American Journal of Applied Sciences 5 (2): 77-82, 2008
ISSN 1546-9239
© 2008 Science Publications

JTst – An Automated Unit Testing Tool for Java Program

Kamal Zuhairi Zamli and Nor Ashidi Mat Isa
School of Electrical and Electronic Engineering,
Universiti Sains Malaysia, Engineering Campus, 14300, Nibong Tebal, Penang, Malaysia

Abstract: Software testing is an integral part of software development lifecycle. Lack of testing can often lead to disastrous consequences including lost of data, fortunes, and even lives. Despite its importance, current software testing practice lacks automation, and is still primarily based on highly manual processes from the generation of test cases 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 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 are the fact that JTst automates the test preparation activities, facilitates the test data generation through recombination, and allows concurrent execution of test data, in order to encourage higher product quality at lower testing costs.

Keywords: Automated Software Testing, Software Engineering

INTRODUCTION

In line with market demands and the need for technological innovations, designing and implementing a useful engineering product is ever growing in complexity. In order to alleviate such complexity, 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. Often, the automation provided by computers avoids the errors that humans make when they get tired after multiple repetitions.

The need for automation (*i.e.* programmatic generation and execution of software test data) is no exception in order to engineer a useful software engineering product, particularly to support software testing activities. Covering as much as 40 to 50 percent of the development costs and resources[1], current software testing practice is still primarily based on highly manual processes from the generation of test cases 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[1]. 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 an automated software testing tool, called JTst, based on the use of Java technology. The main aim of JTst is to automate test preparation activities, facilitate the test data generation through recombination, and allow concurrent execution of test data, in order to encourage higher product quality at lower testing costs.

The gradual shift toward software testing automation is not new. A number of tools do exist either commercially or as research prototypes. As far as Java is concerned, some of these tools are summarized below:

- Jaca[2] is a useful testing tool that permits testing of Java classes by corrupting the method interfaces and attributes. Jaca does not require the application's source code, but it needs the some information about the application such as class name and method interfaces.

Corresponding Author: Kamal Zuhairi Zamli, School of Electrical and Electronic Engineering, Universiti Sains Malaysia, Engineering Campus, 14300 Nibong Tebal, Penang, Malaysia, Tel: +604-5996003, Fax: +604-5941023

- JUnit[3, 4] is a testing tool used to write and run automated and repeatable tests. In JUnit, test engineer need to write a Java unit test case, essentially a collection of tests designed to verify the behavior of a single unit within a user program. The Java unit test case can then be automatically executed by the JUnit environment.
- FIONA[5] is a Java software testing tool for distributed applications. FIONA provides a Java Virtual Machine Tool Interface that enables the inspection and execution of faults of distributed application running in the Java Virtual Machine.
- Simple[6] is a functional testing tool that can be used to assess reliability, robustness and performance of a system as a whole. The aim of simple is to facilitate testing of Java classes used in safety critical applications.
- SoftTest[7] is a testing tool that is based on a predefined test plan. Based on test plan, SoftTest automatically insert and remove in executing code to carry out testing strategies.

Although useful, much of the aforementioned tools do not adequately give the right level of abstraction and automation as required by test engineers. A testing tool must not assume that the user has significant knowledge of Java in order to be able to use the tool (as required Jaca and JUnit). In fact, a helpful tool should be sufficiently high level to facilitate the testing process in the sense that test engineers need not need to do any coding whatsoever in order to perform the actual testing.

Additionally, test automation provided by the tools must be sufficiently intuitive for the test engineers to master. Providing some level of intuition is important to help junior engineers to grasp the testing work context particularly in terms of how each testing activity fits together in the whole picture.

In general, test automation can come in a number of forms. In a nut shell, the test automation should relieve the test engineer from the routine tasks of creating Java test drivers for execution. In addition, test automation should also facilitate the generation and execution of the actual test cases. Here, parallel execution of test cases can help to speed up the testing process. In this manner, test engineers can put significant focus on the job at hand (i.e. coming up with good test cases) and be released from manually writing test drivers.

Apart from the above requirements, test engineers are also burden with generating large data sets for testing purposes. Permitting recombination of test data from the existing data can also be useful to improve test coverage. Based on the aforementioned constraints and requirements as well as building and complementing from earlier work, it is the development of JTst, is the main focus of this paper.

INTRODUCING JTst

JTst consists of a number of related components consisting of the Class Inspector; the Test Editor; the Test Combinator; the Automated Loader; and the Data Logger/Log (see Fig. 2). The process flow for these components is captured as the user's work context within the JTst user interface. The functionalities for each of these components will be discussed next.

