An Automated Software Fault Injection Tool for Robustness Assessment of Java COTs

Kamal Z. Zamli, Mohd Daud Alang Hassan, Nor Ashidi Mat Isa, and Siti Norbaya Azizan
School of Electrical and Electronic Engineering
University Science Malaysia Engineering Campus
14300 Nibong Tebal, Penang, Malaysia

Abstract-In line with market demands and the need for technological innovations, designing and implementing software and hardware components for computing systems is growing in complexity. In order to cope with such complexity whilst meeting market needs, engineers often rely on design integration with commercial-of-the-shelf-components (COTs). In the case where lives and fortunes are at stake, there is a need to ensure dependability of COTs in terms of their robustness before they can be adopted in such an environment. However, it is not often possible to thoroughly test COTs for robustness because their design as well as source codes are usually unavailable. In order to address some of the above issues, we have developed an automated software fault injection tool, called SFIT, based on the use of computational reflection and Java technology. This paper describes our experiences with SFIT performing robustness testing of Java COTs, called Jada, a Linda tuple space implementation.

I. INTRODUCTION

In line with market demands and the need for technological innovations, designing and implementing software and hardware components for computing systems is growing in complexity. In order to cope with such complexity whilst meeting market needs, engineers often rely on design integration with commercial-of-the-shelf-components (COTs). In the case where lives and fortunes are at stake, there is a need to ensure dependability of COTs in terms of their robustness before they can be adopted in such an environment. As an illustration, consider the disaster that occurred during the European Space Agency’s launching of Ariane 5 in 1996. Investigation by independent researchers from Massachusetts Institute of Technology reveals that the disaster is caused by software component faults [5]. The component erroneously puts a 64 bit floating point number in to a 16 bit space, causing overflow error. This overflow error affected the rocket’s alignment function, and hence, causing the rocket to veer off course and eventually exploded a mere 37 seconds after lift off.

As the above example illustrates, it is important to assess robustness of COTs in order to ensure its dependability. However, it is not often possible to thoroughly test COTs for robustness because their design as well as source codes are usually unavailable. In order to address some of the aforementioned issues, we have developed an automated software fault injection tool, called SFIT [1, 3, 4, 24] based on the use of computational reflection and Java technology. This paper describes our experiences with SFIT performing robustness testing of Java COTs, called Jada [13], a Linda tuple space implementation [14].

This paper is organized as follows. Section 2 discusses an overview of fault injection techniques. Section 3 introduces SFIT as well as outlines all of the SFTT components. Section 4 discusses the SFIT case study. Section 5 outlines the running experiment with SFIT and Jada. Section 6 discusses the lessons learned. Finally, section 7 gives our conclusion.

II. OVERVIEW OF FAULT INJECTION TECHNIQUE

Robustness is critical to dependable operation of computer systems. Robustness can be defined as the degree to which software and hardware can function correctly in the presence of invalid inputs or stressful environment conditions. A common technique for ensuring robustness of computer systems is through testing. Although useful for establishing correctness of functionality, traditional testing methods may be insufficient for assessing robustness in the presence of exceptional inputs or extreme conditions.

Fault injection technique is an important testing method for experimentally evaluating robustness as well as accessing dependability of computing systems. Often, fault injection technique improves test coverage as well as facilitates fault forecasting and fault acceleration. Improved test coverage is achieved by exercising extreme input conditions to the particular components or systems of interest. Fault forecasting allows prediction in terms of the behavior of a particular system under fault, that is, whether or not the system maintains its intended operations. Rather than waiting for certain invalid conditions to occur, fault acceleration permits emulation of such invalid conditions to enable early dependability assessment.

According to Ejlali et al [11], there are two main categories of fault injection technique: physical faults injection, and simulation-based fault injection.

Concerning physical fault injection, there are many methods and tools for performing fault injection. Some of the physical fault injection methods are [11]:

- SWIFI (Software Implemented Fault Injection): In this method, faults are injected to a physical system (i.e. software or hardware) by means of software. FIAT [7], FERRARI [17], EXFI [8], FINE [18], Xception [9], and JACA [21] are some examples of tools which use SWIFI.
- SCIFI (Scan Chain Implemented Fault Injection): In this method, faults are injected...
to physical system using scan chain. FIMBUL [12] and GOOFI [2] are some examples of tools which uses SCIFI.

