Comparing the Effectiveness of Automatically Generated Tests by *Randoop*, *JWalk* and μ*Java* with *jUnit* Tests

Nastassia Smeets

July 19, 2009

Abstract

This paper will explore and compare three automated testing techniques with manually contrived *jUnit* tests. These techniques and their respective tools are outlined, after which they are applied to a realistically complex system. Their advantages and disadvantages are discussed, together with possible trade-offs and the type of faults they are able to detect. In conclusion, the findings are summarized and suggestions for future research are made.

1 Introduction

Testing is essential in ensuring software quality and verifying its correct operation; it comes as no surprise then that the practice is widely applied by programmers and testers alike. However, writing these tests is often a tedious and uninspiring task, which potentially could be made easier by automation. Using test generating tools may save time, since they are able to test more cases and test these more more exhaustively, requiring less time or effort invested by the tester. They may even encounter faults that human testers would never think of, but is the quality of those generated tests on par with a carefully thought out test suite?

This paper will explore and compare three automated testing techniques with manually contrived *jUnit* tests. First, the selected testing techniques and representative tools are outlined. We then move on to the hypothesis and criteria that will be used when evaluating the tools the section thereafter, followed by the experiments and their results. The relative merits and disadvantages of every technique are then compared, after which the findings are summarized and suggestions for future research are made.

2 Testing Techniques and Tools

Three different testing techniques (and their associated tools) were selected for comparison. They were chosen as to be unalike: every technique employs a different approach to the same problem.

2.1 Feedback-Directed Random Testing with *Randoop*

The *Randoop*[1] tool requires no additional user input or previously existing test suites, but will automatically generate unit tests for a given set of classes during a limited time interval, which is preset by the tester. It does so by executing random sequences of methods from the class-under-test (CUT). Since the sequences are random, results may vary slightly when examining results from different runs.

Using prior results, more intelligent inputs and sequences will be chosen to avoid redundant or illegal sequences as much as possible, making up the feedback-directed aspect of the tool. For instance, a sequence containing code that will result in an exception is pruned from future runs.[1] The tests that are output can be classified into two categories: first regression tests that capture the current behavior of the code are generated, and secondly, also contract violating tests are produced, which could indicate potentially buggy code.

The author of this tool provides a more detailed and technical explanation of its workings in Sect. 2 of [2].

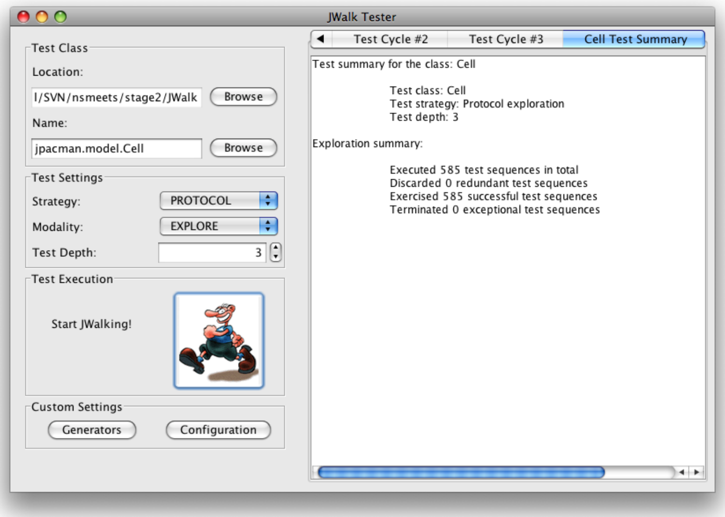
2.2 Lazy Systematic Unit Testing with *JWalk*

JWalk[2] is a *lazy systematic unit testing tool. Lazy* indicates that no formal specifications are required, since they will be inferred from the ever changing code by the tool itself. The lack of specifications is typical in agile, or XP development, for which this tool could prove a useful addition. *Systematic* testing is achieved by executing all transitions between states (caused by sequences of methods) in a replicable, deterministic manner, as opposed to

[1]`http://people.csail.mit.edu/cpacheco/randoop`

[2]`http://www.dcs.shef.ac.uk/~ajhs/jwalk/`

random testing. The class-under-test is exhaustively exercised until a preset depth is reached.[3]



Test Class

Location: I/SVN/nsmeets/stage2/JWalk

Name: jpacman.model.Cell

Test Settings

Strategy: PROTOCOL

Modality: EXPLORE

Test Depth: 3

Test Execution

```
Test summary for the class: Cell

            Test class: Cell
            Test strategy: Protocol exploration
            Test depth: 3

Exploration summary:

            Executed 585 test sequences in total
            Discarded 0 redundant test sequences
            Exercised 585 successful test sequences
            Terminated 0 exceptional test sequences
```

Figure 1: The *JWalk* tool.