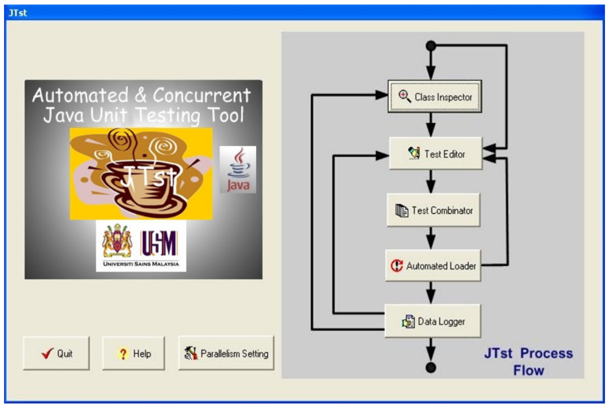
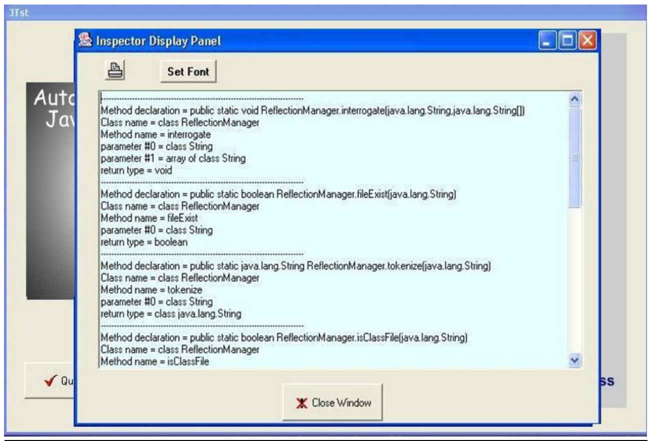


Fig. 1: JTst Workcontext

- Class Inspector – One useful feature of JTst is to allow unit testing in the absence of source codes. In this case, 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 (see Fig. 2). This information can be used to set up the test cases in the fault file (discussed next).



```
-----------------------------------------------------------------
Method declaration = public static void ReflectionManager.interrogate(java.lang.String,java.lang.String[])
Class name = class ReflectionManager
Method name = interrogate
parameter #0 = class String
parameter #1 = array of class String
return type = void
-----------------------------------------------------------------
Method declaration = public static boolean ReflectionManager.fileExist(java.lang.String)
Class name = class ReflectionManager
Method name = fileExist
parameter #0 = class String
return type = boolean
-----------------------------------------------------------------
Method declaration = public static java.lang.String ReflectionManager.tokenize(java.lang.String)
Class name = class ReflectionManager
Method name = tokenize
parameter #0 = class String
return type = class java.lang.String
-----------------------------------------------------------------
Method declaration = public static boolean ReflectionManager.isClassFile(java.lang.String)
Class name = class ReflectionManager
Method name = isClassFile
```

Fig. 2: Interrogating Java Classes in JTst

- Test Editor – 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 definition follows certain predefined formatting rules (shown in bold) in order to facilitate the parsing of data for automatic execution and recombination (see Fig. 3).

```
@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)

...............
```

Fig 3: Sample Fault File

- Test Combinator – Test combinator manipulates the user specified test cases as the *base test cases* in order to generate test data through recombination (see Fig. 4).

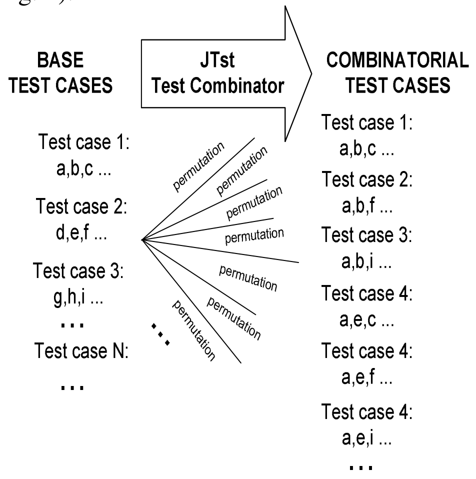


Fig. 4 : JTst Test Combinator

As a unique feature of JTst, this issue warrants further discussion here. 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 recombine and generate new test cases from existing ones.

The current JTst implementation provides two algorithms for recombination of test data. The first algorithm considers one parameter to be the sensitivity variable to be varied whilst the second algorithm considers all parameter to the sensitivity variable to be varied. To illustrate this, two examples will be shown here.

In each example, the following input data will be used (see Table 1). The rationale for using these data inputs stemmed from the fact that historically the same data inputs have been used by other researchers in the area. By adopting the same data inputs, objective comparison may be made amongst different algorithm implementations.