- Pin Level Fault Injection: In this method, faults are injected to the pins of an IC. MESSALINE [6] is an example of pin-level fault injection tools.

- Fault Injection by external disturbance: Heavy ions radiation [15] and power disturbance [20] are some of the techniques for injecting faults by external disturbances.

- Fault Injection in emulated systems: In this method, faults are injected to emulated model of the system implemented using FPGA [10].

As far as simulation-based fault injection is concerned, the physical implementation of the actual system need not be completed in order to perform fault injection. Rather, fault can be injected in the simulated model of the system, for example, in a VHDL model. Some examples of the tools that support simulation-based fault injection are MEFISTO [16] and VERIFY [23].

Building and complementing from earlier work discussed above, this paper focuses on developing SFIT, a new fault injection tool based on computational reflection that employs SWIFI to assess the dependability of Java COTs.

III. INTRODUCING SFIT

SFIT has been developed based on the use of computational reflection offered by the Java programming language [1, 3, 4, 24]. The main novel features of SFIT are:

- It allows robustness testing of Java COTs in the absence of source codes for public, private, and protected methods by using variable types of fault values.

- It permits high level abstraction for fault injection. Because the injector routines are automatically generated (and executed) by SFIT, the test engineer need not be proficient in Java.

- It enables automated support for fault testing. In the current version, robustness testing can be automatically performed up to 2500 test cases per injection.

Referring to Figure 1, SFIT consists of a number of interrelated components:

**Class Inspector**

As its name suggests, the main purpose of class inspector is to inspect COTs in order to obtain details information of the COTs interface. The class inspector exploits the Java Reflection API in order to interrogate COTs for method interfaces including public, private, and protected ones. This information will be used to setup the fault setting (as will be discussed next).

**Fault Setting**

Fault setting is a fault definition file. With the fault setting, the user can define fault test cases to be injected in COTs through its methods interfaces (i.e. up to 2500 interface parameter values per injection as test cases in a pre-defined format). Fault setting also contains the information indicating where will the faults be injected, how will the faults be injected, how many faults (i.e. test cases) are injected, as well as what will be injected.
Loader and Generator

The purpose of the loader and generator component is to load the fault setting file and automatically generate reflective Java code that will be used to inject the faults, although the generated codes are completely hidden to the user. The generated codes will be used by the fault injector component (discussed next).

Fault Injector

The purpose of the fault injector is to load the generated codes into the Java Virtual Machine (JVM) in order to inject the faults as specified by the test cases in the fault setting. The process of generating (i.e., performed by the loader and generator) and injecting faults are performed iteratively based on the number of test cases specified in the fault setting.

Log

Log is a special database for permitting offline analysis of the output and behavior of COTs under test. If the specification of the COTs method exists, conformance analysis can be made using this database.

IV. ON SFIT CASE STUDY - JADA

In order to evaluate its applicability, there is obviously a need to subject SFIT to a real case study problem. Here, Jada [13], a distributed shared memory implementation of Linda in Java, has been chosen. The rationale for choosing Jada stemmed from the fact that it is a public domain Java COTs freely accessible for download in the internet. Like most other COTs implementation, Jada does not come with source codes although an overview of its methods and functionalities are given in the documentation (see reference [13]).

Based on the given documentation and using SFIT as the testing tool, a number of test cases are defined to assess Jada for robustness (see Figure 4). Experiences from the testing Jada are beneficial to evaluate the suitability of SFIT as a general software fault injection tool for Java COTs. In fact, these experiences are valuable to establish whether or not Jada implementation is robust, and hence can be used for safety critical applications.

In order to understand how Jada works, it is necessary to discuss some background on Linda. Linda is a parallel programming model that was proposed by David Gelernter to solve the problem of programming parallel machines [14]. Tuple space, essentially a distributed shared memory, is Linda’s mechanism for creating and coordinating multiple execution threads. Tuple space stores tuples, where a tuple is simply a sequence of typed fields.

