Jumble Java Byte Code to Measure the Effectiveness of Unit Tests

Sean A. Irvine[+], Tin Pavlinic[+*], Leonard Trigg[+], John G. Cleary[+*], Stuart Inglis[+], Mark Utting[*]
Reel Two Ltd.[+]*, University of Waikato*[*]*,*
Hamilton, New Zealand
{sean,tin,len,jcleary,stuart}@reeltwo.com, jcleary@cs.waikato.ac.nz

Abstract

Jumble is a byte code level mutation testing tool for Java which inter-operates with JUnit. It has been designed to operate in an industrial setting with large projects. Heuristics have been included to speed the checking of mutations, for example, noting which test fails for each mutation and running this first in subsequent mutation checks. Significant effort has been put into ensuring that it can test code which uses custom class loading and reflection. This requires careful attention to class path handling and co-existence with foreign class-loaders. Jumble is currently used on a continuous basis within an agile programming environment with approximately 370,000 lines of Java code under source control. This checks out project code every fifteen minutes and runs an incremental set of unit tests and mutation tests for modified classes. Jumble is being made available as open source.

1. Introduction

The motivation for this work was very practical. Reel Two had made a commitment to using agile development methodologies [1, 2] and as a consequence had invested heavily in writing JUnit [3] tests for a significant body of code. However, we had no real measure of the quality of that test code. We were attracted to the use of mutation testing because once a mutation system was available no further programmer effort was forced on us except insofar as deficiencies were found in the unit tests. We could leverage directly off our existing investments in unit testing.

There were, however, stringent performance requirements. Our development environment checks out code every 15 minutes, then compiles and unit tests it. We did not want to compromise on this fast feedback cycle. A survey of existing tools showed that either the mutation testing was too slow, it did not inter-operate with JUnit, or source code was unavailable for further development and adaptation to our environment.

We considered using a simple coverage tool rather than full mutation testing but examination of our unit tests showed that it was easy to exercise code without picking up errors in its execution.

We decided to write our own system. From the start it was clear that the mutation needed to be at the bytecode level to get sufficient speed. Other challenges became apparent as we gained experience. We will describe below the significant issues that arose and how the system meets them. We also give a description of our experience in using Jumble and of future work that is needed.

Jumble has now been made available as an open-source project on SourceForge at http://jumble.sourceforge.net/ [4]

2. Existing Mutation Testing Systems

Several mutation testing systems exist. Three of them will be briefly described, along with their advantages and disadvantages.

2.1 Mothra

Mothra [5] is a mutation testing environment developed in the eighties for use with Fortran 77. While it cannot be used with Java software, it is the first fully featured mutation testing system and a large proportion of research on mutation testing described in the literature is based on it [6, 7, 8]. Mothra's mutation operators are the basis of what is done in Jumble.

2.2 MuJava

MuJava [8] is a mutation testing system developed for the testing of Java programs. Its primary purpose has been to investigate mutation operators specific to object-oriented programming languages like Java. The

prospect of using MuJava in a large-scale software development setting is an attractive one, as its mutation operators represent the state of the art in mutation testing research. However, as it is an experimental system, the extent to which it is scalable is unclear. The source code for MuJava is not publicly available, so its modification for improved scalability is difficult from a legal and practical point of view. Furthermore its test format is not JUnit. It was infeasible for us to port our large legacy set of JUnit tests.

2.3 Jester

Jester [9] is a simple open-source mutation testing system for Java. It was designed to integrate with JUnit unit tests and performs simple mutation operators on Java source code. Source code mutation requires recompilation of the mutated class for each mutation point, which is time-consuming. Additionally, applying mutations to directly to source code is error-prone (we encountered cases where Jester generated mutations within comments). Jester does not apply any sophisticated algorithms developed for mutation testing to speed up the process and is therefore very slow. Furthermore, its range of mutations is limited and it offers little advantage over code coverage tools. It was found that Jester was limited, inefficient and infeasible for use with large systems (at the present time, Reel Two has approximately 370,000 lines of system code and 190,000 lines of test code under source control). These issues motivated the development of Jumble.