Table 1: Data Input

Parameter 1	Parameter 2	Parameter 3	Parameter 4
Netscape	Window	LAN	Local
IE	Macintosh	PPP	Networked
Other	Linux	ISDN	Screen

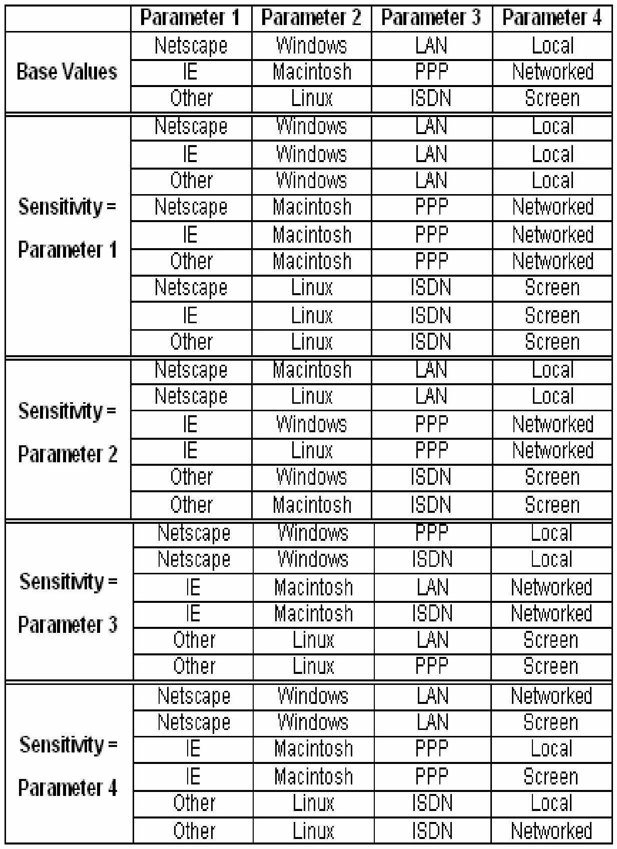
Applying the first algorithm with parameter 2 as the sensitive variable to be varied yields the following results.

Table 2: Output with Parameter 2 as sensitive variable

	Parameter 1	Parameter 2	Parameter 3	Parameter 4
Base Values	Netscape	Windows	LAN	Local
	IE	Macintosh	PPP	Networked
	Other	Linux	ISDN	Screen
Combinational Values	Netscape	Windows	LAN	Local
	Netscape	Windows	PPP	Local
	Netscape	Windows	ISDN	Local
	IE	Macintosh	LAN	Networked
	IE	Macintosh	PPP	Networked
	IE	Macintosh	ISDN	Networked
	Other	Linux	LAN	Screen
	Other	Linux	PPP	Screen
	Other	Linux	ISDN	Screen

Applying the second algorithm with all parameter as sensitive variables yields the following results.

Table 3: Output with All Parameters as sensitive variable



	Parameter 1	Parameter 2	Parameter 3	Parameter 4
Base Values	Netscape	Windows	LAN	Local
	IE	Macintosh	PPP	Networked
	Other	Linux	ISDN	Screen
Sensitivity = Parameter 1	Netscape	Windows	LAN	Local
	IE	Windows	LAN	Local
	Other	Windows	LAN	Local
	Netscape	Macintosh	PPP	Networked
	IE	Macintosh	PPP	Networked
	Other	Macintosh	PPP	Networked
	Netscape	Linux	ISDN	Screen
	IE	Linux	ISDN	Screen
	Other	Linux	ISDN	Screen
Sensitivity = Parameter 2	Netscape	Macintosh	LAN	Local
	Netscape	Linux	LAN	Local
	IE	Windows	PPP	Networked
	IE	Linux	PPP	Networked
	Other	Windows	ISDN	Screen
	Other	Macintosh	ISDN	Screen
Sensitivity = Parameter 3	Netscape	Windows	PPP	Local
	Netscape	Windows	ISDN	Local
	IE	Macintosh	LAN	Networked
	IE	Macintosh	ISDN	Networked
	Other	Linux	LAN	Screen
	Other	Linux	PPP	Screen
Sensitivity = Parameter 4	Netscape	Windows	LAN	Networked
	Netscape	Windows	LAN	Screen
	IE	Macintosh	PPP	Local
	IE	Macintosh	PPP	Screen
	Other	Linux	ISDN	Local
	Other	Linux	ISDN	Networked

As far as predicting the number of generated combinatorial values is concerned, one can use the following expression. In the case of one parameter as sensitivity variable, provided that all the base data values are unique, recombination can regenerate new test cases based on:

The number of generated test cases = n^2

where n = number of defined test cases

Referring to Table 2, the number of generated test cases are = $n^2 = 3^2 = 9$ test cases.

