A Tool for Automated Test Data Generation (and Execution) Based on Combinatorial Approach

Kamal Zuhairi Zamli, Nor Ashidi Mat Isa, Mohamed Fadel Jamil Klaib and Siti Norbaya Azizan
School of Electrical and Electronic Engineering
Universiti Sains Malaysia, Engineering Campus, 14300 Nibong Tebal, Penang, Malaysia
eekamal@eng.usm.my

Abstract

Covering as much as 40 to 50 percent of the development costs and resources, software testing is an integral part of the software development lifecycle. Despite its importance, current software testing practice lacks automation and is still primarily based on highly manual processes from the generation of test cases (i.e. from the specifications documents) up to the actual execution of the test. These manually generated tests are sometimes executed using ad hoc approach, typically requiring the construction of a test driver for the particular application under test. The construction of a test driver is tedious, error prone, and cumbersome process, as it puts extra burden to test engineers. In addition, test engineers are also under pressure to test increasing lines of code in order to meet market demands and deadlines for more software functionalities. To attain the required level of quality and maintain acceptable test coverage, test engineers need to consider a significantly large number of test cases. Many combinations of possible input parameters, hardware/software environments, and system conditions need to be tested and verified against for conformance based on the system’s specification. Often, this results into combinatorial explosion of test cases. Addressing some of the aforementioned issues, this paper describes a new automated and combinatorial software testing tool, called JTst, based on the use of Java technology. The main novel features of JTst are the fact that it permits combinatorial test data generation and automated concurrent execution, enabling higher product quality at lower testing costs.

1. Introduction

Software testing relates to the process of executing a program or system with the intent of finding errors [17]. Although an integral part of software development lifecycle (i.e. covering as much as 40 to 50 percent of the development costs and resources [5]), current software testing practice is still primarily based on highly manual processes from the generation of test cases (i.e. from the specifications documents) up to the actual execution of the test. These manually generated tests are sometimes executed using ad hoc approach, typically requiring the construction of a test driver for the particular application under test. The construction of a test driver is tedious, error prone, and cumbersome process, as it puts extra burden to test engineers.

In addition, test engineers are also under pressure to test increasing lines of code in order to meet market demands and deadlines for more software functionalities. To attain the required level of quality and maintain acceptable test coverage, test engineers need to consider a significantly large number of test cases. Many combinations of possible input parameters, hardware/software environments, and system conditions need to be tested and verified against
for conformance based on the system’s specification. Often, this results into combinatorial explosion of test cases.

As an illustration, consider the option dialog in Microsoft Excel software (see Figure 1 in the next page). Even if only View tab option is considered, there are already 20 possible configurations to be tested. With the exception of Gridlines color which takes 56 possible values, each configuration can take two values (i.e. checked or unchecked). Here, there are $220 \times 56$ (i.e. 58,720,256) combinations of test cases to be evaluated. If the time required for one test case is 5 minutes, then it would require nearly 559 years to complete all tests merely for View tab option. Given the limited time and resources, the research question now is what are the smaller optimum sets of test cases to be used for testing?

![Figure 1. Option Dialog in Microsoft Excel](image)

Addressing some of the aforementioned issues, this paper describes a new automated and combinatorial software testing tool, called JTst, based on the use of Java technology. The main novel features of JTst are the fact that it permits combinatorial test data generation and automated concurrent execution, enabling higher product quality at lower testing costs.

2. Introducing JTst

Our work is developed based on our earlier tool, called SFIT [1-3][19-21]. Despite being useful, SFIT is too rudimentary and primitive with limited automation features. Building and complementing from these experiences, we have opted to develop JTst [22].

A key idea in JTst is the fact that tests are performed based on the values of the interface parameters (i.e. on the data types of the parameter lists) and not on the behavioral specifications. Thus, JTst is suitable for performing automated black box testing particularly involving commercial-of-the-shelves-components (COTs) where no source code and design are usually available apart from some user documentations.
Another key idea in JTst is that the test cases can be combinatorially generated based on some base test cases, the concept borrowed from Ammann and Offutt [4]. Ideally, these base test cases can either be collected from known combination of input variables that causes failures to the module under test from real systems in various application domains [14] or from the program specifications (see Figure 2 and 3 given earlier). As far as JTst is concerned, the base test cases can be both actual test values to be used for test execution or some symbolic data (e.g. On, Off, Left, Right etc). It should be noted that symbolic
combinatorial data values can be used to systematically guide acceptance tests [7] based on some user configuration.

