# A Precise Characterization of Software Component Interfaces

Basem Y. Alkazemi
Computer Science & Engineering
Umm Al-Qura University, Makkah, Saudi Arabia
Email: bykazemi@uqu.edu.sa

***Abstract*— Developing software systems according to the Component-based Software Development (CBSD) model has proven to be one of the prominent development approaches nowadays. However, integrating components to build a complete working system has always been considered as an obstacle that requires substantial effort and may considerably delay the development process. The lack of effective characterization of component interfaces that only focus on the functional characteristics and ignore the architectural characteristics of software components is the main cause of this difficulty. Therefore, this work provides a detailed characterization of software component interfaces in order to expose the implicit architectural characteristics that impact component integration.**

***Index Terms*—architecture, integration, source-code, interface, open-source software, system model.**

## I. INTRODUCTION

Building software system following the Component-based Software Development (CBSD) model become one of the widespread approaches in the software engineering field due to its significance to lower development cost and increase productivity. One of the common activities in CBSD approach is the integration of components to compose new software system. This activity is solely dependent on investigating component's interfaces to identify their suitability for integration into a system. Component's interface exposes the abstract specifications of components and hide their implementation details. It is the first thing that is examined by system developers or tools in order to identify if the component matches the requirements of the system to be built. Thus, component's interfaces must be precisely defined in order to avoid unforeseen problems that might be raised at integration time of software components.

Several works have characterized component interfaces in order to provide clear understanding of components types. However, most of the characterization are coarse-grained in the sense that characteristics cannot be easily mapped to the real source code. Considering the rapid movement of open source software and the urge to utilize the available source code for re-use, addressing component integration problems become an essence [2,3]. The current support provided by the open-source repository system is restricted to string matching that result not only in listing a large number of irrelevant components but also in providing components that cannot fit into the system to be developed. Therefore, an effective characterization of source-code components interfaces is needed in order to refine re-user's search and to generate a list of more focused results.

The scope of this work is built on Marry Shaw's declaration of component characteristics where it was coined that functionality alone is not enough to successfully integrate and re-use component, architectural characteristics must also be considered [10]. This assertion was mainly targeting coarse-grained components obtained at high-level stages of a software development process (e.g. ADL[39]) . However, when it comes to low-level components (i.e. source code) everything is mixed up together in a sense that it may be extremely hard to distinguish between the code responsible for doing functionality and the other parts of code responsible for non-functional businesses. Therefore, the main objective of our work is to evaluate that assertion on low-level software components in order to examine its validity at that level. We proposed a framework for characterizing component interfaces in order to expose the hidden characteristics of source code components that may negatively affect component integration. The work presented here is derived by the business case established in a preceding work that aims at developing a repository system to support software re-use [24]. The term component is used throughout the paper to indicate source code fragment together with any supporting textual files that can be obtained from open-source software repository.

This paper is organized as follows. Section II presents the background work conducted in this research that studied the available interface categorization, software architecture and ADLs, and CBSD field. Section III presents our system model that we adopted to analyze the various architectural characteristics of software components. Section IV identifies the different types of interfaces that software components may have. In Section V, detailed discussion about the various characteristics that our proposed interface might have is given. Section VI describes the syntax of the specification language prototype that we have developed to examine the features of component interface. Section VII presents the evaluation of our approach by experimenting with a number of real software components obtained from Sourceforge.net repository. Section VIII lists a number of potential benefits that our approach can provide to the software development process. Section IX draws the conclusion and the planned future work for this research.


doi:10.4304/jsw.6.3.349-365

## II. BACKGROUND WORK

### A. Interface Categorization

Design by contract (DbC) by Meyer [5] is mainly concerned with defining the formal specifications of component's interface in order ensure that the collaboration between the components of a system is correct. The notion of DbC guides the design of the software system by specifying a set of pre-conditions and post-conditions as part of the interface of a component. Pre-conditions are the requirements that must be made available to a component prior to be able to provide its services (e.g. "You need a debit card to withdraw from a cash machine"). Post-conditions define what a component will provide once a condition is satisfied (e.g. "withdrawing money"). Brown and Short [6] described an interface as a way of summarizing the behaviour and the responsibility of the component. They used an interface to capture all the semantics related to the collaboration between components. The set of operations provided by a component is considered as part of the exhibited interface that a client or a system can use to obtain the required functionality of that component. Sametinger [4] described a component's interface as a way to determine how a component can be re-used and composed with other components in a system. An interface defines the set of operations that characterizes the behavior of a component. Sametinger distinguished between three types of interface namely, data interface, user interface, and programming interface. Data interface concerns the format and transformation of the data between components. User interface captures the protocol of interaction between a component and a user, for example through a simple command line or a graphical user interface. Programming interface captures the possible interactions between components and how they can be composed in a system. The work by Shaw and Clements [38] identified the notion of components and connectors to abstractly classify different architectural styles at the high level design stage. They identified components as the functional building block of a system, while connectors as the mediators that facilitate data conversion and transformation among the interacting components. The work by Mehta et al. [11] identified a fine-grained view of component interaction (i.e. connectors) in an attempt to reflect the high-level principle of connectors with a physical meaning that can be observed during implementation. However, at the source-code level, one may not be able to tell whether a method in the source code is responsible for interaction (i.e. for the connector) or whether it provides functionality For example, if an interaction is defined between two components as a "method invocation", then knowing this will not be of significance to a re-user who wants a precise specification in terms of how the interaction has been accomplished. Arbab *et al.* [7] described the interface as a definition of the observable behavior of components that contains five elements. These are a name, a channel signature, a blocking invariant, pre-condition, and post-condition. The name of an interface is used to uniquely identify an interface from other interfaces. The channel signature captures a set of parameters representing the data input and output of a component. The blocking invariant specifies special cases when a component needs to allow exceptions or perform a special action. The pre-condition refers to the required set of inputs that must be supplied to the component in order for it to operate. The post-condition refers to the set of values that are supplied by the component. They considered the component interface as a way to reason about the correctness of composition of a system from its components. Hondt *et al.* [8] described the notion of a *re-use contract* that concerns capturing the requirements of a component from other components in a system. They considered the interface as a way that not only captures the operations responsible for providing functionality, but also document what a component requires in order to work and what interaction structure is required in order to obtain a correct collaboration between the components of a system. An interface of a component captures the signature of operations without considering any semantics or type of information. The key contribution of the notion of a *re-use contract* is to detect conflicts in component interfaces, in that a conflict indicates that components cannot work together in a system. The notion of *re-use contract* is in line with the contribution of our work, however, we established a more complete set of architectural interfaces to facilitate components integration and re-use that includes external and internal view of component interfaces as described in Section 5. DeLine [12] established a distinction between the functional concern and the architectural concern of software components. His approach focused more on addressing the problems of interaction between the source code responsible for providing functionality and that of the architecture. Moreover, DeLine assumed that the component should be made available to a repository as a source code that purely defines functionality. The source code responsible for capturing the architectural characteristics are left unspecified until a re-user describes the required architectural characteristics. Based on the provided characteristics, the component is then wrapped as necessary to match the characteristics of the architectural style of the re-user. Once a suitable wrapper has been generated, then both the functional and the architectural source codes are combined to form a complete component. Although the work of DeLine is closely related to the work presented in this paper, it may not be applicable in the case of open-source software components where all the source codes that are relevant to functionality and architecture are mixed together. A provider might provide a component that is composed of functional and architectural source code, hence violating the assumption made by DeLine that a component should only be provided as a "ware". In addition, open-source software components usually lack any form of documentation that a packaging specialist might use to identify the architectural source code from the functional one. Even though the specialist was able to use their expertise to identify the architectural characteristics and

split them from the functional characteristics, it would be very difficult and time consuming.

*B. Software Architecture*

People usually refer to the term 'architecture' to indicate the physical construction of a building in terms of external shape, and also how rooms are structured within that building. In software, the word 'architecture' is a term that is in general use, with a number of different interpretations. However, as an analogy to its meaning in civil engineering, it inspires the meaning of creating a product (software system in this case) from a number of selected components rather than building a single monolithic one. So, the way components must be incorporated, the order in which they must be placed, and the mechanism of interaction between them, are parts of what system architecture describes.

Bas et al. [13] defined software architecture as the structure of a system that comprises software elements, their external visible characteristics, and the relationship between them. IEEE 1471 [14] defines software architecture as "the fundamental organization of a system embodied in its components, their relationship to each others and the environment, and the principles guiding its design and evolution". Jones [15] defined architecture as the structure that is composed of components and rules that establish the basis for the interaction between them. All the definitions have agreed upon the fact that architecture is concerned with the constituting parts of a system and the relationship between them.