In the case of all parameters as sensitivity variable, provided that all base data values are unique, recombination can regenerate new test cases based on:

The number of generated test cases = $(p*n^2) - \phi$

where n = number of defined test cases

p = number of input parameters

ϕ = the number of repetitive values

= $n*(p-1)$

Referring to Table 3, the number of generated test cases are = $(p * n^2) - n* (p-1)$ or $(4*3^2) - 3(4-1) = 27$ test cases.

- 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 Fig. 5).

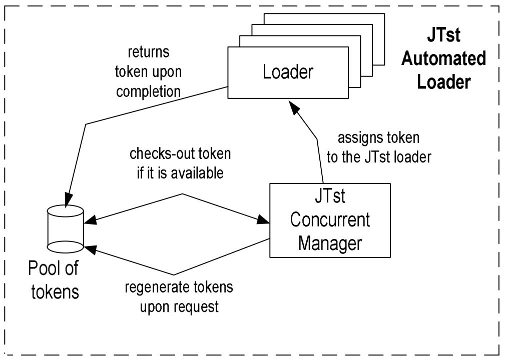
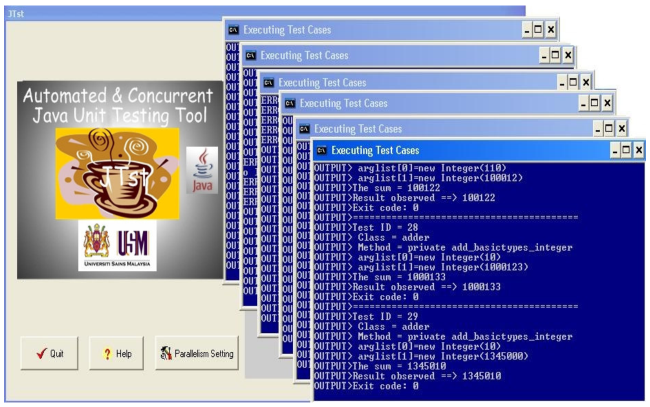

JTst Automated Loader
Loader
returns token upon completion
assigns token to the JTst loader
checks-out token if it is available
JTst Concurrent Manager
Pool of tokens
regenerate tokens upon request

Fig. 5: JTst Automated Loader & Token Passing Mechanism

Concurrent execution is achieved in JTst through a well-known token passing algorithm. Sample concurrent execution of test cases is shown in Fig. 6. In the current version, JTst has been tested to concurrently execute up to 15,000 test cases per execution.


JTst
Automated & Concurrent Java Unit Testing Tool
JTst
Java
USM
Universiti Sains Malaysia
Quit
Help
Parallelism Setting
Executing Test Cases

```
OUTPUT> arglist[0]=new Integer(110)
OUTPUT> arglist[1]=new Integer(100012)
OUTPUT>The sum = 100122
OUTPUT>Result observed ==> 100122
OUTPUT>Exit code: 0
OUTPUT>==========================================
OUTPUT>Test ID = 28
OUTPUT> Class = adder
OUTPUT> Method = private add_basictypes_integer
OUTPUT> arglist[0]=new Integer(10)
OUTPUT> arglist[1]=new Integer(1000123)
OUTPUT>The sum = 1000133
OUTPUT>Result observed ==> 1000133
OUTPUT>Exit code: 0
OUTPUT>==========================================
OUTPUT>Test ID = 29
OUTPUT> Class = adder
OUTPUT> Method = private add_basictypes_integer
OUTPUT> arglist[0]=new Integer(10)
OUTPUT> arglist[1]=new Integer(1345000)
OUTPUT>The sum = 1345010
OUTPUT>Result observed ==> 1345010
OUTPUT>Exit code: 0
```

Fig. 6: Concurrent Execution of Test Data

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. This token setting can be illustrated in Fig. 7.

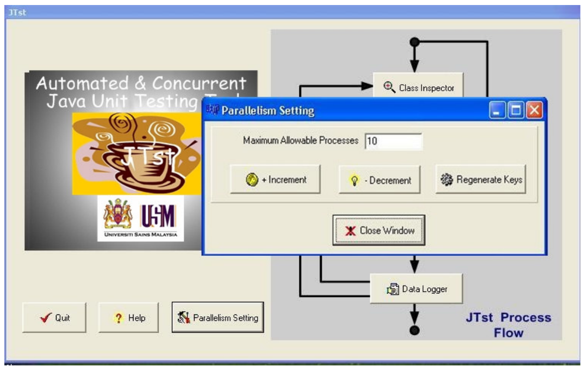


Fig. 7:`Token Generation for Concurrent Execution

- Data Logger –The Data logger is a text browser utility 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 Java module under test (MUT). If the specification of the MUT method exists, conformance analysis can be made using this database. 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.