3. Running Jumble

The primary entry point to Jumble is a single Java class that takes as parameters a class to be mutation tested and one or more JUnit test classes. The output is a simple textual summary of running the tests. This is in the style of JUnit test output and includes an overall score of how many mutations were successfully caught as well as details about those mutations that are not. This includes the source line and the mutation performed. Variants of the output include a version compatible with Java development in emacs where it is possible to click on the line and go to the source line containing the mutation point.

A plugin for the Eclipse IDE [10] has been provided. Currently it permits only jumbling of a single class.

When running the mutation tests a separate JVM [11] is used to prevent runaway or system-exiting code disturbing the testing process. It is given a number which specifies the mutation to start at and it continues until all mutations have been tested or some catastrophic failure occurs. It has been very important to separate the test execution in this way for reasons discussed below. If a failure occurs in one mutation then the child JVM can be restarted and testing continues with the next mutation.

Before running the mutation tests, a full run is done of the unit tests both to ensure that they all pass, to ensure that there are no environmental problems when running the tests in this way and to collect timing information to later detect mutations that lead to infinite loops.

At Reel Two, Jumble is integrated with an internal continual integration system, on several source code repositories. Every fifteen minutes this checks out all the source code, clean compiles it and then runs all the unit tests for packages that have been modified. It also places all modified classes on a queue to be mutation tested. After the unit testing has been done, Jumble is used to test classes until the fifteen minute time limit is exceeded. Overnight more time is dedicated to mutation testing so that any tests that have been accumulated during the day can be cleared.

The results of the Jumble tests for a project are presented in a web interface. These give results for individual classes and accumulate results over the package hierarchy. The web interface permits a manual override of the test queuing so that classes which have not been modified can be retested. This is sometimes necessary when the Jumble code itself has been modified or when a class needs to be retested because of changes in other classes that it interacts with.

In summary we typically use three different ways of running Jumble:

- directly from the command line where the class `com.reeltwo.jumble.Jumble` is provided with the name of the class to be tested and the class(es) which test it (among many other options). Executing directly from the distributed `jumble.jar` will achieve the same effect.
- from the Eclipse IDE where there is a menu item for a class which runs Jumble on that class (it is possible to configure other options for such runs)
- as part of a web based system where all classes are tested incrementally as they are committed and results accumulated in a set of webpages.

4. Jumble System

In this section we discuss individual parts of the Jumble system and the issues that arose in

implementing them.

4.1 Bytecode Mutation

Jumble performs its mutations by directly modifying Java bytecodes using the BCEL package [12]. This allows the mutation testing to be done with no compilations. The bytecode translation approach has been used before. Mothra used an intermediate code form to store and execute mutants. The intermediate form was not a standard code form, however, and was directly designed to represent mutations, which were still done by analyzing source code. MuJava also used bytecode translation for *structural* mutations. *Behavioural* mutations however are still done with source code analysis using the MSG method [6]. The reason for this is that the behavioral mutations in MuJava were intended to be the same mutations as those implemented in Mothra. Thus, while MuJava manages to avoid multiple compilations, it still requires source code analysis to perform the mutations. As mentioned above, Jester does not use any sophisticated techniques and performs a compilation for every mutation.

Basing the mutations entirely on bytecode also simplifies the implementation of Jumble and allows code to be tested even when the source code is not available.

Jumble mutates all the Java bytecode instructions that can be mutated safely in a context-free way. That is, each instruction eligible for mutation is replaced by another instruction independently of the instructions around it. An instruction A can be replaced by an instruction B if A and B operate on the operand stack in the same way – they expect the same number and type of arguments on the stack before the operation and leave the same number and type of arguments on the stack after operation. For example, the `iadd` (integer addition) instruction can be replaced by the `isub` (integer subtraction), since both pop the top two operands from the top of the stack and push the result of the computation (respectively the sum and difference of the two operands) onto the stack. Particularly helpful is BCEL's ability to generate bytecode for the negation of arbitrary conditionals.

A wide range of bytecode instructions are mutated including conditionals, arithmetic operations including increments and decrements, and switch statements. As well inline constants, class pool constants, and return values are mutated. More details can be found in the "Mutations" link in [4]. The mutations are all implemented using the facilities of BCEL and within the limitations of BCEL's facilities the addition of new mutations is straightforward.

