Interface Faults Injection for Component-based Integration Testing

Nor Laily Hashim, Heinz W.Schmidt, Sita Ramakrishnana
Centre for Distributed Systems and Software Engineering
Monash University
900 Dandenong Road
Caulfield, 3145 VIC, Australia
Email: norh, hws, sitar@infotech.monash.edu.au

Abstract—This paper presents a simple and improved technique of interface fault insertion for conducting component integration testing through the use of Aspect-oriented Software Development (AOSD). Taking the advantage of Aspect’s cross-cutting features, this technique only requires additional codes written in AspectJ rather than having a separate tool to perform this operation. These aspect codes act as wrappers around interface services and perform operations such as disabling the implementation of the interface services, raising exceptions or corrupting the inputs and outputs of interface services. Interface faults are inserted into the system under test to evaluate the quality of the test cases by ensuring not only that they detect errors due to the interactions between components, but they are also able to handle exceptions raised when interface faults are triggered.

I. INTRODUCTION

The overall focus of this research is on the component-based integration testing process with consideration of reducing the gaps between the plan for integration testing at architectural level and its actual testing at code level. The motivation of this paper is to introduce a simple and quick technique to insert interface faults in conducting integration testing at the component level through the use of Aspect-oriented Software Development (AOSD). The need for the interface fault technique arises because we want to evaluate the quality of the generated test cases by ensuring that they are not only capable of detecting existing integration faults but also able to detect errors caused by triggered interface faults. There are several researchers focusing on interface faults such as [1, 2]. Interface faults are used because we are conducting component-based integration testing, where the focus is on testing the interconnections between components.

The use of software architectures for planning and managing the overall system testing is one of the motivations in the use of software architectures [3]. The high level information abstraction provided in the software architecture is represented by components and connectors, which complements the component definitions and interface specifications [4]. Among researchers who apply software architecture for test plan in integration testing implementation are [2, 5, 6]. The information gathered from the software architecture provides a broad view of how components are assembled. The structure of a software model derived from a software architecture is useful in deriving integration testing plan and strategy [7, 8]. The information related to testing criteria and inputs are also captured by the elements in the architectural model and can be applied on a test adequacy model [9]. These architecture specifications provide testers with an overall view of the interactions among components and directs them in focusing on the interesting interactions [5]. It can also be used for testing components ordering and electing effective sets of test cases [8].

For testing purposes architectural specifications need to be accurately and explicitly documented. In order to explicitly represent this specification, an architectural definition language (ADL) is used in documenting behavioral relationships and dependencies between architectural elements. This ADL must be precise enough to represent interactions among components. In addition to this, [6] also emphasised that architectural descriptions provide the right abstraction level for devising integration test strategies.

The integration testing at the architecture level concentrates on evaluating the overall system behaviours by analysing individual component behaviours and interactions with the other components, which resembled connectors in software architecture [8]. The emergence of formal notation for the software architecture such as ADL has made testing at the architectural level possible.

The information captured at architectural level is not detailed enough for testers to create test cases. Therefore, we have integrated an approach by [10] into our approach one that allows the extraction of all possible interactions that might not be captured in the architectural specification. [10]’s graph-based approach is an implementation-based component integration testing strategy. They performed static analysis on the source program to derive direct and indirect interactions among components. This feature would definitely help in reducing the gaps with what is implemented at code level and what is planned at architectural level.

The contribution of this paper is a novel technique for interface fault insertion for component-based integration testing. This approach uses a simple one that is easy to create and maintain using Aspect technology. This technique is easily integrated with the JUnit testing framework and other Java programming tools and utilities. Our work is quite similar...
to [1] which focuses more on the fault model using Java Reflection in building its interface fault tool called Jaca. On the other hand, our technique focuses on applying interface faults in component integration testing, concentrating on deriving the test cases, test coverage, etc. In this paper, we also introduce our own methodology in conducting component integration testing.

We start this paper with a brief summary of the existing concepts in examining component-based integration theory and integration faults. In the following section, we discuss the current concepts of interface faults insertion and a tool called Jaca. We also explain briefly a few of AspectJ constructs that are used in this paper. Then we extend the following section by explaining the methodology used to derive the test cases and how faults are inserted by providing snippets of the aspect codes. The section that follows contains the related studies in this area. Lastly, we present our conclusion and future work.

II. COMPONENT-BASED INTEGRATION TESTING

In general, integration testing is to test the interactions between two units [11]. [12] describes integration testing as an evaluation method to test the interactions of interfaces between components. It is to make sure that they are following the assumptions consistently and communicating accurately. An integration-level component test process is performed on larger components aggregation in order to eliminate failures caused by: (1) partial execution of a sequence of messages or protocols which occurs across components, (2) timing failure which occurs when a message sent to a component did not arrive as expected and (3) incompatibility of a component model, for example, by integrating a COM component to a CORBA component model using inappropriate adapter could lead to communication problem [13].

