A FRAMEWORK FOR RESOLVING MISMATCHES THAT MAY OCCUR DURING SYSTEM INTEGRATION USING COTS SOFTWARE

By Horatious Njie Tanyi

B. A, May 1998, University of St. Thomas, St. Paul Minnesota
Master of Software Design & Development May 1992 University of St. Thomas, St. Paul Minnesota

A Dissertation submitted to

The Faculty of

School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Doctor of Science

January 31, 2010

Dissertation directed by

Shmuel Rotenstreich
Professor of Computer Science

The School of Engineering and Applied Science of The George Washington University certifies that Horatious Njie Tanyi has passed the Final Examination for the degree of Doctor of Science as of 12/10/2009. This is the final and approved form of the dissertation.

A FRAMEWORK FOR RESOLVING MISMATCHES THAT MAY OCCUR DURING SYSTEM INTERGRATION USING COTS SOFTWARE

Horatious Njie Tanyi

Dissertation Research Committee:

Shmuel Rotenstreich, Professor of Computer Science, Dissertation Director

Simon Berkovich, Professor of Computer Science, Committee Member

Abdelghani Bellaachia, Professor of Computer Science, Committee Member

Matthew Burke, Professor of Computer Science, Committee Member

Yul Williams, Doctor of Science, Committee Member

Abstract of Dissertation

A FRAMEWORK FOR RESOLVING MISMATCHES THAT MAY OCCUR DURING SYSTEM INTEGRATION USING COTS SOFTWARE

As software systems become increasingly complex and expensive to build, software engineers are challenged with various options to meet these challenges by turning towards pre-build software components known as Commercial Off-The-Shelf (COTS) software. COTS software is usually acquired as binary components, and sometimes their behavior is poorly specified. One of the major challenges faced by software engineers when developing systems by integrating COTS is guaranteeing that the components correctly integrate with each other. In particular, to ensure the interaction behavior of these components and the assumptions they make about the interaction behavior within the external context in which they expects to operate. This dissertation introduces a framework for semi-automatically developing adaptors to resolve signature and protocol mismatches that may occur during COTS integration by unifying various threads of research including byte-code engineering, black-box test case generation, protocol discovery, mismatch patterns and adaptor generation.

TABLE OF CONTENT

List of Figures

CHAPTER 1 INTRODUCTION

As software systems become increasingly complex and expensive to build, software engineers are challenged with various options to meet these challenges by turning towards pre-build software components from third-party vendors known as Commercial Off-The-Shelf (COTS) components. These COTS software components are usually acquired as a black box or binary component were the developers have little or no access to the source code, no influence over the evolution of the component in terms of maintenance, and behavior of the component may be poorly specified to understand its behavior in a composition. One of the major challenges faced by system architects and designers when developing systems by integrating commercial off-the-shelf components is guaranteeing that the components correctly integrate with each other. In particular, to ensure the interaction behavior of these components and the assumptions that they make about the interaction behavior of the external context in which it expects to operate is an effort all on its own. The idea of increasing software productivity by building systems from existing parts, even though seems to be rather attractive, has left software developers with a critical challenge which is, to understand whether these parts being the systems' components correctly integrate with one another. Characteristics of COTS software as identified in [24] that make the development process different from building a system internally or building systems by composing components that are developed externally are:

- Lack of access to the source code to enable developers to customize the code to meet their needs.

- Little or no influence over the evolution of the COTS product since the relationship between the vendor and the customer is one vendor to many customers. The upgrade to a new release might have an adverse effect to already tested functionality or the behavior may have unexpected results when composed with other existing components in the system.
- Complete or sometimes correct behavioral specification is not always available from the vendor because the vendor might loose their competitive edge if such information is released. Lack of sufficient information about the behavior of a component may lead the developers to integrate the component in ways that the component was not intended for or the vendor did not expect.
- Mismatches may occur due to wrong assumptions made by each vendor of a COTS component about the interaction behavior of the external context in which it expects to operate.
- Many COTS are simple standalone applications therefore integrating them to work with other components to build a system is simple not worth the effort.

We consider a component as an entity or a unit of composition with explicit context dependency that has persistent identity, with a contractually specified interface representing the required and provided services. Even though components are independently deployed, they usually interact with other components via their interfaces and the complete set of components that a particular component requires in order to be functional is called the external context of the component.

Unlike a component model, which specifies what a component does, an interface model specifies how the component is used. While components typically have well defined interfaces in terms of specifying the functionalities required – services that must be made available for the component to execute as specified, or provided by the component – services that are provided to other components, the sequencing constraints that is, which call must be invoked in which order is formally not specified. One of the important roles of an interface is to guide integration with other components by the consumer in order to adapt their implementation to the expected usage context without breaking the black-box model. Lack of formally specifying the order in which to invoke the services provided by a component, compromises re-use and makes it difficult to integrate with other components.

1.1 Motivation

During the integration of COTS components to build an application, mismatches can occur at different levels of abstraction. A classification of potential functional mismatch patterns that may occur during integration of COTS components includes:

1.1.1 Operation Signature Mismatch

In order for two components to transfer or share data during a collaboration, their interfaces need to agree on its representation, which entails the operation name, parameter name, data type and/or data format as well as parameter constraints (accepted value range) on the input/output messages. The signature of a component's interface refers to the profile of the interface specifying the structure of parameters and related data types. Signature mismatches can be further categorized into syntactic-level and semantic-level mismatches. The syntactic-level signature mismatches are those occurring at the structure of a single operation's signature and the basic mismatch patterns can be enumerated as follows:

1.1.1.1 Syntactic-level Operation Mismatch Patterns

- **Extra parameters mismatch pattern**

  The consumer's provides interface has extra parameters that the provider's requires interface does not need to fulfill the consumer's request.

- **Missing parameters mismatch pattern**

  The consumer's provides interface has missing parameters that the provider's required interface needs. In this case the missing parameter may be part of a different operation within the same interface specification.

- **Splitting parameters mismatch pattern**

  The consumer's provides interface has parameters that the provider's requires interface needs them split into two or more parameters in order for the provider to fulfill the consumer's request.

- **Merging parameters mismatch pattern**

  Two or more parameters of the consumers provides interface needs to be merged into one parameter for the provider's required interface.

- **Ordering of parameters mismatch pattern**

  The sequence of parameter list of the consumer's provides interface needs to be reordered to match that of the provider's required interface.

1.1.1.2 Semantic-level signature mismatch pattern

Semantic-level signature mismatch patterns are conceptual mismatches referring to the signature of the provided and required interfaces between the consumer and provider and the basic patterns can be enumerated as follows:

- **Parameter name mismatch pattern**

  This is the situation were the parameter names between the provides and requires interface of the consumer and provider don't match.

- **Parameter Constraints mismatch pattern**

  Parameters in the provider's provides and requires interfaces are constraint but the consumer is unaware of any such constraints. For example, certain parameters in a providers requires interface may have a value range constraint which is unknown by the consumer.

- **Parameter type mismatch pattern**

  The consumer may use a character value "Y/N" to represent a particular parameter while the provider uses a Boolean for the same parameter.

  Equally, the consumer may store a particular document in a MS word format while the provider stores the same document in word perfect format.

- **Operation name mismatch pattern**

  The name of operations in the consumer's provides interface doesn't match those of the provider's requires interface.

1.1.2 Protocol Mismatch

In order for two components to interact with one another, they must agree as to the order in which messages are exchanged between them. This means agreeing on the number and order of individual transfers of data or control. For example, a consumer may expose one signature in its provides interface which is equivalent to two or more signatures in the provider's requires interface thereby requiring the consumer to call the provider twice to fulfill its service. Equally a provider may require an initialization operation in its interface to be executed before executing any other operation. The Basic patterns of the syntactic-level protocol mismatches can be enumerated as follows:

1.1.2.1 Syntactic-level protocol mismatch pattern

- **Extra message mismatch pattern**

  The consumer's requires interface for a given operation has some extra messages that the provider's provides interface does not expect to send. Likewise the provider's requires interface for a given operation has extra messages that the consumer's provides interface does not expect to send.

- **Missing message mismatch pattern**

  The consumer's provides interface for a given operation, does not have some messages/parameters that the provider's requires interface expects to receive.

- **Splitting messages mismatch pattern**

  The consumer's provides interface has some messages that the provider's requires interface expects to split to receive or vice versa.

- **Merging message mismatch pattern**

The consumer's provides interface has some messages that the provider's requires interface expects to merge in order to receive or vice versa.

- **Message exchange sequence mismatch pattern**

  The order in which to exchanges messages between the consumer's provides interface and the provider's requires interface is not compatible or unknown by the consumer.

1.1.3 Control transfer

To eliminate mismatch between two components, they must agree on the mechanism and the direction of control transfer. For example, a single threaded component requires completion of execution before control is transferred, but a component that does not block can continue to execute. Hence a problem exist whereby, both components might not be ready or willing to interact at the expected time.

1.2 Limitations with State of the Art

To facilitate the interoperability, management, execution and communication between heterogeneous components (i, e components build with different languages, across different platforms at different locations), current component platforms such as .NET, DCOM or CORBA-Interface Description Language (IDL) already provide mechanisms for component interoperability and focuses on the syntactic aspects of what the component provides and possible requires. While the provision of IDL type interfaces is an important step towards integration of components because they provide communication and data exchange mechanisms for letting them interact, the interaction between methods or the order in which to exchange messages between collaborating components, that is, the interaction protocol is missing in this interface model thereby compromising integration due to potential protocol mismatch (differences in component behavior).

As a result of these drawbacks, there have been several proposals in the field of black box component mismatch resolution including enhancing component interfaces with specification of their behaviors or interaction protocol in the form of Π-calculus [75,76], finite state machine [28,47], regular expression as well as the use of adaptors [28], wrappers [44,46], superimposition [40,41], Binary Component Adaptation [42] and Architecture Description Languages (ADL) [43] to name a few.

Despite these advances, resolving mismatches such as protocol mismatch during component integration continues to be a challenging effort because most components in the industry today still depend on the classic CORBA-IDL interface model. Additionally, the target audience for an ADL specification of components has been the designers of the original component themselves, and not the consumer of these components. While mismatches at the operation level can be resolved by inspection of the CORBA-IDL type interfaces or addressed statically by using architectural specification or type theory of a system and its intended components thereby enabling the engineer to reason about the interactions early-on and at a high level of abstraction or even provide substantial help in detecting and preventing errors in mismatched static properties such as operation names, parameter name, type and format, the most difficult to identify and resolve are those that occur at run time which includes the transfer control and interaction protocol mismatches because by themselves, these components may work correctly, but at the global level, their assumptions as to the external context or environment in which they are expected to operate seems to be in conflict. A practical approach or mechanism that can address this problem using tools that are readily available to software engineers is clearly needed.

1.3 Goal of this Thesis

After analyzing other approaches related to adaptor generation as related to resolving protocol mismatch during component integration, it was realized that none of this approaches lend themselves to a practical solution to the problem. Particularly, even though enhancing component interfaces with specification of their behaviors or interaction protocol in the form of Π-calculus [75,76], finite state machine [28,47], regular expression provides a solid foundation for generating appropriate adaptors for mismatch resolution during component integration, their applicability or utility has been questionable given that COTS products still follow the classic IDL type interfaces. For the purpose of solving the component mismatch problem during integration, it is necessary to adopt a different approach for adaptor generation that leverages tools that are readily available to software engineers.

This dissertation introduces a framework for semi-automatically developing adaptors to resolve mismatches that can occur during the integration of COTS components by unifying various threads of research including byte-code engineering, black-box test case generation, protocol discovery, mismatch patterns and adaptor generation.

1.4 Organization of this Thesis

The organization of this dissertation consists of 7 major chapters. Chapter 2 will provide background information on component mismatch resolution as well as background on various technologies and lines of research used in this work. Chapter 3 will introduce the requirements for a mismatch adaptor generation framework including the framework. Followed by a detail design of the framework in chapter 4 with a simple example. Chapter 5 will introduce a case study for our proof of concept. In chapter 6, we discuss adaptor analyses including component compatibility and adaptor compatibility and finally in chapter 7, we offer our conclusion and future work.

Chapter 2 Component Mismatch Resolution Background

This section reviews work in the field of component mismatch resolution. We begin with an overview of formal specification approaches including Architecture Description Languages and Architectural Styles then we discuss black-box adaptation techniques that lend themselves to mismatch resolution such as binary component adaptation, wrappers, adaptors, and superimposition followed by various technologies that relate to the work in this dissertation such as byte code engineering, component models, interface models, component protocols, dynamic protocol inference, black-box testing, and test case generation.

2.1 Formal Specification

2.1.1 Architecture Description Languages

There has been several research work done in the area of formal specification in addressing uncovering mismatches in component behavior. Architecture Description Language's (ADL) which is a language that provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation [43], uses formal specification theories including partially-ordered event sets [48], communicating sequential processes (CSP) [49], π-calculus [43], and model-based formalisms such as the chemical abstract machine (CHAM) [9] to model various aspects of a system's behavior thereby providing the ability to successfully uncover mismatches in static properties of a system. Different ADL's focus on different system domains, architectural styles, or aspects of the architectures they model and they use different terminology to specify the same aspect of the system. For example, Darwin, UniCon [50], and Wright models components and refer to them as simply components, whereas in Rapide, components are referred to as interfaces. Unlike components, an interface point in Wright is referred to as a port, and in UniCon a player. The, provides and requires interfaces in Rapide are referred to as functions and specify synchronous communication, while in and out actions are used to represent asynchronous events. UniCon supports a predefined set of common player types, including *RoutineDef*, *RoutineCall*, *GlobalDataDef*, *GlobalDataUse*, *ReadFile*, *Write-File*, *RPCDef*, and *RPCCall*. Wright, Rapide and UniCon supports specification of relatively complex component communication protocols including the expected component behavior or constraints on component usage relevant to each point of interaction thereby providing a suitable means to describe and reason about behavioral analysis as related to mismatches.

Daniel Compare, Paola Inverardi, and Alexander L. Wolf [5] developed a method that depends on a monolithic specification and analysis of a whole system's component interaction behavior. This method was further refined due to its limitation to address situations were the intent was to build systems from existing components by permitting the individual specification of a component's interaction behavior together with a specification of the assumptions that the component makes about the expected interaction behavior of other components with which it might have to interact. The method would then use the specification to discover mismatches among the components at system integration or configuration time [9]. Both methods are based on the CHAM (Chemical Abstract Machine) formalism [38] were an architecture is specified with processing, data, and connecting elements, modeled as an abstract machine fashioned after chemicals and chemical reactions. A CHAM is specified by defining molecules, their solutions, and transformation rules that specify how solutions evolve. The interface of a process and its connecting elements are implied by their topology and the data elements the current configuration allows them to exchange.

2.1.2 Architectural style

Another significant source for detecting architectural mismatches comes from mismatches among different architectural styles. As defined by [18] an architectural style defines a family of systems based on a common structural organization. An architectural style defines a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done [29]. It provides a vocabulary of design elements like components, connectors such as pipes and design rules or constraints that determine the permitted compositions of the elements. For example, a rule might prohibit cycles in a particular pipe-and-filter style. It further provides semantic interpretation whereby, compositions of design elements, suitable constrained by the design rules have a well-defined meaning. These constraints are not style specific. A constraint like having an explicit data connector for data transfer between components is not limited to the pipe-and-filter style but can be shared by another style like the distributed process. The absence of these constraints in some styles is just as important. For example, the main-subroutine and event-based styles do not have explicit data connectors, but rather use shared variables for data transfer. Analysis of common architectural styles suggest that they are discriminated by a variety of features or sometimes called conceptual feature such as data transfer, how data and control interact, what type of reasoning is compatible with the style, how control is shared, allocated and transferred among the components. Features that distinguish one style from the other helps understand why a particular style is an appropriate solution for one type of problem and not the other. Mismatches may occur because the subsystems have different choices for some particular feature. For example, one is multi-threaded and the other isn't thereby causing a potential for a synchronization problem.

2.2 Black-Box Adaptation

With black-box adaptation technique, component reuse is as-is, the adaptation technique needs no knowledge of the internal structure or implementation details of the component. This technique only requires a good understanding of the interface of the component.

2.2.1 Wrappers

With wrapping, one or more components can be declared as part of an encapsulating component the *wrapper*, which is responsible for forwarding request to the wrapped component with minor changes. There is no clear boundary between wrapping and aggregation, but wrapping is used to adapt the behavior of the enclosed component whereas aggregation is used to compose new functionality out of existing components providing relevant functionality. A drawback with wrappers is that, it may result in considerable implementation overhead since the complete interface of the wrapped component needs to be handled by the wrapper, including those interface elements that need not be adapted. Also, wrapping may lead to excessive amounts of adaptation code and serious performance reductions [51]. Wrappers introduce additional redundancy into the system because they duplicate part of the interfaces of the components. Therefore, if a component interface changes, additional work is required to adapt the program. Even though wrappers can be used to solve some interface mismatch problems, they may also require a large number of wrapper types and can introduce potentially severe run-time and space overheads. Wrapping components can be a tedious process, and components using wrappers are often harder to understand and therefore harder to maintain.