Care needs to be taken to avoid mutating instructions in some parts of the code. For example, it makes no sense to mutate assertion statements. Detection of assertion statements is done by their reference to the class level flag which indicates when assertions are enabled. The pattern of code generated by the compiler is then used to detect the end of the assertion. Unfortunately such patterns are compiler dependent and care is needed when moving to new compilers and new JDK releases. Other code that needs to be excluded are conditionals generated when class constants are accessed, lengths for certain array allocations, and switch code generated for enumerations.

A facility is provided to globally exclude certain named methods from mutation. In practice this is used to exclude main methods (this is coupled with a coding standard that main methods should be as short as possible and that they should call another method with such things as input and output files as parameters). Also excluded are "integrity" methods which can be called on a class to check that its internal state is currently consistent. These effectively function as post-conditions and like assertions should be excluded.

Jumble also mutates constants, both those that occur inline and those that occur in the constant pool. An integer constant x is transformed to (-x + 1) which works correctly even when the integer represents a boolean. In Java boolean, short, and char datatypes, are implemented in bytecode as 32-bit ints. Consequently, Jumble cannot readily discriminate among these types.

Constants in the constant pool (such as String literals) are also mutated. The constant pool for a class often contains entries not referenced by any line of code or only referenced by unmutateable code such as assertions. Care is taken not to modify such literals.

4.2 Class Loader

One feature of the BCEL which has been extremely useful in the development of Jumble is its customizable class loader. It allows classes to be modified as they are loaded into the JVM. The process is much simpler than creating a class loader from scratch, as the processing of the actual class file is done by the BCEL and the user only has to implement a `modifyClass` method to perform bytecode modifications on the fly.

The Jumble class loader is derived from the BCEL class loader. Given the name of the class to modify (the target class) and the mutation number, it loads the class with the mutation inserted. All classes other than the target class are loaded normally with no modifications.

The Jumble test suite is an extension of the JUnit test suite. It runs a JUnit test in a way suitable for

Jumble testing, by running the tests until one fails. A "PASS" message is returned at that stage and no further tests are run. If no tests fail, a "FAIL" message is returned. The Jumble test suite is given the name of the test to run as a string and loads the test class dynamically (using `Class.forName`). The Jumble test suite is intended to be loaded in the Jumble class loader so that the tests are run with a mutated class.

When running several mutations inside a single JVM, a separate class loader must be used for each mutated version of the class being tested. To ensure independence between mutation tests, the entire set of (non-system) classes are reloaded in each class loader, and the classes being tested are not available directly from the system class loader.

Heavy usage of class loaders in this way can use a lot of memory in the permgen space in the JVM, particularly for classes that make use of many third-party libraries. In practice it has been difficult to prevent increasing usage of the permgen space as the tests are run. This is dealt with in two ways. Firstly the child JVM that is used to run the tests is given additional permgen space when it is started. Secondly after each test is run a check is done to see if the available permgen space is nearly exhausted, if so the child process is terminated and restarted (it will then start at the next mutation).

Some code remains difficult to run within the Jumble environment. Typically this involves code that attempts to directly access the system class loader (such code is usually incorrectly written, and unlikely to function when run in similar environments, such as within the Tomcat servlet container).

4.3 Detecting and Terminating Infinite Loops

Some mutations become stuck in an infinite loop. It is reasonable to consider an infinite loop as a failed test and hence mutation points which cause infinite loops can be considered tested. Thus Jumble needs to be able to detect infinite loops caused by mutations and terminate them.

Infinite loops are detected by timing the original test running on a class without any mutations. Timing measurements are made using the `System.currentTimeMillis` method. This introduces two difficulties. Firstly, the granularity of the value returned from the method is dependent on the underlying operating system. Secondly, the value returned is a measure of elapsed time, not CPU time allocated to the current process. Thus, the time measures obtained are somewhat imprecise and non-deterministic. This is not a problem as long as the non-determinism is accounted for.

