with specific characteristics defined for components A , B , and C in which it is known that A does not interact with C . We assume that the application (APP) has a requirement that the control topology of the participating components should be arbitrary. Empirical analysis shows that

$$CT.Hierarchical(B) \rightarrow \{F1, F2\} \leftarrow CS.Decentralized(C)$$

$$CT.Hierarchical(B) \rightarrow \{F1, F2\} \leftarrow CT.Arbitrary(C)$$

$$CT.Hierarchical(A) \rightarrow \{\emptyset\} \leftarrow CT.Hierarchical(B)$$

$$CT.Arbitrary(APP) \rightarrow \{F1, F2\} \leftarrow CT.Hierarchical(A), \\ CT.Hierarchical(B)$$

where the CT and CS refer to the control topology and control structure, respectively. These relations indicate that both conflicts occur when B and C must interact. The conflicts are unioned so that their direct relationship is observable. In general, this problematic interaction can indicate a single solution. Indeed, for this example, mediation can resolve both problems by determining the appropriate sequence of multiple control transfers and directing their point of entry.

The interaction assessment does not end with component-component interactions. By providing indicators for application requirements in the form of architectural characteristics, the AIC allows further analysis for application-component interaction assessment. In the example, it appears that no component-component conflicts occur between A and B . The absence of conflicts indicates the potential for simple connectors to be deployed. However, when application requirements are examined, new problematic interactions with respect to A and B are discovered. In this case, the requirements for the application's control topology forces components that

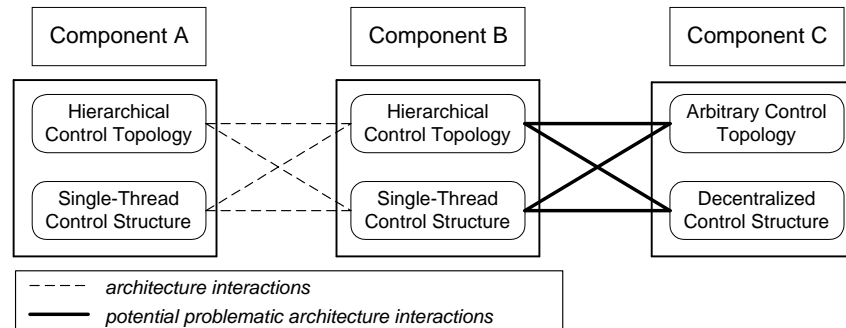


Figure 2: Architecture Interaction Types

normally have a direct control exchange to communicate in a more decoupled manner. Now, intervention is needed in the form of external integration services, such as a mediator, to conduct the control transfer.

The characteristic indicators in the AIC are basic, yet powerful enough for values to be directly assigned and assessed for each component that participates in the integration, as well as the resulting application. If a value cannot be assigned, the indicator is simply not used in the assessment. There are two ways to address the limited number of conflicts that can be directly detected during analysis of partial specifications. First, because generic solutions will be designed to resolve the known conflicts, it is likely that they will also cover those that will not be discovered until the application design matures. Second, as the theory is expanded, values, as well as conflicts, may be derived from partial specifications.

4.2 Protocol Information Assessment

Though the protocol information indicators may seem simplistic, they afford two types of assessment: generic assessment of required behavior, and translation to ADLs.

Generic Protocol Assessment

Using the protocol information provided by the AIC, and assuming that a global naming system is used for role and protocol names, we can make certain interoperability determinations. First, given that components have already been chosen, we can check if the expected behavior of the application has been fulfilled. Alternatively, we can make a choice among candidate components for integration according to whether the component satisfies a needed role. Finally, given the AIC protocol information, we can uncover problems related to missing components or an excess of components attempting to fill a desired role.

As an example of this instance of interoperability determination, consider the following banking example in Figure 3, adapted from Cho and McGregor in [7]. Information that is required by AIC indicators is in bold print. The example is based on the instance of a customer that wishes to withdraw money from an ATM. There are three components involved in the communication exchange: the ATM, the customer database, and the transaction controller.

Now, two banks are completing a merger, and they wish to combine their ATM withdrawal systems, using a central customer database. The new components' protocol information is illustrated in Figure 4. In this setting, the ATM_New component can compete for the role filled by the original ATM component. With two ATM components, their requests to participate in the ValidatePIN protocol with the CustomerDatabase component must be sequenced. Otherwise, the components will not interoperate.