DISCUSSION

Over the past two years, JTst has been extensively used as a tool to test Java program. In fact, JTst has been used to evaluate a Linda based distributed shared memory implementation[8] as well as the prototype runtime environment for a visual language[9]. Interested readers are referred to our earlier work [9,10].

In order to discuss the usefulness of JTst, it is necessary to revisit the aim of implementing JTst. As discussed earlier, the main aim of JTst is to automate test preparation activities, facilitate the test data generation through recombination, and allow concurrent execution of test data.

Indeed, JTst has successfully achieved the first aim to automate the test preparation activities. As discussed earlier, the test engineers merely need to concentrate on getting the good the test data. Unlike JUnit [6] where test engineers need to manually specify test drivers and execute them for testing, the process of generating test drivers as well as executing the test data is done automatically by JTst.

JTst approach is similar to JACA in the sense that JACA also uses computational reflection in order to execute faults in a Java program. At a glance, JACA appears to have all the features of JTst. Nevertheless, a closer look reveals that, unlike JTst, JACA requires that the test engineer who performs the testing have substantial knowledge of Java in order to undertake the testing process, that is, in order to manually write the test driver program. In JTst, the driver code are automatically generated and executed in a single-click of a button. Furthermore, the testing process in JTst is highly automated allowing 15,000 concurrent test cases to be executed at a particular instant. As such, JTst can be seen as offering a high level of abstraction for testing. In fact, with concurrent execution, test engineers can do multi-tasking activities without having to wait for a particular test execution to finish before moving on to the next testing assignments.

Finally, JTst also permits recombination of test data. Apart from enhancing test coverage, some earlier work in the literature suggests that, in some software implementation, the execution of combinatorial test data based on interaction of two or more variables can typically uncover 50% to 75% of faults in a program[11].

CONCLUSION

Summing up, development of an automated testing tool like JTst is crucial in order to assist test engineers at work. Although useful as an automated testing tool, much work needs to be done before JTst can truly be a practicable tool for testing Java program. In line with such a vision, we are currently implementing a parallel version of JTst to support test data execution over heterogeneous distributed environment such as the GRID.

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".

REFERENCES

1. Beizer, B., 1990. Software Testing Technique. Thomson Computer Press.
2. Moraes, R.L.O. and Martins, E., 2003. Jaca – A Software Fault Injection Tool". Proceedings of the 2003 Intl. IEEE Conf. on Dependable Systems and Networks, IEEE CS Press, p.667.
3. JUnit Website - URL http://www.junit.org
4. Matt, A. 2003. Testing Java Interface with JUnit. Dr Dobb's Journal, 28(2): 24:28.
5. Silva, G.J., R.J. Drebes, J. Gerchman, and T.S. Weber, 2004. FIONA: A Fault Injector for Dependability Evaluation of Java-based Network Applications". Proceedings of the 3rd Intl. Symposium on Network Computing and Applications (NC'04).
6. Acantilado, N.J.P. and C.P. Acantilado, 2002. Simple: A Prototype Software Fault Injection Tool, Unpublished MSc Thesis, Naval Postgraduate School, Monterey, California, December 2002.
7. Childers, B., M.L. Soffa, J. Beaver, L. Ber, K. Camarata, T. Kane, J. Litman, J., and J. Misurda, 2003. SoftTest: A Framework for Software Testing of Java Programs. Eclipse Technology Workshop, Anaheim, CA, 2003.
8. Ciancarini, P., and D. Rossi, 1997. Jada: A Coordination Toolkit for Java, Unpublished Technical Report UBLCS-96-15, Department of Computer Science, University of Bologna, Italy, 1997.
9. Zamli, K.Z., N.A. Mat Isa, and N. Khamis, 2005. The Design and Implementation of the VRPML Support Environment. Malaysia Journal of Computer Science 18(1):57-69.
10. Alang Hassan, M.D., 2005. Enhancing and Evaluating A Software Fault Injection Tool, Unpublished MSc ESDE Dissertation, School of Electrical and Electronics, University Science Malaysia, 2005.
11. Kuhn, D.R. and D.R. Wallace, 2004. Software Fault Interactions and Implications for Software Testing. IEEE Transactions on Software Engineering 30(6): 418-421.