Each mutated test run can then be timed and the runtime can be periodically compared against a runtime limit and an infinite loop is considered detected if the limit is exceeded. The formula for the runtime limit is:

RUNTIME LIMIT = 10 x ORIGINAL RUNTIME + 2 s

This formula is somewhat arbitrary but works reasonably well in practice. The most difficult situation is when a test is the first to be run, then effects such as classes being loaded and static code being executed can increase the apparent execution time.

Once an infinite loop is detected, it needs to be terminated. One option is to run the mutation testing in a separate Java thread, and terminate the thread when an infinite loop is detected. The problem with this approach is that there is no inherently safe way of terminating a Java thread while being guaranteed to leave the rest of the system in a consistent state. Use of the `Thread.stop()` method is strongly discouraged.

The method used in Jumble is to terminate the child JVM that is running the tests and for it to note that the test has been successful. Then the child JVM is restarted at the next mutation.

4.4 Applicability and Limitations

The Jumble system will run with code generated for Java 1.3 to 1.6. It has been run on code that uses multi-threading and concurrent processes. The biggest difficulties have been with systems that implement their own class loaders (see discussion above) and with variations of different compilers. The greatest difficulties with compilers is in detecting code such as assertions where the patterns may differ from compiler to compiler. Jumble has been successfully used with javac, jikes and the Java compiler in the Eclipse IDE [10].

5. Performance Issues

The expectation for Jumble is that it would be run frequently as code was committed to a common repository. However, mutation testing can be computationally expensive. All tests must be run for every mutation point. One approach to reducing this cost is to use 'weak mutation' [13], where a mutation is killed if a test case causes the mutated expression to produce a different value. This can be more efficient, because one test run can kill many mutants. But it is weaker, because even if a JUnit test uses a value that causes a mutated expression to return a different value, there is no guarantee that the JUnit test will detect that

different value. For this reason, Jumble uses strong mutation testing rather than weak.

This project has focused on reducing overheads associated with running tests and reducing the number of tests which need to be run before a mutation fails. One issue which seems not to have been addressed in the literature is modifying the order in which tests are run to avoid having to run unnecessary tests.

Mutation testing usually executes multiple test cases for each mutation. As soon as one test case fails for the mutation, the remaining test cases do not need to be run as it is already known that the mutation point is covered by a test case. An interesting question arises: Is it possible to automatically determine the order in which to run the test cases so that a test fails as soon as possible?

This section describes three heuristics used by Jumble to try to make sure that failures happen as soon as possible. Note that these heuristics can only give performance improvements for relatively well tested code. If a mutation has not been covered by a test, all the test cases must be run in order to show this and the order is irrelevant. Hence, the heuristics will not produce performance improvements for poorly tested code.

5.1 Timing Order (Heuristic 1)

Test cases often vary in their runtimes. Some appear to take negligible time while others can be very complex and take several minutes to complete. It is the long tests that take up the most time in mutation testing. This first heuristic attempts to avoid running the longest tests, if possible. First, the tests are run without any mutations and their runtimes are recorded. During mutation testing, the tests are sorted in order of runtime so that the shortest tests run first. It is hoped that one of the short tests fails before the long ones are attempted. That way the long tests are only run when there is no shorter test that has covered the mutation.

5.2 Remembering the Test Case for Each Method (Heuristic 2)

JUnit examples suggest the convention that a separate test case should be developed for every method being tested. Sometimes this is hard to do, as there are methods which are never used in isolation, but only in conjunction with other methods. Anecdotal evidence shows however, that in most cases, a method is tested by only one test case. During mutation testing, once a test case has been identified as the test for a given mutation point, it seems reasonable to run that test case for each mutation point inside the same method before trying the other test cases. The second heuristic does precisely that. Once a failure is detected, the test case is remembered and run first for every other mutation point inside the method. Often, this test case will fail for most mutation points inside the method.

5.3 Remembering the Last Failure (Heuristic 3)

The third heuristic, closely related to the second heuristic above applies when Jumble is run subsequent times. When Jumble is run on code a second time, the code may have been unchanged, or changes could have been made to the code itself or to the tests. If only small changes were made before re-running the tests, as advocated by Extreme Programming, most of the Jumble results will stay the same. Specifically, modified mutation points will be detected by the same test as before. Hence, for each mutation point first try the test case that failed last time. This heuristic stores the test case which fails for each mutation point in a cache file. On subsequent Jumble runs, the cache file is loaded and for each mutation point, the test that failed the last time is executed first.