The Linda model is often embedded in a computation language (such as C, Lisp, and Java) and the result is a parallel programming language. The Linda model defines four operations on tuple space:

- out(t); Causes tuple t to be added in the tuple space.
- in(s); Causes some tuple t that matches the template s to be withdrawn from the tuple space. The values of the actuals of t are assigned to the formals of s and the executing process continues. If no matching t is available when in(s) executes, the executing process suspends until one is (i.e. blocking). If many matching t’s are available, one is chosen arbitrarily.
- read(s); Its operation is the same as in(s); expect that the matching tuple is not withdrawn from the tuple space.
- eval(t); Causes tuple t to be added to the tuple space but t is evaluated after rather than before it enters the tuple space. A new process is created to perform the evaluation.

As far as Jada is concerned, it implements most of the Linda operations including the non-blocking version of out(t), in(s) and rd(s), although, the eval(t) has not been implemented.

Because no source code is available, it is impossible to know the exact class structure and dependencies of Jada. Nevertheless, the Jada documentation highlights the following Jada class hierarchy:

```java
class java.lang.Object
  o class jada.primitiveSerializer (implements jada.Serializer)
    o class jada.integerSerializer (implements jada.Serializer)
  o class jada.lisp
    o interface jada.LispItem
    o interface jada.LispItemConst
  o class jada.net.JadeNetD
    o interface jada.JadeNetSerializ
    o class jada.net.ObjectSpace
      o class jada.net.ObjectServer (implements jada.JadeNetConst, jada.net.JadeNetSerializ)
    o class jada.StringSerializer (implements jada.Serializer)
      o class java.lang.String
        o class java.lang.Exception
          o class jada.jadaException
`
Although testing all the classes defined in Jada is equally beneficial (see Figure 2), the Jada Object Space class that is the focus of this work. The rationale for such a focus is due to the fact that it is the Object Space class that implements the Linda tuple space operations.

It should be noted that although given in the Jada documentation, the methods for manipulating tuple space defined in the Object Space Class can also be discovered automatically using the SFIT Class Inspector (see Figure 3).

V. RUNNING EXPERIMENTS WITH SFIT AND JADA

The running experiments follow the methodology summarized in Figure 4. 15 experiments have been devised to evaluate the robustness of Jada operations manipulating the tuple spaces. These experiments are summarized below:

- Experiment 1: public void out (Object item)
- Experiment 2: public void out (Object objects [], int n_objects)
- Experiment 3: public Object in (Object match)
- Experiment 4: public Object in (Object match [], int n_objects)
- Experiment 5: public Object in (Object match [], int n_objects, long timeout)
- Experiment 6: public Object in (Object match [], int n_objects, long timeout)
- Experiment 7: public Object in_nb (Object match)
- Experiment 8: public Object in_nb (Object match [], int n_objects)
- Experiment 9: public Object read (Object match)
- Experiment 10: public Object read (Object match, long timeout)
- Experiment 11: public Object read (Object match [], int n_objects)
- Experiment 12: public Object read (Object match [], int n_objects, long timeout)
- Experiment 13: public Object read_nb (Object match)
- Experiment 14: public Object read_nb (Object match [], int n_objects)
- Experiment 15: public Object read (object match) with max possible test cases of 2500.

With the exception of Experiment 15 where 2500 test cases were used, at least 500 test cases have been employed in each experiment. The complete discussion of each experiment is beyond the scope of this paper (see reference [3] for details), but the summary of the experimentations will be highlighted in the next section.

VI. LESSONS LEARNED

As discussed earlier, the main issues under consideration in this section relates to the applicability and suitability of SFIT as a software fault injection tool as well as on the robustness assessment of Jada.
Applicability and Suitability of SFIT

The fact that SFIT can seamlessly test all the relevant Jada methods is a positive indication of its applicability. The features offered by SFIT appear sufficiently complete in order to permit robustness testing of Java COTs utilizing the fault injection technique. The injection process in SFIT is also highly automated whereby the test engineer can define up to 2500 test cases in a single experiment. This feature is helpful to relieve the test engineer from the mundane tasks inherent in the testing process.

The ability to keep historical data is also another significant feature in of SFIT. This feature permits offline conformance analysis of the test results as well as allows test archive to be setup for traceability purposes.

The fact that SFIT can interrogate a class for method interfaces (i.e. through the SFIT class inspector) is also useful when no source code for that class is available. In this way, robustness testing is also possible using SFIT for classes (or even COTs) in the absence of source codes.