2.2.2 Binary Component Adaptation

Unlike creating new classes as with the wrapper technique, with Binary Component Adaptation (BCA), the definition of the original component is modified. The main requirement for BCA is that the component in binary form should contain enough high-level information about the underlying component to allow for inspection and modification of its structure. Binary component adaptation rewrites component binaries before (statically) they are loaded or (dynamically) while they are loaded. By directly modifying or rewriting binaries, BCA takes advantage of the flexibility of source code level modifications without incurring its disadvantages.

BCA is good for dealing with interface incompatibilities such as adding new methods, misnamed methods, method argument type mismatch, ordering of parameters, renaming methods, and changes to the inheritance or sub typing hierarchy. The BCA approach is such that, the adaptation system requires two inputs. The original component in binary form and a list of modifications or the delta file, and in turn, produces a modified component including the desired modifications or additions. For example, suppose Component A needs to collaborate with Component B, but there is a method name mismatch between CalcArea in Component A and CalculateArea in Component B. Thereby requiring the need to change CalcArea in component A to match CalculateArea in Component B. To implement this adaptation, the adaptation system must perform two changes. When reading-in component A, it must update its method table by replacing CalcArea to CalculateArea. Then, the system must update all references to CalcArea to CalculateArea within its environment. In order to preserve consistency, the adaptation system must have access to references within the environment that could possibly be impacted by the modification. If the complete set of references within the environment can be determined statically, the system will modify the entire environment with these changes. The static approach has the disadvantage in that, it physically duplicates all components with reference to the CalcArea and thus increases disk space usage since each application potentially needs its own copy of every component in every library. On the other hand, static adaptation completely eliminates any runtime overhead since no modifications are needed at load time or during component execution.

With dynamic binary component adaptation all adaptations are done when components are loaded into memory during runtime thereby imposing some overhead during the loading phase and requires delta files to be distributed with the application, thereby requiring users to use a runtime environment that is BCA-aware.

2.2.3 Adaptors

An adaptor or Glue is a piece of code that resides between two components that are functionally compatible but their interfaces are not. That is, the services provided by one component are equivalent to services required by the other but their interfaces are not compatible. The adaptor compensates for the mismatches between their interfaces including operation signature mismatch, control flow mismatch and protocol mismatch. Using adaptors to resolve mismatches between two components has been the de facto approach in the industry. Yellin and Strom laid the foundation for component adaptation in [28] were a finite state machine was used to specify a components' behavior and introduced the notion of adaptor as a software entity capable of enabling two mismatch components to be integrated.

2.2.4 Superimposition

Superimposition is a black-box adaptation technique that provides the ability to impose predefined, but configurable types of functionality on reusable components by introducing a layer that encapsulates the component to be adapted and all message exchange to and from the encapsulated component are intercepted by this layer thereby providing the ability to superimpose multiple predefined adaptation behavior types that can be configured for a given component. One advantage of this approach over traditional wrappers is that the layers are transparent and provide reuse of adaptation behavior. The notion of superimposition in computing systems was earlier identified within the context of Communicating Sequential Processes (CSP). Bouge and Francez in [41] define the superimposition R of P over Q as the additional superimposed control P over the basic algorithm Q. Analogously, superimposition S of B over O can be defined as the additional overriding behavior B over the behavior of component O [40]. The principle underlying superimposition is that the encapsulated component and the functionality adapting the component (the superimposed layer) are two separate first class entities that need to be very tightly integrated.

2.3 Related Technologies

2.3.1 Component Model

There are various definitions of a component in software engineering but for the sake of this work a component as previously defined is *an entity or a unit of composition with explicit context dependency that has persistent identity, with a contractually specified interface representing the required and provided services*. Components may have multiple interfaces as they can also be nested to form a hierarchy. A higher-level component may be composed of several mutually interconnected cooperating subcomponents. A component model defines the basic architecture of a component as well as specifies the structure of their interfaces, mechanism by which they interact with the environment, their internal structure and also provides guidelines in relation to their creation, implementation and assembly into an application. There are three widely used component models for distributed computing, supporting middleware platforms namely; Microsoft's Component Object Model COM+/.NET [53], Sun Microsystems' EJB [52] and the Object Management Group's Common Object Request Broker Architecture (CORBA) Component Model (CCM) [54].

2.3.2 COM+/.NET

COM+ is the name of the COM-based services and technologies first released in Windows 2000. COM+ brought together the technology of COM components and the application host of Microsoft Transaction Server (MTS). COM+ automatically handles difficult programming tasks such as resource pooling, disconnected applications, event publication and subscription and distributed transactions. COM+ infrastructure also provides services to .NET.

Microsoft COM (Component Object Model) technology in the Microsoft Windows-family of Operating Systems. It enables software components to communicate. COM is used by developers to create re-usable software components, link components together to build applications, and take advantage of Windows services. The family of COM technologies includes COM+, Distributed COM (DCOM) and ActiveX® Controls. COM and .NET are complementary development technologies. The .NET Common Language Runtime provides bi-directional, transparent integration with COM. This means that COM and .NET applications and components can use functionality from each system.

The .NET Framework is a development and execution environment that allows different programming languages & libraries to work together seamlessly to create Windows-based applications that are easier to build, manage, deploy, and integrate with other networked systems [55]. Integrated across the Microsoft platform, .NET technology provides the ability to quickly build, deploy, manage, and use connected, security-enhanced solutions with Web services.

At the heart of .NET is the Common Language Runtime, commonly referred to as the CLR. The CLR is made up of a number of different parts including language independence. Language independence is attained through the use of an intermediate language (IL). What this means is that instead of code being compiled in actual machine code (code that the CPU would run), it is instead compiled into a high-level generic language. Whatever language you write your code in, when you compile it with .NET it will become IL. Since all languages eventually get translated into the intermediate language, the runtime only has to worry about understanding and working with the intermediate language instead of the plethora of languages that you could actually use to write code. Other features of .NET include Just-in-Time Compilation and Memory Management.

2.3.3 Enterprise Java Bean

The Enterprise JavaBeans (EJB) component model is basically an extension to the JavaBeans component model to support server components, which are reusable, prepackaged pieces of software (bean) designed to run in an application server. They can also be combined with other software components to create custom applications. The EJB specification defines interfaces and required behavior for both - enterprise beans and their containers.

An Enterprise JavaBeans (EJB) container provides a run-time environment for enterprise beans within the application server. The container handles all aspects of an enterprise bean's operation within the application server and acts as an intermediary between the user-written business logic within the bean and the rest of the application server environment. The EJB container provides many services to the enterprise bean, including the following:

- Beginning, committing, and rolling back transactions as necessary.
- Maintaining pools of enterprise bean instances ready for incoming requests and moving these instances between the inactive pools and an active state, ensuring that threading conditions within the bean are satisfied.
- Most importantly, automatically synchronizing data in an entity bean's instance variables with corresponding data items stored in persistent storage.
- Transaction management, security management, and persistence management.

Access to a bean is handled by the *home* and *remote* interfaces. While the remote interface defines the bean's business methods, the home interface specifies methods for the bean's creation and destruction, as well as so-called finder methods - methods for querying the population of beans to find a particular bean instance. By default, both interfaces are accessible via RMI over IIOP protocol [64]. To obtain a reference to the bean's home interface, the Java Naming and Directory Interface (JNDI) can be used. EJB supports distributed flat transactions defined in Java Transaction Service (JTS). Every client method invocation on a bean is supervised by the bean's container, which makes it possible to manage the transactions according to the transaction attributes that are specified in the corresponding bean's deployment descriptor (an XML-based document containing the bean's basic characteristics, usage of the services provided by the container, and requested references to other beans).

There are three types of enterprise beans - session beans, entity beans, and message-driven beans. Session beans are short-lived objects existing on behalf of a single client that do not represent any shared and/or persistent data. Depending upon its conversational state, a session bean can be stateful or stateless. On the other hand, an entity bean is usually a long-lived, transactional object representing persistent data (usually stored in a database) that can be shared by multiple clients. Entity bean persistence is managed either by the bean itself (bean-managed persistence), or it is driven by the container (container-managed persistence) in compliance with the abstract persistence schema defined in the bean's deployment descriptor. A message-driven bean is in fact a stateless session bean, whose execution is driven by messages delivered through Java Message Service (JMS).

2.3.4 CORBA Component Model

Unlike the EJB Component Model which is only intended to be used in the Java environment, the CORBA Component Model (CCM) [54] specified by the OMG's aims at enabling open interconnection of distributed components that are heterogeneous in nature (implemented using different programming languages and running on various platforms). The CCM specification defines four basic models.

The CCM abstract model covers specification of component interfaces and their mutual interconnections. The Interface Definition Language (IDL) has been extended for this purpose. A CORBA component can have multiple interfaces (ports in the CCM terminology) to either provide or require functionality to/from its clients. Those interfaces support two interaction modes: facets and receptacles can be used for synchronous method invocations, event sources and sinks can be used for asynchronous notifications. Like EJB, the CCM abstract model also defines component homes - instance managers serving as component factories and finders.

The Component Implementation Framework (CIF) and its associated Component Implementation Definition Language (CIDL) form the base of the CCM programming model. The main purpose of the CCM programming model is to describe component implementations and their non-functional properties/system requirements (security, persistent state, transactions, etc).

The CCM deployment model defines how to assemble an application composed of components, pack it into a software package, and install the application on various sites. The deployment information is provided in a form of various descriptors (there are four kinds of descriptors defined by the CCM) using the XML vocabulary of the Open Software Description.

The CCM execution model defines containers as a runtime environment for component instances and their respective homes. A single container server can host several containers. Containers hide the complexity of the underlying system services such as the POA, transactions, persistence, security, etc.

While these middleware platforms such as CORBA, EJB and .NET facilitate the development of distributed applications by providing certain infrastructure services required in many distributed systems such as name services, remote method calls, parameter marshalling etc., these platforms fail to support the development of distributed systems with independent components and their component models do not provide sufficient information for component interoperability checks or component adaptation [56].

2.3.5 Interface Model

The interface of a component, which is the only point of access, is used to expose the observable behavior of a component. Specifically, it defines the component and serves as the basis for reasoning about its use and implementation. In order for a consumer to reuse or consume the services from a component, the components producer provides an interface, which describes the required and/or provided functionalities of the component. To facilitate the interoperability, management, execution and communication between heterogeneous components (components build with different languages, across different platforms at different locations), current industry specification frameworks for components include the classic signature-list based interface models, stemming from component-oriented platforms, such as the CORBA-Interface Description Language (IDL), which mainly focuses on the syntactic aspects of what the component provides and possible requires.

Whereas the provision of IDL type interfaces is an important step towards integration or composition of components, because signature type mismatches can be identified, and resolved through adaptation [45, 57, 35], the interaction between methods or the order in which to exchange messages between components, that is, the interaction protocol is missing in this interface model thereby compromising integration due to potential protocol mismatch (differences in component behavior).

2.3.6 Component Protocol

Component interface protocols, also called component protocols, are sequencing constraints that a component should obey while collaborating with another component. A component protocol can be bidirectional that is, it specifies sequencing constraints on its provides and requires interface or unidirectional in which case it specifies sequencing constraints only on its requires interface (messages that can be received). Component protocols can be specified as part of documentation of the component to enable the consumer to use static verification tools for analysis to determine if the component usage conforms to the protocols. Despite the fact that understanding of the protocol of a component is rather useful in assuring the correct component usage without compromising reusability, most components have no protocol specification or documentation. Also as stated above, current component platforms have a serious limitation in that, they do not have suitable means to describe and reason on the concurrent behavior of interacting components. To address this problem, there's been extensive research in inferring or mining protocols from components dynamically or statically [70, 71, 72, and 73].

Dynamic protocol inference techniques discover protocols from execution traces collected while the component is in use [62, 67] while static protocol inference techniques deduce sequencing constraints by statically analyzing component code [63, 66]. Given that the static protocol inference approach requires source code and this work addresses black box components were the code is not available, we will limit our discussion to the dynamic inference approach.

2.3.7 Dynamic Protocol Inference

A common way to express protocols is to model them as Finite State Automaton (FSA). When an FSA protocol captures all the legal method sequences in an interface, any method call that does not follow the transitions in the FSA is considered as protocol violation. However, an FSA protocol can leave out method calls that have no sequencing constraints. If a method in a component interface does not occur on any transition edge of the FSA protocol, it is legal to call this method at whatever state the component is in. The underlying assumption is that calling this method does not change the state of the component.

The problem of the dynamic protocol inference is such that, given an interface with a set of services and the traces collected while the services of the interface were in use; infer the sequencing constraints on the interface in the form of FSA. There has been several algorithms already proposed in the literature for dynamic protocol inference including [58, 59, 60, 61, 62].

The dynamic protocol inference is generally performed by observing interactions of a component at runtime and the process is divided into four steps. The trace collection step, the interaction scenario extraction step, the protocol inference step, and the protocol testing or usage step. In the trace collection step, specific types of trace data for a service in a component interface are collected when the execution of the service begins and terminates. In the scenario extraction step, the recorded traces are analyzed and all interdependent call sequences for each service are grouped into a set of scenarios to serve as input to the protocol inference step. In the protocol inference step, the derivation of the FSA is depending on the approach used. In [62] all interactions generated from the same service are clustered to infer the FSA. In [58] calls to the same objects are group together while [61] uses dependencies between requests flows. Some algorithms employ a very strict inference technique to the point were legal sequences or possible behaviors are rejected from the FSA (over-restrictiveness) while others generalize the observed behavior to the point were illegal sequences or behaviors that should be excluded are part of the FSA (overgeneralization). In the protocol usage phase, the inferred protocols are used to characterize the test suite [63], validate the process [59] or validate the trace [61].

2.3.8 Black Box Component Testing

Black box testing, also called functional testing and behavioral testing, focuses on validating the functional and behavioral requirements of a black box component based solely on the outputs generated in response to selected inputs as specified by the interface since the implementation or internal workings of the component is not known. Given that the tester is dependent solely on the exposed behavior as specified in the interface, an efficient test plan that describes the scope, test activities and identifies all test items, features to be tested, and individual test cases is very critical. A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

2.3.9 Test Case Generation

Typically, component test case generation involves three phases namely; component analyzer, path selector and test case generation. During the analysis phase, the component is analyzed with an analyzer, which produces input typically in the form of a flow graph or a data-dependency graph for the path selector and the test data generator phase. The path selector inspects the input in order to find paths leading to high code coverage. The path selector phase is very critical to the system as a whole since the effectiveness of the entire system is as good as the paths selected for test data generation. The objective is to produce paths that provide enough coverage. Which means the coverage criterion has to be strong enough to cover all paths, statements, branches and conditions. The paths are then passed as arguments to the test data generator, which derives suitable input values that traverses the given paths. The object of the test data generation phase is to generate data that traverses all the paths received from the selector phase. This involves identifying the path predicate and then, solving the path predicate in terms of its input variables. There are various methods in the literature for deriving test data including various search strategies, and simulated annealing [64, 65].

In the literature, there are several approaches proposed for test case generation, mainly random, path-oriented, goal-oriented and the intelligent approach.

The random approach determines test cases based on assumptions concerning fault distribution. It mostly does not perform well in terms of coverage since it relies merely on probability; it has quite a low chance in finding fault that is revealed only by a small percentage of the component input. It executes components with random input and then observes the component structures executed. It works well for very small components.

The Goal-oriented approach identifies test case coverage for a selected goal such as a statement or branch, irrespective of the path taken. The goal-oriented approach is much stronger than the random approach, in the sense that it provides guidance towards a certain set of paths. Instead of letting the test case generator generate input that traverses from the entry to the exit of a component, it generates input that traverses a given statement, branch or path. Because of this, it is sufficient for the generator to find input for any path. Intelligent techniques of automated test case generation rely on complex computations to identify test cases.

The Path-oriented approach generally uses control flow information of the component to identify a set of paths to be covered and test cases are generated for these paths, which generally lead to better coverage. The Path-oriented approach is the strongest among the three approaches since it uses specific paths for generating test cases as oppose to providing the generator with a possibility of selecting among a set of paths. In this way it is the same as a goal-oriented test data generation, except for the use of specific paths.

