Proceedings of the Third IASTED International Conference
ADVANCES IN COMPUTER SCIENCE AND TECHNOLOGY
April 2–4, 2007, Phuket, Thailand
ISBN CD: 978-0-88986-656-0

DESIGNING A COMBINATORIAL JAVA UNIT TESTING TOOL

Kamal Zuhairi Zamli, Nor Ashidi Mat Isa, Mohammed Fadel Jamil Klaib, Siti Norbaya Azizan
School of Electrical and Electronic Engineering,
Universiti Sains Malaysia, Engineering Campus,
14300 Nibong Tebal, Pulau Pinang,
Malaysia
Email: {eekamal, ashidi}@eng.usm.my

ABSTRACT
Software testing relates to the process of executing a program or system with the intent of finding errors. 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. Although the emergence of helpful automated testing tools in the market is blooming, their adoptions are lacking as they do not adequately provide the right level of abstraction and automation required by test engineers.

JTst is a Java based automated unit testing tool that addresses some of the aforementioned issues. The main novel features of JTst are the fact that it permits combinatorial test case generations as well as automated and concurrent execution of test cases for Java classes, enabling higher product quality at lower testing costs.

KEY WORDS
Test Automation, Combinatorial Testing Tool

1. Introduction

Computing technology has gone a long way since the first Babbage computer. Today, many chores that were once manual have been taken over by computers. Factories use computers to control manufacturing equipments. Electronics manufacturing use computers to test everything from microelectronics to circuit card assemblies. The automation provided by computers avoids the errors that humans make when they get tired after multiple repetitions.

Software testing, as a subset of software engineering, is one area which can also benefit from automation (*i.e.* programmatic generation and execution of software test data). Although an integral part of software development lifecycle (i.e. covering as much as 40 to 50 percent of the development costs and resources [2]), 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 especially if the test cases are significantly large.

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, test engineers need to maintain high test coverage, typically requiring large number of test cases per module. While there are significant proliferations of helpful testing tool support in the market, much of which runs sequentially and does not adequately provides the right level of abstraction and automation required by test engineers.

In order to address some of the aforementioned issues, this paper describes a new automated 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 case generations as well as automated and concurrent execution of test cases for Java classes, enabling higher product quality at lower testing costs.

This paper is organized as follows. Section 2 outlines the scope of JTst along with all of its components. Section 3 discusses the JTst case study. Section 4 illustrates our running experiment with JTst. Section 5 discusses our overall assessments and lessons learned. Finally, section 6 gives our conclusion as well as outlines the possible future work.

2. Introducing JTst

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 [1]. 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 [9] or from the program specifications. Kuhn and Okum [10] 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 all *t-way* combinations of discrete values (i.e. using equivalent class or boundary value analysis for continuous value variables). Empirical evidence suggests that in some software implementation, the execution of these test cases can typically uncover 50% to 75% of faults in a program [6][11][12].

Considering ideas discussed in previous paragraphs, JTst has been implemented with a number of components consisting of the Class Inspector; the Test Editor; the Test Combinator; the Automated Loader; and the Data Logger/Log (see Figure 1). 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. 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 (discussed earlier).