In the literature, many of the available works have explained the significance of considering architecture in software development (especially in the CBSD paradigm). One reason for considering software architecture is to help our understanding of complex software systems. Shaw and Garlan [16] suggested that architecture can be used to define the overall design of a system. Garlan and Perry [17] identified the benefits of considering software architecture in software development as providing support for re-using, evolving, analyzing, and managing software. Budged [18] considered software architecture to be a way of describing the constructional aspects of a software system at a high-level of abstraction (e.g. design stage). Allen [20] identified architecture as being the vehicle to communicate between the requirement and the implementation stages. Szyperski et al.[19] suggested that architecture is important for establishing a context for software systems representing standards and platform requirements.

Garlan et al. [21] identified a number of architectural characteristics that might cause a mismatch to occur in terms of component interaction within a system. These characteristics are:

- The infrastructure that a component is primarily built on.
- Control issues of whether a component can generate a control signal or not.
- The data type manipulated by a system and the way it is transferred between components.
- The pattern of interaction between components.
- The sequence that components must be instantiated and invoked with.

From the re-users point of view, these characteristics are significant in order to identify whether or not a component can be integrated into their system and to build an understanding of the different characteristics of the architecture at hand. Consequently, a component that supports a single thread of control will not be suitable for integration in a system that assumes its components must be thread-safe. Also, a component that communicates through RPC will not be integrated in a system that uses message passing to transfer data, hence a mismatch might occur. Yakimovitch et al. [22] refined the work of Garlan and identified five variables that describe assumptions about components' interactions, namely packaging, control, information flow, synchronization, and binding. Their main motivation was to establish a mapping between architectural assumptions and a number of problem domains that conform to some standard architectural types. They demonstrated that the defined variables can be used to abstractly classify different software architectures.

Based on analyzing the current works in the field, software architecture seems to consider another view of a system that is not tightly relevant to functionality. This view examines the structure of a system and tries to identify components and define the possible interaction that a component can have in order to avoid the occurrence of fault due to a potential mismatch between components in a system. The development of the AESOP system [23] from large-scale components demonstrated the difficulty of incorporating components, and emphasized that the main reason for the observed difficulty is due to architectural mismatch between the various components. Even though the various components of the AESOP system were providing the required functionality as the developers needed, the integration of the various components to form a complete system was impossible without major modifications. The problems encountered by the AESOP developers were in favor of the assertion by Shaw [10] that stated considering functionality alone is not enough to successfully re-use components. As a result, exposing the architectural characteristics is necessary in order to integrate components correctly, and this is the main focus of the paper.

*C. Architecture Description Languages (ADLs)*

ADLs are specification languages for defining the structure of software systems at a high level of abstraction by identifying elements and the relationship between them [39, 40]. ADLs provide a description of the conceptual architecture of a system [47]. A general characterization of ADLs' capability was given in [44]. ADLs aim to support architecture-based software development by establishing notations that are appropriate for defining system architecture and its

constituting elements. They formalize the definition of a system at the architectural level in a graphical way that can be communicated to humans. Moreover, instead of drawing boxes and lines that may not involve rules that govern connections between them (i.e. boxes and lines), ADLs provide a semantic check of whether two elements can be linked together and what the requirements are that need to be satisfied in order to successfully create the links between these elements.

ADLs are built on the notion of components, connectors and constraints that have been described in the software architecture field. They provide a basis for analyzing and verifying the design of a software system. There are many ADLs available nowadays such as ACME [43], that can be used to represent the architecture of the system to be built (Darwin [42], Rapide [46] and many others). ADLs possess several characteristics that are relevant to the CBSD field as many of them facilitate the automatic generation of glue codes to form a system [45]. Despite the variety of ADLs, they are not widely adopted by the software industry, because they are not general enough as they only support specific architectural styles [41].

*D. Component-Based Software Development (CBSD)*

The notion of CBSD is not new. It was firstly coined by McIlory [25] who established the need to componentize software (i.e. building software from components) as a way of resolving some issues identified by the software crisis that concerns the case of building large and reliable software in a controlled way. CBSD is concerned with the assembly of software systems from pre-existing software components. One of the main objectives of the CBSD approach is to promote the re-use of previously developed components to allow the building of a new system. The notion of building a system from components can reduce development costs and increase the quality of the final system.

Building a software system from re-usable components requires a clear understanding of the aspects related to the characteristics of the overall system, the characteristics of software components, and aspects related to obtaining and integrating components [26].

A common model for CBSD is that a re-user who wants to add functionality to their system might find a repository to search for re-usable components. The re-user then gathers their ideas about the characteristics of the component they are looking for. After that, the re-user types a searching query that formulates their thoughts about the characteristics of the required component, either as free-text or in the form of a specification model. Alternatively, the re-user could browse the available categories in case they are not fully aware of the representation method used by the repository to organize the component. In this way, browsing can build up their knowledge [27]. In response, the repository may list a number of results that are relevant to what the re-user needs. Consequently, the re-user can examine the characteristics of every component on the list until they find a best match in terms of the required characteristics. Sometimes, the re-user might need to modify the component they have found in order to exactly match the requirements of the system to be built, so they might apply some adaptation techniques to accomplish the modification. Once the component matches the required characteristics, it can be integrated safely into the system.

The above model identifies a number of key activities with respect to development according to the CBSD approach, including:

- Identification: Identifying components involves recognizing the potential of re-usable ones, based on their exhibited characteristics from a list of components. This activity involves searching and browsing software components. The selection of the appropriate component from a list of components is done by matching the characteristics of the component to the specifications of the system to be built. This requires a precise definition of the components' characteristics in order to facilitate an understanding of them by their users and also to classify them for re-use . The success and soundness of the identification of the component is a major factor for the success of the CBSD approach as components cannot be re-used unless they are found. The key element for the success of the identification activity is the availability of an effective organizing scheme with regard to the software component. Software components can be identified in various ways. Some of the common ways of identifying components are based on matching their behavior [28], their signature [29] and their specifications [30]. Behavioral matching identifies components based on a set of predicates (i.e. pre- post- ) that are used to execute components. The resulting values of the execution are then used as representative "terms" to identify the corresponding components. Signature matching identifies components based on the signature of the functions within a component and the type of parameters. For example,in ML a function "hd" can be identified by the type of its input and output parameters "a list □ a". A whole component that is composed of several functions can be identified by the signature of the functions within the component. Specification matching is derived from the behavioral matching approach. However, it relies on predicates of the entire component's operation. The set of predicates are written using formal specification languages such as Z language [31] or OCL [32].
- Validation: Validation is a way of checking the characteristics of the component against a pre-defined specification. Two kinds of validation are relevant to the CBSD paradigm - unit test and integration test. The unit test is done by a component developer to ensure that the provided behavior of the component is correct. Testing a component's behavior could either be done as black-box testing by providing a set of inputs and examining the resulting output, or white-box testing by inspecting the source code. The integration test is undertaken by a component re-user to determine whether or not the component can interact with the other components in the system and is not going to raise any structural problems. In addition, integration

tests can be done in some cases to measure the quality of a component in order to decide whether or not the component can be trusted for re-use [5].

- Integration: The activity of integrating a component can be seen as a mechanical action involving connecting components by means of matching their syntax and semantics to form a system [33]. Part of the integration activity is related to checking the compatibility of the components to match the characteristics of a system. The main issue to address in this activity is related to solving potential mismatches between components. One reason behind the occurrence of a mismatch in a component's characteristics is due to the fact that the component's producers may be unaware of the potential usages (i.e. context) that their component might be re-used in, hence their assumptions are different from the assumptions considered by the components initial users. Thus, it is extremely important that the software components are produced with a well-defined interface in order to understand the assumptions that components can match, and also can be connected with at runtime. In a system that is built locally, integrating heterogeneous component can be achieved using a wrapper or glue code to bridge the differences between the components' interfaces. So, if a component that takes two parameters as an input is composed with a component that provides three values as an output, then a glue code can be used to map the input and the output of the components. In building a distributed system, the interaction between the components can be addressed using the notion of middleware (e.g. CORBA) that unifies the components' interfaces to enable their interaction across a network.
- Evolution: This activity is concerned with replacing components from a system with other components that conform to the same interface, so that we can substitute the replaced component without affecting the other components of the system [34]. The reasons for replacing the components could be to fix bugs in the system or to extend the functionality of the system by incorporating new components that provide the desired behaviour into it. Consequently, this activity is important in the notion of CBSD.

With regard to the above activities, integrating components is a significant issue that needs to be investigated in depth. Addressing component integration is especially important when dealing with heterogeneous components, as they might cause lots of interoperability problems when developers need to integrate components into their systems. Components can either be integrated statically or dynamically [35]. Statically integrated components are those that are bound by programming mechanisms (e.g. method invocation) at compile time and usually conform to an architectural style. The dynamically integrated components are those that are bound at run-time and they are identified by the services (i.e. behavior) they can provide.