These techniques can further be classified as static and dynamic. Static techniques are often based on symbolic execution. Instead of using actual values, variable substitution is used. This approach is best suited for white box component test generation. Unlike the static technique, the dynamic techniques obtain the necessary data by executing the component under test. Values of variables are known at any time of the execution. By monitoring the flow during execution, the system can determine if the intended path was taken. Using different kinds of search methods, the flow can be altered by manipulating the input in a way that the intended branch is taken. This approach is best suited for black box test case generation.

Chapter 3 The Mismatch Adaptor Generation Framework

In this chapter, we first identify requirements for a mismatch adaptor generation framework before introducing the framework designed to generate adaptors for resolving mismatches based on mismatch patterns at the signature as well as the behavioral level which unifies various threads of research in software engineering including byte code engineering, test case generation and protocol inference to infer sequencing constraints from execution traces collected while the component is in use. Bytecode engineering is used for analyzing component interfaces and extracting properties of the interface relevant for integration as well as internals of the component such as subcomponent calls that are not exposed at the interface level. We also provide an interface mapping facility to facilitate mapping of operations, and parameters including their domain between the to-be integrated components thereby resolving any semantic mismatches as well as a facility to infer behavioral protocols from the observed behavior in the form FSA from traces collected while the interface of the to-be integrated component is in use, and a facility for test case generation to validate the inferred protocol as well as identify any potential parameter constraints.

3.1 Requirements for mismatch adaptor generation framework

3.1.1 Transparency

The adapted component should be unaware of the adaptor between the consumer and the adapted component. This requirement emphasizes the fact that, the derived adaptor is not a wrapper. Given that wrapping a component requires the wrapper to forward all requests to the component, including those that don't require adaptation, aspects of the component that don't require adaptation should be accessible without the use of the derived adaptor.

3.1.2 Interface Inspection

Given that integration is based on the component interface only, the framework must provide the capability to inspect interfaces and extract relevant information pertaining to potential mismatch at least at the signature level as well as subcomponent calls that are not exposed in the interface to enable the engineer to reason about potential blocking problems [5] which may arise after integration.

3.1.3 Protocol Inference

To address behavioral mismatches such as the order in which a component expects its services or operations to be invoked (protocol mismatch), the framework should provide a facility to capture traces while the services of the interface of the component were in use. This will further be analyzed by various algorithms [63, 59, 61] to infer the sequencing constraints of the interface.

3.1.4 Test Case Generation

To identify any parameter constraints as well as validate the inferred protocol, the framework must provide the ability to automatically generate test cases. As previously stated, component test case generation involves three phases namely; component analyzer, path selector and test case generation. During the analysis phase the interface (class file) of the component to be adapted will serve as input and the output could be in the form of a control flow graph for the test case generator phase. It is desirable to implement the Path-Oriented approach for test case generation given that it lends itself to better coverage strong enough to cover the entire path, statements, branches and conditions. The objective of the test case generation phase is to generate test cases that traverse the entire path. This involves identifying the path predicate and then, solving the path predicate in terms of its input variables.

3.1.5 Compatibility

The generated adaptor must be compatible with the consuming component. Component compatibility can be described as the ability for two components to work properly together if connected. Whenever a collaboration between two components P_1 and P_2 can reach the point where $P1$ ($P2$) is in a state where it can send a message m, its mate will be in a state where it can receive that message, and hence there exists some collaboration history in which that message is exchanged at that point. All exchanged messages between both components are understood by each other, and that their communication is deadlock-free [28].

3.1.6 Overgeneralization/over-restrictiveness of observed behavior

Given that an inferred FSA protocol is accurate if it accepts all legal sequences and rejects all illegal sequences, exceptions that occur within the component during execution may be considered a legal sequence or expected behavior of the component and should not be ignored. For example, an exception may occur as a result of a parameter constraint violation but the call to the method that threw the exception is a legal sequence. If an inferred FSA rejects legal sequences in the observed behaviors, it is said to be over-restrictive and if it accepts some illegal sequences, it is said to be over-generalized.

3.1.7 Parameter Constraints Identification

The framework should provide a facility to isolate test cases that resulted in exceptions within the component from test cases that were able to traverse the entire paths in the control flow graph thereby forming the bases for further investigation as related to parameter constrain violation.

3.1.8 Reuse

The framework should be reusable within the context of mismatch adaptor generation such that, programmers should be able to use it without actually having to understand how the framework works.

3.1.9 Extensibility

In the case of a white-box framework, it should be extensible enough to enable users to easily add and/or modify functionality by replacing various components as needed as well as provide hooks so users can customize the behavior of the framework by deriving new classes.

3.2 Assumptions

In this section, we state the assumptions of our framework, and then identify the various components of the framework in detail followed by a simple example that motivates our framework.

- We assume that components are already tested and functions as expected. Testing components in this work doesn't include functional testing but rather is for the purposes of test case generation and validation of the derived protocol.
- All components are black-box components with no access to source code. Integration is at the interface level only.
- Component interfaces have no behavioral specification extensions.

3.3 The MAG Framework

The mismatch adaptor generation framework is composed of a set of collaborating components that make up a reusable design for the generation of adaptors to resolve mismatches including signature mismatch, and transfer protocol mismatch that may occur during system integration. It provides the mechanism and a plug-and-play infrastructure for an engineer/developer to implement different components from those used in this work by partitioning the design into encapsulated components with well-defined roles and responsibilities as well as their interactions. In the remainder of this section, we will describe how to use this framework followed by the detail design of the framework. To motivate our framework, consider the following simple example. The caller component ConsumeMathCalc requires services provided by the callee component CalculateArea as shown in figure 1. Further more, the requires interface of CalculateArea component is constraint to accepts even integers only and the maximum value for each parameter of its requires interface can't exceed the value 998 for length p1 and width p2. Which can be expressed as:

$\forall p_1$: integers • $0 \geq p_1 \leq 999 \wedge p_1 \bmod 2 = 0$

$\forall p_2$: integers • $0 \geq p_2 \leq 999 \wedge p_2 \bmod 2 = 0$

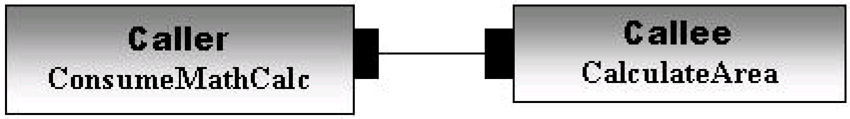


Figure 1: Simple Example

The caller interface is: double calculateArea (double length, double width).

The callee interface is:

```
public interface CalculateAreaInterface
{
    public float calcArea(double brdth, double len);
     public void setArea(double a);
     public double getArea();
}
```

From inspecting both interfaces, the following mismatch patterns were identified.

Parameter type mismatch

The caller (ConsumeMathCalc) is expecting a return value as double while the callee (CalculateArea) returns a float.

Operation name mismatch

The operation (calculateArea) invoked by the caller doesn't match the operation (calcArea) providing the required service in the callee's interface.

Parameter Constraints mismatch

The caller does not know the parameter constraint of the callee interface.

Parameter name mismatch

Parameter names of the caller ConsumeMathCalc doesn't match those of the callee CalculateArea.

Parameter ordering mismatch

The ordering of parameters in the caller's provides interface doesn't match the order in the callee's requires interface.

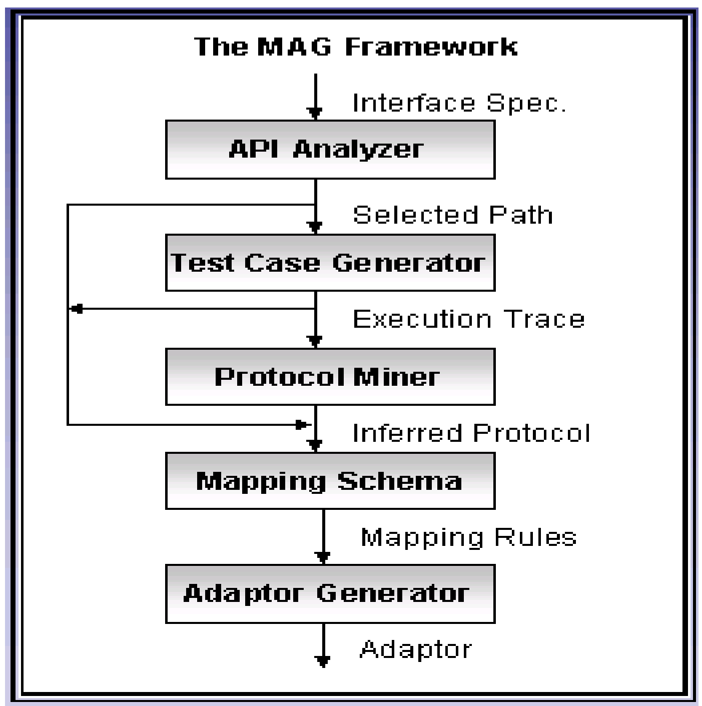


Figure 2: The Mismatch Adaptor Generation (MAG) Framework

3.3.1 Interface Analyzer

The Interface Analyzer component which is made up of two main subcomponents the InterfaceDissector component and the InterfaceMapper component takes as input the interface specification of the component(s) to be analyzed and realizes three artifacts for the framework namely, integration relevant data and mapping rules between the interfaces to be integrated which serves as input to the Adaptor Generator Component, and the selected path to be used by the Test Case Generator component.

3.3.2 The Interface Dissector component

In our case, the InterfaceDissector component is developed using the Javassist byte code manipulation library [69], which provides source-level abstraction of binary code and is intended to give users a convenient possibility to analyze, create, and manipulate binary Java class files. With Javassit, classes are represented by objects, which contain all the symbolic information of the given class methods, fields and byte code instructions, in particular. The InterfaceDissector class takes as input the names of the component interface (.class file) to be analyzed and initiates the creation of the Adaptor Mapping Rules Schema file with relevant information about the interfaces including:

- All operation names
- All input and output parameters for each operation
- All read/write operations for each parameter and
- The Control flow between operations

After analyzing both the ConsumeMathCalc and CalculateArea interface specification .class files using the InterfaceDissector component, the following output is derived and stored in the adaptor mapping rules schema file as shown in figure 3.

To obtain the control flow between operations as well as read/write operations for each parameter in the provider interface specification, the Interface Dissect component uses classes of the javassist library to obtain the flow as shown in figure 4. The path identified serves as input for Test Generator Component.

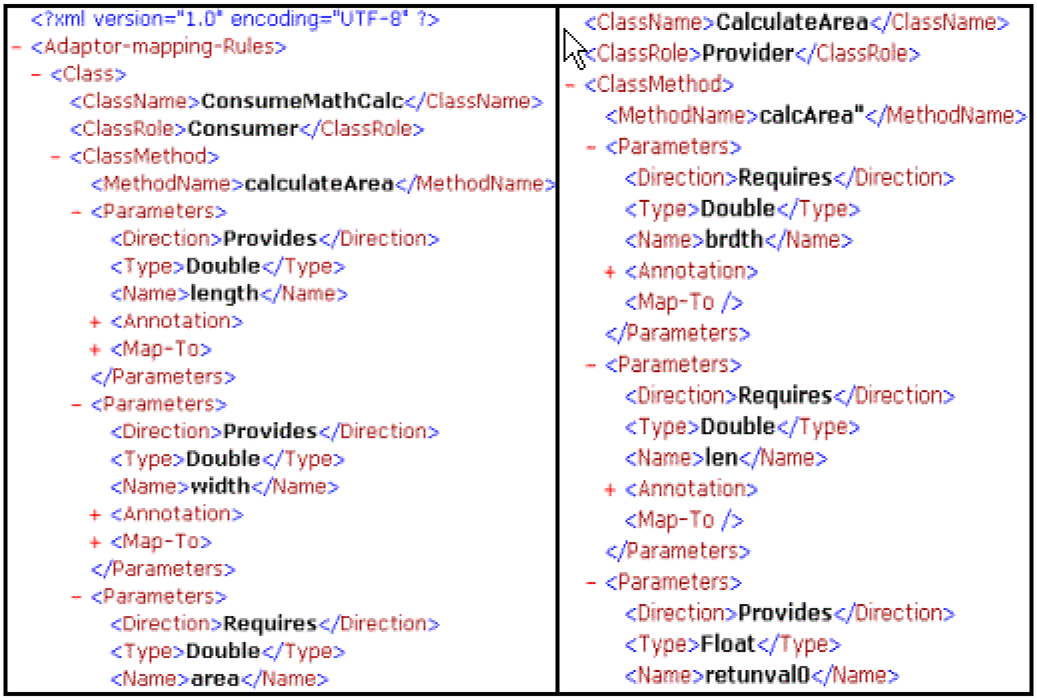


Figure 3: Interface Dissector Example Output for CalculateArea

```
call to CalculateArea.setArea in calcArea(CalculateArea.java:7)
call to CalculateArea.getArea in calcArea(CalculateArea.java:8)
write of CalculateArea.area in setArea(CalculateArea.java:13)
read of CalculateArea.area in getArea(CalculateArea.java:18)
method name = calcArea
                param #0 double
                param #1 double
                return type = float
Field name  = area
decl class = class CalculateArea
```

Figure 4: Interface Dissect Control Flow example for CalculateArea

3.3.3 The Interface Mapper Component

The Interface Mapper component, which uses a graphical user interface as shown in figure 5, provides the ability to map profiles between the component interfaces to be integrated including operations and parameters thereby resolving semantic mismatches between the interfaces. It also provides a facility in the form of a dialogue box for a user to identify various types of mismatches between the interfaces to be integrated as shown in figure 6. When you check on the check box next to a parameter from the consumer class then do the same to its mate on the provider class and click on the map button, the system checks on the parameter types and if there's a mismatch, the mismatch dialogue box will appear with a drop down list for the user to select the type of mismatch from a list of pre-populated mismatches supported by the framework. After which the adaptor mapping rules schema file is updated as shown in figure7.

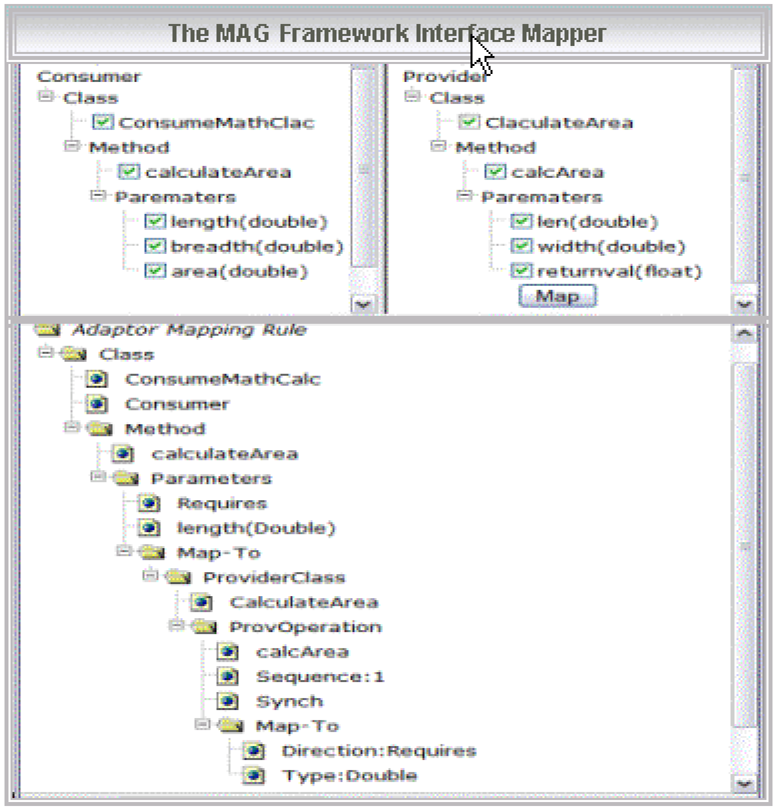


Figure 5: The MAG Framework Interface Mapper

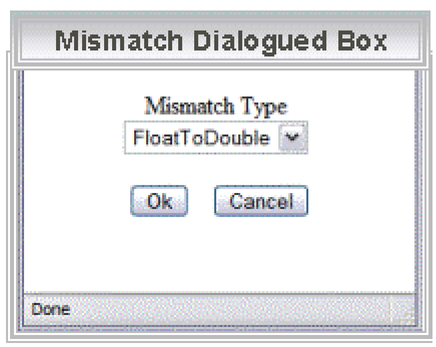


Figure 6: Mismatch Dialogue Box

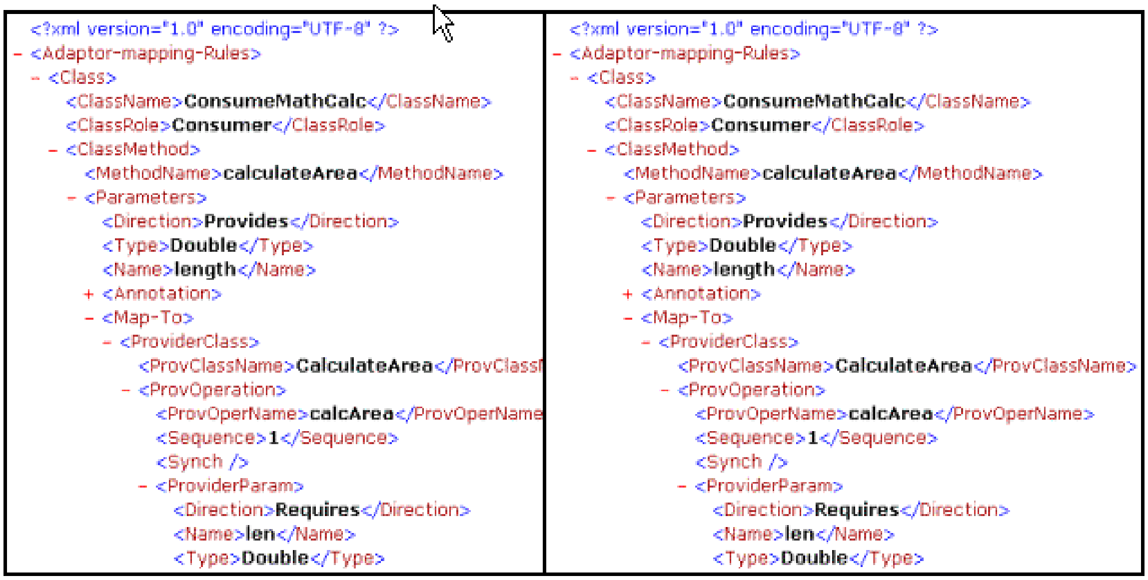


Figure 7: Interface Mapping Example

3.3.4 Test Case Generator

The objectives of the test case generator component after receiving the selected path as input from the Interface Analyzer component are twofold. Firstly, it's for the purpose of generating test cases to identify all possible parameter constraints based on the path received from the Interface Analyzer component. Secondly, it's to collect positive sample of traces from the provider (new) component while in use which will serve as input to the protocol miner component. The recorded events are analyzed, and all call sequences belonging to each service provided by the component are collected and serves as input to the Protocol Miner.