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 and execution (see Figure 2).

```
@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 2 – Sample Fault File

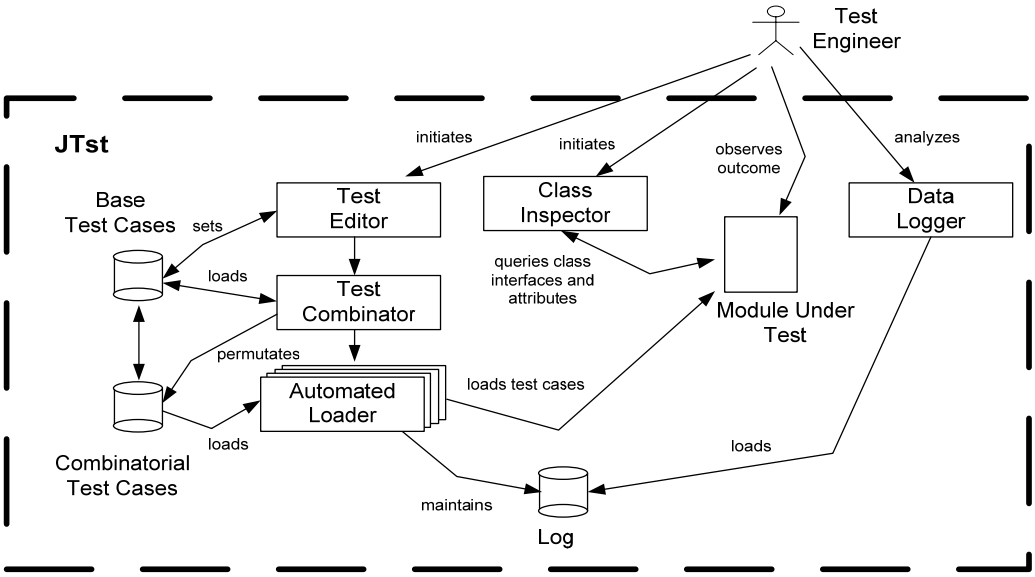

Figure 1 – JTst Main Components

Test Combinator

JTst test combinator manipulates the base test case in order to generate combinatorial test cases. To illustrate the functionality of the JTst test combinator, consider the following running example. Let a method M_1 has four inputs variables X = {A,B,C,D}. For simplicity sake, let us assume that the base test case for M_1 has been identified in Table 1.

Table 1. Base Data Values for Method M_1

	Input Variables			
Base Values	A	B	C	D
	T1	99	C1	Large
	T2	2000	C3	Small

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. Table 2 depicts the possible combinatorial test cases with the sensitivity variable set to A.

Table 2. Base and Combinatorial Data Values for Method M_1 With Sensitivity = A

	Input Variables			
Base Values	**A**	**B**	**C**	**D**
	T1	99	C1	Large
	T2	2000	C3	Small
Combinatorial Values	T1	2000	C3	Small
	T2	99	C1	Large

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). The algorithm employed by JTst test combinator is based on the modified *greedy algorithm* as discussed in [3][4][12]. Here, JTst test combinator ensures that at least one test case will be randomly generated to cover each of 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 *t-way* combinations involving triplets (ACD).

Table 3. Base and Combinatorial Data Values for Method M_1 With Sensitivity = Random *t-way* (ACD)

	Input Variables			
Base Values	**A**	**B**	**C**	**D**
	T1	99	C1	Large
	T2	2000	C3	Small
Combinatorial Values	T1	2000	C1	Large
	T1	99	C1	Small
	T1	99	C3	Large
	T1	99	C3	Small
	T2	99	C1	Large
	T2	2000	C1	Small
	T2	99	C3	Large
	T2	99	C3	Small

Automated Loader

JTst automated loader have two main responsibilities. The first responsibility is to iteratively parse the JTst *fault files*, and automatically generates and executes the appropriate Java code driver.

The second responsibility is to manage concurrent execution of test cases. 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. Obviously, the more tokens are allowed, the slower the test case executions will be. In the current version, JTst has been tested to concurrently execute up to 15,000 test cases per execution.

Data Logger/Log

Data logger is a text browser with customised search capability to perform offline analysis of the output captured by the automated loader (see Figure 1) 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

In order to evaluate its features, there is obviously a need to subject JTst to a case study problem. Here, Jada [5], 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 [5]), 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 [7]. 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.

- class java.lang.Object
  - class jada.FloatSerializer (implements jada.JadaSerializer)
  - class jada.IntegerSerializer (implements jada.JadaSerializer)
  - class jada.Jada
  - interface jada.JadaItem
  - interface jada.JadaNetConst
  - class jada.net.JadaNetIO
  - interface jada.JadaSerializer
  - class jada.net.ObjectServer (implements java.lang.Runnable, jada.JadaNetConst, jada.net.JadaNetOpcodes)
  - class jada.ObjectSpace
    - class jada.net.ObjectClient (implements jada.JadaNetConst, jada.net.JadaNetOpcodes)
  - class jada.StringSerializer (implements jada.JadaSerializer)
  - class java.lang.Throwable
    - class java.lang.Exception
      - class jada.JadaException
        - class jada.JadaFormatException
        - class jada.JadaIOException
        - classjada.JadaItemIsNotStreamableException
  - class jada.Tuple (implements jada.JadaItem)
    - class jada.TupleString