Integrating components involves adapting components to resolve potential mismatches in the characteristics of a component and the characteristics of the system to be built. Adaptation refers to modifying the interface of the software components by means of using a wrapper, glue code, or a translator to eliminate the unnecessary characteristics of a software component and also to add additional characteristics to its current interface in order to meet the requirements of a developer. Specifically, the adaptation of the component is mainly concerned with solving potential interaction problems that are caused due to potential architectural mismatches between components interfaces.

Several attempts have been made to try to tackle the problem of integrating components in a system. Eclipse [36] has established a framework by means of plug-ins that encapsulate components in order to unify their interfaces. So, different components that conform to different assumptions can be incorporated into an Eclipse if they are wrapped with the necessary plug-ins' architectural characteristics. The Vienna Component Framework (VCF) [37] has established a framework similar to that of Eclipse, but its authors claim that it is has an advantage over Eclipse in the sense that it provides uniform access through different component models. VCF has defined general characteristics for software components that are common among different types of components. These characteristics are:

- Life-cycle: every component must implement a set of methods that allows a system to control it when it must be initialized and destroyed.
- Persistence: this allows a component to be stored and retrieved from storage.
- Method: this characteristic gives a handle to the real methods provided by a component that are responsible for functionality aspects.
- Property: this characteristic allows for the manipulation of the component's state.
- Event: this characteristic allows components to be registered as listeners to be notified about events.

Integration problems are experienced in various situations where the CBSD approach is used to build a system. One of the prominent examples that demonstrates this problem is the integration of Commercial Off the Shelf products (COTS). The problems encountered at integration time are primarily caused due to potential mismatches in the architectural assumptions between the components that are planned to be re-used and for the system to be built. One may find, somehow, a component that seems to satisfy their functionality. However, that component does not fit into the system under development due to incompatibility in the programming language, operating system, or the database schemes used to write the component and that of the system. These mismatches are additional difficulties that a system builder might need to take into consideration when considering CBSD model. As a result, a new approach is needed to identify the implicit architectural characteristics of software components to avoid potential

mismatches, and this is one of the key contribution of this paper.

## III. System Model

A common model of a system structure is to consider a system as being composed from a set of components. A component might be complex in a scene that it can be composed of sub-components, and sub-components might also be composed of sub-sub-components, and so on until a point is reached where a component cannot be decomposed any further, hence can be called a simple component. Each component itself can be considered as a system, with the above description being applied recursively. A system might be a stand-alone application that can be installed and run on an environment or a part of a larger system that can be incorporated to a predefined system framework, but the model permits the general term of system to be used to cover such eventualities. Since the developer is building a system, components are what they may try to examine and integrate, and are the dependencies that the system utilizes for providing the necessary functionality. Every system has some characteristics that must be matched by software components. If the characteristics required by the system are matched by the characteristics exhibited by the component, that component will be a fit candidate in the system.

Following from the system model, the notions of system and component are interchangeable in the sense that a system can be considered as a component if a developer decided to integrate it in another system, while a component can be considered as a system in its own right, for example, if the developer is interested in examining its composing sub-components. So, all the characteristics (to be defined) that are relevant to a system are applicable to a component and vice-versa. Components provide functional characteristics that a system requires through standard interfaces to the system. In turn, a system provides architectural characteristics that components require to work in the system through standard interfaces defined by the system.

Figure 1 below illustrates the ontology of the described system model. While this is a simple system model, and clearly does not capture all the complexity of a software system's structure, it is sufficient to use as the model for identifying the important characteristics necessary for component integration.

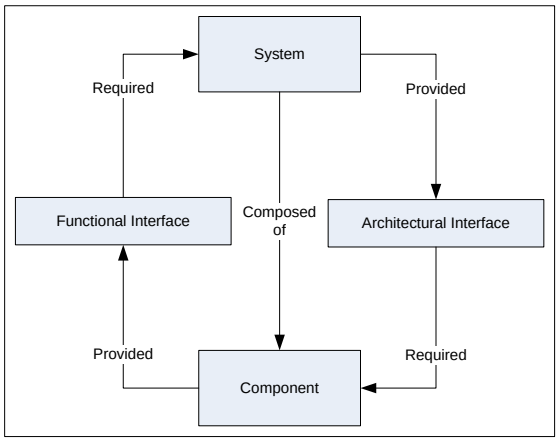


Figure 1. Ontology of System Model

## IV. Types of Interfaces

We describe component interface abstractly as a contract of fit. An interface is, in fact, a kind of contract of communication between a component and a system. Both a component and a system must agree upon a defined contract in order to allow for a component to be integrated into a system. The characteristics defined by an interface capture the functional and non-functional aspects of software components. Based on the exhibited characteristics of a component's interface, a component can be identified and integrated. Component's interface can be represented directly in the code of the component (e.g. Java Interfaces) or by using additional files (e.g. a textual file) that describes the external attributes and methods of the component.

Two types of interfaces are distinguished that we adopted the terminologies *functional interface*, *architectural interface*. The functional interface exposes the set of services that a component can provide. Obviously, this interface is the key to identify if a component is of any interest to a developer as it is the first thing that developers examine. However, it is the least concern with respect to the CBSD as components will not be of any use if they cannot be integrated into a system. Therefore, functional interface is not discussed in this paper due to its irrelevance to component integration. On the other hand, the architectural interface exposes the characteristics that components must match in order to compile, link, and work successfully in a system regardless of the semantics of the components. In fact, this interface is significant to examine aspects about components integration, as it is responsible for identifying whether components can physically fit into a system or not. For example, an Oracle 6i form will not fit directly into Microsoft SharePoint system due the lack of web-based support indicating a mismatch occurrence in the architectural characteristics. Thus, the notion of architectural interface is the main focus of this paper and is described in detail in the next section. Overlooking functional interface in this research does not meant that it is not of any importance, but our aim is to support software re-users to refine their functional based

searching criteria with additional characteristics that are defined by architectural interface. The aim of architectural interface is to ensure that software components that are obtained from external vendor (e.g. open source repository) can compile, link, and run into a system without trouble.

## V. The Characteristics Defined by the Architectural Interfaces

We identified two types of architectural interfaces namely external and internal. The external interface of a component captures the characteristics that must be exposed to satisfy the requirements of a system. It can also be used to identify whether the system can be integrated as a component into another system. The internal interface of a system is significant in identifying the characteristics of the composing components, that are the dependencies of the system, and also the characteristics defining how components can interact with each other. For example, if a system requires its components to provide a method called `public void start()` to control when the component starts running, this requirement forms part of that system's internal interface, and the method must be part of the external interface of any potentially re-usable component. Similarly, if a system uses some special libraries to implement its functionality, the characteristics of the library must be part of the internal interface that the system must provide to its components. An analogy with jigsaw pieces is useful to express the idea of the two interfaces. The things needed are the "hole" in a jigsaw piece, and things provided are the "protruding bobble". So, an external interface of a piece of a jigsaw has holes that it needs, and bobbles that represent what it provides; similarly for the internal interface. Both interfaces identify characteristics that dictate whether a component can be successfully integrated in a system.

Consider the Eclipse as an example of a system that a developer wants to add some functionality to by incorporating new "plug-ins". The Eclipse system provides an extensible environment that precisely defines how new plug-ins can be added to the system, and also establishes the basis for defining the relationships between plug-ins. Mapping the Eclipse system to the system model introduced in this section, the internal interface of the Eclipse system requires the following methods as part of the characteristics that the external interface of a component (i.e. a plug-in) must match in order to fit in the system:

- `public void start(BundleContext)`
- `public void stop(BundleContext)`

A plug-in might have interaction with other plug-ins in the Eclipse system or it may need sub-components to accomplish its desired functionality. For example, a file-transfer protocol (FTP) plug-in needs to interact with the "org.eclipse.osgi" plug-in, which is part of the Eclipse system, to facilitate launching the FTP plug-in in the system. So, the "org.eclipse.osgi" plug-in must be defined as a part of the characteristics that the external interface of the FTP plug-in must capture, as it is one of the external dependencies of the FTP plug-in that is required by the Eclipse system. The FTP plug-in uses a Java class called "newSocket" that is not part of what the Eclipse system requires, hence the "newSocket" Java class must be captured by the internal interface of the FTP plug-in as one of its internal dependencies.