Based on a user's functional experience, execution logs for executions that lead to exceptions within the provider (to-be integrated) component are isolated from other executions for further investigation for potential identification of parameter constraints since these exceptions exhibits the expected behavior of the component. However, exceptions that occur at the interface level of the provider (to-be integrated) component are ignored since we are only interested in the expected behavior of the provider component. All parameters that could not traverse all edges in the path received from the Interface Analyzer component are identified as constraints and represented in the Adaptor Mapping Rules data structure as constraints under the annotation element as shown in figure 8.

```
 <ClassName>CalculateArea</ClassName>
 <ClassRole>Provider</ClassRole>
- <ClassMethod>
    <MethodName>calcArea</MethodName>
  - <Parameters>
      <Direction>Requires</Direction>
      <Type>Double</Type>
      <Name>brdth</Name>
    - <Annotation>
        <Condition>@pre</Condition>
        <Constraints>> 0</Constraints>
        <Condition>@pre</Condition>
        <Constraints>< 999</Constraints>
        <Condition>@pre</Condition>
        <Constraints>mod 2 =</Constraints>
      </Annotation>
```

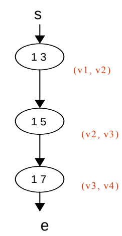
Figure 8: Sample Schema with Annotations

The test case generation problem is such that for a given component *C* and a path u, generate input x $\in I$, so that x traverses u. The goal is to find input values that will

traverse the paths received from the Interface Analyzer. This is achieved by first finding the path predicate for the path and then solve the path predicate in terms of the input variables. With our simplified example, the objective is the find integers that traverses the first edge (v1, v2) such that $T_{(v1,v2)}(p_1)$ = true and those that do not traverse the edge in this case the odd numbers will be considered as parameter constraints for the operation calcArea of CalculateArea. The class file and control flow from the Interface Analyzer component is shown in figure 9 and a sample execution log collected while CalculateArea was in use is shown in figure 10.

```
public class CalculateArea
{
// Constructors
//
 public CalculateArea() { }

// Fields
//
 double area;

 // Methods
 //
 public float calcArea(double p1, double p2) { }

 public void setArea(double p1) { }

 public double getArea() { }
```

Class File for CalculateArea



Control Flow Graph for CalcArea

Figure 9: Class File and Control Flow Graph for CalculateArea

```
5     CalculateArea calArea = new CalculateArea();
2   public  class CalculateArea
5     CalculateArea calArea = new CalculateArea();
6     float area =
calArea.calcArea(Double.parseDouble(args[0]),Double.parseDouble(args[1]));
 args[0] = "7"
 args[1] = "6"
7       setArea(breadth * length);
 breadth = 7.0
 length = 6.0
13      area = a;
 a = 42.0
 area = 0.0
 area = 42.0
8       return (float)getArea();
18        return area;
 area = 42.0
8       return (float)getArea();
6     float area = calArea.calcArea(Double.parseDouble(args[0]),Double.parseDou
ble(args[1]));
```

Figure 10: Sample Execution Log for CalculateArea

3.3.5 Protocol Miner

The objective of the Protocol Miner is to infer the posible interaction protocol for the provider's interface from the traces collected during the test case generation phase represented in the form of a finite state automaton (FSA). A FSA is a tuple P = (S,s_0,F,M,T), where S is the set of states of the protocol, M is the set of messages supported by the component, $T \sqsubseteq S^2 \times M$ is the set of transitions, s_0 is the initial state, and F represents the finite set of final states. A transitions from state s to state s' triggered by the message m is denoted by the triplet (s, s',m). In FSM-based protocol models, states represent the different phases through which a component may go during its interactions with a consumer. Each state is labeled with a logical name. A protocol has one initial state and one or more final states. Transitions are labeled with message names, with the semantics that the exchange of a message causes a state transition to occur.

The recorded events from the Test Case Generation phase are analyzed, and all call sequences belonging to each service provided by the component are collected to form the traces provided as input to the inference engine of kBehaviour [62]. A distinct FSA is generated for each single service. The FSA of a service S represents interactions that can be generated when S is executed. All exceptions that occur at the interface level of the components are exempted because they can generated traces that doesn't represent the expected behavior of the provider component while all exceptions that occurs within the provider component are part of our positive sample since these represents the expected behavior of the component. After the FSA is derived, the sequence element of the adaptor mapping rules schema is updated representing the order in which to invoke operations in the requires interface of the provider component which is analogous to the order of the states in the FSA as represented in figure 11. The Map-to.ProviderClass.ProvOperation.sequence of the Adaptor Mapping Rules data structure represents the sequence in which to exchange messages to the provider's requires interface as shown in figure 7. An example FSA obtained from the execution trace of figure 10 is shown below.

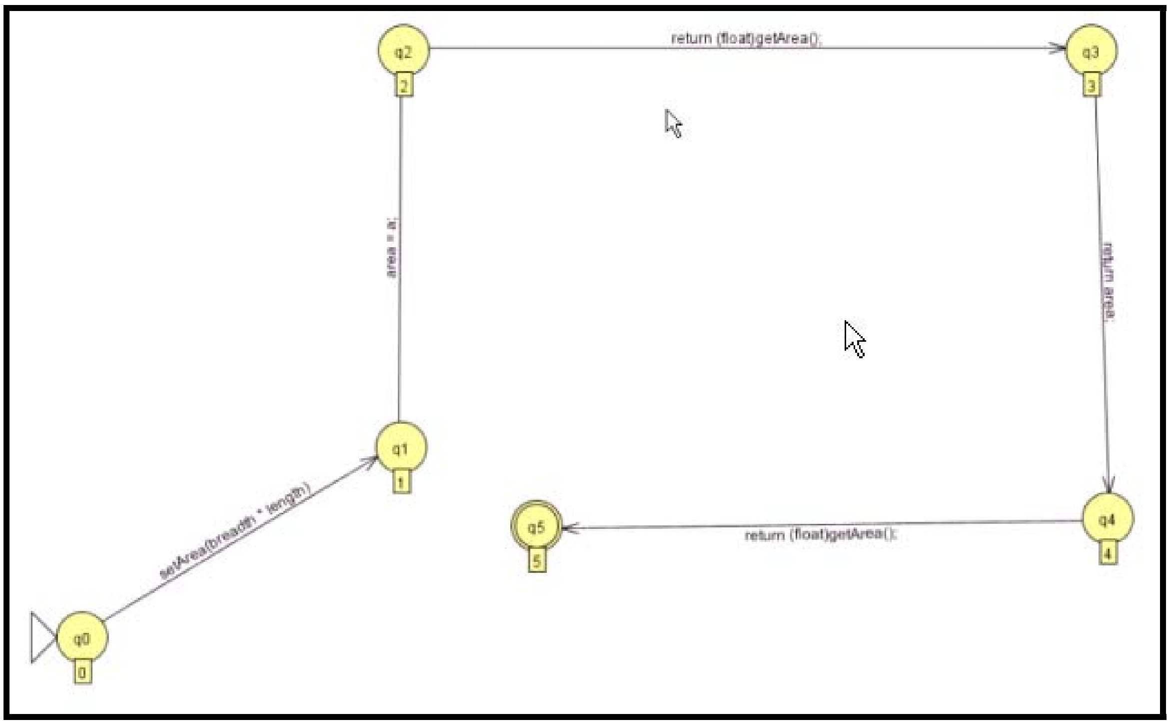


Figure 11: The FSA obtained from the Trace file of Figure 10 using the KBehavior Algorithm

3.3.6 Adaptor Generator

The adaptor generator component uses the mapping rules schema file developed progressively by the various components of the framework as well as the adaptor generation algorithm which is based on the signature mismatch pattern and the protocol mismatch pattern adaptor templates as shown in figures 12 and 13 to generate the adaptor as a java program using Microsoft WordPad. Parameters or operations with annotations from the data structure derived by the test case generator component are translated into JMSAssert special tags (@pre, @post and @inv) using Javadoc comments to specify method pre and post conditions and class invariants. JMSAssert is then run on the Java source code, which results in the automatic creation of certain contract files that contain code in JMScript™, a Java-based scripting language developed by Man Machine Systems. The generated JMScript code actually represents triggers that are called by the assertion runtime to enforce the explicitly stated contractual obligations on the part of the consumer and provider, which in this case is the adaptor.

3.3.6.1 Mismatch Pattern Adaptor Template

The main difference between the protocol and signature mismatch adaptor template is such that, with signature mismatches, there is no need for the framework to use the test case generator and protocol miner abstract components because the mismatches can be resolved by using the mapping rules schema file, adaptor generation algorithm and the data type conversion library if there are any parameter type mismatches. Some of the steps within the adaptor generator engine are optional such as perform transformation after invoking the provider if there are no parameter mismatches between the provides interface of the provider and the requires interface of the consumer. Likewise, the perform transformation before calling the provider may not be executed if there is no mismatch between the consumer provides interface and the provider requires interface.

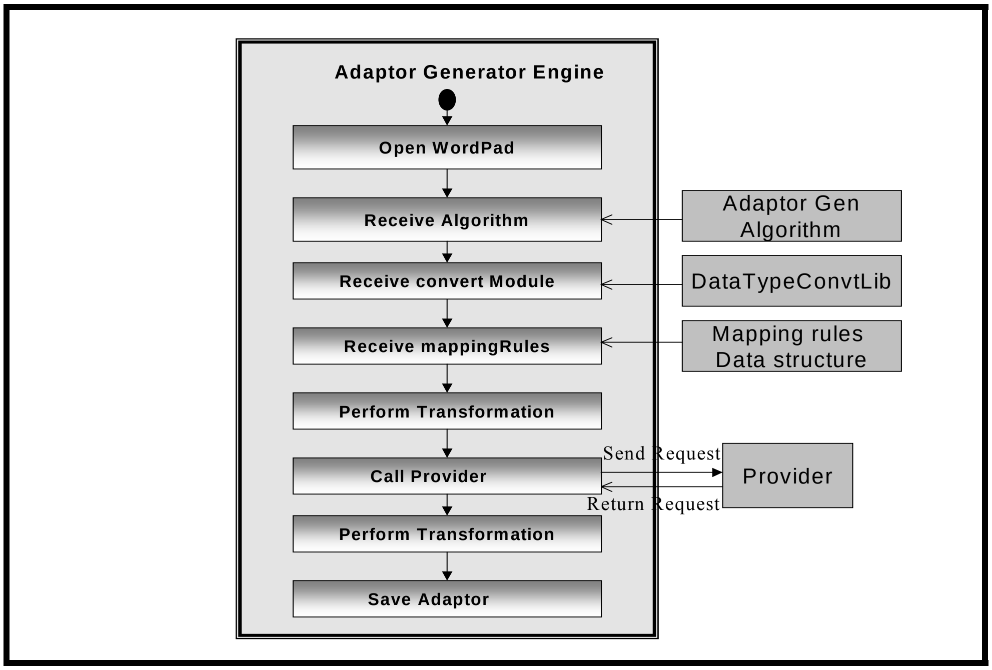


Figure 12: Signature Mismatch Pattern Adaptor Template

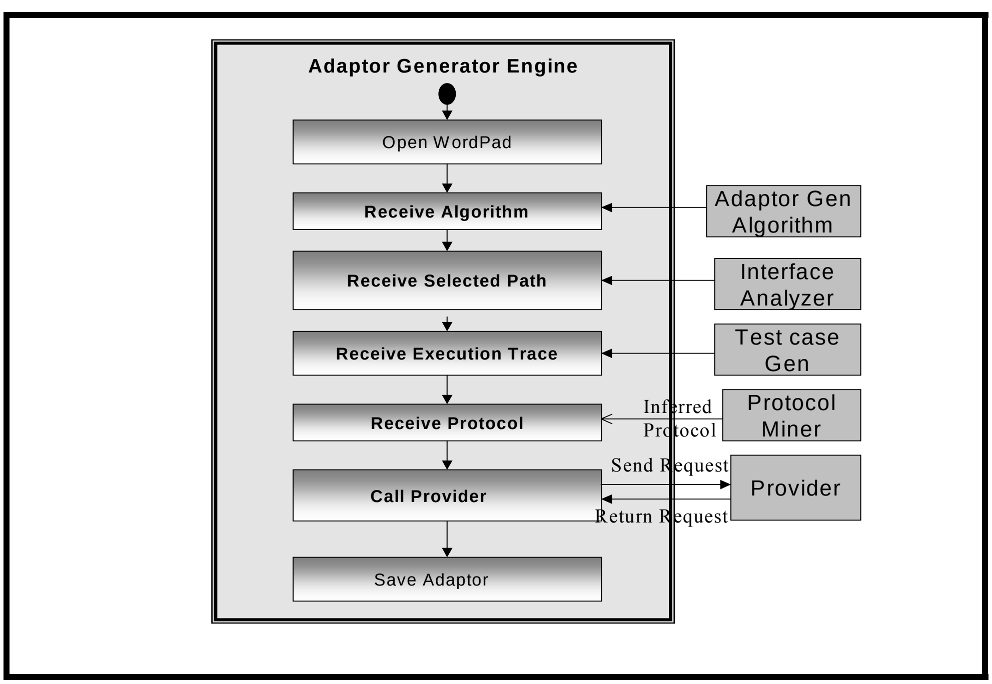


Figure 13: Protocol Mismatch Pattern Adaptor Template

Chapter 4 The Framework Design

This section describes the detail design of the framework in which we identify and describe the inner workings of each of the four abstract components and how the various subcomponents or classes within each abstract component interact with each other to fulfill the objective of each abstract component within the context of design patterns as applicable, as well as how the various abstract components collaborate to fulfill the objective of the framework.

4.1 Interface Analyzer

The Interface Analyzer abstract component as shown in figure 14, which is made up of the Interface Dissect class, Javassist and the Interface Mapper component receives the interface specification (.class file) of both the consumer and provider components to be integrated as input and initiates the creation of the Adaptor Mapping Rules Schema containing the class name, operations within each class, parameters for each operation and their types for both the provider and consumer. This file is then used by the Interface Mapper component which uses a graphical user interface to facilitate mappings between the interfaces to be integrated thereby eliminating any semantic mismatches between the interfaces and then, initiates the updates to the Adaptor Mapping Rules schema with the mapping information.