Translation to ADLs

As mentioned earlier, ADLs often specify protocols, and are useful tools for detailed assessment. The protocol information within the AIC can be used as a basis for an ADL specification. Translation from the protocol indicators to Darwin [20] and Wright [3] is discussed below.

Because of its focus on components, Darwin is a good fit for specifications resulting from the AIC protocol information. Recall from Section 2 that, in Darwin, a component is specified in terms of the services provided and required. This representation is in the form of π -calculus [23]. The access names in Darwin of the provided and required services are mapped to the AIC role name indicators of the components. The service name in Darwin is mapped to a component's instantiated name, when that name becomes known.

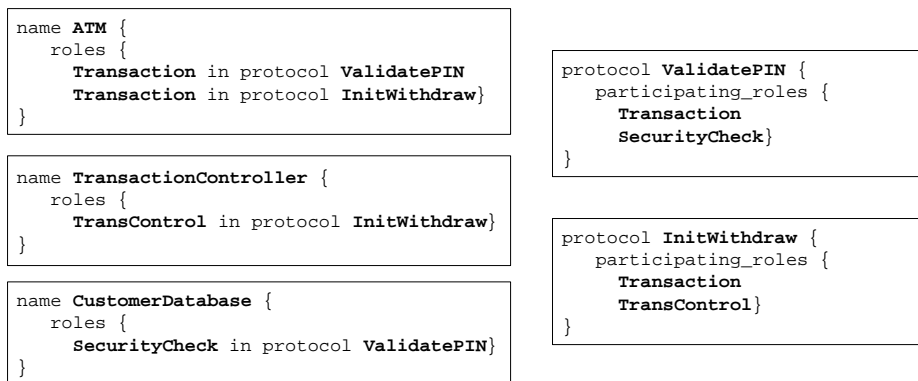


Figure 3: Example Protocol Information

```

name ATM_New {
  roles {
    Transaction in protocol ValidatePIN
    Transaction in protocol InitWithdraw}
}

name TransactionController_New {
  roles {
    TransControl in protocol InitWithdraw}
}

```

Figure 4: Role Assessment Using Protocol Information

Consider the banking example discussed earlier (Figure 3). Part of a Darwin translation from the available AIC information for the ATM component is shown in Figure 5. Italics indicate the information that is not available for specification until later in the design process.

```

ATM(Transaction) = (v s)(PROV(Transaction, s) | ATM'(s))
where
  (PROV(Transaction, s) = !(Transaction(o).5s)

```

Figure 5: Translation to Darwin

This partial specification shows that the ATM component provides the service Transaction. Later in design, it will be possible to specify the instantiated name that a service is referenced by, *s*, and the location at which the service should be utilized, *o*. For purposes of brevity, we do not illustrate the entire specification of behavior for the component. Though not shown, the component also requires services -- those roles that are present in the protocols it utilizes, but not present with the component itself.

A translation from AIC information to a Wright specification is also possible. The operational semantics of protocols in Wright are expressed using a variant of CSP [17]. This specification has two parts: an interface part and a computation part. The protocol information from the AIC indicators can be used to fill in some of the interface part. For instance, the protocol name(s) can be directly mapped to port name(s) of the Wright component interface. Additionally the role names participating in that protocol can form the outline for the port protocol sequence that is provided in Wright. The ATM component in the previous banking example is translated to in Wright in Figure 6.

```

Component ATM {
  port ValidatePIN [ValidatePIN protocol]
  port InitWithdraw [InitWithdraw protocol]
  comp spec [behavioral specification]

```

Figure 6: Translation to Wright

In this specification, the boldface print indicates Wright

keywords and the italics indicate comments. While we cannot make an entire mapping of the ValidatePIN and InitWithdraw protocol specifications in Wright, skeletal specifications of these protocols would indicate the role names required to achieve each protocol.

The resulting Wright specification is more simplistic than that obtained from the translation to Darwin. Because Wright focuses on expressing connectors as first class entities, the information from the AIC, which focuses on component specification, is more of a sketch in its translation as compared to the translation to Darwin.

In this section, we have only examined the potential for translation of protocol information to Wright and Darwin. Other ADLs may serve as guides to expanding the generic protocol information in the AIC.

4.3 Non-Functional Property Assessment

Given the properties, values, and rankings in Section 3.4, we can employ an analysis style akin to the ranking system of the military security policy [26]. The nature of this assessment affords us ease of expression with well-founded reasoning as its basis.

The military security policy assigns to information an ordered pair consisting of <rank; {compartments}>. An individual is not allowed access to a piece of information unless their rank is above that of the information, and they possess access to the compartments listed in the ordered pair. Similarly, each non-functional indicator utilizes values (compartments) and ranks. However, the rank is associated with the value and not related in an ordered pair.