Kuhn and Okum [15] suggest that from empirical observation, the number of input variables involved in software failures is relatively small (i.e. in the order of 3 to 6), in some classes of software. If t or fewer variables are known to cause fault, test cases can be generated on some t-way combinations (i.e. resulting into a smaller set of test cases for consideration). Empirical evidence suggests that in some software implementation, the execution of these smaller subset of test cases can typically uncover 50% to 75% of faults in a program [10][16][18].

Concerning its implementation, JTst consists of a number of related components consisting of the Class Inspector, Test Editor, Test Combinator, Automated Loader and Data Logger (see Figure 4). The working context (i.e. process flow and the main user interface) for these components are captured in the JTst main interface.
The functionalities for each of these components will be discussed next.

**Class Inspector**

In the absence of source codes, the class inspector can optionally be used to obtain details information of the Java class interface (see Figure 6). To do so, the class inspector exploits Java Reflection API in order interrogate Java classes for method interfaces including public, private, and protected ones. This information can be used to set up base test cases.

![Figure 6. JTst Class Inspector](image)

**Test Editor**

As its name suggests, the test editor allows the user to edit and setup the test cases (i.e. including the base test cases) in a JTst fault file. Here, the test case definitions (up to 15,000 test cases) can be straightforwardly described using JTst predefined markup language in order to facilitate the parsing of test case data values for automatic recombination (see Figure 7). As noted earlier, to allow execution, test data must use actual values and not symbolic data.

```plaintext
@FaultFile
/////////////////////////////////////////
Common Header Definition
/////////////////////////////////////////
classname  : adder
methodname : add_basictypes_integer
specifier   : private
paramtypes   : 2
returntype  : int
parameter   : partypes[0]=Integer.TYPE
parameter   : partypes[1]=Integer.TYPE
```
Body - Test case 0
arglist[arglist[0]]=new Integer(Integer.MAX_VALUE)
arglist[arglist[1]]=new Integer(Integer.MAX_VALUE)

Body - Test case 1
arglist[arglist[0]]=new Integer(Integer.MIN_VALUE)
arglist[arglist[1]]=new Integer(Integer.MIN_VALUE)

Figure 7. Sample Fault File

Test Combinator
Test combinator manipulates the base test case in order to generate combinatorial test cases (see Figure 8). The combinator interface can be seen in Figure 9.

Figure 8. Test Combinator

Figure 9. JTst Combinator Interface
To illustrate the functionality of the JTst test combinator, consider the following running example. Let a system or a method M1 has four possible inputs variables X = \{A,B,C,D\}. For simplicity sake, let us assume that the base test cases for M1 have been identified in Table 1.

### Table 1. Base Data Values for Method M1

<table>
<thead>
<tr>
<th>Input Variables</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Values</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>99</td>
<td>C1</td>
<td>Large</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>2000</td>
<td>C3</td>
<td>Small</td>
<td></td>
</tr>
</tbody>
</table>

The test cases data can be viewed as a matrix with specified columns and rows. Here, one can traverse one column at a time (called sensitivity variable in JTst implementation), whilst keeping other column fixed to permutate and generate new test cases from existing ones. For instance, Table 2 depicts the possible combinatorial test cases with the sensitivity variable set to A.