Concerning comparison with other tools, SFIT approach is similar to JACA [21] in the sense that JACA also uses computational reflection in order to inject faults in a Java program. At a glance, JACA appears to have all the features of SFIT. Nevertheless, a closer look reveals that, unlike SFIT, JACA requires that the test engineer who performs the testing have substantial knowledge of Java in order to undertake the injection process, that is, in order to manually write the test driver program. In SFIT, the driver code for are automatically generated and executed in a single-click of a button. As discussed earlier, the injection process in SFIT is also highly automated allowing 2500 test cases to be injected at a particular instant (tested in experiment 15). As such, SFIT can be seen as offering a high level of abstraction for robustness testing of COTs.

Although complements of each other, one obvious limitation of SFIT is the fact that the loader and generator, fault injector, and the inspector are not integrated together (i.e. they are composed of two separate entities). The fact that the test engineer needs to run two separate programs (i.e. one for inspecting the relevant class and one for testing that class) make SFIT an inconvenience tool to use. This issue is considered an urgent matter in the future version of SFIT in order to qualify SFIT as a complete testing suite.

Robustness Assessment of Jada

Referring to all of the experiments undertaken, a number of observations can be made on Object Space class of Jada. It seems that all the methods behave as expectation when the classes such as String, Integer and Float are being used as the passing parameter. However, when the Long, Double, and user defined class are used, most methods fails to respond properly. For example, in experiment involving the method public void out (Object item), the method unexpectedly blocks when a Long, Double or a user defined class is used as the passing parameter for the variable item. Similarly, in experiment 2 (involving the method public out (Object [] item, int n_objects), the methods also blocks when a Long, Double or a user defined class is used as the passing parameter for the array item. In fact, this observation is true to other experiments as well.

Although defined as objects, the fact that only String, Integer and Float are supported as the valid passing parameters for the object variable in all the methods of the Object Space class raises an issue relating the usefulness of Jada. At a glance, it may appear that Jada implementation might not be sufficiently extensive for manipulating distributed shared memory. Nevertheless, a counter argument suggests that ad hoc approach may be adopted in order to simulate the use Long, Double and user defined class as passing parameter, for instance, by representing the required passing parameter (e.g. item) as String. One known extension of Jada addressing this issue does exist, solving this problem by creating a form tuple that can hold any object types [19]. In this manner, matching rule for that item in the tuple space can also be simplified.

Testing Jada Object Space class with values more the allowable boundary also causes problems in Jada. For instance, when manipulating extreme operation on Java predefined Integer.MIN_VALUE, Integer.MAX_VALUE, Float.MAX_VALUE, and Float.MIN_VALUE or any of their combination in the passing parameters, there is a tendency for all the methods to hang (and never return). This can be seen, for example, in experiment 5 involving the method public object in (object item, long timeout). Here, when the variable timeout is given a boundary value, the method unexpectedly hangs. Similar observation can be seen in other methods, for example, involving experiment 14. Here, the method under testing is given as public Object read_nb (Object match [], int n_objects). If the variable n_objects uses boundary value (and out of range value) the method also unexpectedly hangs.

Another observation worth mentioning is on the Jada Object Space methods involving array values as in experiment 8. In this case, the method is defined as public object in_nb (Object [] match, int n_objects). From the Jada specification, the number of values defines in the array of object match must ideally tally with the values of n_objects. However, our observation indicates that Jada is flexible enough to accept any values of n_objects and yet still gives the expected result provided that the values are within range (see the previous paragraph). Again, similar observation can be seen in all other methods involving array values.

Overall, while useful for manipulating distributed shared memory, Jada appears to be unsuitable for highly available and safety critical applications. As seen above, Jada lacks robustness, that is, it always fails to behave accordingly when unsupported or out of range input values are used.
An application employing COTs may be correct in many occasions, but may fail in just one occasion. In a safety-critical application, such a failure may not be tolerable as lives and fortunes could be lost as a result. Thus, development of a fault injection tool, like SFIT, is essential for assessing an application’s robustness as well as its dependability issues.

REFERENCES