Based on examining various components types, a number of key architectural characteristics are identified to characterize the external interface of software components, they are:

- Format: this characteristic specifies the underlying syntax used to write a component. For example, at the source-code level, a programming language will represent the format of a component. So if a system requires a component written in Java then a component written in FORTRAN will not be directly suitable for integration.
- Entry point: the entry point is the first block of code that should be invoked to initialize a component. Some components may provide special methods that must be executed to provide initialization, while others may require the presence of special tools or files for their initialization. For example, a standalone Java application must have a method called public static void main() to be initialized, while an Eclipse plug-in can be initialized by reading a file called "plugin.xml" and the presence of a method called public void start(BundleContext). So, a component must match the initialization mechanism that a system requires in order to fit successfully in the system.
- Handling failures: if a fault occurred in a component at any stage during its execution then the failure handling mechanism implemented by the component must conform to the one expected by the system. For example, if a system assumes that its composing components must provide a specific recovery action in case of failure, then components must implement the necessary action to fit.
- Using external dependencies: a software system may require its composing components to use dependencies that it provides for them to fit in the system. For instance, a Java system requires its composing components (i.e. Java classes) to use a library called "java.io" to achieve the basic input and output functionality. Also, an Eclipse system requires its components (i.e. plug-ins) to use a plug-in called "org.eclipse.osgi" to allow the system to control their execution. So, components must use the external dependencies that are provided by a system in order to be integrated successfully in the system.
- Data I/O: after a component is initialized it will be ready to receive data for processing and sending out. The mechanism of handling data must be defined according to the requirement of the system under development in order to avoid potential mismatches. For example, a component that receives data via parameters may not fit into a system that requires their components to read data input from a file. Both system and components must agree upon a data exchanging model. So, a component employs the push-model will

not fit into a system assuming their components to exchange data according to the pull-model [9].

- Control flow: the way control is exchanged can be different from one component to another. One component may synchronize its execution with a system, so the component can return control to the system upon the completion of its execution. Another component might execute asynchronously with the system. Thus, identifying the different mechanisms of control flow is necessary for integrating components successfully into a system.

The internal interface of software components represents the set of composing sub-components that the component depends on in order to provide its functionality in addition to the protocol of interaction between sub-components. The set of characteristics identified by the internal interface are:

- Sequence of execution: a software system must invoke components in the correct sequence otherwise some of the composing components of the system may not execute correctly. For example, consider a simple parser system composed from a reader component that reads from a file and stores data in a temporary buffer for processing, and an analyzer component that analyzes the data and identifies their semantics. The parser system must invoke the reader component first and then invoke the analyzer component. If the analyzer component is invoked prior to invoking the reader component then this might cause the analyzer component to raise an error, and hence cause the system's execution to fail.
- Internal dependencies: components in a system may depend on sub-components in order to provide its intended functionality. Considering the internal dependencies is significant in the case of extracting a component from a system prior to integrating it into another system as all sub-components must be extracted together with the component in order to work successfully in the new system.

Figure 2 summarizes the different types of architectural interface with their corresponding characteristics classified based on these two types.

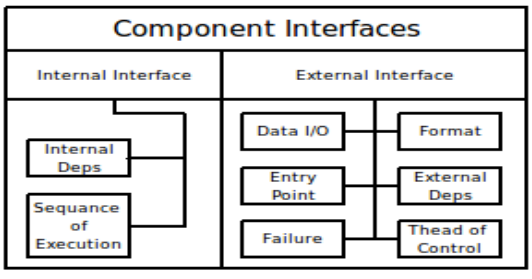


Figure 2.Classification of Component Characteristics

Identifying the possible values of the characteristics identified in the external architectural interface is necessary to determine whether a component can fit architecturally into a system or not. The values of the identified characteristics are defined by what we adopted the term architectural type. Hence, there is a need to specify these characteristics and their corresponding values in a precise manner that could be identified in the source code of a given component. Figure 3 depicts a fine-grained ontology of the system model describing the relationship between the notion of architectural interface and architectural type.

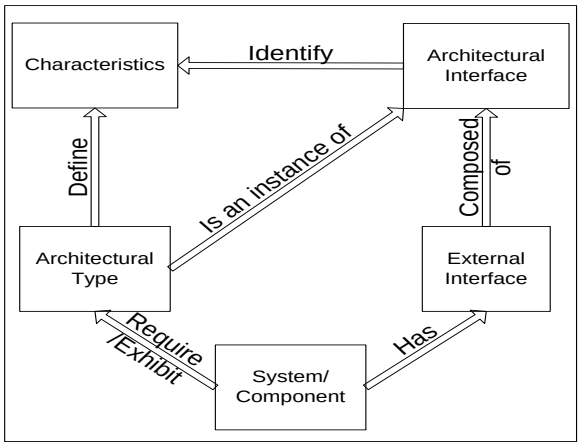


Figure 3. Fine-grained Ontology of System Model

A prototype of a specification language namely *ArchInt* is developed to formalize some of the characteristics identified by the architectural interface. We started our experimentation with a prototype that examines only the characteristics identified by the external architectural interface. The internal interface is left for future experimentation after evaluating the soundness of the external architectural interface. The specification language describes things with respect to syntactic constructions that can be identified in the component regardless of semantics concerning what the construct means. For example, the syntactic aspects of identifying what a method is (e.g. a Java Method), must be separated from the semantics that express what that method can do (e.g. it corresponds to handling a failure). The main consideration of our developed language is to return a "Yes/No" answer with respect to matching a components against the characteristics defined by an architectural type. This consideration has the advantage of facilitating a tool to check automatically the availability of the characteristics in the architectural interface of software components without human intervention. For instance, if an architectural type that is required by a system defines one of its characteristics as requiring a UNIX process with standard inputs and outputs then the architectural type of a component must define this requirement in order to match the architectural type required by the system. The semantics of the UNIX process is not important to fit into the system as that can be considered as part of the functional interface that we have excluded in this work.

## VI. ARCHINT SPECIFICATION

ArchInt is developed as a prototype of the required specification language to evaluate some aspects of architectural interface. ArchInt represents a document that contains the set of values that comprise a particular

architectural type; it is used to match characteristics represented in software component against the architectural type. An ArchInt document has a specific structure that needs to be processed by a tool. Hence, XML seemed a sensible choice for representing the information to help in detecting errors in the document itself.

Every document must start with a pair of opening and closing tags called <ArchInt> to identify the boundaries of a document written in ArchInt and also to indicate that the defined XML document is an ArchInt document. The opening and closing tags must be the first and last tags in any ArchInt document. In a repository system, every architectural type must have a name to distinguish it from other architectural types in the repository, this is captured by ArchInt using a pair of tags called <name>. Every ArchInt document must contain only one name and the tag corresponding to the name must be the first tag that appears after the <ArchInt> tag. One key characteristic that is identified by architectural interface was the "Format"; this characteristic specifies the programming language that is used to write a component. ArchInt captures the "Format" characteristic using a pair of tags called <programming_language>. This tag is necessary to identify how software components can be processed. The <programming_language> tag identifies the appropriate tool to be used in order to check the conformance of software components to an architectural type. As will be seen later in this paper, the compiler associated with a programming language is used as a tool to examine the characteristics of software components. The three tags described earlier represent the basic features of the ArchInt language and must be present in every ArchInt document to identify the type of the document (i.e. conforming to ArchInt specification), to identify the name of an architectural type, and to identify the tool that can process a component.

The characteristics that need to be matched in the software components against an architectural type description are captured by ArchInt using the pair of tags called <must_have>. This pair of tags indicates that the content between them is related to the requirement of architectural fit. So, if an architectural type that is required by a system defines a method called "public static void main(String arg)", then this method should be described between the <must_have> tags pair.

ArchInt captures part of the requirements of architectural fit by the pair of complex tags (i.e. composed of sub-tags) called <Block>. The fundamental idea that is captured by the pair of <Block> tags is related to identifying the address within a component where data is exchanged, and also the type and sequence of data input and output of the component in that address. For example, in Java the <Block> tag corresponds to the *methods* defined in a Java class, while in Eiffel the tag corresponds to the *features* defined by an Eiffel class, and in FORTRAN the tag corresponds to *sub-routines*. Within the body of the tag <Block> the name of the block is captured using the pair of tags <name>. The data exchanged by a block is represented in ArchInt by the pair of tags <Data_Input> and <Data_Output>. The type of the input and output data exchanged by a block is captured by ArchInt using the pair of tags <type>. A block may have more than one type of data that need to be processed in a defined sequence, this is represented by ArchInt using the pair of tags called <sequence>. A failure that might be raised by a block is represented in ArchInt using the tag <Failure>. Software components might include some descriptive files that might satisfy special requirements of a system in addition to the source code of the component. As a result, ArchInt identifies a pair of complex tags called <File> to capture the additional files that might be defined by an architectural type. This tag will be part of the characteristics that should be defined between the <must_have> pair of tags in an ArchInt document. Within the body of the <File> tag the name of a file is captured using the pair of tags <name> to identify a file from any other files that might also be defined by an architectural type. Every file must have a type that represents its format (e.g. XML, Doc, TXT). The type of a file defined by an architectural type is captured by ArchInt using a pair of tags called <type>. This tag identifies what tool can be used to check whether a file is well-formed or not. In some cases, a file might need to have a specialized format that is inherited from a generic structure type. For example, the file plugin.xml conforms to the XML document structure type and also conform to a specific Exlipse formatting structure, hence a pair of tags called <sub-type> is used to capture such specialization. A system might require its composing components to hold temporary data during their lifetime in the system or to define values for some specific attributes of components required by the system. ArchInt captures this requirement of a system using a pair of complex tags called <Storage>. In the source code of components, fields and member variables are the concern of this tag. Every storage must have a name that distinguishes it from other one in a component, hence a pair of sub-tags called <name>. The data held by a storage must be of a certain type, hence a pair of sub-tags called <type> is defined by ArchInt. ArchInt can reduce the effort of writing new ArchInt document of an architectural type that, part of its defined characteristics, is captured by another ArchInt document. So, old ArchInt documents can be extended instead of replicating the same characteristics in a new ArchInt document. ArchInt captures the feature of extending old ArchInt documents through a pair of tags called <uses_ArchInt>. This tag refers to the ArchInt documents that are going to be extended by their names, hence a pair of tags called <name> is introduced. The dependencies that a system must provides to components is captured by ArchInt using the pair of tags <External_Dependencies>. Table 1 summarizes the syntax of the ArchInt specification language.