5.4 Combination of Heuristics

Jumble uses a combination of the three heuristics to determine its test order. First, if the cache file exists, the test that failed last time is run. Next, the last test which failed with the mutation in the current method is run. Finally, the remaining tests are run in increasing runtime order.

This combination allows the most effective heuristic to take precedence, depending on the stage that the Jumble testing is currently at and the information available. When the testing begins, no information about the appropriate test order is known so the tests are run in runtime order, according to Heuristic 1. As the testing proceeds, more information about method - test case correspondence is known so Heuristic 2 is used first. Finally, after the testing is finished, the failing test case is known for every mutation point so the next time Jumble is run, Heuristic 3 applies first.

6. Evaluation and Experience

6.1 Developer Experience and Acceptance

Jumble is being used for several different projects within Reel Two. These range in size from 2,500 lines

of code and 500 lines of unit tests to 310,000 lines of code and 150,000 lines of tests. It has now been used continuously for over a year and is fully integrated into the software development process.

The developers using it are all committed to agile development techniques and have found the presence of a score for their unit tests a strong incentive to improve them. In general most of the programmers were surprised at how poor their scores were for their normal testing practices. The scores also provide a strong management tool for assessing the state and quality of software. All the scores are available to everyone in the development group and can be seen by the programmers, their peers and their managers. This provides strong incentives not to let quality drop.

It has proved feasible in most cases to obtain scores over 95% although the last 10% of this often requires checking back against the mutation failures. Such *post-hoc* testing is less valuable than blind testing. The Jumble scores are computed only on a sample of all possible mutations. Tests which take cognizance of the actual mutations can say less about the other unsampled mutations.

The reasons for being unable to achieve 100% scores are mainly environmental or timing related. It can be impossible or very inconvenient to test external situations – for example invalid database states or extreme situations such as attempting to allocate large arrays. Another problem is with conditional code which is present solely for performance reasons where mutating a conditional to its negation has no effect on the results. The inconsistency and variability of timing make it infeasible to test such cases.

As a rule of thumb it has been found that the Jumble scores get above 95% when there is approximately as much test code as original code.

Jumble has also been used in a group software engineering project course at the University of Waikato. It was found that the feedback of a score was a strong motivator for students to write and improve their unit tests. It was also invaluable when assessing the quality of the students code and tests.

6.2 System and Performance

The single most important decision about the system architecture was ensuring that all tests were run in a separate JVM. This significantly improved execution speeds, gave greater control over the environment the tests ran in, and gave reliable recovery from errors and infinite loops.

The time to do a mutation test of a class varies widely, both with the size of the class and with its complexity. The most time consuming code is heavily numeric code with a high density of conditionals and a complex flow control.

The general experience is that the mutation testing queuing system keeps up with the demand with tests seldom needing to be held over for more than a few 15 minute check out cycles.

The mutation of string literals has been controversial. Such mutations imply that all results in exception messages and the details of all output strings including warnings and internal logging messages need to be checked. Some projects have elected to turn off these mutations (there is a command line parameter in Jumble for this).

The performance heuristics that were implemented as part of Jumble were somewhat disappointing in the extent to which they sped up testing. Detailed results are given by T. Pavlinic in [14] which show that Heuristic 2 improved testing time by up to 68% in some cases but that in general performance was improved by only 5% to 10%. The major practical speedups were obtained through the careful use of a child JVM to execute multiple tests in one run and management of a queue of pending tests.

7. Future Work

7.1 System and Performance

Our experience with the speedup and heuristics for performance indicates that one further speedup technique may be worthwhile. That is to do a coverage analysis when initially running the unit tests. This would record which tests actually passed through which mutation points. Then at mutation test time only the subset of tests which actually exercise the mutation need be run. This technique will be of greatest benefit in cases where the Jumble scores are low. That is, there are few unit tests and usually all of them will need to be run before discovering that none of them fail. In the case of high Jumble scores the heuristic that remembers the test that failed on the last run may be more valuable.