4.1.1 The InterfaceDissect Class

The interfaceDissect class receives the component interface specification file (.class) and uses classes from the bytecode manipulation library Javassist.jar for the following:

- Identify the .class file (interface specification)
- Identify all methods within the class
- Identify all arguments for each operation and their types
- Identify all returns for each operation
- Identify uses of each operation or fields in the bytecode (.class file) and
- Report on all operation calls in the loaded class including method calls that are not exposed in the original interface specification.

After which, it updates the schema file with the class name, methods within each class, input/output parameters for each method with their types and the method calls within each class is used to derive the control graph of that class.

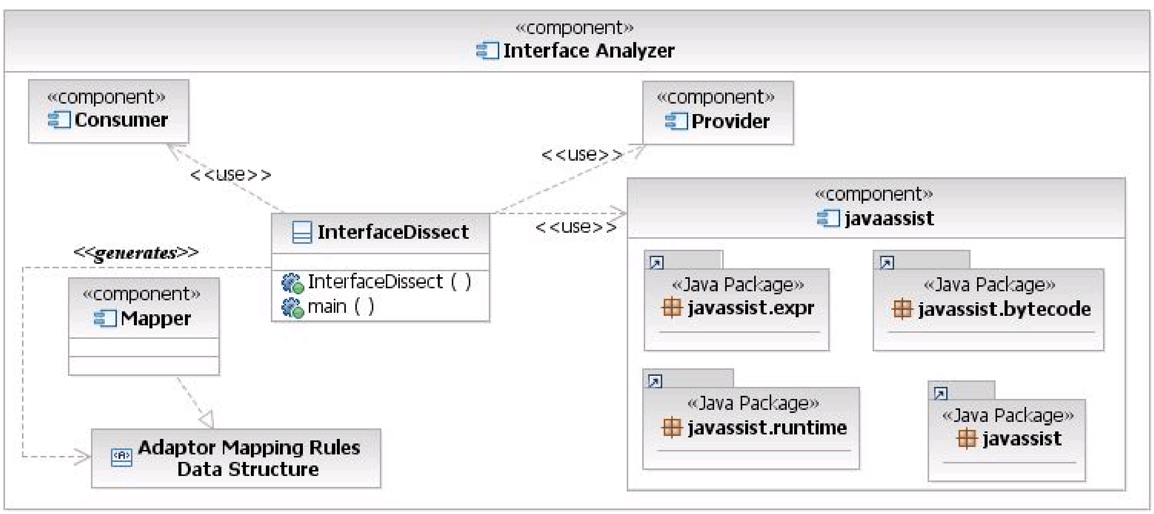


Figure 14: Component Model for the Interface Analyzer

4.1.2 The Javassist Component

Javassist (*Java* programming *assist*ant) is a class library for editing bytecodes in Java, and provides users with convenient possibilities to analyze, create, and manipulate binary

Java class files. Using Javassit, classes are represented by objects, which contain all the symbolic information of a given class, methods, fields and byte code instructions in particular and it allows a user to inspect and manipulate the structure of a class before it is loaded by the ClassLoader. In order to take advantage of classes from the javassist.jar file, the InterfaceDissect class implements the Translator interface and extends the ExprEditor class from the Javassist core API. The following classes from the javassist.jar were used in this framework.

4.1.2.1 Translator API

The InterfaceDissect class implements the Translator API which is an observer of the JVM Loader to attach an instance of the interface specification (.class file) to a `Loader` object so that it can translate a class file when the class file is loaded into the JVM.

4.1.2.2 Javassist.ClassPool

Javassist.ClassPool extends the java.lang.Object and is a container of the `CtClass` object. It creates a `CtClass` object, which represents an instance of class files. The created object is returned to the caller. Methods of the CtClass object used for this framework includes:

getName( ) Obtains the fully-qualified name of the class.

getMethods( ) Returns an array containing `CtMethod` objects representing all the non-private methods of the class.

getFields() Returns an array containing `CtField` objects representing all the non-private fields of the class.

4.1.2.3 Javassist.CtBehavior

A public abstract class, which extends CtMember (an instance of `CtMember` represents a field, a constructor, or a method). `CtBehavior` represents a method, a constructor, or a static constructor (class initializer). It is the abstract super class of `CtMethod` and `CtConstructor`. The following operations of the CtMember class were used for this framework.

getName( ) Obtains the name of the member.

getSignature( ) Returns the character string representing the signature of the member.

getParameterTypes( ) Obtains parameter types of this method/constructor.

where( ) Returns the method or constructor containing the method-call expression represented by this object.

4.1.2.4 Javassist.expr

This package contains the classes for modifying a method body.

4.1.2.4.1 javassist.expr.FieldAccess

Expression for accessing a field. The following operations of this class were used.

getClassName( ) Returns the name of the class in which the field is declared.

`getField`( ) Returns the field accessed by this expression.

getFieldName() Returns the name of the field.

getFileName() Returns the source file containing the field access.

getSignature() Returns the signature of the field type.

isReader()Returns true if the field is read.

IsWriter() Returns true if the field is written in.

4.1.2.4.2 javassist.expr.MethodCall

For identifying calls to methods.

`getClassName``()` Returns the class name, which the method is called on.

`getLineNumber``()` Returns the line number of the source line containing the method call.

`getMethod``()` Returns the called method.

`getMethodName``()` Returns the name of the called method.

`getSignature``()` Returns the method signature (the parameter types and the return type).

4.1.2.5 javassist.expr.ExprEditor

A subclass of this class is defined by the user to customize how to modify a method body. The overall architecture is similar to the strategy pattern. If `instrument()` is called in `CtMethod`, the method body is scanned from the beginning to the end. Whenever an expression, such as a method call and a new expression (object creation), is found, `edit()` is called in `ExprEdit`. `edit()` can inspect and modify the given expression.

4.1.3 The Mapper

The mapper component uses a graphical user interface to map the relationships between the consumer and provider using the information already captured in the mapping rules schema file. This also provide the ability to resolve all possible semantic mismatches between operations and parameter names of the consumer and provider interfaces as well as the ability to identify parameter type mismatches between the interfaces. All mappings between the interfaces and mismatches identified by the user are stored in the adaptor mapping rules data structure.

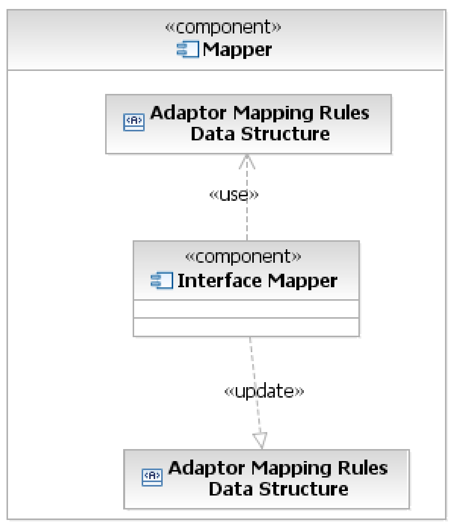


Figure 15: The Mapper Component

4.2 Test Case Generator

4.2.1 Background

A component *C* can be considered as a function, $C: I \rightarrow O$, where *I* is the set of all possible inputs and *O* the set of all possible outputs. Formally *I* is the set of all vectors x = (d1; d2;…..; dn) such that $d_i \in D_{xi}$ where D_{xi} is the domain of input variable xi. An input variable x of *C* is a variable that either appears as an input parameter of *C* or in an input statement of *C*, e.g. read(x). The execution of *C* for a certain input x is denoted by *C* (x).

A control flowgraph of an operation of *C* is a directed graph G = (V;E; s; e) consisting of a set of nodes V and a set of edges E $\subseteq V^2$ connecting the nodes with one entry node s and one exit node e. Each node is defined as a basic block, which is an uninterrupted consecutive sequence of instructions, where the flow of control enters in the beginning and leaves at the end without halt or possibility of branching except at the end. If any statement of the block is executed, then the whole block is executed. An edge between two nodes n and m corresponds to a possible transfer from n to m. All edges are labeled with a condition or a branch predicate. In order to traverse the edge the condition of the edge must be true. If a node has more than one outgoing edge it is referred to as a condition and the edges as branches.

A path P in G is defined as a tuple $(v_1, \ldots, v_m) \in V^m$ of nodes with $(v_j, v_{j+1}) \in E$ for $1 \leq j < m$, v_1 and v_m being the initial node s and final node e of G, respectively.

Whenever the execution of *C* (x) traverses a path p, we say that x traverses p. A path is absolutely feasible if there exists an input x ∈ *I* that traverses the path, otherwise the path is absolutely infeasible. A path that begins with the entry node and ends with the exit node is called a complete path otherwise an incomplete path or a path segment.

Let p = (p1; p2;…..; p_n) and w = (w1; w2;…..; w_n) be two paths, then pw = (p1; p2;…..; p_n, w1; w2;…..; w_n) denotes the concatenation of p and w. Let first(p) denote the first node p1 of path p and let last(p) denote the last node p_n of p. If (last(p); first(w)) ∈ E connect, then they are said to be connecting paths., where E is the set of edges. If p and w are two specific paths, pw is an unspecific path if p and w do not connect.

Let O be the operation under test and *C* the component providing this operation. Furthermore, let $a_1, \ldots, a_l$ designates the arguments of *O* and attributes of *C*, and D_{ai} with $1 \leq i \leq l$ be the set of all values which can be assigned to a_i.

1) The domain D of *C* is defined as the cross product $D_{a1} \times \cdots \times D_{al}$,
2) An input of *C* as an element x in D and
3) A test case x_O as an input, which satisfies a testing-relevant objective O.

The Test Case Generator abstract component is made up of the Tracer, which uses the BINTEST algorithm [64] to generate test cases for the component to be integrated. Traces generated by test cases that traverse all the branches in the path supplied by the Interface Analyzer component are identified as Expected Component Behavior Interaction Trace and isolated from test cases that don't which are further analyzed by the Constraint Identifier component for possible constraint for that branch were the exception occurred. If the exception occurred in a branch within the path that is not exposed in the components interface in the case of a subcomponent, the identified constraints are then identified as constraints to the main operation that is exposed at the interface. The high level component model for the Test Case Generator component is illustrated in figure 16.

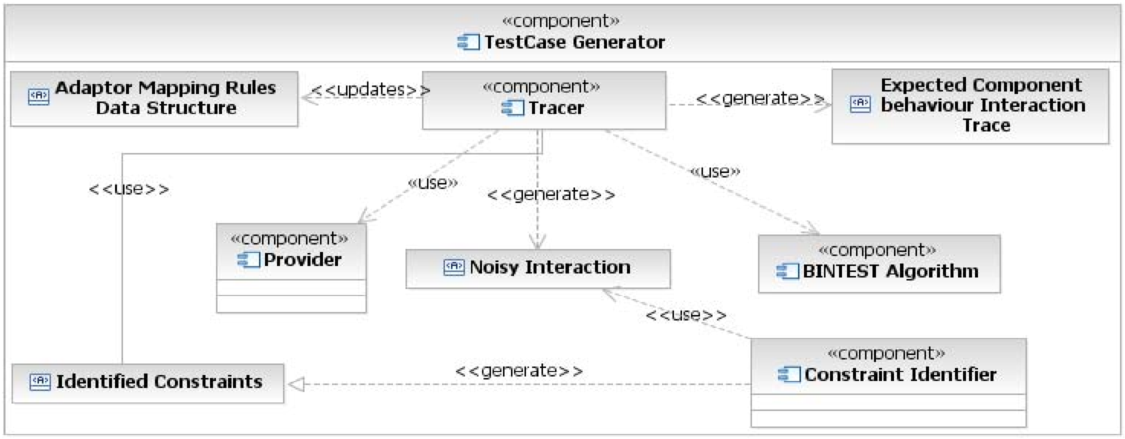


Figure 16: Component Model for Test Case Generator

4.2.2 The Tracer

Given that the BINTEST algorithm is based on a binary search strategy, the Tracer Class acting as a proxy to the consumer initiates the search for test cases by determining the median element of the argument for the provider's operation under test and invokes the provider initially. If the value passed by the Tracer class traverses all the branches in the identified path without throwing an exception, the trace is annotated. After the tracer class determines the first value, subsequent values are derived based on the BINTEST algorithm. Based on the value determined by the algorithm, the component is executed again by the Tracer class and if there's an exception within the component, the trace for that execution is annotated as such and the Tracer class stores the value that caused the exception. This process is repeated until we run out of values in the list for that argument.

4.2.3 The BINTEST Algorithm

The BINTEST algorithm [64] is such that the path to be covered is not considered as a whole, but rather divided into its basic constituents, which are then considered in the sequence respecting their order on the path. A test case x_P is approached by iteratively generating inputs successively covering each of the edges of P. A particular edge is only traversed by a subset of all possible inputs. The algorithm receives the initial input x_0 from the Tracer class and evaluates the traversal condition of the first edge (v_1, v_2) of P with respect to this value. Traversal condition $T_{(v1,v2)}$ is generally not met for all inputs in D but for values in a certain subset $D_1 \sqsubseteq D$ and the initial input is therefore changed to a value $x_1 \in D_1$ ensuring the traversal of edge (v1, v2), i.e. $T_{(v1,v2)}(x_1) = true$. Input values that can't traverse a particular edge are stored to be evaluated for potential parameter constraints for that operation. In the next step, the traversal condition of the second edge (v2, v3) on P is evaluated given that arguments of *O* and attributes of *C* are set to the values specified by x_1. Again, $T_{(v2,v3)}$ is generally only satisfied by a subset $D_2 \sqsubseteq D_1$ and x_1 needs to be modified if it does not lie in D_2. Hence, $D2 \sqsubseteq D1 \sqsubseteq D$. This procedure is continued for all edges on P until either an input is found fulfilling all traversal conditions or a contradiction among these conditions is detected. In such a case, the traversal conditions cannot be fulfilled entirely and the path is infeasible. For more details about the BINTEST Algorithm, please see [64].

4.2.4 Constraint Identifier

This class is used to identify constraints for the operation in which an exception occurred during execution of the component to be integrated by the Tracer class while generating test cases. Parameter constraints for a particular operation are represented with special tags or markers such as "@pre" followed by a boolean expression and @post followed by a boolean condition for the pre and post conditions respectively. The Tracer class then updates the annotation element associated with the operation of the Adaptor Mapping Rules schema file with the identified constraints. These constraints (@Pre and @post conditions) will then be defined within Javadoc comments preceding the respective operations during the adaptor generation phase.

4.3 Protocol Miner

This component uses the expected component interaction trace log generated by the Test Case Generator as input to the Kbehaviour algorithm to infer the interaction protocol in the form of a FSA. After which the AdaptorStructureUpd class takes the FSA as input and updates the mapping rules structure with the sequence in which the operations in the provider component is expected to be executed.

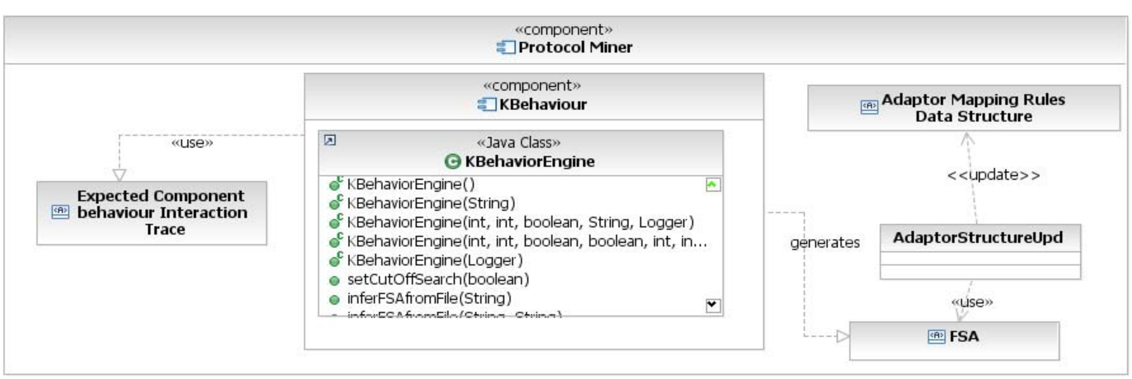


Figure 17: Component Model for the Protocol Miner

4.4 Adaptor Generator

The adaptor generator component model as shown in figure 18 uses the adaptor mapping rules data structure as input to generate an adaptor to resolve mismatches based on the mismatch pattern adaptor template. It has two components, the adaptor generator engine and the DataConvLib and uses three artifacts; the adaptor generation algorithm, signature mismatch pattern adaptor template and the protocol mismatch pattern adaptor template. The generator engine which is designed based on the strategy pattern, uses the adaptor mapping rules schema as input and depending on the mismatch pattern, signature or protocol, it then uses the adaptor generation algorithm, and delegates conversion of any parameter type mismatch to the DataTypeConvLib to resolve and then generates an adaptor with delegation by delegating functionality from the provider class as oppose to inheritance which implements the interface of the provider with the adaptor as shown in figure 19 and 20 respectively. The annotations from the adaptor mapping rules schema representing pre and post conditions for operations within a class, will be translated as Javadoc comments during the adaptor generation process. The next step is to run jmsassert on the Java source code, which results in the automatic creation of certain contract files that contain code in JMScript™, a Java-based scripting language developed by Man Machine Systems. The generated JMScript code actually represents triggers that are called by the assertion runtime to enforce the explicitly stated contractual obligations on the part of the provider and consumer.