Table 1. ArchInt Specification

| Tag | Description |
|---|---|
| <ArchInt> | identify a document as ArchInt specification |
| <ArchInt>\<name> | define the name of an architectural type |
| <ArchInt>\<uses_ArchInt> | define parent architectural type |
| <must_have> | defines the specification of architectural types |
| <must_have>\<Block> | identify address in source code |
| <must_have>\<External_Dependencies> | define the dependencies that a system must provide |
| <Block>\<name> | defines the name of a block of code |
| <Block>\<Data_Input> | define input channel |
| <Block>\<Data_Output> | define output channel |
| <Block>\<Failure> | define exception handling mechanism |
| <Block>\<File> | define external file that architectural type use |
| <Block>\<Storage> | define temporary memory |
| <Data_Input>\<sequence> | identify the sequence of input data |
| <Data_Output>\<sequence> | identify the sequence of output data |
| <sequence>\<type> | define data type |
| <Failure>\<type> | define the type of exception handling |
| <External_Dependencies>\<lib> | define required resources |
| <File>\<name> | define the name of file |
| <File>\<type> | define the type of file |
| <Storage>\<name> | define memory address |
| <Storage>\<type> | define memory type |
| <type>\<sub-type> | define specialized file type of a generic one |

The defined tags in this section are the ones that the current prototype of the ArchInt specification language has defined at the moment for evaluating the external architectural interface. Although these tags can be seen as a small set of tags, in fact they capture some of the essential and key features of architectural interface. Therefore, this set of tags formed the basis for a set of experiments designed to evaluate the feasibility of architectural interface to define component interfaces. The experiments have been confined to Java examples since this was sufficient to demonstrate the soundness of the basic idea to start with rather that attempting to generate completely a general solution at this stage of the development of the language.

## VII. EVALUATION

We have developed a tool called *ArchIntParser* purely in Java using a common Java editor that check components against an ArchInt description. The approach followed by the ArchIntParse tool for matching an architectural type document to a provided component is based on utilizing the tool associated with the programming language identified by the `<programming_language>` tag to check the syntax of a file or a component. So, if the component was source code then the corresponding compiler can be used to check for matching types and also identify whether a component is missing any of its required sub-components (i.e. internal dependencies). In the case of textual files, then the corresponding tools can be used to check conformance to the syntax and styling required by the language (e.g. well-formed xml document). Our experiments assumes that components use Java architectural type hence the programming language is java as illustrated in figure 4.

```
<ArchInt>
  <name>Java Class</name>
  <programming_language>Java</programming_language>
</ArchInt>
```

Figure 4. Java Architectural Type

In the case of source code check, the tool works by automatically generating a "TestSuite" Java class from an architectural type document. The TestSuite class contains code to exercise all of the features specified in the architectural type document. The tool then compiles and links the generated Java class with the source code of the provided component. If no compile-time or link-time errors are raised, this indicates that the provided source code matches the architectural type that was used to generate the TestSuite Java class, and the tool returns a positive result. If compile or link errors are raised, this reflects a mismatch and the tool returns a false match result. Figure 5 illustrates the operations performed by *ArchIntParser* tool to examine different components obtained from Sourceforge.net.

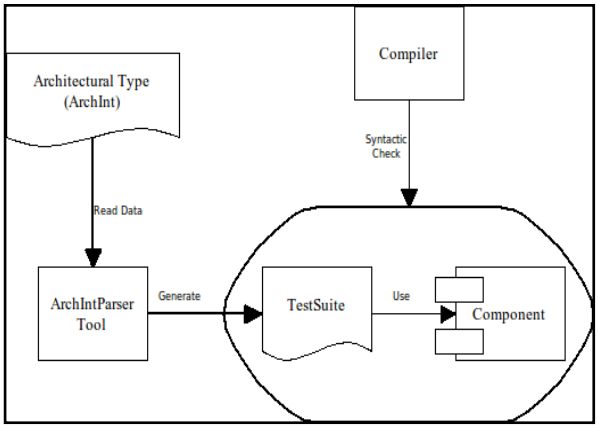


Figure 5. ArchIntParser Operations

The generated TestSuite class contains method calls representing invocations of all of the methods identified in the `<must_have>` tag. Figure 6 illustrates an example of the automatically generated TestSuite java class to match the source code of an Eclipse plug-in to the Eclipse plug-in architectural type.

Sourceforge.net is selected as a component supplier for conducting our experimental work since it is among the prominent open-source repository systems nowadays. Sourceforge.net supports searching queries written between quotations and also queries without quotations. A query that is written between quotations seems to return more focused results (i.e. exact match) than the one written without quotations. This study considered searching for software components using queries surrounded by quotations. The selection of components was done randomly using the formula "`1 + (int)(N * Math.random())`" . Results that are listed in Sourceforge.net without their corresponding source code were discarded. Moreover, components returned by Sourceforge.net that are written in different programming languages than Java were not considered at this stage.

```
package Deserializer;
import tmp.com.nilswinkler.eclipse.accessmodifier.AccessModifierPlugin;
public class TestSuite
{
    AccessModifierPlugin instAccessModifierPlugin = new
AccessModifierPlugin();
    public void test()
    {
       org.osgi.framework.BundleContext a0 = null;
       org.osgi.framework.BundleContext a1 = null;
       try
        {
             instAccessModifierPlugin.start(a0);
        }
        catch(Exception e){}
       try
        {
             instAccessModifierPlugin.stop(a1);
        }
        catch(Exception e){}
    }
}
```

Figure **Error! No text of specified style in document.**. Sample of TestSuite Class

The terms used to search for software components are those that were observed common among various repository systems or those that precisely state the name of an architectural type. However, there might be other expressions of use that were beyond our knowledge; hence the selected terms were not claimed to be exhaustive. The tool then compiles and links the generated Java class with the source code of the provided component. Figure 7 illustrates an extract of the output generated from executing the ArchIntParse tool on a number of Eclipse plug-ins.

```
CCL: Compiling ...
TestSuite  against ComponentsTesting\com\jcraft\eclipse\sftp\internal\SFtpCommunicationException:compilation has failed!
CCL: Compiling ...
TestSuite  against ComponentsTesting\com\jcraft\eclipse\sftp\internal\SFTPDirectoryEntry:compilation has failed!
CCL: Compiling ...
TestSuite against ComponentsTesting\com\jcraft\eclipse\sftp\internal\SFtpPlugin :compilation is completed successfully
CCL: Compiling ...
TestSuite  against ComponentsTesting\com\jcraft\eclipse\sftp\internal\SFtpServerException:compilation has failed!
```

Figure 7. ArchIntParse Tool Output

If no compile-time or link-time errors are raised, this indicates that the provided source code matches the architectural type that was used to generate the TestSuite Java class, and the tool returns a positive result. If compile or link errors are raised, this reflects a mismatch and the tool returns a false match result.

### *A. Experiment 1: Eclipse plug-in architectural type*

Our selected test-bed sample was Eclipse plug-in architectural type. We have generated an ArchInt description that ArchIntParser tool can utilize to check if software components match this architectural type. An extract of the Eclipse ArchInt is given below in Figure 8. Sourceforge.net was searched for Eclipse plug-in components, using the normal text matching search, for the phrase "Eclipse plugin" provided by the Sourceforge.net repository. The searching phrase returned 512 components as at 12/2009 that only contain the phrase "Eclipse plugin".