The tool can be used for both exploring the class as for generating an oracle, based on automatic analysis, predictive rules and interaction with the tester. It distinguishes between mutator methods and observer methods, which respectively do and do not cause a change in state in the system, for faster and more reliable results. When the software, and therefore its specification, is modified, *JWalk* will only prompt for verification of altered properties, in this way providing a regression test for the CUT.

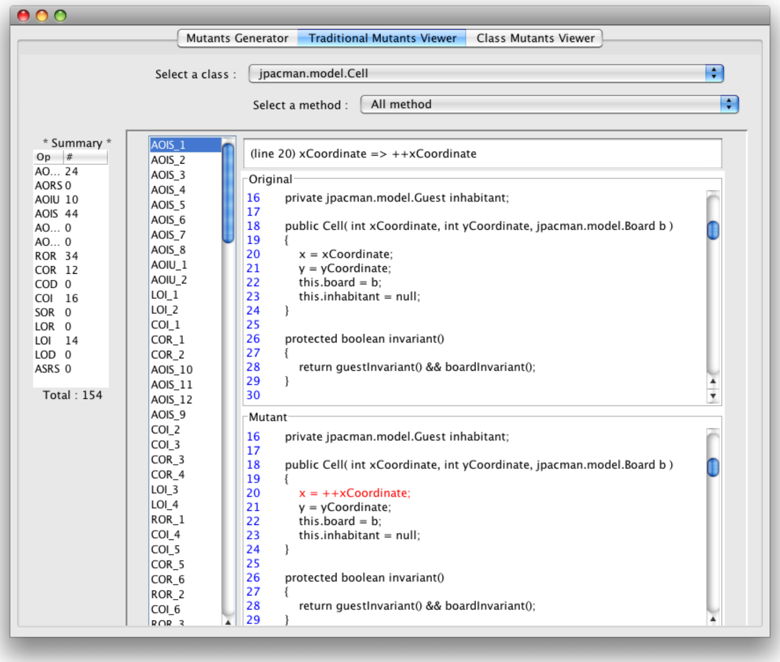
A notable detail is that this tool does not generate unit tests, only reports.[4]

2.3 Mutation Testing with μ*Java*

μ*Java*[3] will introduce small modifications, or mutations, to the code, both on class and method level. Since the testing occurs on the implementation level, this is a white box technique. Its intent is to trace code that is rarely called or showing that the existing unit tests are inadequate (when the injected fault is not discovered by the test suite). In a way, this resembles coverage measurements – it also attempts to indicate which code needs improved testing. The difference is that mutation testing can find code that is executed (and thus is recorded as covered), but is not actually tested. Since

[3]`http://cs.gmu.edu/~offutt/mujava/`

this technique's concept is more the testing of the tests, *μJava* indeed does require preexisting tests. It will only generate a modified implementation of the system.[5] A screenshot is presented in Figure 2.



Select a class : jpacman.model.Cell

Select a method : All method

* Summary *

Op	#
AO...	24
AORS	0
AOIU	10
AOIS	44
AO...	0
AO...	0
ROR	34
COR	12
COD	0
COI	16
SOR	0
LOR	0
LOI	14
LOD	0
ASRS	0

Total : 154

```
AOIS_1
AOIS_2
AOIS_3
AOIS_4
AOIS_5
AOIS_6
AOIS_7
AOIS_8
AOIU_1
AOIU_2
LOI_1
LOI_2
COI_1
COR_1
COR_2
AOIS_10
AOIS_11
AOIS_12
AOIS_9
COI_2
COI_3
COR_3
COR_4
LOI_3
LOI_4
ROR_1
COI_4
COI_5
COR_5
COR_6
ROR_2
COI_6
ROR_3
```