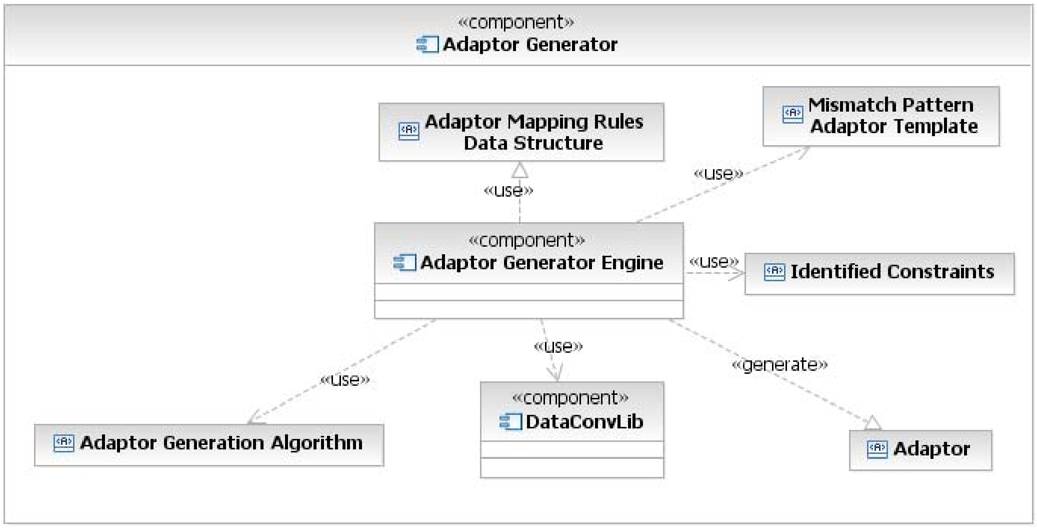


Figure 18: The Adaptor Generator Component Model

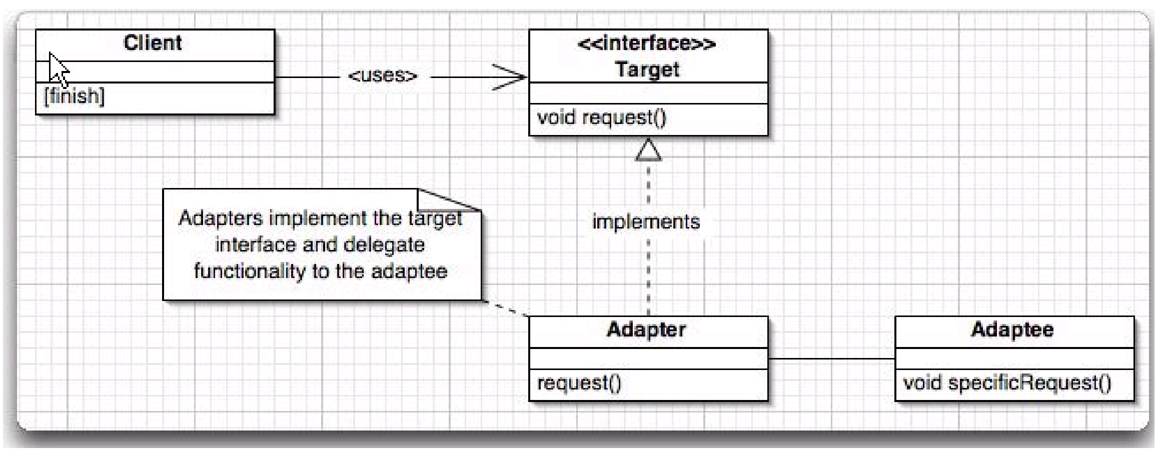


Figure 19: Adaptor with Delegation

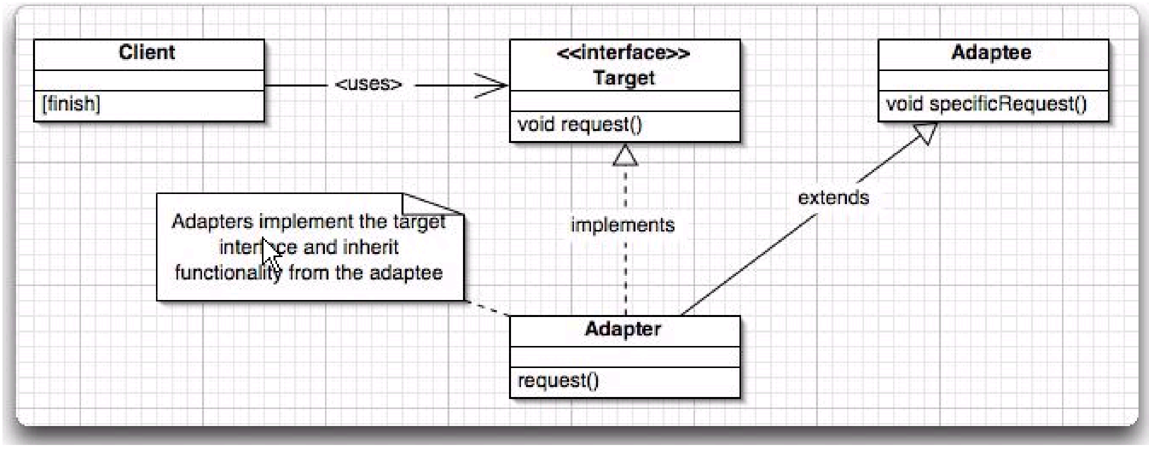


Figure 20: Adaptor with Inheritance

4.4.1 The Adaptor Generation Algorithm

For each parameter for the consumer and provider interface, define a member variable with default access method of *public*. Append *"l"* to the parameter name and get the type from the parameter type element *double.* If a method returns a value but there is no explicit name such as *float calcArea (double brdth, double len)*, the adaptor generator engine uses the default name of *returnval* with the last letter denoting the occurrence of any such default name. In this case the first default name will be *returnval0* with *0* denoting the first occurrence.

Public double lreturnval0;

Public double lLen;

Public double lBrdth;

If the parameter has an annotation element, generate a Javadoc comment.

* @pre len > 0

For each parameter create a setter method with a default access method *public.* Since setter methods don't have outputs, we use *void.* Append *"set"* to the member variable just created *setreturnval0*. Use the parameter type from the parameter type element *double.* Assign the parameter to member variable just created *lreturnval0 = returnval0.*

Public void setreturnval0 (double returnval0)

{

l returnval0 = returnval0;

}

For each parameter, create a getter method with default access method for all objects as *public.* Since getter methods have outputs, get the parameter type *double and a*ppend "get" to member variable just created g*etreturnval0( ).* Since getter methods return some value use the Java keyword *{return.* The value to return is the member variable defined earlier *lreturnval0;}*

Public double getreturnval0 ( )

{

Return lreturnval0;

}

For each parameter if there's a map-to element, create an adaptor method. Generate the method using the member variable name appended to *"operation" returnval0Operation* with default access as *Public.* If the map-to provider operation accepts input, get the name(s) and data type from the map-to parameter name and type *double len double brdth.* Append *rtn* to the consumer parameter name *area*.

public double returnval0Operation(double len, double brdth)

{

double rtnarea = 0;

try

{

A constructor always has the form provider class name *CalculateArea* the instance name is always "Adaptor" plus the object's name *adaptorcalculateArea* Keyword = *new* provider class name *CalculateArea ( );*

CalculateArea adaptorcalculatearea = new CalculateArea();

float larea = adaptorcalculatearea.calcArea(len,brdth);

If there's a parameter mismatch, which in our simple example there is between float and double, generates a constructor for the data type conversion library as

DataTypeConvLib cnvLib = new DataTypeConvLib();

and then assign the value returned from invoking the appropriate module from the *DataTypeConvLib rtnarea.*

rtnarea = cnvLib.Convert2Double(larea);

}

catch (Exception e){

e.printStackTrace();

}

return rtnarea;

}

Input: The Adaptor Mapping Rules Schema file; Mismatch Pattern Adaptor Template, DataTypeConvLib
Output: The Adaptor
Begin
Open File

1) For all Consumer & Provider Interfaces Do:
   a) Identify all required and provides parameter names and their types
   b) For each parameter define a member variable
      i. Default access method for all parameters is
      ii. Append "l" to the parameter name
   c) Generate Setters Methods for each parameters
      i. Default access method for all parameter is
      ii. Since setter methods don't have outputs
      iii. Append "set" to parameter name
      iv. Assign parameter to member variable
   d) Generate Getter Methods for each parameter
      i. Default access method for all object is
      ii. Append "get" to object name
      iii. Since getter methods return some value use the Java keyword *{ return*
      iv. The value to return is the member variable defined earlier *{return lmemberVariable;}*

2) For each parameter from the Provider Interface Do:
   a) Identify those with annotations
   b) Generate constraints (Pre-Post conditions)
3) For each parameter of the Consumer's Provides Interface Do:
   a) Check for the existence of a Parameter Mismatch tag
   b) If Mismatch tag exist
      i. Determine type if Mismatch
      ii. Perform conversion using the DataTypeConvLib

4) Fo each Parameter of the Consumer Requires Class with Map-to element:
    a) Get the Map-to Class
    b) For each Map-to Class
        a. Get all the map-to operations
            i. For each operation
                1. Get all the map-to parameters
    b) If Protocol Mismatch tag exist
        For each Map-class.method
            Get the class name
            Get the method name
            Get the method sequence
            Get the corresponding parameter name & type
        End for
    End If
End For

6) Generate the Constructor Statement
6) Generate Call Statement to Provider Requires Operation
7) For each parameter of the Consumer's Requires Interface Do:
    Check for the existence of a Parameter Mismatch tag
        If Mismatch tag exist
            i. Determine type if Mismatch
            ii. Perform conversion using the DataTypeConvLib
            iii. Assign Return Value to the Consumer's Requires Interface.
        End If
    End For
8) Save File.

Figure 21: The Adaptor Generation Algorithm

```
import component.convertor.lib.DataTypeConvLib;
public class ContractExample{

 public ContractExample ()
 {

 }
public double lreturnval0;
/**
* sets the value of the returnval0 property
*
* @pre len > 0     " Length must be greater than Zeroes"
* @pre len < 999   "Length Must be less than 999"
* @param double
*/

 public  void setreturnval0 ( double returnval0)
 {
lreturnval0=returnval0;

 }
/**
* Gets the value of the returnval0property
*
* @post returnval0 != 0   " returnval0 cannot be Zeros"
* @post returnval0 < 999   "returnval0 must be Less than 999"
* @return double
*/
 public double getreturnval0 ()
 {

 return lreturnval0;
 }
```

```
 /**
* sets the value of the returnval0 property
*
* @pre len > 0  " Length must be greater than Zeros"
* @pre len < 999 "Length Must be less than 999"
* @return
*/
public  double returnval0Operation(double len, double br)
{
 double rtnarea=0;
try
 {
CalculateArea adaptorcalculatearea = new CalculateArea();
float larea = adaptorcalculatearea.calcArea(len, br);
DataTypeConvLib cnvLib = new DataTypeConvLib();
 rtnarea = cnvLib.Convert2Double(larea);
  }
 catch (Exception e){
 e.printStackTrace();

 }
return rtnarea;

 }
 public double llength;
/**
* sets the value of the length property
*
* @pre len > 0  " Length must be greater than Zeros"
* @pre len < 999 "Length Must be less than 999"
* @param double
*/

 public synchronized void setlength ( double length)
 {
llength=length;

 }
```

```
/**
* Gets the value of the lengthproperty
*
* @return double
*/

 public double getlength ()
 {

 return llength;
 }
public double lbreadth;
/**
* sets the value of the breadth property
*
* @param double
*/

 public  void setbreadth ( double breadth)
 {
lbreadth=breadth;

 }
/**
* Gets the value of the breadthproperty
*
* @return double
*/

 public double getbreadth ()
 {

 return lbreadth;
 }
public float lfloatval;
/**
* sets the value of the floatval property
*
* @param float
*/

 public  void setfloatval ( float floatval)
 {
lfloatval=floatval;
 }
/**
* Gets the value of the floatvalproperty
*
* @return float
*/
 public float getfloatval ()
 {

 return lfloatval;
 }
}
```

Figure 22: Generated Adaptor for Simple Example

Chapter 5 Case Study

5.1 The Bidder/Auctioneer Interaction

In this chapter we use a complex example to show how our framework can be applied to generate an adaptor to resolve various mismatches between a bidder and auctioneer collaboration example which first appeared in [28]. Based on the interface specification of the bidder and auctioneer components in [28], a replica of both components has been developed in Java to demonstrate the utility of our framework.

The collaboration between the bidder and auctioneer is such that, when the auction begins or when a bidder attaches to the auction in progress, the auctioneer sends a newItem message to the bidder containing information about the current item being auctioned along with an id for the bidder to use on subsequent bids for that item. The auctioneer then enters state B, representing the fact that this bidder is not the current high bidder for the item. In this state the auctioneer will inform the bidder about new high bids for the item by way of an update message, or it will send the bidder an itemsold message, indicating that the auction for this item is over. When in state B the auctioneer protocol may also receive a bid message from the bidder, in which case it enters state C and evaluates the bid. The auctioneer protocol then responds to the bidder protocol that the bid is either rejected or accepted. In the former case, the auctioneer moves back to state B; in the latter case it moves to state D, representing the fact that this bidder now owns the high bid for the item. From this state the bidder is not expected to bid, but the auctioneer can either inform the bidder of subsequent higher bids for the item received from competing bidders by way of update messages, or that the auction is over and that this bidder has bought the item. Each update message is expected to be acknowledged by the bidder with an updateAck message. The interaction model for the auctioneer component can be illustrated as shown in figure 23 and the interface specification for both the bidder and auctioneer is shown in figure 24 and figure 25 respectively.

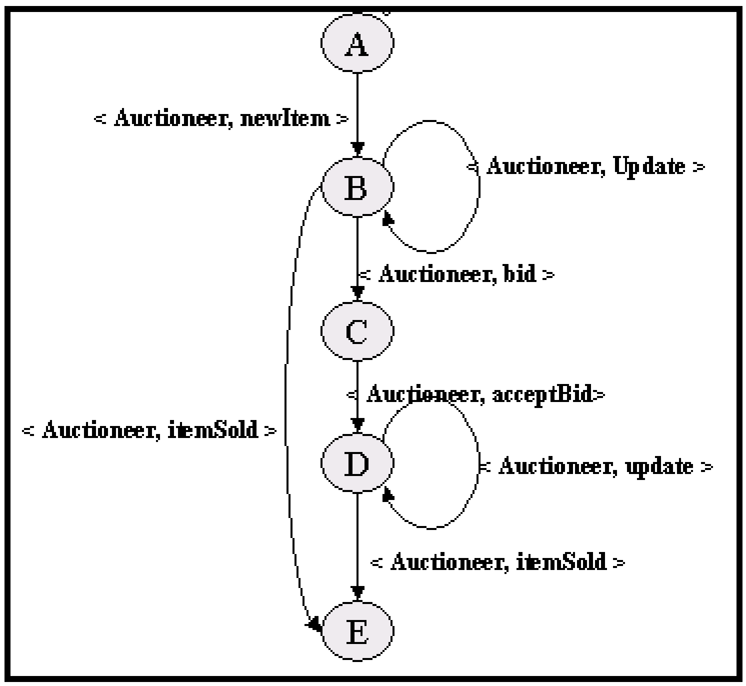


Figure 23: Interaction Model for Auctioneer

5.1.1 Interface Analysis

After analyzing the interfaces for both the bidder and auctioneer as shown in figure 25 and figure 24 respectively, the following mismatch patterns were identified.

5.1.2 Mismatch Patterns

5.1.2.1 Extra message mismatch pattern

The consumer's requires interface has some extra messages that the provider's provides interface does not expect to send. There are two instances of this mismatch pattern in this example. First, the bidder.auctionBegin and bidder.canBid operations of the bidder's requires interface maps to auctioneer.newItem operation of the auctioneer's provides interface. In order for this message exchange to occur, the adaptor most receive the newItem message from the auctioneer and then call the auctionBegin and canBid operations of the bidder interface. The second example is the case were the Bidder.newBid and Bidder.requestToBib operations of the bidder's provides interface maps to auctioneer.bid of the auctioneers requires interface.