Components in this paper refers to Java packages within a software program. We refer them as internal components. These internal components might invoke functions from other external independent programs, such as from several open source programs. The packages within the external programs are referred to as external components. Let us say, if an application program is developed by collaborating functions from two open source programs, the integration testing will test the interconnections between the internal components and also external components.

The interconnections are actually the interactions of interfaces between components. An interface defines the types of services that a component provides to others. A component is accessed through its interfaces. For example, a client component can only request the services defined in a server component interface. When an interface is invoked in response to a request, an action is triggered to process the request. The triggered action is referred to as an event. Events can either be invocation invocations or user actions from inside or outside of the components which are triggered through their interfaces.

The integration testing strategy used in this paper is an execution-based type, which traces the execution between two units to identify defective interactions [14] by using software architecture for planning the test. In this paper, we are assuming each component has been well tested in the unit/component testing.

A. Component Interaction Graph

[10]’s graph-based approach is one of implementation-based component integration testing strategies. The implementation-based approach tackles the system from bottom-up, and is only suitable if the source code is available. The interactions between the components are used to form test requirements and later transform to test cases to detect inter-operability faults between interacting components. Four test elements are identified in this approach:

- Interfaces for all components must be tested at least once.
- Events from interface invocations which are either followed by an event execution or a sequence of event executions. The same interface can be invoked in different ways and would lead to different event executions. Every event should be tested at least once. The following are the types of event classifications:
  1) A message sends to an interface, requesting its services.
  2) A user action such as a mouse clicked on a GUI frame, which is executed by an event or a sequence of events
  3) An exception that is triggered, for example, when a file is not found, an SQL exception and others.
- A context dependence relationship refers to a path of event invocations when components interact. This is similar to the control flow dependencies concept. For example, if an event \( e_1 \) directly or indirectly invokes event \( e_2 \), there exists an execution path between these two events. Therefore, there exists a context dependence relationship among interfaces or events.
- A content dependence relationship refers to any content dependence relationships that are associated when the interface of each path has data dependence relationship. This is similar to the data flow concept in the traditional program. For example, if a function in interface A depends on the data used by another defined function in interface B.

Direct interactions are covered by interface and event test elements. Meanwhile, the indirect interactions are represented by content dependence relationship and context dependence relationships. A loop-free event path derived from context dependence relationships can be used to detect integration faults between components, while a loop-free event path identified from content dependence can be used to generate test cases and detecting faults.

The graph created using this approach is called a Component Interaction Graph (CIG). The graph \( G = (V, E) \), where \( V \) is a set of nodes and \( E \) is a set of directed edges. \( V = V_I \cup V_E \), where \( V_I \) represents a set of interface services and \( V_E \) represents a set of events which may be generated in the components. For example, an edge from an
interface \( v_i \) to \( v_j \) is represented as \( e_{i,j} = (v_i, v_j) \). A path (or subpath) is a finite sequence of vertices \((v_0, v_1, ..., v_k)\) such that \( e_{i,j} = (v_i, v_{i+1}) \in E \) for \( i = 0, 1, ..., k-1 \). A loop free path is a path \((v_0, v_1, ..., v_k)\) such that for any \( i \) and \( j \), \( i \neq j \) and \( v_i \neq u_j \) [15].

B. Integration faults

A fault is defined as "a program defect that has caused or can potentially cause a failure" [16]. [17] provided a comprehensive study of the possible faults in component-based testing. Faults specifically related to integration testing are semantic, faulty connector and topology.

Semantical faults are caused by failures in execution of interactions or failures in caller components due to wrong return values from callee components. There are four types of semantical fault: (1) misunderstood behaviours, (2) misunderstood parameters, (3) misunderstood event generations, which means that a component which received an event has wrongly interpreted it, (4) misunderstood interaction protocols, which occurs when two components do not agree on the used interaction protocols. [17] proposed the usage of contract, formal specifications or state machines to detect semantic faults.

A faulty connector is caused by mismatches of protocol faults or incompatible data models. A mismatch protocol is caused by imprecise agreements in the protocol used in handling interactions between components. An incompatible data model fault occurs when there is a failure in transmitting the desired data. This incompatibility happens because component developers during a development phase often make assumptions of the type of infrastructures, services and structure of the system, which the component will be in used.