Our assessment method specifies that if a weaker rank for a non-functional indicator is present in a collection of components, the rank for the application as a whole must abide by each lowest ranked property value, again loosely emulating the lattice structure inherent in the military security policy. For instance, should one component have non-modifiable functions, the system's functions as a whole would be considered non-modifiable.

When considering the comparison of non-functional properties for interoperability assessment, all components in the system must be analyzed at the application-level.

Pairwise comparison will not produce useful knowledge, in that any pairwise analysis can be compromised if a separate assessment yields a weaker value. An analysis across the components of the application is useful in determining both the lowest common denominator concerning non-functional aspects of an application and in choosing components, should particular non-functional attributes be needed to hold in the final application.

We will use the security property to illustrate our method of analysis. Given three components *P*, *Q*, and *R* with security value rankings listed in Table 2, some suppositions about the different components can be made. Component *P* appears to be a well-trusted system since it ranks high in all the security classifications in the AIC, while components *Q* and *R* score lower on the whole. When comparing the three components in each value classification, Table 2 shows that the ranks of the final application will be encryption.low, authentication.low, mediation.none, and audit.high. Therefore, a system built out of these components likely will not have a high level of security.

Now consider that the end-user wants a secure system. It is then a matter of whether these components are integral to the final system, such that additional coding must be undertaken to ensure high security in all areas or whether new, more secure components can be chosen.

SECURITY VALUE	P	Q	R	APP
<i>Encryption</i>	high	med	low	low
<i>Authentication</i>	high	high	low	low
<i>Mediation</i>	high	low	none	none
<i>Audit</i>	high	high	high	high

Table 2: A Security Example

In this paper we focus on non-functional requirements for components and applications, and do not take into account non-functional requirements of middleware. Integrated applications comprised of heterogeneous components, which utilize middleware, sometimes have trouble fulfilling their non-functional requirements even if the individual components themselves did. Future work will help determine if the AIC and its non-functional property assessment technique can help predict middleware's effect on an application comprised of components that have high performance, as well as secure and reliable.

5 COMPATIBILITY WITH INTERFACE DEFINITION FRAMEWORKS

The purpose of the AIC is most similar to an interface definition framework developed with the goal of forming a complete component specification to facilitate CBSE [16]. For instance, this framework includes behavioral information, but requires that each component know the identity of those components with which it communicates.

The AIC is scoped differently, allowing for analysis of the protocol specification independent of direct communication information. With respect to non-functional requirements, those expressed in the framework maintain "goal" values, as opposed to the resulting values found in the AIC for most components. Finally, only type information is represented in the framework that is similar to the architectural characteristic indicators in the AIC. This provides only limited analysis.

By this comparison, we can see that using the AIC could open a door of opportunity for multi-dimensional analysis. To realize this potential, we propose that it be included within the larger interface definition framework to address major architectural aspects of interoperability assessment.

6 DISCUSSION AND CONCLUSION

In this paper, we have presented an architecture interaction conspectus formed from a minimal, realizable set of architectural indicators. The intended use of the AIC is primarily for a first pass interoperability assessment, resulting in fundamental, yet relevant analysis information. AIC analysis provides information that concerns the general composition of components and aids in choosing among component alternatives. Thus, the AIC is one of the essential building blocks for a comprehensive interoperability assessment methodology.

We realize that middleware is an important entity to consider in interoperability analysis. Middleware specific characteristics will add complexity to an AIC, as the services must be addressed both in the context of a potential solution to interoperability problems and as a participating component. Determining the indicators for middleware is part of future research.

ACKNOWLEDGEMENTS

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

REFERENCES

1. Abd-Allah, A. Composing Heterogeneous Software Architectures. Ph. D. Dissertation, Computer Science, University of Southern California, 1996.
2. Abowd, G., Allen, A., Garlan, D. Formalizing Style to Understand Descriptions of Software Architecture. *ACM TOSEM*, 4(4): 319-364, 1995.
3. Allen, R. A Formal Approach to Software Architecture. Ph. D. Dissertation, Computer Science, Carnegie Mellon University, 1997.
4. Allen, R., Garlan, D. A Formal Basis for Architectural Connection. *ACM TOSEM*, 6(3): 213-249, 1997.
5. Barret, D., Clarke, L., Tarr, P., Wise, A. An Event-Based Software Integration Framework, TR 95-048. Laboratory for Advances Software Engineering