5.1.2.2 Missing message mismatch pattern

The bidder's requires interface has the bidNotOk operation were as the auctioneer does not provide such a message. The auctioneer.rejectBid operation of the auctioneer provides interface maps to the bidder.cannotBid operation of the bidder's requires interface.

5.1.2.3 Message exchange sequence mismatch pattern

Even though the auctioneer component is functionally compatible with the bidder component, their interfaces are not. The bidder interface has more messages than the auctioneer interface and the order in which to exchange messages between the two interfaces is totally unknown to the bidder. In this case, the framework will have to infer the interaction protocol to enable both components to interact with one another.

5.1.2.4 Parameter name mismatch pattern

The parameter mismatch between the bidder and auctioneer interfaces is such that:

Bidder.autionBegin(auctionInfo) maps to Auctioneer.newItem(itemDescr)

Bidder.newBid(myBid) maps to Auctioneer.bid(amount)

Bidder.requestToBid(bidItem) maps to Auctioneer.bid(itemBiddingOn)

Bidder.cannotBid.(why) maps to Auctioneer.rejectBid(reason)

Bidder.newHighBid(price) maps to Auctioneer.update(highBid)

5.1.2.5 Operation name mismatch pattern

The following operation name mismatches were identified:

Bidder.newHighBid maps to Auctioneer.update

Bidder.acceptBid maps to Auctioneer.bidOk

Bidder.rejectBid maps to Auctioneer.cannotBid

Bidder.gave1 maps to Auctioneer.itemSOld

5.1.2.6 Extra parameters mismatch pattern

The bidder.requestToBid operation contains a name parameter that the auctioneer doesn't need to provide in order to fulfill the services requested. In this case the generated adaptor will drop this parameter before calling the auctioneer.

5.1.2.7 Missing parameters mismatch pattern

The bidder's provides interface provives bidder.newBid with a missing parameter.bidItem which is a parameter in the bidder.requestToBid message. In this situation, before calling auction.bid , the bidder most first provide to the adaptor both messages bidder.requestToBid and bidder.newBid for the adaptor to call autioneer.bid.

```
public interface Auctioneer
{
public void receiveMessage_bid(String bidderId, String  itemBiddingOn, int amount);
public boolean receiveMessage_updateAck();
public boolean sendMessage_itemSold();
public boolean sendMessage_acceptBid();
public String sendMessage_rejectBid(String reason);
public void sendMessage_newItem(String itemDescr, String bidderId);
public void sendMessage_update(int highBid);
}
```

Figure 24: Interface Specification for Auctioneer

```
public interface CollaborationBidder
{
 private boolean receiveMessage_Gave1();
 private boolean receiveMessage_bidNotOk();
 private boolean receiveMessage_bidOk();
 private void receiveMessage_newHighBid(int price);
 private void receiveMessage_cannotBid(String why);
 private void receiveMessage_canBid(int bidId);
 private void receiveMessage_AuctionBegin(String auctionInfo);
 private boolean sendMessage_highBidAck();
 private void sendMessage_newBid(String bidId, int myBid);
 private void sendMessage_requestToBid(String name, String bidItem);
 }
```

Figure 25: Interface Specification for Bidder

5.2 Interface Analyzer

After analyzing both interface specification files for the bidder and auctioneer using the InterfaceDissect class, the interface analyzer component initiates the creation of the Adaptor Mapping Rules Schema file. After which, the mapper component is used to map the various operations and parameters of both interfaces. A portion of the Adaptor Mapping Rules Schema showing the mappings between the bidder.requestToBid and bidder.newBid mapped to autioneer.bid is shown in figure 26. You will notice that the operation bidder.requestToBid is not mapped to any operation of the auctioneer interface whereas, the others are. The Interface Mapper GUI also provides the facility to resolve semantic mismatches such as differences in parameter types or names as well as parameter ordering. The challenges with the bidder-auctioneer example lies mostly with the mismatch patterns identified above as oppose to parameter constraint mismatch. Therefore the Test Case Generation step of the framework will focus mostly on generating interaction logs of the auctioneer component while in use thereby providing input to the protocol miner component.

Figure 26: Example - Bidder Auctioneer Interface Mapping

5.3 Test Case Generation

For the purposes of observing the behavior of the auctioneer component while in use, the Tracer class acting as a proxy to the bidder, initiates the collaboration after receiving a newItem from the auctioneer. The Tracer then invokes bidder.newBid and bidder.requestToBid and then passes the request to auctioneer.bid. The auctioneer then receives the bid and returns a reject message back to the Tracer class which then sends another bid. For the sake of this example, the auctioneer rejects bids from the Tracer four times and then accepts on the fifth try by sending a acceptBid message followed by an itemSold message. Even though multiple bidders can interact with the auctioneer, for purposes of this presentation we only use one bidder to demonstrate the utility of the framework. Part of the conversation log between the Tracer and the auctioneer is depicted in figure 27. In this example, given that the auctioneer accepts any value passed by the

Tracer and will either accept or reject, there was no need to exhaust all possible integer values with the range. The test case generation algorithm was used to derive the other four bids before stopping execution.

```
RunAuctionBidder.main(), line=5 bci=0
5    TracerExample cntrct = new TracerExample.();
TracerExample..<init>(), line=4 bci=0
TracerExample.<init>(), line=6 bci=4
RunAuctionBidder.main(), line=5 bci=7
5    TracerExample cntrct = new TracerExample();
RunAuctionBidder.main(), line=6 bci=8
6    cntrct.Begin_Auction_processOperation();
TracerExample.Begin_Auction_processOperation(),
 line=836 bci=0
836    Auct_snd_newItemOperation();
TracerExample.Begin_Auction_processOperation(),
 line=837 bci=4
837    Bid_rcv_auctionbeginOperation();
TracerExample.Begin_Auction_processOperation(),
 line=838 bci=8
838    Bid_snd_requestToBidOperation();
TracerExample.Begin_Auction_processOperation(),
 line=839 bci=12
839    Bid_snd_newBidOperation();
TracerExample.Begin_Auction_processOperation(),
 line=840 bci=16
840    Auct_rcv_bidOperation();
TracerExample.Begin_Auction_processOperation(),
 line=841 bci=20
841    Auct_snd_updateOperation();
TracerExample.Begin_Auction_processOperation(),
 line=842 bci=24
842    Bid_rcv_newHighBidOperation();
TracerExample.Begin_Auction_processOperation(),
 line=848 bci=28
848    }
TracerExample.Begin_Auction_processOperation(),
 line=849 bci=36
849    }
RunAuctionBidder.main(), line=8 bci=12
```

Figure 27: Conversation Log between the Tracer and Auctioneer

5.3 The Protocol Miner

After the observed behavior between the Tracer and the auctioneer was analyzed and all call sequences to the auctioneer recorded as input to the Kbehavior algorithm, the generated FSA was derived as identified below. The order of operations which represents states of the FSA are translated to the sequence in which to invoke operations of the auctioneer interfaces and are stored in the Adaptor Mapping Rules schema under the sequence element which is a sub element of the ProvOperName element.

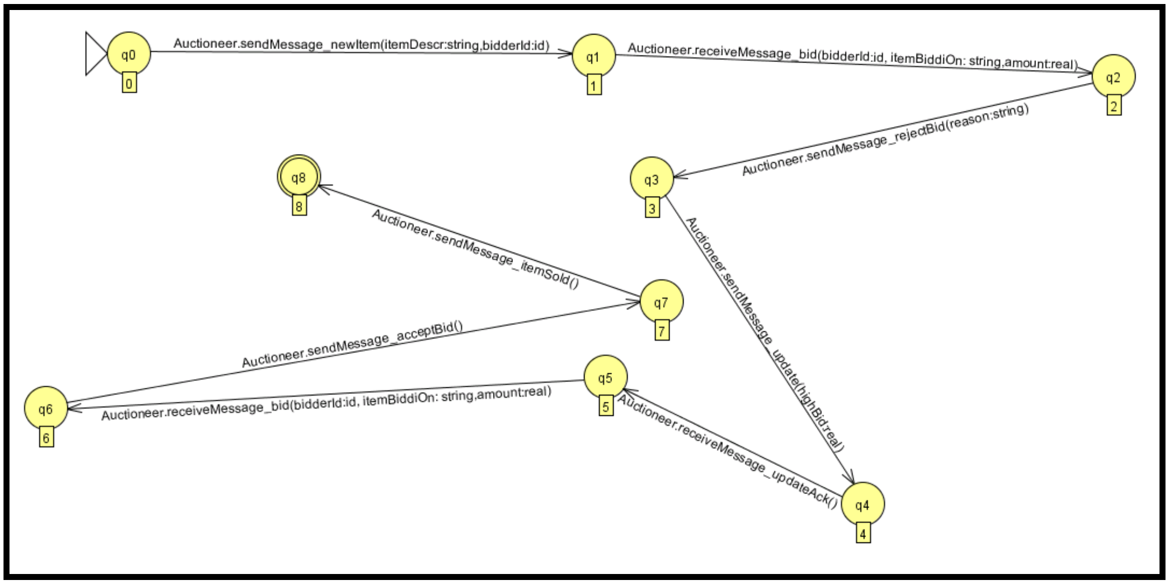


Figure 28: FSA derived from conversation log between the Tracer and auctioneer using the KBehavior algorithm

5.4 Adaptor Generator

The adaptor generator engine uses the adaptor mapping rules file created by the framework then it opens a new WordPad document. For each parameter element for both the provider and consumer, it creates a member variable by appending the letter "*l*" to the parameter element name *"llength"* with default access method as public resulting to *"public double llength"*. It then generates setter methods for each of the parameter elements with default access method as *public,* it then appends *"set"* to the parameter name before assigning the parameter to the member variable just created resulting in:

```
public void setlength (double length)
{
llength=length;
}
```

it then generates getter methods for parameter elements by making the access method public and appending get to the parameter name and since getter methods returns some value, the keyword return is generated before the name of the member variable just created resulting in:

```
public double getlength ()
 {
  return llength;
 }
```

For each parameter in the provider class with annotation, generate the constraints by inserting comments such as shown below to be translated by the JMSassert as constraints.

```
/**
*
* @pre len > 0
```

If the mismatch tag exists for any parameter in the consumer class, the adaptor generator engine uses the identified mismatch as input and delegates the DataConvLib, which performs the conversion and returns the right value. The engine then checks for the existence of the map-to element for parameters in the consumers requires interface if it exists, it retrieves the class, operation and the parameter name.

5.5 Test Observation Before and After Adaptation

Prior to adaptation, we encountered various problems ranging from compilation errors while compiling the bidder component due to argument type mismatch problems between the bidder and auctioneer component to situations where the auctioneer component could not be invoked because some of the operations had to be merged in order to collaborate with the bidder component. The bidder component interface exposed operations such as canBid() and cannotBid() that the auctioneer component didn't provide in order to collaborate with the bidder or fulfill its objective. Without adaptation, it impossible for both components to collaborate even though they are functionally compatible. After instrumentation of the auctioneer component to better understand how to use its interface, we then had a better understanding on how to map the operations between the two interfaces.

However after the adaptation, we were able to successfully establish exchange of information between the two components. The auctioneer component was able to initiate an auction by sending a newItem message and responding to various bids from 5 different bidders and finally the bidder with the highest bid received the acceptBid message from the auctioneer all the other four bidders received the itemSold message. Figure 29 below depicts our experience prior to adaptation.

Bidder \ Auctioneer	Bid()	UpdateAck()	itemSold()	acceptBid()	rejectBid()	newItem()	Update()
Gave1()			√				
bidNotOk()					x		
bidOk()				√			
newHighBid()							√
cannotBid()							
canBid()							
auctionBegin()						0_2	
highBidAck()		√					
newBid()	0_1					0_2	
requestToBid()	0_1						

Legend:
√ Signature matched
X Signature mismatch - Compilation error
0_1 Message requires merging
0_2 Message requires merging

Figure 29: Auctioneer - Bidder Test Observation

Chapter 6 Adaptor Analysis

6.1 Component Compatibility

In order to resolve mismatches between two components that are functionally compatible but their interfaces are not, an adaptor is placed between the two components to compensate for their differences. The adaptor will mediate the interaction between the two components even if their interfaces are not protocol compatible or they support a different set of messages or their parameters are not type compatible.

Assuming that a consumer component supports protocol P1 and the provider supports protocol P2, in [28] which is the foundation of this work, Yellin and Strom defines an adaptor to be well-formed w.r.t P1 and P2 iff It is memory consistent and protocol safe w.r.t. P1 and P2. An adaptor is protocol safe w.r.t P1 and P2 iff in its communication with Component A it uses a protocol compatible with P1 and in its communication with Component B it uses a protocol compatible with P2. For an interface mapping between component A and B supporting protocol P1 and P2 respectively, they define an adaptor to be valid iff the adaptor is well-formed w.r.t. P1 and P2 and is correct w.r.t the interface mapping between the two component interfaces. An adaptor is said to be correct w.r.t its interface mapping between the two components if there exists a mapping rule in the interface mapping of the form val → parm iff the adaptor satisfies the property val ▹ parm. Where val ▹ parm means there exist some adaptor rule that synthesizes the parm parameter of a message from the value val. Using our simple example above, this can be expressed as (a ▹ x) as there exist a rule in the interface mapping that forwards the *len* parameter from component A to that of the component B. One limitation with this proof is the lack of any property constraining (val ▹ parm**)** to a set of permitted values or a valid range as in $\{x > 0, x \bmod 2 = 0,\}$. In our simple example, the values of the parameters are constraint therefore an extension to the Interface Mapping rule to address the limitation cited above can be expressed in the form:

If and adaptor is correct w.r.t the interface mapping between two components, then there exists a mapping rule in the interface mapping of the form val → parm such that val ∈ parm or val ⊂ parm iff the adaptor satisfies the property val ▹ parm.

6.2 Adaptor Protocol Compatibility

Yellin and Strom's [28] definition of protocol compatibility requires that when one party can send a message *m*, then the other party must be willing to receive that message. However, the protocols are compatible even when one party can receive a message *m*, yet the other party cannot send that message. An adaptor is compatible with protocols P1 and P2 iff they have no unspecified receptions and are deadlock free. P1 and P2 have no unspecified receptions iff, whenever a collaboration history for P1 and P2 can reach the point where *P*1 (*P*2) is in a state where it can send a message *m*, its mate will be in a state where it can receive that message, and hence there exists some collaboration history in which that message is exchanged at that point. P1 and P2 are deadlock free iff the collaboration history of P1 and P2 ends with both protocols in final states, or the collaboration can continue. Thus Protocols P1 and P2 are compatible iff they have no unspecified receptions, and are deadlock free.

In CSP when two processes P1 and P2 interact with each other, it is assumed that the alphabets of the two processes are the same; represented by the parallel composition operator ‖ for interacting process:

$$\alpha\,(P1 \parallel P2) = \alpha P1 \cup \alpha P2$$

The process (P1 ‖ P2) is an interaction were both P1 and P2 permit the same behaviors.