III. INTERFACE FAULT INJECTION

There are two reasons for injecting faults into a system: (1) to evaluate the level of fault tolerance capability of a system and (2) to enhance the testing process. In this paper, we apply fault insertion for the latter reason, that is to test the quality of our test cases by ensuring they are capable of detecting existing errors and the faults when they are triggered. Inserting faults randomly into a component is pointless because it might lead to testing another component than the intended. A better approach in inserting faults into a component involves triggering them on the interface. This can be conducted by placing wrappers around the interfaces. The wrapper function in this paper is provided by AspectJ, an Aspect programming language. Examples of research in fault injection techniques are [18, 19, 20].

There are several issues to consider in fault injection such as what the faults inserted are, the combination of different faults type, the faults location, the faults frequency, etc. The main focus of fault insertion is to identify all possible ways a software product can fail. Due to the large amount of space involved in a software product, the fault injection process requires planning.

Jaca [1] is a fault insertion tool for Java applications which uses Java Reflection. This research has been extended to provide a systematic approach of conducting interface fault insertion to test the interconnections between integrated components at the architectural level [2]. This approach allows components and system behaviours to be observed. The interface faults are in the form of corrupted methods’ parameters and methods’ return values, which are inserted into the system under test. However, this tool is a research tool and is not available for public use. Therefore, we have to create our own technique to insert interface faults into our own test environment.

We are adopting the similar approach to [2] that focuses on testing the interactions between components and uses software architecture for planning integration testing and fault insertion. The generation of errors is by corrupting the inputs and outputs data of the interfaces invoked. [2]’s technique only allows the corruption of the interface inputs and outputs data using simple primitives like integer, real and boolean and String. Our technique has a few additional features such as allowing the corruption of data from user-defined object types. It is also capable of raising relevant exceptions to test how well the test cases handle exceptions. Moreover, our technique also allows faults to occur by invoking other polymorphic methods.

A. AspectJ

Aspect-oriented software development facilitates easy maintenance and development as it allows the separation of concern in software development through its modular cross-cutting implementation. This section contains a brief explanation of a few constructs of AspectJ, one of the aspect-oriented programming languages. The selected constructs are join points, pointcut, advice, and aspects [21].

Join points refer to specific locations in a program that can be invoked by any advices that called them. They can either refer to the invocations or executions of methods or to constructor, field assignments, objects initialisation, or exceptions handling.

A pointcut is a way of announcing an interest in a join point, which is later used in advices. A pointcut can also be built up with several pointcuts from various types. It can be defined using two crosscut types: name-based or property-based [21]. The former crosscut type allows a programmer to explicitly specify the name of a set of pointcuts or methods. The latter type of crosscut uses wildcard in specifying method signatures.

An advice is a piece of code executed when Aspects is invoked. It has similar form to method in Java, where it combines together a pointcut and a body of codes defining how Aspect is going to implement it when a join point has been invoked. It contains a set of rules defining when it should be invoked in relation to the join point. Advices can be invoked in defining timing such as before, after or around a pointcut. The body of codes within a before advice is executed by Aspect just before the pointcut is triggered. An after advice is executed after the pointcut declared in it is triggered. An around advice
blocks the execution of the join point by allowing the body of codes within the advice to be executed first. The remaining action of the actual the join point can be invoked again by using `proceed` call.

An *aspect* has almost the equivalence of a class in Java, where it may contain pointcuts, and advices, as well as method declarations that are permitted in class declarations.

From fault insertion context, AspectJ is used as an interception mechanism to block invocations of interface services made by the program. By intercepting calls to the interface services, we are able to plant faults for either the inputs or outputs of the interface services.

### IV. METHODOLOGY

In this experiment, the system under test is a free Java-based decompiling tool, called Cavaj version 1.0, where it produces java source program from binary files, which can be in the form of .class, .jar or .zip [22]. This tool uses Jode, an open source Java decompiling engine [23]. The Jode version used in this tool is of embedded version, which is with less functions compared to its full version. Cavaj also uses SkinLF component by [24], as its graphical interface. SkinLF has additional graphical functions, which are not supported by Java Swing, including PropertySheet component, a collapsible task pane, a button bar, a font chooser, a directory chooser and able to read GTK (The Gimp ToolKit) and KDE (The K Desktop Environment) skins. Cavaj is composed of three main components: Sureshot, Jode and SkinLF. The Sureshot component acts as a manager and integrator for Jode and SkinLF, and also provides features for file management, error handling, help functions etc.

#### A. Test cases

Based on the derived paths derived from the CIG, the information from each path is used to form test cases. In this paper, we only concentrate on the first three of the test elements, ignoring the content dependence relationship, as it would be too detailed for handling at the component level.