### Table 2. Base and Combinatorial Data Values for Method M1 With Sensitivity variable = A

<table>
<thead>
<tr>
<th>Input Variables</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Values</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>99</td>
<td>C1</td>
<td>Large</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>2000</td>
<td>C3</td>
<td>Small</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Combinatorial Values</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>2000</td>
<td>C3</td>
<td>Small</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>99</td>
<td>C1</td>
<td>Large</td>
<td></td>
</tr>
</tbody>
</table>

Apart from permitting sensitivity column to be a single column, JTst test combinator also allows the sensitivity column to be a combination of 2 or more columns (t-way combinations). Here, the algorithm employed by JTst test combinator is based on the modified greedy algorithm as discussed in [6][8][18]. In the case of t-way combinations, JTst test combinator ensures that for each t-way definition of input variables, at least one test case will be randomly generated to cover the required t-way combinations. Although not discussed in this paper, it should be noted here that because the generation of t-way combinations of test cases is random, it is highly unlikely that the same sets of test cases are generated even during multiple t-way combinations exercise using the same fault file.

To illustrate the greedy algorithm implemented as part of JTst test combinator, Table 3 depicts the possible combinatorial test cases generated with sensitivity variable set to triplets (ACD) as compared to Table 4 with sensitivity variable set to all variables. Obviously, less number of test cases can be seen in the case when t-way combinations are used (see Table 3 and Table 4 for comparison). In fact, the number of test cases would be even smaller given the smaller t-way combination (e.g. AB).

**Automated Loader**

JTst automated loader have two main responsibilities. The first responsibility is to iteratively parse the test cases (defined in JTst fault files), and automatically generates and
executes the appropriate Java code driver. The second responsibility is to manage concurrent execution of test cases. Here, the JTst automated loader is actually consists of two sub-components: Loader and Concurrent Manager (see Figure 10).

**Table 3.** Base and Combinatorial Data Values for Method M₁ With Sensitivity variable = t-way (ACD)

<table>
<thead>
<tr>
<th>Base Values</th>
<th>Input Variables</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>99</td>
<td>C₁</td>
<td>Large</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td>2000</td>
<td>C₃</td>
<td>Small</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Combinatorial Values</th>
<th>Input Variables</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>2000</td>
<td>C₃</td>
<td>Small</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td>99</td>
<td>C₁</td>
<td>Large</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>2000</td>
<td>C₁</td>
<td>Large</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td>99</td>
<td>C₃</td>
<td>Small</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>99</td>
<td>C₁</td>
<td>Small</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td>2000</td>
<td>C₁</td>
<td>Small</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>99</td>
<td>C₁</td>
<td>Small</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td>99</td>
<td>C₃</td>
<td>Large</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.** Base and Combinatorial Data Values for Method M₁ With Sensitivity variable = All (ABCD)

<table>
<thead>
<tr>
<th>Base Values</th>
<th>Input Variables</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>99</td>
<td>C₁</td>
<td>Large</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td>2000</td>
<td>C₃</td>
<td>Small</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Combinatorial Values</th>
<th>Input Variables</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>99</td>
<td>C₁</td>
<td>Small</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>99</td>
<td>C₁</td>
<td>Large</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>99</td>
<td>C₃</td>
<td>Small</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>2000</td>
<td>C₁</td>
<td>Large</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>2000</td>
<td>C₃</td>
<td>Large</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>2000</td>
<td>C₃</td>
<td>Small</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₁</td>
<td>99</td>
<td>C₁</td>
<td>Large</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T₂</td>
<td>99</td>
<td>C₃</td>
<td>Large</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Concurrent execution is achieved in JTst through a well-known token passing algorithm. Here, a token is always associated for each concurrent execution. Once all the tokens have been used up, no further concurrent execution is allowed until one or more concurrent executions have terminated (i.e. release its token). Here, the number of defined tokens in the pool of tokens can be dynamically configured through the user interface provided should the need arise (see Figure 11). Obviously, the more tokens are allowed, the slower the test case executions will be. In the current version, JTst has been tested to execute up to 10 concurrent execution of test cases, each of which can support up to 15,000 test cases per execution (see Figure 12).
Pool of tokens assigns token to the JTst loader checks-out token if it is available JTst Loader returns token upon completion JTst Concurrent Manager checks-out token if it is available assigns token to the JTst loader Pool of tokens regenerate tokens upon request

Figure 10. JTst Automated Loader & Token Passing Mechanism

Figure 11. JTst Concurrent Setting for Token Generations

Figure 12. JTst Concurrent Execution of Test Cases

Data Logger/Log
Data logger is a text browser equipped with customized search capability to perform offline analysis of the output captured by the automated loader in the form of logs. Here, logs are special database storing the input output behavior of the module under test (MUT). If the specification of the MUT method exists, conformance analysis can be made using this database.