- Research, Computer Science Dept., Univ. of Massachusetts, 1995 (revised 1/96).
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.
 7. Cho, I., McGregor, J., Krause, L. A Protocol Based Approach to Specifying Interoperability Between Objects. In, *26th Int'l Conf. on Technology of Object-Oriented Languages and Systems.*, 1998.
 8. Chung, L., Nixon, B., Yu, E. Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design. In, *The 1st Int'l Wksp. on Architectures for Software Systems*. 1995.
 9. Davis, L., Payton, J., Gamble, R. How System Architectures Impede Interoperability, *To appear in the 2nd Int'l Wksp. on Software and Performance*, 2000.
 10. Davis, L., Payton, J., Gamble, R. Toward Identifying The Impact Of COTS Evolution On Integrated Systems. In, *2nd Wksp. on Commercial off the Shelf Software*, 2000.
 11. Davis, L., Gamble, R., Payton, J., Kelkar, A. From Feature Interaction to Architecture Interaction: Adapting Processes for Conflict Detection, 2000.
 12. Franch, X., Botella, P. Putting Non-Functional Requirements into Software Architecture. In, *9th Int'l Wksp. on Software Specification & Design*. 60-67, 1998.
 13. 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.
 14. Garlan, D. Higher-Order Connectors, *Wksp. on Compositional Software Architectures*, 1998.
 15. Garlan, D., Allen, A., Ockerbloom, J. Architectural Mismatch, or Why it is hard to build systems out of existing parts. In, *17th Int'l Conf. on Software Engineering*. Seattle, WA, 1995.
 16. Han, J. A Comprehensive Interface Definition Framework for Software Components. In, *Asia Pacific Software Engineering Conference*, 1998.
 17. Hoare, A. *Communicating Sequential Processes*. Prentice-Hall, 1985.
 18. Kelkar, A., Gamble, R. Understanding the Architectural Characteristics behind Middleware Choices. *1st Int'l Conf. in Information Reuse & Integration*, 1999.
 19. Keshav, R., Gamble, R. Towards a Taxonomy of Architecture Integration Strategies, *3rd Int'l Software Architecture Wksp.*, 1-2, November, 1998.
 20. Magee, J., Dulay, N., Eisenbach, S., Kramer, J. Specifying Distributed Software Architectures. In, *The 5th European Software Engineering Conf.* 1995.
 21. Medvidovic, N., Gamble, R., Rosenblum, D. Towards Software Multioperability: Bridging Heterogeneous Software Interoperability Platforms, *In 4th Int'l Software Architecture Wksp.*, 2000.
 22. Mehta, N., Medvidovic, N., Phadke, S. Towards a Taxonomy of Software Connectors. In, *22nd Int'l Conf. on Software Engineering*, 2000.
 23. Milner, R. The Polyadic PI-Calculus: a Tutorial. In Bauer, F., Brauer, W., Schwichttberg, H., eds. *Logic and Algebra of Specification*. Springer Verlag, 1993.
 24. Payton, J., Gamble, R., Kimsen, S., Davis, L. The Opportunity for Formal Models of Integration. In, *2nd Int'l Conf. on Information Reuse & Integration*, 2000.
 25. Perry, D., Wolf, A. Foundations for the Study of Software Architecture. *ACM SIGSOFT*,17(4): 40-52, 1992.
 26. Pfleeger, C. *Security in Computing -- Rev. ed.* Upper Saddle River, NJ: Prentice-Hall, Inc., 1997.
 27. Shaw, M. Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does. In, *8th Int'l Wksp. on Software Specification and Design*, 1996.
 28. Shaw, M., Clements, P. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In, *1st Int'l Computer Software and Applications Conference*. 6-17, 1997.
 29. Shaw, M., Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall, 1996.
 30. Sitaraman, R. Integration of Software Systems at an Abstract Architectural Level. M.S. Thesis, Dept. Mathematical & Computer Sciences: University of Tulsa, 1997.
 31. Stiger, P. An Assessment of Architectural Styles and Integration Components. M.S. Thesis, Dept. of Mathematical & Computer Sciences: University of Tulsa, 1997.
 32. Yakimovich, D., Bieman, J., Basili, V. Software Architecture Classification for Estimating The Cost Of COTS Integration. In, *the 21st Int'l Conf. on Software Engineering*. Los Angeles, CA, 296-302, 1999.
 33. Zaremski, A., Wing, J. Specification Matching of Software Components. *TOSEM*,16(4): 333-69, 1997.