It seems likely that more mutations could be added to the system. In particular permuting the actual parameters in a method call would be valuable. This would be done only for parameters with the same formal parameter type. One advantage is that it would go someway to exercising the usage of code from external packages.

Currently the test class(es) associated with a class are automatically determined by a simple naming convention where the test class name has “Test” appended to it together with a crude global list which allows the test classes to be explicitly listed for a

particular class. This is insufficiently flexible for some cases. For example abstract classes would be well served by having the test classes for all their concrete subclasses associated with them. We also intend to replace the global test list with an annotation based system to allow more explicit assignment of classes to their tests.

7.2 Interoperability

It would be of significant utility if the Jumble system could inter-operate with other unit testing environments such as testNG and JUnit 4. Both in order to use their more sophisticated facilities and to make it easier to retro-fit Jumble to code brought in from external sources.

Also keeping Jumble current is a burden. Changes in compilers and Java system releases often require updates to the system because of changes in the sequences for assertions and other automatically generated code.

These issues have all motivated the move to make Jumble available as open source.

7.3 Model Based Testing

Jumble measures the effectiveness of a test suite, so is a natural complement to model-based testing [15], which generates test suites. Model-based testing is typically used to generate system tests, but recent tools like ModelJUnit [15, 16] have also used it to generate unit tests for Java classes. In the future, we want to experiment with using Jumble as a feedback mechanism for ModelJUnit, to tell the tester how well his model and test generation choices are testing the current class.

8. Acknowledgements

We would like to thank the programmers at Reel Two for patience during the development of Jumble. Tin Pavlinic would like to thank the University of Waikato Department of Computer Science and the New Zealand Energy Education Trust for financial support.

9. References

[1] K. Beck and C. Andres, "Extreme programming explained: embrace change," 2nd ed. Boston, MA: Addison-Wesley, 2004.

[2] F. Maurer and D. Wells, *Extreme programming and agile methods : XP/Agile Universe 2003 : third XP Agile Universe Conference, New Orleans, LA, USA, August 10-13, 2003 : proceedings*. Berlin ; New York: Springer, 2003.

[3] E. Gamma and K. Beck, "JUnit." Retrieved March 2005 from http://www.junit. org

[4] "Jumble." Retrieved June 2007 from http://jumble.sourceforge.net/

[5] R. A. DeMillo and E. H. Spafford, "The Mothra software testing environment," presented at The 11th NASA Software Engineering Laboratory Workshop, Goddard Space Center, 1986.

[6] J. Offutt and R. H. Untch, " Mutation 2000: Uniting the Orthogonal," in *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*. San Jose, CA, 2000, pp. 45-55.

[7] J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, " An Experimental Determination of Sufficient Mutant Operators," *ACM Transactions on Software Engineering Methodology*, vol. 5, pp. 99-118, 1996.

[8] J. Offutt, Y.-S. Ma, and Y. R. Kwon, " MuJava : An Automated Class Mutation System," *Software Testing, Verification and Reliability*, vol. 15, pp. 97-133, 2005.

[9] I. Moore, "Jester – a JUnit Test Tester," presented at eXtreme Programming and Flexible Processes in Software Engineering - XP2000, 2000.

[10] S. Shavor, J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy, *The Java Developer's Guide to Eclipse*: Addison-Wesley, 2003.

[11] T. Lindholm and F. Yellin, *The Java virtual machine specification*, 2nd ed. Reading, MA: Addison-Wesley, 1999.

[12] Apache Software Foundation, "Byte Code Engineering Library," 2003. Retrieved March 2005 from http://jakarta.apache.org/bcel/

[13] J. Offutt, and S. Lee. “How strong is weak mutation?” In Proceedings of the Symposium on Testing, Analysis, and Verification. ACM Press, New York. 1991

[14] T. Pavlinic, “Jumble: A practical mutation testing tool for Java”, Dept. of Computer Science, University of Waikato, Hamilton, New Zealand, 2005.

[15] M. Utting and B. Legeard. "Practical Model-Based Testing: A Tools Approach", Morgan-Kaufmann 2007.

[16] Mark. Utting. ModelJUnit web site, http://www.cs.waikato.ac.nz/mbt/modeljunit Accessed on 17 April 2007.