Using an automated JUnit test generator tool, called JTest [25], test cases for methods in each class are generated. The quality of some of the automated generation of test cases might not be very good because the automated creation of stubs within each test case still need to be modified manually to increase the branch coverage of each test case. The process of constructing stubs is very time consuming and requires semantic knowledge of the methods. Using the paths derived from CIG, the unit tests generated are selected to represent the methods sequence involve in each of the paths. In selecting these unit tests, we choose the one with higher branch coverage value, so that we have a better quality test cases. These derived paths would assist testers in focusing only on the generation of test cases that are involved in the interactions between components. The unit tests that are not involved in the derived paths can be ignored.

For each of the component interface, a test suite is created. Each of the paths that invokes the interface service is tested in this test suite. An interface service can be invoked in several contexts. Each path contains a sequence of unit tests that leads to the invocation of the interface service.

#### B. Faults insertion

Based on the decided scenario, the usage profile of the system is collected. From the TPTP Eclipse Profiling tool [26], we are able to monitor all calls to the methods in the system. This information allows users to identify the components that are frequently called and the changes of method calls of the interface services for different scenarios. However, the information that is important are the calls to the interface functions. From the CIG, we are able to extract all possible paths that exist between components regardless of the scenarios. This profiling information helps us to identify the actual interface functions that are called at runtime. Therefore, this makes the test cases generated more focused and efficient as this would guide the testers to prepare the test suites of the interface services which are involved in the execution of the scenarios. The profiling information also assists testers in specifying the probability for the faults injection. We are currently using the number of times the interface services are invoked to assign the probability value. The more frequently an interface service is called, the higher the probability value is assigned. Currently, we are using probability values generated from a method called `nextGaussian()` from Random class in Java 2 SE v1.4.2. This method returns a Gaussian distributed double value with mean 0.0 and standard deviation 1.0 from this random number generator’s sequence. This method has its limitation where any two Random objects created within the same millisecond will have the same sequence of random numbers.

Faults are triggered by the program itself, whenever the program calls the interface services. This fault is of interface faults type. We are using AspectJ using around advices to intercept the method calls to the interface functions. Depending on the type of interface services, it can either:

- invoke an appropriate exception
- skip an execution of an interface service
- skip an execution of an interface service and invoked an appropriate exception
- skip an execution of an interface method and return a corrupted value, either null or from different type

Figure 1 shows a portion of Cavaj code listing three of the interface definitions from three packages.

In Figure 2, the first advice shows an example of how to corrupt the input of `load(String)` method. The actual input for this method is replaced with a `null` value. Therefore, when this method continues its operation through the `proceed()` call, this method will have its input value changed. The following advice shows an example of a fault that returns a null value, where the actual method is expecting an array of Image type.
V. RELATED WORK

In a survey conducted by [27], he reported the testing strategies for unit and system level software components for both component producers and component consumers. [17] reported several known testing techniques specifically for conducting component-based integration testing: (1) Flow-graph Based Testing/Component Interaction Graph (CIG) [10], (2) Finite State Machine (FSM) Based Testing, (3) Component-based Adequacy Criteria [28], (4) Dependence Analysis, (5) UML-Based Testing [29], (6) Self-Test Component and (7) Run-Time Verification.

[1] highlights the usage of heuristics methods to assess the risk of components. Even though the components can be prioritised according to the risk assessment, expert’s opinion is still important in selecting the components to be tested. Components can be assigned one of the following types of risks: (1) new- New developed components, (2) changes- Existing components that are modified, (3) upstream dependency- Failures in the components that have caused rippling effects to the rest of the system, (4) upstream dependency- Failures within the system that populate in the component), (5) Critical- Any failures triggered by this component that brings significant damage, (6) popular- A component frequently used by other components, (7) strategic- A component with important features, for example contains business features), (8) third-party- A commercial component), (9) distributed- A component that is distributed physically, but contains elements that are important.

VI. CONCLUSIONS

This paper introduces an improved, simple and easy technique of interface faults insertion using AspectJ for Java component-based applications. Our technique is capable of ignoring the entire execution of an interface service, corrupting its input values and returning a bogus return value. Our technique has a few additional features, such as allowing the corruption of data from user-defined object types, raising relevant exceptions and allowing faults to occur by invoking other polymorphic methods. Examples presented in this paper
only show a simple way of how interface faults can be triggered. This technique can also be extended by adding counters in the interface faults codes to monitor how often an interface service is called, what components often invoked a particular interface service, etc.

For our future work, we plan to expand this component integration testing approach for measuring test coverage and the degree of correctness of the test cases created and also apply this technique in component-based product lines application.

REFERENCES