We applied our ArchIntParser tool on the returned components by Sourceforge.net. The tool identified 493 components out of the 512 as conforming to the Eclipse plug-in architectural type, while 19 components were not identified as conforming ones. To check the validity of the generated results, all 493 Eclipse components were tried as plug-ins in an Eclipse system. The 493 components that were identified by the ArchIntParse tool as conforming to the Eclipse plug-in architectural type were all recognized and run successfully in the Eclipse system. The remaining 19 components exhibited variations. A number of 14 components out of the 19 did not fit in the Eclipse system. While the remaining five components were recognized by the Eclipse system, however they never provide any functionality.

Visual inspection of the component confirmed that all the 19 non-conforming components did not implement the methods defined by the Eclipse plug-in architectural type to control their life-cycle. In addition, 14 components of them were also missing the plugin.xml file so the Eclipse system was not able to recognize and initialize them. This explains why the resulted behaviour of the 19 components was not presented as expected.

```
<ArchInt>
        <name>Eclipse Plugin</name>
        <uses_ArchInt>
                <name>Java Class</name>
                <name>Eclipse XML</name>
        </uses_ArchInt>
        <must_have>
                <Block>
                        <name>start</name>
                        <Data_Input>
                                <sequence>
                                        <type>BundleContext</type>
                                </sequence>
                        </Data_Input>
                        <Data_Output>
                                <sequence>
                                        <type>void</type>
                                </sequence>
                        </Data_Output>
                        <Failure>
                                <type>interrupt</type>
                        </Failure>
                </Block>
                <External_Dependencies>
                        <lib>osgi.jar</lib>
                </External_Dependencies>
                <File>
                        <name>plugin</name>
                        <type>xml</type>
                                <sub-type>Eclipse</sub-type>
                </File>
                .
                .
                .
        </must_have>
</ArchInt>
```

Figure 8. Partial Listing of Eclipse Plug-in Architectural type

### *B. Experiment 2: Applet Architectural Type*

This experiment involved the Applet architectural type. Text matching in SourceForge.net was used again, but this time with the string "Java Applet". A list of 235 results that contained the phrase "Java Applet" was returned by Sourceforge.net as in 6/2010. We applied the ArchIntParse tool against all the 235 components

returned by Sourceforge.net in order to identify if they match our ArchInt description of the Applet architectural type. ArchInt description for the Applet architectural type is given partially in Figure 9.

```
<ArchInt>
        <name>Applet</name>
        <uses_ArchInt>
                <name>Java Class</name>
                <name>Serializable</name>
                <name>Accessible</name>
                <name>MenuContainer</name>
                <name>ImageObserver</name>
        </uses_ArchInt>
        <must_have>
                <Block>
                        <name>init</name>
                                <Data_Input>
                                        <sequence>
                                                <type>void</type>
                                        </sequence>
                                </Data_Input>
                                <Data_Output>
                                        <sequence>
                                                <type>void</type>
                                        </sequence>
                                </Data_Output>

                </Block>
                <Block>
                        <name>setStub</name>
                                <Data_Input>
                                        <sequence>
                                                <type>AppletStub</type>
                                        </sequence>
                                </Data_Input>
                                <Data_Output>
                                        <sequence>
                                                <type>void</type>
                                        </sequence>
                                </Data_Output>
                </Block>

                <External_Dependencies>
                        <lib>java.awt.Panel</lib>
                </External_Dependencies>
                .
                .
                .
        </must_have>
</ArchInt>
```

Figure 9. Partial Listing of Applet Architectural type

The ArchIntParse tool identified that 218 of the 235 components matched the Applet architectural type document. The remaining 17 components were flagged as not matching. To check the validity of the generated results, all the 235 components that Sourceforge.net identified as Applets were examined on an Applet system using a normal appletviewer utility. It was found that 229 components out of the 235 components ran successfully, including those components that the ArchIntParse tool identified as being instances of the Applet architectural type. The remaining components did not run successfully, all of which were correctly identified by the tool as not being instances of that architecture type. The 11 components that apparently were successfully executed as Applets and were not correctly identified as matching by the tool are discussed further below.

Inspecting by hand the source code of the 11 components that returned negative result by the ArchIntParse tool showed that all the components matched the characteristics defined by the Applet architectural type. After examining the possible reasons for the conflict in results obtained by the ArchIntParse tool and by trying the components on the Applet system, the reason for the conflict was identified. The 11 components for which negative results were returned by the ArchIntParse tool were delivered by the Sourceforge.net repository missing some of their internal dependencies. As a result, the compile-and-link process in the ArchIntParse tool failed. This was the real reason that caused the ArchIntParse to return negative results and not because these components were not conforming to the Applet architectural type. So, this result is considered a false negative result as the failure in the compilation was not caused due to missing any of the characteristics of the Applet architectural type but it was related to missing internal dependencies that allow the components to work in an Applet system. Despite these false negative results, the results obtained in this experiment are promising.

Overall, this experiment demonstrated that the Applet architectural type represented by ArchInt has worked successfully to check and identify automatically the conformance of software components to the Applet architectural type.

### *C. Experiment 3: Model-view-controller (MVC) Architectural Type*

Figure 10 illustrates the architecture of one implementation of an MVC-based system. This experiment involved replacing a component that matched the observed characteristics of the Model architectural type as identified in this system (i.e. “ContactModel”) with another Model component obtained from a OSS repository system.

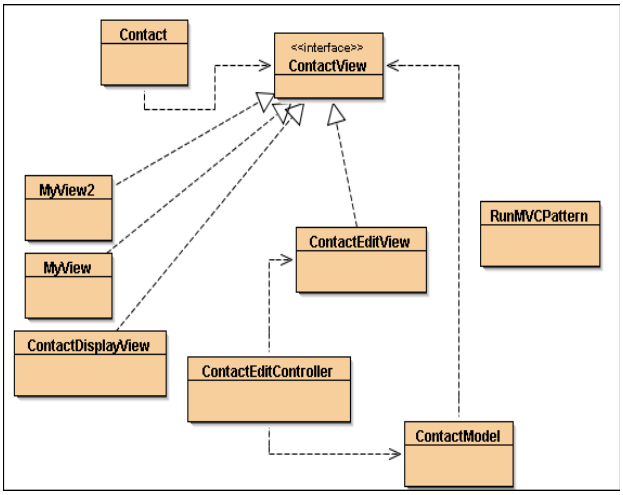


Figure 10. Contact MVC System

The Java source code of “ContactModel” is inspected by hand in order to identify the architectural characteristics that would constitute a description for the Model architectural type according to the implementation of the “ContactModel” in this system. The observed characteristics of the Model architectural type of this system are illustrated in Figure 11. The *ArchIntParse* tool was used to check the generated ArchInt document for the Model architectural type against all the Java classes in this system and the “ContactModel” Java class was verified as the only one that matched the generated ArchInt document.

```
<ArchInt>
        <name>Model</name>
        <uses_ArchInt>
                <name>Java Class</name>
        </uses_ArchInt>
        <must_have>
                <Block>
                        <name>refreshContactView</name>
                                <Data_Input>
                                        <sequence>
                                                <type>String</type>
                                                <type>String</type>
                                                <type>String</type>
                                                <type>String</type>
                                        </sequence>
                                </Data_Input>
                                <Data_Output>
                                        <sequence>
                                                <type>void</type>
                                        </sequence>
                                </Data_Output>

                </Block>
                <Block>
                        <name>addContactView</name>
                                <Data_Input>
                                        <sequence>
                                                <type>ContactView</type>
                                        </sequence>
                                </Data_Input>
                                <Data_Output>
                                        <sequence>
                                                <type>void</type>
                                        </sequence>
                                </Data_Output>
                </Block>
                <Block>
                        <name>removeContactView</name>
                                <Data_Input>
                                        <sequence>
                                                <type>ContactView</type>
                                        </sequence>
                                </Data_Input>
                                <Data_Output>
                                        <sequence>
                                                <type>void</type>
                                        </sequence>
                                </Data_Output>
                </Block>
                <External_Dependencies>
                        <lib>java.io</lib>
                </External_Dependencies>
        </must_have>
</ArchInt>
```

Figure 11. Model ArchInt

An attempt was made to try to find a component from open-source repository to fit into this system. Although, finding component was not intended to be part of the evaluation in this experiment, it was done to examine whether the identified characteristics of the Model architectural type in this example are common to all Model architectural types. It was not possible to search open-source repository systems using the characteristics of the Model architectural type identified above as open-source repository systems do not currently support searching for components based on the characteristics defined by an ArchInt document. The only possible way to search was by searching the open-source repositories using the text-matching approach of instances of the Model architectural type available in the repository ("e.g. "servlet", "JavaBeans"). However, the searches retrieved results that were not conforming to the architectural type of the above MVC system in this case.

We tried to examine the impact of ArchInt to facilitate understanding the structure of software systems and allow for modifying their components. So, the defined architectural characteristics in Figure 11 was generated to define the boundaries of the Model component in the selected system so it can be replaced safely. Based on the generated ArchInt description we managed to replace the Model component successfully from the system with another Java class that was generated manually and implemented conforming to the Model architectural type of this system. The new added component fit in the system and ran as expected.