Nevertheless, in the absence of source codes and formal specification, the trivial outcome of “doesn’t hang and doesn’t crash” suffices to determine whether MUT passes the minimum testing requirement. In this case, the operating system can be queried if the test program terminates abnormally and a process monitor can be employed to detect hangs. A key issue here is the fact that the faults can always be reproducible with the same sets of inputs.

3. On JTst Case Study – Jada

In order to evaluate its features, there is obviously a need to subject JTst to a case study problem. Here, Jada [9], 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 library freely accessible for download in the internet. Although an overview of its methods and functionalities are given in the documentation (see reference [9]), Jada does not come with complete source codes.

The focus of this case study is two folds. The first is to evaluate the applicability of JTst, and the second is to perform robustness assessment of Jada.

In order to perform robustness assessment of Jada, there is a need to understand how Jada works. Jada is actually the Java implementation of Linda. Linda is a parallel programming model that was proposed by David Gelernter to solve the problem of programming parallel machines [11]. Tuple space, essentially a distributed shared memory, is the 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.
Although testing all the classes defined in Jada is equally beneficial (see Figure 13), 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 JTst Class Inspector.
4. Running Experiment with Jada

16 experiments have been devised to evaluate the robustness of Jada primitives for manipulating the tuple spaces. With the exception of Experiments 15 and 16 where 15,000 test cases were used (i.e. combinatorially generated), 40 base case values were identified based on equivalent class and boundary value analysis. Using these base values, combinatorial values were generated using sensitivity = all variables. It should be noted that the total combinatorial test cases are not uniform in each experiment (i.e. depending on the parameter values).

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, 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 15,000 test cases
- Experiment 16: public Object read (object match) with 5 concurrent test cases each with 15,000 test cases

The complete description of these experiments is beyond the scope of this paper. Interested readers are referred to our earlier work described in [2][21]. Nevertheless, the summary of the experiments will be highlighted in the next section.

5. Lessons Learned

As discussed earlier, the main issues under consideration in this section relates to the applicability JTst as well as on the robustness assessment of Jada.

Applicability of JTst

The fact that JTst can seamlessly test all the relevant Jada methods is a positive indication of its applicability. The features offered by JTst appear sufficiently complete in order to permit robustness testing of Jada.

Test automation is the key feature of JTst. As a comparison, Figure 14 depicts traditional testing versus automated testing provided by JTst.
Start Testing Java Code

**...the old way**
- Must gain access to source code
- Develop test cases for each method in a class file
- Develop Java test drivers for each test case in every class
- Build the test drivers
- Compile time errors?
- Execute each test driver sequentially
- Runtime errors?
- Manually analyse results

**...using JTst**
- Gain access to either source code or class files
- Develop test cases in the fault file
- Optionally use recombinator to permute test cases
- Execute test cases defined in fault file automatically in parallel
- Analyse categorised log files

JTst Automation

Run tests

Conformance analysis

Java Code Tested

Figure 14. Traditional vs JTst Automated Testing
As the figure illustrates, a number of saving in terms of test driver generations and feedback loops can be observed. In this respect, JTst can be compared to JUnit [12]. In order to use JUnit, programmers would still be required to write test drivers. In JTst, no programming would be required as the test drivers are automatically generated by the JTst automated loader. Such a feature is helpful to relieve the test engineers from the mundane tasks inherent in the testing process.

The execution of the test cases can also be concurrent in JTst. In experiments 15 and 16, we have managed to demonstrate this aforementioned feature. Obviously, this feature can significantly alleviate the execution bottlenecks of test cases. Theoretically, JTst can support unlimited number of concurrent executions of test cases. Nevertheless, our experience with JTst indicates that allowing more than 10 concurrent executions can seriously slows down overall execution of each concurrent process.