Each event of P1 can occur only when P2 permits it to occur; where as P2 can engage independently in the action ($\alpha P2 - \alpha P1$), without the permission or knowledge from P1. The set of all events that are logically possible for the system is simply the union of the alphabets of both processes. As an example, suppose component B the adaptor is a service provider capable of computing by delegation the area, volume, and the square root of an object such that:

CompB = (length → width → CompB
| length → width → height → CompB
| integer → CompB)

and CompA the consumer, requires only the area of an object and sometimes the square root value to support some internal functionality such that:

CompA = (length → width → CompA
| integer → CompA)

The interaction of both components can be expressed as;

(CompA ‖ CompB) = μX ● (length → width → X | integer → X)

Consequently, each event that occurs must be a possible event in the independent behavior of each process and each sequence of such event must be possible for both these operands such that:

traces (P1 ‖ P2) = *traces* (P1) ∩ *traces* (P2)

If t is the trace of (P1 || P2), then every event in t such that $t \in \alpha P1$ has been an event in the life of P1 and every event in t such that $t \notin \alpha P1$ has occurred without P1's participation. Therefore ($t \upharpoonright \alpha P1$) is a trace of all events that P1 has participated in and therefore a trace of P1. Similarly ($t \upharpoonright \alpha P2$) is a trace of P2. Consequently, every event in t must be either in $\alpha P1$ or αP2. Therefore

$$\textit{traces}\ (\text{P1} \parallel \text{P2}) = \textit{traces}\ (\text{P1}) \cap \textit{traces}\ (\text{P2}) =$$

$$\{t \mid (t \upharpoonright \alpha P1) \in \text{traces}\ (\alpha P1) \wedge (t \upharpoonright \alpha \text{P2}) \in \text{traces}\ (\alpha \text{P2}) \wedge t \in (\alpha P1 \cup \alpha \text{P2})^* \}$$

Let $t1 = \langle$ request, area, reply $\rangle$ then

$$t \upharpoonright \alpha P1 = \langle \text{request, area} \rangle$$

$$t \upharpoonright \alpha \text{P2} = \langle \text{request, reply} \rangle$$

therefore

$$t1 \in \text{traces}\ (\text{P1} \parallel \text{P2})$$

Within the context of integrating COTS components, how do we guarantee that a request from the consumer is compatibility to that expected by the provider? More specifically, we need to make sure that process P2 obeys all the rules of process P1 in its collaboration. Using CSP where a process is modeled as a triple (*A, F*, *D*) of alphabet, failures and divergences, the notion of one process obeying the rules of another process is captured formally by means of a subordinate relationship between the processes. A process P1 is subordinate to process P2 expressed as $P1 \sqsubseteq P2$ if the alphabets of P1 are a subset of the alphabets of P2, the failures of P1 are a subset of the failures of P2 and the divergences of P1 are a subset of the divergences of P2. Formally,

Definition 1 (subordination)

A process $P1 = (\alpha_{P1}, F_{P1}, D_{P1})$ is a subordinate process to $P2 = (\alpha_{p2}, F_{p2}, D_{p2})$, written as $P1 \sqsubseteq P2 \equiv ((\alpha P1 \sqsubseteq \alpha P2) \wedge (F_{p1} \sqsubseteq F_{p2}) \wedge (D_{p1} \sqsubseteq D_{p2}))$

It should be noted that subordination enables the construction of generalized processes that can be used to provide services to other specialized processes thereby promoting reuse. We can now introduce a rather strict definition for process compatibility such that; for two process to be compatible, they must have the same alphabets, the set of all events that are logically possible for the system is simply the union of the alphabets of both processes and the caller must be a subordinate to the callee.

Definition 2 (process compatibility)

P1 is compatible with $P2 \equiv \alpha\,(P1 \parallel P2) \wedge \mathit{traces}\,(P1 \parallel P2) = \mathit{traces}\,(P1) \cap \mathit{traces}\,(P2) \wedge$

$$P1 \sqsubseteq P2.$$

Chapter 7 Conclusion and future work

7.1 Conclusion

This thesis starts with a summary of the state-of-the-art in the area of COTS integration with emphases on integration issues. Specifically signature and protocol mismatches that may occur during the integration of COTS component were articulated as a serious drawback in the area of COTS integration. A necessary background to this topic together with a comprehensive list of references to related work was provided. But despite these advances, resolving mismatches such as protocol mismatch during component integration continues to be a challenging effort. After analyzing other approaches related to adaptor generation as related to resolving protocol mismatch during component integration, it was realized that none of this approaches lend themselves to a practical solution to the problem. Particularly, even though enhancing component interfaces with specification of their behaviors or interaction protocol provides a solid foundation for generating appropriate adaptors for mismatch resolution during component integration, their applicability or utility has been questionable given that COTS products still follow the classic IDL type interfaces. For the purpose of solving the component mismatch problem during integration, it is necessary to adopt a different approach for adaptor generation that leverages tools that are readily available to software engineers.

In this dissertation, we have introduced requirements for a framework for generating adaptors to resolves mismatches such as signature mismatch and protocol mismatch. We've also introduced the MAG framework with a proof of concept which not only demonstrates its utility in developing adaptors to resolve protocol and signature mismatch problems but also flexible and extensible to allow a user to use different components from those proposed in this work.

7.2 Future Work

There are various opportunities for future work that could be investigated based upon the work presented in this dissertation such as,

- Investigating the use of byte code engineering to identify type mismatches and dynamically resolve them as oppose to the use of a graphical user interface.
- The graphical user interface presented in this work could be enhanced to drive the other components of the framework.
- Investigate the use of Byte Code Engineering Libraries (BCEL) that is language independent.
- Even though algorithms other than the BINTEST Algorithm were investigated for purposes of test case generation, there are still opportunities to investigate other algorithms with better performance.
- There are various frameworks and algorithms for inferring possible protocols from execution traces based on the requirements identified in this dissertation. We used the kBehaviour algorithm for this work because it lends itself to our environment but other frameworks and algorithms may be used.

Chapter 8 Bibliography

[01] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse Is Still so Hard: IEEE *Software*, vol. 26, no. 4, pp. 66-69, July/Aug. 2009.

[02] Abd-Allah, A. Composing Heterogeneous Software Architecture. Ph. D. Dissertation, Computer Science, University of Southern California, 1996.

[03] Allen, R., Garlan, D. A Formal Basis for Architectural Connection. *ACM TOSEM* (1997), 6(3): 213-49

[04] Abowd, G., Allen, A., Garlan, D. Formalizing Style to Understand Descriptions of Software Architecture. *ACM TOSEM* (1995),4(4): 319-64.

[05] Compare, D., Inverardi, P., Wolf, A. Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming* (1999), 33:101-31.

[06] Davis, L., Payton, J., Gamble, R. Toward Identifying the Impact of COTS Evolution on Integrated Systems. *2nd Workshop on Successful COTS,* (2000).

[07] Gacek, C. Detecting Architectural Mismatches During System Composition USC/CSE-97-TR-506. Center for Software Engineering, USC, 1997.

[08] Garlan, D., Allen, A., Ockerbloom, J. Architectural Mismatch, or Why it is hard to build systems out of existing parts, In *ICSE-95* (Seattle, WA, 1995).

[09] Inverardi, P., Wolf, A., Yankelevich, D. Static Checking of System Behaviors Using Derived Component Assumptions. *ACM TOSEM* (2000), 9(3): 239-72.

[10] Keshav, R., Gamble, R. Towards Taxonomy of Architecture Integration Strategies, *3rd ISAW* (1998).

[11] Mehta, N., Medvi., Medvidovic, N., Phadke, S. Towards a Taxonomy of Software Connectors. *22nd ICSE*, (Limerick, Ireland, 2000).

[12] Perry, D., Wolf, A. Foundation for the Study of Software Architecture. *ACM SIGSOFT* (1992). 17(4): 40-52.

[13] Stiger, P. An Assessment of Architectural Styles and Integration Components. CS M.S. Thesis, University of Tulsa, 1997.

[14] Boehm, B. and Scherlis, W. L. (1992) "Megaprogramming" Proceedings of the DARPA Software Technology Conference, April (available via USC Center for Software Engineering, Los Angeles, CA 90089-0781).

[15] Gacek, C., Abd-Allah, A., Clark, B., and Boehm, B. (1995) "on the Definition of Software Architecture," in Preceedings of the First International Workshop on Architectures for Software Systems – In Cooperation with the 17th International Conference on Software Engineering, D. Garlan (ed), Seattle, Wa, pp. 85-95.

[16] Gacek, C. and Boehm, B. (1998) "Composing Components: How does one Detect Potential Architecture Mismatches?," Proceedings of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, January.

[17] Sage, Andrew P., Lynch, Charles L. (1998) "Systems Integration and Architecting: An overview of principles, Practices, and Perspectives," System Engineering, The Journal of the International Council on Systems Engineering, Wiley Publishers, volume 1 November 3, pp 176-226.

[18] Shaw, M., and Garlan, D. (1996) "Software Architecture: Perspectives and an Emerging Discipline," Prentice Hall.

[19] Egyed, A.; Medvidovic, N.; Gacek, C. "Component-based perspective on software mismatch detection" In IEE Proceedings-Software, Volume: 147 Issue: 6, December 2000.

[20] Chrysanthos Dellarocas. "Towards a design handbook for integrating software components." In Proc Symposium on Assessment of Software Tools and Technologies. 1997. Sloan School of Management MIT Cambridge, MA ust02139, U.S.A.

[21] Bertrand Meyer. "Systematic Concurrent Object-Oriented Programming" Communications of the ACM, September 1993/Vol. 36 No. 9.

[22] Bertrand Meyer. Applying "Design by Contract" Interactive Software Engineering. IEEE COMPUTER, Oct(0018-9162/92/1000):40--51, October 1992. http://csdl.computer.org/comp/mags/co/1992/10/rx040abs.htm

[23] Christine A. Mingins, Chee Y. Chan. "Building Trust in Third-party Components using Component Wrappers in the .NET Frameworks." School of Computer Science and Software Engineering. Monash University VIC Australia. http://crpit.com/confpapers/CRPITV10Mingins.pdf

[24] Mark R. Vigder, John Dean. "An Architectural Approach in Building Systems from COTS Software Components," National Research Council. http://seg.iit.nrc.ca/papers/NRC40221.pdf

[25] Using Assertions in Java Programming: A novel Approach. http://www.mmsindia.com/DBCForJava.html

[26] Fraser, T., Badger, L., Feldman, M. (1999); Hardening COTS components with generic software wrappers. *Proc. 1999 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press.

[27] Paola Inverardi and Massimo Tivoli. The Role of architecture in component assembly. University of L'Aquila (Dip. Informatica) via Vetoio 1, 67100 L' Aquila Italy.

[28] Daniel M Yellin and Robert E. Strom. "Interfaces, protocol, and semi-automatic construction of software adaptors," In *ACM OOPSLA*, 1994.

[29] Mary Shaw and Paul Clement. "A field guide to bxology: Preliminary classification of architectural styles for software systems." In *Proc. International Computer Software and Applications Conference*. 1997.

[30] Robert Deline. "A Catalo of Techniques for Resolving Packaging Mismatch." Symposium of Software Reusability, Los Angeles CA May 1999.

[31] Amy Moormann Zaremski and Jeannette M. Wing. "Signature Matching: A Key to Reuse," School of Computer Science. Carnegie Mellon University.

[32] Cynthia Della Torre Cicalese. "Behavoiral Specification of Distributed Software Component Interface," The George Washington University. 1999.

[33] Yi Liu and H. Conrad Cunningham. "Software Component Specification Using Design by Contract,". Department of Computer and Information Science. University of Mississippi

[34] Bracciali A., Brogi A., and Canal C.. Dynamically adapting the behaviour of software components. In F. Arbab and C. Talcott (editors), *COORDINATION*

2002: Fifth International Conference on Coordination Models and Languages, Lecture Notes in Computer Science 2315, pages 88-95, 2002. Springer-Verlag.

[35] Bracciali A., Brogi A., and Canal C.. Adapting components with mismatching behaviours. In J. Bishop and S. Herrmann (editors), *Component Deployment, First International IFIP/ACM Working Conference*, Lecture Notes in Computer Science 2370, pages 185-199, 2002, Springer-Verlag.

[36] Bracciali A., Brogi A., and Canal C.. Systematic component adaption. *Electronic Notes in Theoretical Computer Science*, 66(4), 2002.

[37] Bracciali A., Brogi A., and Canal C.. A formal approach to component adaptation. To appear in *Journal of Systems and Sofware*, 2003.

[38] Robert J Allen. A formal approach to software architecture. school of computer science. Carnegie Mellon University. Pittsburgh, PA 15213

[39] C. A. R Hoare. Communicating sequential process March 28, 2003.

[40] Jan Bosch. Superimposition: A component adaptation technique. Technical Report TR, Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, September 1997.

[41] L. Bougé, N. Francez, 'A Compositional Approach to Superimposition,' Proceedings POPL'88, pp. 240-249, 1988.

[42] Ralph Keller and Urs Hölzle: Supporting the Integration and Evolution of Components Through Binary Component Adaptation. University of California at Santa Barbara , Technical Report TRCS97-15 September.

[43] Nenad Medvidovic and Richard N. Taylor A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, 26(1):70-93 January 2000]

[44] Hölzle U.: Integrating Independently-Developed Components in Object-Oriented Languages. In: proceedings of ECOOP '93. Lecture Notes in Computer Science, Vol. 707. Springer-Verlag, (1993) pp. 36-56

[45] Purtilo J. and Atlee J. Module Reuse by Interface Adaptation. In Software practice and Experience, Vol. 21, No. 6, 1991.

[46] Bridget Spitznagel, David Garlan: A Compositional Formalization of Connector Wrappers. *Proceedings of the 2003 International Conference on Software Engineering (ICSE'03).*

[47] Heinz W. Schmidt, Ralf H. Reussner: Generating adapters for concurrent component protocol synchronisation. Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems V Pages: 213 - 229

[48] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, Walter Mann. Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering, 21,(4):336-355, April 1995.

[49] Robert J. Allen. A Formal Approach to Software Architecture. Ph.D. Thesis, Carnegie Mellon University,CMU Technical Report CMU-CS-97-144, May 1997

[50] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M.

Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering, 21(4):314–335, April 1995].

[51] Urs Hölzle: Integrating Independently-Developed Components in Object-Oriented Languages. *ECOOP '93 Proceedings, Springer Verlag Lecture Notes on Computer Science.*

[52] Sun Microsystem Corporation, The Enterprise Java Beans homepage http://java.sun.com/products/ejb/

[53] Microsoft Corp., The .NET homepage. http://www.microsoft.com/net/default.asp

[54] Object Management Group (OMG), The CORBA homepage http://www.corba.org

[55] Microsoft .NET Homepage, http://www.microsoft.com/net/

[56] R.H. Reussner, Automatic component protocol adaptation with the CoConut/J tool suite, *Future Generation Computer Systems* **19** (2003) (5), pp. 627–639.

[57] George T. Heinernan , "An Evaluation of Component Adaptation Techniques. " Computer Science Department, Worcester Polytechnic Institute, WPI-CS-TR-99-04

[58] S. P. Reiss and M. Renieris. Encoding program executions. In Proceedings of the International Conference on Software Engineering, pp. 221-230, May 2001.

[59] J. Cook and A. Wolf. Discovering models of software processes from event-based data. ACM Transactions on Software Engineering and Methodology, 7(3):215–249, 1998. Austria, 2001. ACM Press.

[60] A. Biermann and J. Feldman. On the synthesis of finite state machines from samples of their behavior. IEEE Transactions on Computer, 21:592–597, June 1972.

[61] G. Ammons, R. Bodik and J. R. Larus. Mining Specification. In Proceedings of Principles of Programming Languages (POPL02), Portland, Oregon, pp. 4-16, January 2002.

[62] Leonardo Mariana, Mauro Perez. Inference of Component Protocols by the Kbehavior Algorithm. Technical Report LTA:2004:5 via Bicocca degli Arcimboldi 8 20126 Milano Italia.

[63] John Whaley Michael C. Martin Monica S. Lam. Automatic Extraction of Object-Oriented Component Interfaces. Computer Systems Laboratory Stanford University

[64] Sami Baydeda, Volker Gruhn, 2003, "BINTEST – binary search-based test case generation", In Computer Software and Applications Conference (COMPSAC), IEEE Computer Society Press, 2003.

[65] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, Paul Ammann, March 2003, "Generating Test data from State based Specifications", The Journal of Software Testing, Verification and Reliability, 13(1):25-53.

[66] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In Proceedings of Eighteenth ACM Symposium on Operating Systems Principles, pp. 57-72, October 2001.

[67] D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In Proceedings of the 28th Annual International Symposium on Computer Architecture, pp. 192-203, July 2001.]

[68] D. Wagner and D. Dean. Intrusion detection via static analysis. In Proceedings of the IEEE Symposium on Security and Privacy, pp. 156-169, May 2001.]

[69] Shigeru Chiba: Javassist --- A Reflection Based Programming Wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java.* October 1998.

[70] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In Proceedings of the ACM Conference on Programming Language Design and Implementation, pp. 59-69, June 2001.

[71] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces, SPIN 2001,Workshop on Model Checking of Software, LNCS 2057, pp. 103-122, May 2001.

[72] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, June 2002.

[73] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and tool support for debugging object protocols. In Proceedings of the 8th ACM International Symposium on the Foundations of Software Engineering, pp. 50-59, November 2000.

[74] IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology

[75] Carlos Canal, Lidia Fuentes, Jos´e M. Troya, and Antonio Vallecillo. Extending CORBA interfaces with π-calculus for protocol compatibility. In *Proc. of TOOLS Europe 2000*, pages 208–225, France, June 2000. IEEE Press.

[76] A. Vallecillo, J. Hern´andez, and J. M. Troya. Object interoperability. In Object-Oriented Technology: ECOOP'99 Workshop Reader, number 1743 in LNCS, pages 1–21. Springer-Verlag, 1999.

[77] L. Davis, R Gamble, J. Payton, G. Jonsdottir, D. Underwood "A notation for problematic architecture Interactions" Department of Mathematical and Computer Sciences, University of Tulsa, OK