### *D. Discussion*

The results obtained demonstrated that the notion of architectural interface is significant and must be considered carefully when dealing with software components for integration. These results are, in fact, emphasising the assertion made by Mary Shaw who firstly coined the essence of investigating architectural characteristics in order to ensure that components can fit into a system. So, addressing the architectural characteristics are important even when dealing with low-level software components, and these two experiments revealed that explicitly. The experiments also demonstrated that ArchInt successfully identified the salient characteristics of architectural fit into an Eclipse and Applet based systems, and captured that in an architectural type description using the ArchInt prototype language. In addition, the experiments showed that the defined characteristics of the Eclipse plug-in and Applet architectural types represented by ArchInt have been successfully matched automatically by the tool without the need for any human intervention. These experiments also revealed a weakness in Sourceforge.net as it listed components that do not match the characteristics of the Eclipse architectural type, but the repository has considered them mistakenly as matching ones. A possible justification of listing these erroneous results by Sourceforge.net is that the provider of these components seemed to assume that re-users of the components should be responsible for implementing the required architectural characteristics. The providers only focus on producing components that provide certain behaviour without completely concerning about their architectural aspects. As a result, the providers of these components considered them as Eclipse plug-ins, even though they do not practically match the full characteristics of the Eclipse architectural type. This problem could have been avoided if Sourceforge.net used checking mechanism to validate components' characteristics against the claimed architectural type of components by their providers.

With respect to experiment 3, a problem encountered when generating the ArchInt document for the Model architectural type of the MVC system was that the identified architectural characteristics are not fixed for every Model architectural type as it is observed that different characteristics for the Model architectural type were available. For example, one implementation of the Model architectural type might consider implementing data exchange between the instance (i.e. component that conforms to an architectural type) of the Model and the instances of the other architectural types (e.g. Controller and View) using a push-model. The push-model concerns transferring data out of an instance of the Model to other components whenever changes in the state of the instance of the Model occurs, hence requiring the View component to register with the Model component. According to this implementation, a Model component must have a method called "`public void addContactView(ContactView)`" as defined in the Model architectural type in this study. Another implementation of the Model architectural type, however,

might be to exchange data by applying a pull-model. An instance of the Controller component would then need to keep checking changes in the state of the Model component and pull data from the Model component as appropriate, thus requiring the View component to register with the Controller. According to this implementation, the component that conforms to the Controller architectural type is the one that must have a method "`public void addContactView (ContactView)`" and not the Model as described earlier. This variation in the implementation of the various components of the MVC system indicates the lack of a precise definition of what the characteristics of the Model, View, and Controller architectural types are.

It seems that the variety in describing the characteristics of the Model, and also View and Controller, of an MVC system is caused as the three architectural types are in fact metaphors normally used at the design stage to identify the high-level architecture of a system. The component that is responsible for storing and manipulating data in a system can be considered abstractly as an instance of a Model. The architectural types of an MVC system are defined abstractly but their definitive characteristics are left for programmers to determine at implementation time, and hence variety in the characteristics of the Model, View, and Controller architectural types resulted.

An advantage of ArchInt is observed in this experiment indicating that the identified characteristics of the Model architectural type depicted in Figure 11 can be used to understand what is required to modify a component to match the Model architectural type in that MVC system. If ArchInt was not provided, then a developer would need to identify the interfaces of the components of the system at hand manually, which can be difficult and time consuming.

## VIII. Aspects of Architectural Interfaces

Experimenting with architectural interfaces has uncovered some interesting aspects on the overall approach of this approach:

- Cohesive software development environment: consider an IDE with the notion of architectural interface integrated into it. A developer can be given help by automatically generating the source-code that represents the architectural framework for the system based on the design at hand. This will help developers to focus only on writing the source code that will provide the functionality for the components of the system to be developed. Moreover, the IDE can advise the developer about the potential components that match the architectural interfaces of their system, so the developer can re-use components without worrying about any architectural mismatches as that could be dealt with automatically by the IDE. Equally, if a software developer needs to apply some modifications to the architecture of the system, then the IDE can reflect the changes on the high-level artifacts (e.g. design, requirement) of the system and presents them to the developer. This kind of support that is provided by the IDE would not have been possible without the support provided by the notion of architectural interface. The usefulness of architectural interface is to maintain the links between the high-level artifacts and the low level implementation.
- Identity for components: a developer might indicate "I want an Applet component that counts the number of visitors to a webpage"; that would be a more accurate description of the search requirement than "I want a component that counts the number of visitors to a webpage". Instead of describing only behavior to search for components it would be useful to know what components are in the first place. The architectural characteristics defined by architectural types can represent identity for components as the characteristics can be used to discriminate one architectural type from another. In the above example, the identity of the component that the developer was looking for was Applet.
- Source-code documentation: most of the source code available in open-source repository lacks documentation that explains the meaning of the written source code and also how to use it. The lack of documentation is an obstacle that could hinder re-using source code. Architectural interfaces represented in ArchInt provide a means of documenting source-code components. A fully implemented ArchInt specification language will generate all the necessary information that developer need to know in order to re-use components (e.g. how a component can to be registered with a system). However, the current version still provides useful information to describe the architectural characteristics of components that developers can utilize to build their decision about re-using the component. So, developers can identify if the component at hand can fit into their system.
- Formalizing high-level artifacts: the design of a software system is usually an abstract specification of the components of a system and their interaction. System developers are required to map these abstractions into a concrete implementation, and the flexibility they have for doing this is precisely the reason for the difficulty of finding matching re-usable components. Architectural interfaces have been shown to address this issue. If the designer of a software system has provided the description of the architectural types of the system to be built, this will reduce the effort on the implementation stage as developers can use the generated architectural type description to find re-usable components or build their own that conform to the provided architectural type description.
- Enhanced support of component re-use: the external interface of software components can be used by a repository system to automatically classify and organize those components, while a set of characteristics that a re-user requires can be specified and used by the repository to identify candidate components. Moreover, the internal interface is useful to help the repository system retrieve a re-usable

component together with all of its required dependencies (i.e. sub-components). Thus, the repository can provide a complete component to a re-user, without requiring the re-user undertake this action manually.

- Establish classification scheme for software components: the external and internal interfaces can be used to build an organisational hierarchy; Figure 12 illustrates an example. Assume that T is a component that defines the characteristic Y in its internal interface. Sub-components T1 and T2 are both identified as providing the characteristic Y in their external interface. However, sub-component T1 defines the characteristic A in its internal interface while sub-component T2 defines B as a characteristic in its internal interface. As a result of the difference in the characteristics defined by T1's and T2's internal interfaces, the two sub-components can be discriminated from each other. The example indicates that the external interface of a sub-component identifies the potential parent in a hierarchy and the internal interface discriminates a component (or sub-components) from other components.

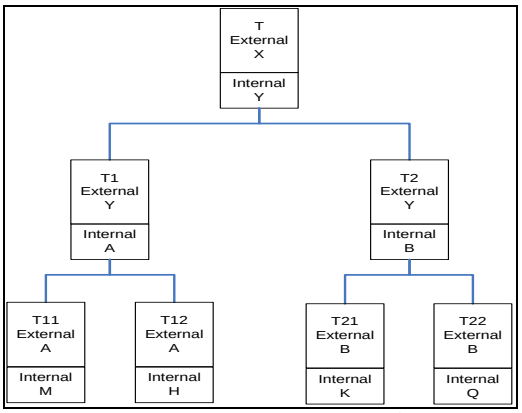

Figure 12.Using External/Internal Interfaces to Organize Components

- Facilitate component modifications: understanding the external interface of a component and internal interface of the system under development might influence the modifications that the re-user might wish to make. A re-user could modify the internal interface of the system under development to match the external interface of a potentially re-usable component, or the re-user could modify the external interface of a component to match the internal interface of the system.

## IX. Conclusion and Future work

This paper has presented a new approach for defining component interfaces in order to address some of the difficulties encountered in components integration. New interface called architectural interface was identified to address integration problem. The approach was formalized by a specification language called ArchInt that was used to validate the notion of architectural interface. Our experimental work revealed that the proposed approach is efficient to identify the architectural characteristics of software components. This work uncovers a significant research direction that need to be considered in depth for successful component integration. The paper proves that considering functionality alone is not enough even at the source code level, and this is in fact in favor of Shaw's assertion.

The current approach is limited to small to medium sized software components written in Java. As a result, our planned future work is to evaluate the notion of architectural interface on a wider range of architectural types including components written in different programming languages. Also, we are going to evaluate the applicability of our approach to supplement the searching criteria of the current open-source search engines in order to facilitate retrieving components that not only provide the required functionality but also fit architecturally into a system. Moreover, we are going to investigate the potential of re-using component that are partially matching the characteristics defined by architectural interface as the current approach is limited only to exact match. So further modifications to software components can be estimated and planned more carefully.