(line 20) xCoordinate => ++xCoordinate

Original

```
16      private jpacman.model.Guest inhabitant;
17
18      public Cell( int xCoordinate, int yCoordinate, jpacman.model.Board b )
19      {
20          x = xCoordinate;
21          y = yCoordinate;
22          this.board = b;
23          this.inhabitant = null;
24      }
25
26      protected boolean invariant()
27      {
28          return guestInvariant() && boardInvariant();
29      }
30
```

Mutant

```
16      private jpacman.model.Guest inhabitant;
17
18      public Cell( int xCoordinate, int yCoordinate, jpacman.model.Board b )
19      {
20          x = ++xCoordinate;
21          y = yCoordinate;
22          this.board = b;
23          this.inhabitant = null;
24      }
25
26      protected boolean invariant()
27      {
28          return guestInvariant() && boardInvariant();
29      }
```

Figure 2: The *μJava* Mutant Generator screen.

3 Hypothesis and Criteria

As previously stated, it may prove interesting to see how well test generating tools perform when compared to an actual human tester. Though it is highly unlikely that any tool could ever fully replace an experienced software tester, it just might make the latter one's job easier.

Obviously, a sample software system is needed when comparing the tools. We opted for *JPacman*, an implementation of the famed game in *Java*, that is used as a teaching example at the Delft University of Technology. It was chosen for its well-documented unit and integration tests, known to be of sound quality from previous experience in testing class. Moreover, the system is one of relative complexity and thus serves as a more realistic sample system as opposed to e.g. a mere stack implementation.

Coverage is an unbiased, verifiable and easily measured criterion when using tools such as *Emma*[4]. It is however unsuitable in this case, since neither *JWalk* nor *μJava* actually generate unit tests. Besides, *μJava* is in a way similar to measuring coverage, since it indicates which sections of the code are insufficiently executed by tests, if executed at all.

Time (or effort) spent seems like a good alternative, but it is much more problematic to quantify precisely. For instance, measurements of the time invested in composing the non-generated *jUnit* tests are not available and impartially estimating the actual effort spent learning the tools and reading papers are particularly hard problems. Since running the tests usually happens very quickly, it is also not straightforward to reliably measure how long this takes. It is also biased, seeing that *Randoop* generates tests in a preset time limit, as opposed to the other tools.

Therefore, comparisons will primarily be made based on the type of faults the tools can detect. Further points of distinction may be the amount of extra work a tester has to invest when using one of these tools and maintainability of any software that needs to be manually written by the tester, or that was automatically generated by the tool. A potential user can then weigh the pros and cons of usage of the tool.

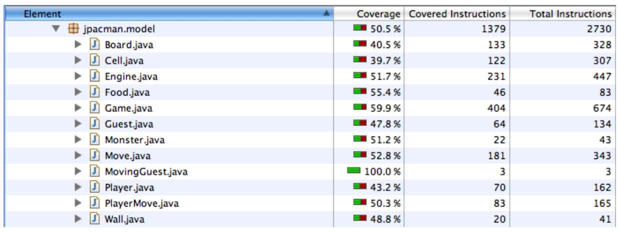
4 Experiments

4.1 *Randoop*

Both easy in installation and use, the *Randoop* tool randomly generates interleavings of methods, bounded by time, not depth. The unit tests it like so generates, essentially capture the current behavior of the class-under-test. The tool was ran with a time limit of 1 second, 10 seconds and 30 seconds respectively. Coverage measurements using Emma were conducted to get a quick overview of the quality of the generated regression tests. Figure 3 shows coverage achieved when running the test suite generated by *Randoop* in 30 seconds. Overall coverage in the `jpacman.model` package when using *Randoop*'s default 10 seconds setting is only slightly less (49.2%) than when using 30 seconds (50.5%). As can be expected, 1 second yielded far worse results (39.9% coverage).

