

# A Complete Automation of Unit Testing for Java Programs

Yoonsik Cheon, Myoung Yee Kim, and Ashaveena Perumandla

TR #05-05

February 2005; revised July 2005

**Keywords:** unit testing, test automation, genetic algorithms, runtime assertion checking, formal interface specifications, design by contract, Java Modeling Language (JML), Java language

**2000 CR Categories:** D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — computer-aided software engineering (CASE); D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, tools, validation, JML; D.2.5 [*Software Engineering*] Testing and Debugging — Debugging aids, design, monitors, test data generators, testing tools, theory; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

Appeared in *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05)*, Las Vegas, Nevada, USA, June 27-30, 2005, pages 290–295.

Copyright © 2005 by CSREA Press

Department of Computer Science  
The University of Texas at El Paso  
500 West University Avenue  
El Paso, Texas 79968-0518, U.S.A.

# A Complete Automation of Unit Testing for Java Programs

Yoonsik Cheon, Myoung Yee Kim, and Ashaveena Perumandla  
Department of Computer Science  
University of Texas at El Paso  
El Paso, TX 79968  
cheon@cs.utep.edu, {mkim2, aperumandla}@utep.edu

**Abstract**—Program testing is expensive and labor-intensive, often consuming more than half of the total development costs, and yet it is frequently not done well and the results are not always satisfactory. However, testing is the primary method to ensure that programs comply with requirements. We describe our on-going project that attempts to completely automate unit testing of object-oriented programs. Our project investigates the use of an evolutionary approach, called genetic algorithms, for test data generation and the use of program specifications, written in JML, for test result determination. A proof-of-concept tool has been implemented and shows that a complete automation is feasible for unit testing Java programs. Automated testing techniques such as ours can complement manual testing by testing significant portion of object-oriented programs, as methods in object-oriented programs tend to be small; manual testing can focus more interesting problems, e.g., inter-class testing.

## I. INTRODUCTION

Testing—ubiquitous in software development—is the process of exercising a program with the intent to yield measurable errors. It is the primary method to achieve confidence of software, i.e., to ensure that the design and implementation of programs comply with the specified requirements. However, software testing is an expensive and labor-intensive process, typically consuming at least 50% of the total costs involved in developing software [1], while adding nothing to the functionality of the product. For example, Microsoft is said to have one tester per developer [2] and the global testing market is estimated to be a \$13 billion industry. Yet, testing is frequently not done well and the results are seldom quite optimal; e.g., how often do you have to click the “new automatic updates available” messages on your computers?

Object-orientation is the most recent and popular programming paradigm. However, testing object-oriented programs is difficult and not well established yet. The object-oriented features, such as encapsulation, polymorphism, and inheritance, complicate program testing. For example, encapsulation makes it difficult to ascertain whether the internal data of the class being tested is correct. It may even be impossible to access some

of the internal data because no accessor methods are provided. Thus, creating appropriate test data (objects) and deciding test success or failure are difficult for object-oriented programs.

In this paper we describe our on-going project that addresses the problem of reducing the high cost of software testing. Software testing should be less costly, less time-consuming, and more automated. We focus on unit testing of object oriented programs; unit testing is testing each module separately. The aim of our project is to investigate techniques and tools for complete automation of unit testing. In particular, we investigate the use of an evolutionary approach, called *genetic algorithms* (see Section II-A), for automatic test data generation and the use of formal program specifications, written in JML (see Section II-B), for automatic determination of test results.

In the next section, we describe a background including unit testing, JML, and genetic algorithms. In Section III, we explain our approach informally through an example and also discuss core research issues. In Section IV, we describe the current status of our project, focusing on a proof-concept-tool that we developed. In Section V, we discuss related work and we conclude in Section VI.

## II. BACKGROUND

### A. Unit Testing

We focus on a particular kind of software testing, called *unit testing*, that tests each program module (or unit) separately. In object-oriented programs, a unit can be a method, a class, or a set of closely related classes. Unit testing is usually performed by programmers and is the foundation of all other tests such as integration testing and system testing.

In general, software testing consists of three components: test case design (or test data selection), test case execution, and comparison of the results of the execution with expected results. The last component is called a *test oracle*, for it decides test success or failure. For a complete automation of testing, all three components must be automated.

The black-box approach and the white-box approach are two well-known test data selection techniques. In the *black-box approach*, test data is selected based solely on the descriptions of inputs and outputs of the program being tested, i.e., without knowing its internal workings. In the *white-box approach*, specific knowledge of program code—such as the program’s control flow or data flow properties—is used to select test data. The two approaches are complementary to each other and both are required for complete testing. Our work support both approaches.