  - class jada.VectorSerializer (implements jada.JadaSerializer)

Figure 4 – Jada Class Hierarchy

Although testing all the classes defined in Jada is equally beneficial (see Figure 4), 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 Experiments 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. Nevertheless, the summary of the experiments will be highlighted in the next section.

5. Lesson Learned

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. The JTst test combinator is also useful to generate *t-way* combinatorial test cases that can be used to locate faults. The execution of the test cases can also be concurrent and automated. In experiments 15 and 16, we have managed to demonstrate this aforementioned feature. Obviously, this feature is helpful to relieve the test engineers from the mundane tasks inherent in the testing process.

Combinatorial techniques often cause test data explosion. 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. Undoubtedly, 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 [3] and IPO [12]. Similar to AETG and IPO, *t-way* test cases can be combinatorially generated for analysis (i.e. both as actual and symbolic data values). 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 comparative performance measure is yet available.

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 [8]. 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 operation on Java predefined

MIN and MAX value (i.e. 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 first prototype implementation tool has been completed and used to perform robustness assessment of Linda implementation called Jada. Currently, our work is still progressing to improve JTst combinatorial algorithm (discussed earlier) as well as to support parallel execution over heterogeneous distributed environment.

Acknowledgement

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".

References

[1] P.E. Ammann and A.J. Offutt, Using Formal Methods to Derive Test Frames in Category-Partition Testing. *Proc. of the 9th Annual Conference on Computer Assurance (COMPASS'94)*, IEEE CS Press, June 1994, pp. 69-80.

[2] B. Beizer. *Software Testing Technique*. Thomson Computer Press, 2nd Edition, 1990.

[3] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton, The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering 23(7)*, July 1997, pp. 437-444.

[4] C.J. Colbourn, M.B. Cohen, and R.C. Turban, A Deterministic Density Algorithm for Pairwise Interaction Coverage. *Proc. of the IASTED Intl. Conference on Software Engineering* (February 2004), pp. 345–352.

[5] P. Ciancarini, and D. Rossi, Jada: A Coordination Toolkit for Java. Technical Report UBLCS-96-15, Department of Computer Science, University of Bologna, Italy, 1997.

[6] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, Model-Based Testing In Practice. *Proc. of the 21st Intl. Conf. on Software Engineering*, ACM Press, May 1999, pp. 285–294.

[7] D. Gelernter, Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems, 7* (1), 1985, pp. 80-112.

[8] D. Kuhni. APROCO: A Programmable Coordination Medium. Diploma Thesis, Faculty of Science, University of Berne, Switzerland, 1998.

[9] D.R. Kuhn and M.J. Reilly, An Investigation of the Applicability of Design of Experiments to Software Testing. *Proc. of the 27th NASA/IEEE Software Engineering Workshop*, IEEE CS Press, December 2002, pp. 69-80.

[10] D.R. Kuhn and V. Okun, Pseudo-Exhaustive Testing for Software. *Proc. of the 30th NASA/IEEE Software Engineering Workshop*, IEEE CS Press, April 2006.

[11] D.R. Kuhn and D.R. Wallace, Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering 30*(6), June 2004, pp. 418-421.

[12] K.C. Tai and Y. Lei, A Test Generation Strategy for Pairwise Testing. *IEEE Transactions on Software Engineering 28*(1), January 2002, pp. 109-111.