Randoop's main attraction is the automated generation of a full regression test suite. This can be helpful when working on a project that has no tests whatsoever and for which it is crucial to ensure current behavior is preserved

[4]`http://emma.sourceforge.net/`



Element	Coverage	Covered Instructions	Total Instructions
jpacman.model	50.5 %	1379	2730
Board.java	40.5 %	133	328
Cell.java	39.7 %	122	307
Engine.java	51.7 %	231	447
Food.java	55.4 %	46	83
Game.java	59.9 %	404	674
Guest.java	47.8 %	64	134
Monster.java	51.2 %	22	43
Move.java	52.8 %	181	343
MovingGuest.java	100.0 %	3	3
Player.java	43.2 %	70	162
PlayerMove.java	50.3 %	83	165
Wall.java	48.8 %	20	41

Figure 3: *Coverage achieved by running the test suite generated by Randoop in 30 seconds.*

when updating the code. Even though one gets free and effortless regression tests, it comes at a excessively high cost in maintenance. When running *Randoop* for 30 seconds, it yields some 96,000 lines of testing code, accounting for more than 400 test cases. Some of these cases use just under a hundred variables that are named by the unhelpful convention `var_number`, making any deep understanding of the tests practically impossible. Another limitation of Randoop is the lack of support for category partition testing. There is no way for the tester to supply carefully chosen values for arguments.

It is claimed in [1] that the efficiency and quality of the resulting test suite can be improved by using method level annotations and checkers. First, three method level annotations are announced to be available in *Randoop*: `@Omit`, `@Observer` and `@ObjectInvariant`, each respectively ignoring the method it is added to, indicating the method it is attached to has no side-effects and finally declaring that this parameterless method that returns a boolean value, is actually a class invariant. The latter seemed especially relevant to *JPacman* since it employs class invariants, but an implementation of this annotation was not found (the first and second ones are available though). Likewise, checkers should provide a way of translating the assert-statements that already occur in the *JPacman* code into contracts that *Randoop* can interpret and use. To make matters worse, the class that implements this was also nowhere to be found in this distribution of the tool, nor was there any documentation to be found concerning custom checkers.

The tests cannot be generated with the existing assertions in mind, but they can be switched on when executing the test suite. Some tests will return an `AssertionException`, as is to be expected, but this only occured in a small fraction of the tests.

4.2 *JWalk*

JWalk recently hit version 1.0 and was easy to install and comprehend, through clear and detailed instructions on its website[5]. There are multiple options available for analyzing the selected class: first there are three modes, then there are three strategies.

The first mode is the *inspect* mode, which just gives a summarizing overview of the methods in the class. The results depend on which of the three strategies is chosen. The first and simplest strategy, the *protocol* strategy, takes into consideration all available methods in that class and the overview will only distinguish between constructors and public methods. The *algebra* strategy attempts to separate methods that have side-effects (mutator[6] or transformer operations) and observers, methods that do not change the current state of the system, e.g. getters, invariants, etc. Finally, the *states* mode attempts to guess the various high level states the class can be in.

The second mode is the *explore* mode. Here, paths from the constructor and all possible interleavings till the selected depth are generated. Using the protocol strategy, all distinct interleavings of the methods in the class may lead to a potentially different state, so, again, none are pruned. This is the most exhaustive, but also the most time consuming strategy. The algebra strategy applies only distinct sequences of constructors and mutators, allowing for faster results since observer methods need not to be considered. The states strategy attempts to find all abstract design states by executing every unique combination of state predicates over the objects variables.

All three strategies report the amount of discarded and non-redundant sequences that were found, and how many of the latter executed successfully or ended in an exception; an example is shown in Listing 1. All non-redundant sequences and their results can be browsed in the tool.

```
Test summary for the class: Cell
        Test class: Cell
        Test strategy: Algebraic exploration
        Test depth: 3
Exploration summary:
        Executed 9 test sequences in total
        Discarded 576 redundant test sequences
        Exercised 9 successful test sequences
        Terminated 0 exceptional test sequences
```

Listing 1: *Example summary for* `jpacman.model.Cell` *generated by JWalk, using the algebraic setting and maximum test depth 3.*

[5]`http://www.dcs.shef.ac.uk/~ajhs/jwalk/userguide.html`
[6]Not to be confused with mutators as employed in *μJava*.