The test combinator is also another key feature of JTst. The test combinator is useful to generate t-way combinatorial test cases that can be used to locate unknown faults. Combinatorial techniques are known to cause test data explosion [6]. Suppose that test input variables are 10, each had 3 values say 0, 1, and 2. Then, there are $3^{10} = 59,049$ possible parameter combinations. Now, if the test input variables are increased to 13, then there are $3^{13} = 1,594,323$ possible parameter combinations. This simple example illustrates that a small change in the input parameters can cause massive increase in the parameter combinations. Exhaustive testing for all combinations is desirable, of course, in the expense of costs. For these reasons, partitioning into t-way combinations as implemented in JTst can offer trade-offs between testing efforts, costs, and quality assurances. As discussed earlier, JTst ensures that at least one test case will be generated to cover the required t-way combinations.

Comparatively, JTst can be seen as a complement to AETG [6] and IPO [18]. Similar to AETG and IPO, t-way test cases can be combinatorially generated for analysis (i.e. both as symbolic data and actual data values). Concerning combinatorial generation based on symbolic data values, JTst, like AETG and IPO, can even be used to suggest systematic test values for embedded system modules involving hardware, that is, through the t-way combination features. To illustrate this point, consider a hardware product with 20 on/off switches. To test all possible combination would require $2^{20}$ test cases. If the time required for one test case is 5 minutes, then it would take nearly 10 years for a complete test. As discussed earlier, execution of a smaller set of test cases based on t-way combinations can significantly reduces the test data space, in turn, reducing costs and efforts.

Unlike AETG and IPO, JTst also supports concurrent execution (and automation) of the generated test cases for actual data values. As the work is still progressing, we are still investigating ways to improve the modified greedy algorithm used in JTst. For this reason, no performance measure is yet available.

The work context exemplified by JTst components and adopted as the JTst user interface can also be helpful to put the testing process into perspective (see Figure 5). In this manner, a subtle guidance can be given to the test engineers to assist them to understand how each piece fits together into the whole picture.

The fact that JTst can interrogate a class for method interfaces (i.e. through the JTst class inspector) is also useful when no source code for that class is available. In this way, unit testing is also possible using JTst for classes (or even COTs) in the absence of source codes.
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 method public void out (Object item), the method unexpectedly blocks when a Long, Double or a user defined class are 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 [13]. 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.
6. Conclusion

In conclusion, this paper has described the design and implementation of an automated and combinatorial Java unit testing tool called JTst. The prototype implementation tool has been completed and used to perform robustness assessment of Linda implementation called Jada. As part of our on-going work, we are currently implementing a parallel version of JTst to support execution over heterogeneous distributed environment such as the GRID.

7. Acknowledgements

This research is partially funded by the eScience Fund – “Development of a Software Fault Injection Tool to Ensure Dependability of Commercial-off-the-Shelf Components (COTs) for Embedded System Applications” and the USM Short Term Grants – “Development of An Automated Unit Testing Tool for Java Program”.

8. References


Authors

Kamal Zuhairi Zamli
Received a B.Sc. degree in Electrical Engineering from WPI, USA, 1992, and MSc degree (Real Time Software Engineering) from Universiti Teknologi Malaysia, 2001, and PhD degree in Software Engineering from University of Newcastle upon Tyne, UK, 2003. His research interests include Software Engineering, Testing Automation, and Algorithm design.

Nor Ashidi Mat Isa
Received a B.Eng in Electrical Engineering from Universiti Sains Malaysia in 2000, and PhD in Image Processing and Neural Networks from the same university in 2003. His research interests include Image Processing, Neural Networks, and Algorithm design.

Mohamed Fadel Jamil Klaib
Received a B.Eng in Electronic Engineering from Al-Quds University in Jerusalem in 2002, and MSc in Computer Engineering from Near East University, Cyprus, 2004. Currently, a PhD candidate in the area of testing automation and combinatorial testing.

Siti Norbaya Azizan
Received a B.Eng in Electrical Engineering from Universiti Sains Malaysia in 2005. Currently undertaking MSc by research in the area of combinatorial testing.