## References

[1] Oberleitner, J., Gschwind, T. and Jazayeri, M.,The Vienna Component Framework enabling composition across component models. *In Proceedings of the 25th International Conference on Software Engineering*.2003. doi:10.1109/ICSE.2003.1201185.

[2] Brown, A., Booch, G., Reusing Open-Source Software and Practices: The Impact of Open-Source on Commercial Vendors. *In Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*.2002 Springer-Verlag. doi:10.1007/3-540-46020-9.

[3] Spivey, J., *The Z Notation: A Reference Manual*, Prentice Hall, 1991.

[4] Sametinger, J., *Software Engineering with Reusable Components*, Springer-Verlag,1997.

[5] Meyer, B., The Grand Challenge of Trusted Components, *in 25th International Conference on Software Engineering (ICSE'03)*. 2003, IEEE Computer Society. doi:10.1109/ICSE.2003.1201252

[6] Brown, A., Short, K.,On Components and Objects: The Foundations of Component-Based Development. *In Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST '97)*.1997.IEEE Computer Society. doi:10.1109/AST.1997.599921

[7] Arbab, F., Boer, F. and Bonsangue, M.,A Logical Interface Description Language for Components. *In Proceedings of the 4th International Conference on Coordination Languages and Models*.2000.Springer-Verlag. doi:10.1007/3-540-45263-X

[8] Hondt, K., Lucas, C. and Steyaert, P., Reuse Contracts as Component Interface Descriptions. *In Proceedings of the Workshops on Object-Oriented Technology.1997.Springer-Verlag*.doi:10.1007/3-540-69687-3.

[9] Duller, M., Tamosevicius, R., Alonso, G. and Kossmann, D., XTream: Personal Data Streams. *In Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.2007.ACM,1088 – 1090.

[10] Shaw, M., Architectural issues in software reuse: it's not just the functionality, it's the packaging. *In Proceedings of the 1995 Symposium on Software reusability*.1995.ACM. doi:10.1145/211782.211783.

[11] Mehta, N., Medvidovic, N. and Phadke, S., Towards a taxonomy of software connectors. *In Proceedings of the 22nd international conference on Software engineering*.2000.ACM. doi:10.1145/337180.337201.

[12] DeLine, R., Avoiding packaging mismatch with flexible packaging. *IEEE Transactions on Software Engineering, 2001. 27(2): p. 124-143*. doi:10.1109/32.908958.

[13] Bass, L., Clements, P. and Kazman, R., *Software Architecture in Practice*, Addison-Wesley, 2003.

[14] ISO/IEC. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std 1471. 2000, Accessed 11 - 2008, Available from: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00875998.

[15] Jones, A., The Maturing of Software Architecture. *In Software Engineering Symposium*.1993. Software Engineering Institute.

[16] Shaw, M., Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[17] Garlan, D., Perry, D., Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering, 1995. 21(4): p. 269-274.*

[18] Budgen, D., *Software Design*, second edition, Pearson Addison-Wesley, 2003.

[19] Szyperski, C., Gruntz, D. and Murer, M., *Component Software - Beyond Object-Oriented Programming*. 2nd edition, Addison-Wesley(ACM Press), 2002.

[20] Allen, R.,*A Formal Approach to Software Architecture, PhD Dissertation*, Carnegie Mellon University, 1997.

[21] Garlan, D., Allen, A. and Ockerbloom, J., Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software, 1995. 12(6): p. 17-26. doi:10.1109/52.469757.*

[22] Yakimovitch, D., Bieman, J. and Basili, V., Software architecture classification for estimating the cost of COTS integration. *In Proceedings of the 21st international conference on Software engineering*. 1999. IEEE Computer Society Press. doi:10.1145/302405.302643.

[23] Garlan, D., Allen, A. and Ockerbloom, J., Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software, 1995. 12(6): p. 17-26*. doi:10.1109/52.469757.

[24] Alkazemi, B., Prototype of Repository System to Support Software Component Re-Use. *In the Proceedings of the Sixth IASTED International Conference on Advances in Computer Science and Engineering*.2010. doi:10.2316/P.2010.689-003.

[25] McIlroy, M., Mass Produced Software Components. *In Software Engineering: Report on a Conference by the NATO Science Committee*.1968.NATO Science Affairs Division,138-150.

[26] Mahmood, S., Lai, R. and Kim, Y, Survey of component-based software development. *IET Software, 2007. 1(2): p. 57-66*. doi:10.1049/iet-sen:20060045.

[27] Li, G., Zhang, L., Li, Y., Xie, B. and Shao, W., Shortening retrieval sequences in browsing-based component retrieval using information entropy. *Journal of Systems and Software, 2006. 79(2): p. 216 – 230*. doi:10.1016/j.jss.2005.04.035.

[28] Ostertag, E., Hendler,J., Prieto Díaz,R. and Braun, C., Computing Similarity in a Reuse Library System: An AI-Based Approach. *ACM Transactions on Software Engineering and Methodology, 1992. 1(3): p. 205- 228.* doi:10.1145/131736.131739.

[29] Zaremski, A.M., Wing, J.M., Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology, 1995. 4(2): p. 146–170*. doi:10.1145/210134.210179.

[30] Hemer, D., Lindsay, P., Specification-based retrieval strategies for module reuse. *In Proceedings 2001 Australian Software Engineering Conference*.2001.IEEE Computer Society. doi:10.1109/ASWEC.2001.948517.

[31] Spivey, J., *The Z Notation: A Reference Manual*, Prentice Hall, 1991.

[32] Fischer, B., Specification-based browsing of software component libraries. *Automated Software Engineering, 2000. 7(2): p. 179 – 200*. doi:10.1109/ASE.1998.732577.

[33] Cechich, A., Piattini, M., Quantifying COTS component functional adaptation. *In 8th International Conference on Software Reuse: Methods,Techniques and Tools (ICSR 2004).2004.LNCS(3107)*.

[34] Assman, U., *Invasive Software Composition*, Springer Verlag, 2003.

[35] Budgen, D., Brereton, P. and Turner, M. Codifying a Service Architectural Style. *In Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04) - Volume 01 2004*.IEEE Computer Society. doi:10.1109/CMPSAC.2004.1342800.

[36] ClayBerg, E., Rubel, D., *Eclipse: Building Commercial-Quality Plug-ins. 3rd edition*, Addison-Wesley, 2004.

[37] Oberleitner, J., Gschwind, T. and Jazayeri, M., The Vienna Component Framework enabling composition across component models. *In Proceedings of the 25th International Conference on Software Engineering*.2003.IEEE Computer Society. doi:10.1109/ICSE.2003.1201185.

[38] Shaw, M., Clements, P., A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. *In Proceedings of the 21st International Computer Software and Applications Conference*.1997.IEEE Computer Society. doi:10.1109/CMPSAC.1997.624691.

[39] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley Professional, 2003.

[40] Gacek, C., *Detecting Architectural Mismatches During Systems Composition*, PhD thesis, University of Southern California, 1998.

[41] Besnard, D., Gacek, C. and Jones, C., *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, Springer, 2005.

[42] Crane, S. Darwin: an Architectural Description Language. 1997, Accessed 11 - 2008, Available from: http://www.doc.ic.ac.uk/~jsc/research/darwin.html.

[43] Garlan, D., Monroe, R.T. and Wile, T., ACME: An Architecture Description Interchange Language. *In Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*.1997.IBM Press,169-183.

[44] Garlan, D., *Shaw, M., An Introduction to Software Architecture*,1994, Technical Report:CS-94-166,Carnegie Mellon University.

[45] Jefferson, N., *Dependable Compositions: A Formal Approach*, PhD Thesis, Newcastle University, 2006.

[46] Luckham, D., *Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events*,1996, Technical Report:CSL-TR-96-705,Stanford University.

[47] Medvidovic, N., Taylor, R., A Classification and Comparison Framework for Software Architecture

Description Languages. *IEEE Transactions on Software Engineering, 2000. 26(1): p. 70-93*. doi:10.1109/32.825767.

**Basem Y. Alkazemi**, is an assistant professor at Umm Al-Qura University (UQU) in Saudi Arabia under the school of computer science & Engineering. He obtained his PhD in 2009 from Newcastle University in U.K. His PhD topic was concerned with addressing the complexity of re-using open-source software components. Basem is currently holding the position of managing director of the e-government project at UQU that leads to the integration of all the university software systems in a unified model. He is a member in the IEEE, SIGSOFT-ACM, and SEI societies. His main research interests include software architectural patterns, software product lines, Aspect-oriented SE, and CBSE.