The third and last mode is the *validate* mode. Here, the programmer is asked to validate the results of a sequence of methods that is shown in a pop up window. This way, one can concentrate on checking expected results, as opposed to writing unit tests. However, reading, understanding and validating the proposed result (for a sequence of methods taking parameters) is a tedious task at first, especially when a considerable depth has been selected. However, *JWalk* attempts to predict results, requiring less interaction. Also, when an oracle has been generated and the code is updated, only novel sequences need to be verified. So, effort is only substantial the first run, making this method scalable. If it encounters an unexpected exception, it prompts the tester – this could possibly indicate, for example, a new fault or a broken precondition. An example oracle that was generated by *JWalk* is printed in Listing 2. (The three strategies are fully analogous to the previous modes.)

```
new(1,2,Board#1)=Cell#1
new(1,2,Board#1).getY()=2
new(1,2,Board#1).getX()=1
new(1,2,Board#1).guestInvariant()=true
new(1,2,Board#1).getInhabitant()=null
new(1,2,Board#1).isOccupied()=false
new(1,2,Board#1).cellAtOffset(5,6)=null
new(1,2,Board#1).getBoard()=Board#1
new(1,2,Board#1).adjacent(Cell#1)=false
```

Listing 2: *Oracle for `jpacman.model.Cell` generated by JWalk, using the algebraic setting and maximum test depth 2.*

An downside to this tool is that the automatically generated values for parameters are practically useless for exhaustive testing. For instance, the tool will always generate a game board of size 1 by 2, then attempt to retrieve the cell located at position $(3, 4)$. This will always yield the same exception again and again; we are obviously also interested in the correct retrieval of cells that effectively are on the board, making category partition testing virtually impossible with *JWalk* (support for category partition is however currently under development [4]). This problem can be solved by implementing a custom generator (an example is shown in Listing 3).

```
package generators;

import org.jwalk.GeneratorException;
import org.jwalk.gen.CustomGenerator;
import org.jwalk.core.MasterGenerator;


public class BoardGenerator implements CustomGenerator {
        private int seed = -1;
        private int ctr = 0;
        int[] num = {5, 10, 3, 5, 0, 0, 4, 9, -1, -1, 5, 10};

        public boolean canCreate(Class<?> type) {
                return type == int.class;
        }

        public Object nextValue(Class<?> type) throws GeneratorException {
                if (type != int.class) {
                        throw new GeneratorException(type);
                }

                Integer res = new Integer(num[ctr]);
                if (ctr == num.length-1) {
                        ctr = 0;
                } else {
                        ctr++;
                }
                return res;
        }

        public void setOwner(MasterGenerator generator) {
                // Null op
        }
}
```

Listing 3: *Custom generator for `jpacman.model.Board`*

Again, these were easily written using the available instructions, but yet, they provide no elegant solution – a depth far greater than probably is required to achieve the same result in *jUnit* tests is needed here. Therefore, this testing technique and tool is not recommended for classes that rely heavily on category partition testing. It is, however, very useful for the exhaustive testing of all possible interleavings of methods, in just a fraction of the time it takes to do this manually. This makes effort for using this tool very low.

Also note that *JWalk* does not allow the option of enabling assertions.

4.3 μ*Java*

The latest version of the tool (v.3) was downloaded from the μ*Java* website[7]. Installation instructions can be retrieved when scrolling down the page, but these were difficult to follow at times and occasionally omitted details. A somewhat clearer guide is found in the appendices of [5].

Notwithstanding the fact that the μ*Java* homepage claims there is support

[7]`http://cs.gmu.edu/~offutt/mujava/#Links`

for *Java 1.5* and *1.6*, this proved to be untrue when applying the tool to the *JPacman* code. Very little, if any, debug output is provided to the tester (generally a mere “cannot parse” is output when encountering an error), making any attempts at locating the cause of the problems far more trying. After a good deal of study, it became apparent that there is no support for *Java 1.5* or *1.6* language constructs such as annotations (for instance `@Override`, `@Test`, etc). It also does not allow generics (it should be noted that this issue is indicated on the website though), and worse, `assert`-statements, which are plentiful in *JPacman*. Therefore, all files had to be rewritten and compiled to Java version *1.4*. Fortunately, once this problem was fixed, *μJava* was fairly easy to work with.

First, mutants were generated. All available mutants were selected for all classes in the `jpacman.model` package. These mutators are listed in Tables 3 and 4 and the Java equivalents for the latter are found in Table 5 (compiled from the information found in [5] [6] [7]). The `mujava.gui.GenMutantsMain` GUI allows for easy comparison between the original and mutated class, as could be previously seen in Figure 2.

Class	**Method-level**	**Class-level**
`Board`	167	9
`Cell`	154	15
`Engine`	81	5
`Food`	28	1
`Game`	147	9
`Guest`	(abstract)	(abstract)
`Monster`	0	0
`Move`	(abstract)	(abstract)
`MovingGuest`	(abstract)	(abstract)
`Player`	49	5
`PlayerMove`	36	9
`Wall`	0	0

Table 1: *Classes in the* `jpacman.model` *package and their associated number of mutants created by μJava.*

Since *μJava* predates *jUnit*, the tests originally supplied with *JPacman* need to be rewritten as well. No modifications were made other than revising each test to now output a string, which will be used for comparison when running tests on both the original and the mutated code. This process is called *killing mutants* and is conducted using the `mujava.gui.RunTestMain` GUI by selecting the CUT and its associated unit test. However, *JPacman* does not supply a separate unit test for all of the classes, resulting in a very low number of mutants killed for some.

	Method-Level			Class-Level			Coverage
Class	**Killed**	**Total**	**Score**	**Killed**	**Total**	**Score**	**Instr.**
`Board`	108	167	64%	5	9	55%	39%
`Cell`	109	154	70%	7	15	46%	46%
`Engine`	1	81	1%	0	5	0%	47%
`Game`	31	147	21%	0	9	0%	49%
`PlayerMove`	0	36	0%	0	9	0%	50%

Table 2: *Percentages of mutants killed for classes that have an associated unit test in JPacman, contrasted with block coverage scores.*

The higher the score, the better – it signifies that more mutations, or possible subtle mistakes by the programmer, can be detected by the test suite. It is fairly high for the `Board` and `Cell` classes (especially for mutations on the method level). This is not surprising, since these are conceived as actual unit tests. However, the `Game` and `Engine` tests do no provide an exhaustive testing of all possible paths or transitions between states, so it comes as no surprise their mutation scores are very low. Also, when comparing to traditional coverage results, one can deduce the individual tests are far from complete as well.

Mutation testing with *μJava* has clearly highlighted where there is room for improvement in the tests for the *JPacman* system. However, the time and effort invested by the tester is exceedingly high, since all code and tests have to be converted to Java 1.4. This operation may not even prove possible or feasible for other systems. The `assert`-statements provided with the system were lost as well. All this, and the fact that maintaining the same code base in multiple Java versions for the sole sake of using *μJava* borders on insanity, makes the tool practically unusable for realistic systems unless they are written in compliance with version 1.4 to start with.

5 Conclusions

In conclusion, one can state that the *Randoop* tool provides an easy regression test suite for testers that lack time to write one themselves, but it does come with the high cost of unmaintainable tests.

JWalk provides exhaustive testing by executing all methods, but the lack of support for sound category partition testing limits its practical uses. A solution is however in the make. Furthermore, only CUTs with 5 to 15 public operations, executed till a depth of 3 to 5 methods[4], are realistically

Category	**Operator**	**Description**
Encapsulation	*AMC*	Access Modifier Change
Inheritance	*IHD*	Hiding Variable Deletion
	IHI	Hiding Variable Insertion
	IOD	Overriding Method Deletion
	IOP	Overriding Method Calling Position Change
	IOR	Overriding Method Rename
	ISI	`super` Insertion
	ISD	`super` Deletion
	IPC	Explicit Call To Parent's Constructor Deletion
Polymorphism	*PNC*	New Method Call With Child Class Type
	PMD	Member Variable Declaration With Parent Class Type
	PPD	Member Variable Declaration With Child Class Type
	PCI	Type Cast Operator Insertion
	PCC	Cast Type Change
	PCD	Type Cast Operator Deletion
	PRV	Reference Assignment With Other Compatible Variable
	OMR	Overloading Method Contents Replacement
	OMD	Overloading Method Deletion
	OAN	Arguments Of Overloading Method Call Change
Java-Specific Features	*JTI*	`this` Keyword Insertion
	JTD	`this` Keyword Deletion
	JSI	`static` Keyword Insertion
	JSD	`static` Keyword Deletion
	JID	Member Variable Initialization Deletion
	JDC	Java-Supported Default Constructor Creation
	EOA	Reference Assignment and Content Assignment Replacement
	EOC	Reference Comparison and Content Assignment Replacement
	EAM	Accessor Method Change
	EMM	Modifier Method Change

Table 3: *The 29 class-level mutation operators available in MuJava.*

Category	**Operator**	**Description**
Arithmetic	AOR_B	Arithmetic Operator Replacement (binary)
	AOR_U	Arithmetic Operator Replacement (unary)
	AOR_S	Arithmetic Operator Replacement (shortcut)
	AOI_U	Arithmetic Operator Insertion (unary)
	AOI_S	Arithmetic Operator Insertion (shortcut)
	AOD_U	Arithmetic Operator Deletion (unary)
	AOD_S	Arithmetic Operator Deletion (shortcut)
Relational	ROR	Relational Operator Replacement
Conditional	COR	Conditional Operator Replacement
	COI	Conditional Operator Insertion
	COD	Conditional Operator Deletion
Shift	SOR	Shift Operator Replacement
Logical	LOR	Logical Operator Replacement
	LOI	Logical Operator Insertion
	LOD	Logical Operator Deletion
Assignment	ASR_S	Assignment Operator Replacement (shortcut)

Table 4: *The twelve traditional, or method-level, mutation operators available in MuJava.*

Category	**Operator**	**Description**
Arithmetic	Binary: `+, -, *, /, %`	addition, subtraction, multiplication, division and modulus of two numerical variables
	Unary: `+, -`	indicating a positive or negative value for a number
	Shortcut: `++, --`	both pre- and post-increments
Relational	$<, <=, >, >=, ==, !=$	
Conditional	Binary: `&&, ||, &, |, ^`	conditional and bitwise AND/ORs, bitwise XOR
	Unary: `!`	negation
Shift	$>>, <<, >>>$	signed shift right/left, unsigned shift right
Logical	Binary: `&, |, ^`	AND, OR, XOR
	Unary: `~`	complement
Assignment	`+=, -=, *=, /=, %=, ...`	assignment shortcuts

Table 5: *MuJava's method-level mutation operators in Java.*

feasible when deploying this tool.

Mutation testing provides a fresh view on assuring test quality and a helpful addition to coverage measurements, but the μ*Java* tool is unfortunately in desperate need of an update to be of any realistic use.

Even though the tools outlined in this paper could be of use to an experienced manual tester, it is clear that there is still a lot of work that needs to be done before (if ever!) the latter can be fully replaced by them. The tools used only work on unit test level; there still will be a need for integration and system tests conducted by experts. Supporting fully automatic category partition testing will prove to be a complex challenge, especially when basic support for this does not yet exists. Also, these three techniques are but a small sample of the various approaches that exist to automated test generation, such as evolutionary testing, model-based testing, robustness testing, classification tree testing, assertion-based testing and many more.

References

[1] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *OOPSLA 2007 Companion*, (Montreal, Canada), ACM, 2007.

[2] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, (Minneapolis, MN, USA), IEEE Computer Society, 2007.

[3] A. Simons and C. Thomson, "Lazy systematic unit testing: JWalk versus JUnit," in *Testing: Academic and Industrial Conference Practice and Research Techniques-Mutation, 2007. Taicpart-Mutation 2007*, pp. 138–138, 2007.

[4] A. Simons, "JWalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction," *Automated Software Engineering*, vol. 14, no. 4, pp. 369–418, 2007.

[5] M. Umar, "An Evaluation of Mutation Operators for Equivalent Mutants," Master's thesis, King's College, London, UK, 2006.

[6] Y.-S. Ma and J. Offutt, "Description of method-level mutation mutation operators for java," 2005.

[7] Y.-S. Ma and J. Offutt, "Description of class mutation mutation operators for java," 2005.

[8] A. Bacchelli, P. Ciancarini, and D. Rossi, "On the effectiveness of manual and automatic unit test generation," in *Proceedings of the 2008 The Third International Conference on Software Engineering Advances-Volume 00*, pp. 252–257, IEEE Computer Society Washington, DC, USA, 2008.

[9] Y. Ma, J. Offutt, and Y. Kwon, "MuJava: An automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.