### B. Java Modeling Language

The Java Modeling Language (JML) [3] [4] is a formal behavioral interface specification language for Java. In our approach, JML is used to write formal specifications, which act as test oracles, of program modules to be tested. The specification of a program module describes what the module does, but not how it does it.

JML specifies both the syntactic interface and the behavior of Java classes and interfaces (see Section III-A for an example specification). The syntactic interface of a Java class or interface, commonly called an application programming interface (API), consists of the signatures of its methods and the names and types of its fields. The behavior is specified in assertions, given as pre and postconditions and class invariants. The assertions in JML are written using a subset of Java expressions and are annotated in the source code. This makes JML a practical tool, for one of the main hurdles to using a new specification-centric tool is often the lack of familiarity with the associated specification language.

JML supports a suite of tools [5], including runtime assertion checker [6]. The runtime assertion checker detects violations of JML assertions at run-time and, thus, can turn JML specifications into test oracles [7].

### C. Genetic Algorithms

We use an evolutionary approach, called *genetic algorithms*, to generate object-oriented test data automatically. Genetic algorithms are rooted in the mechanisms of evolution and natural genetics and manipulate a population of potential solutions to an optimization or search problem [8] [9]. A set of potential solutions are represented as *chromosomes*, consisting of a sequence of *genes*, equivalent to the genetic material of individuals in nature. In testing, each test data may be represented as a chromosome [10] [11]. Each solution is associated with a *fitness value* that reflects how good it is, compared with other solutions in the population. In testing, test coverage may be used for fitness. As in nature, fitness plays the driving force for better solutions to survive. That is, selective breeding is used to obtain new potential solutions that have characteristics inherited from each parent solution. Successive populations

called generations are evolved using genetic operations, such as *crossover* and *mutation*. The aim is to let the population converge towards a global solution. From an initial population of randomly generated solutions, the fitness value of each solution is calculated. Based on the fitness values, members of the population are selected to become parents. The parents are combined to form the offspring by using the crossover operator that concatenates the genes of two chromosomes. The aim of crossover is to produce offspring that combine the best features from both parents to result in a better offspring. A number of the offspring are then mutated to introduce diversity into the population. A new generation is then selected from the offspring and old population. This generational process is repeated until a termination condition is reached. Although the application of genetic algorithms to testing is relatively new, the approach has been shown to be effective. However, Previous work has been limited to simple data types, and relatively little attention has been given to object-orientation [12]. Our work extends genetic algorithms to object-oriented programs by addressing the issues of genetic encoding, genetic operations, and fitness of objects.

## III. OUR APPROACH

The ultimate goal of our project is to completely automate unit testing of object-oriented program. A programmer should be able to perform unit testing by a single click of button or a single command execution.

A complete automation of program testing involves automating three components of testing: test data selection, test oracle, and test execution. The essence of our approach is to combine JML and genetic algorithms (see Fig. 1). In addition, JUnit [13] is used as a test execution platform. JUnit is a open-source unit testing framework for Java and provides a way to organize test data and perform test execution. In JUnit, one has to write Java code, called a *test class*, that describes test data, invokes the methods to be tested, and determines test results.

JML is used both as a tool for describing test oracles and as a basis for generating test data. The approach uses specifications as test oracles, and JML is used to write such specifications. Each class to be tested is assumed to be annotated with JML assertions, such as pre and postconditions and class invariants that describe the behavior of the class. As in the earlier work of one of the authors [7], the JML’s runtime assertion checker is used to detect assertions violations at run-time and to interpret them as either test success or failure. For this, a JUnit test class called a *test oracle class* is generated automatically.

One of important aspects of our work is to automate test data generation. This is done by using genetic algorithms (see Fig. 1). An initial set of test data is randomly

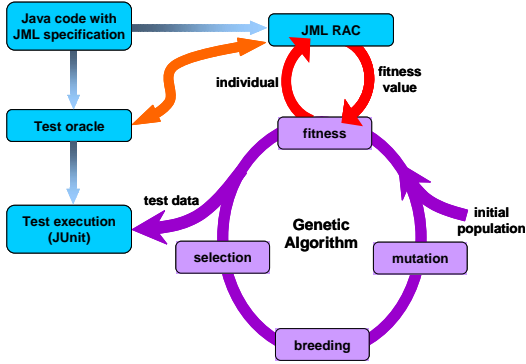


Fig. 1. Automation of unit testing with JML, JUnit, and genetic algorithms.

generated based on the signature information of classes. The fitness value of individual test data is calculated based on either black-box coverage (e.g., condition coverage on postconditions) or white-box coverage (e.g., branch coverage). The set of test data is filtered by using the precondition to eliminate so called *meaningless tests* [6]. Thus, JML assertions are used both as test oracles and to generate test data. If the set of test data satisfies a preset coverage criterion, a suitable set of test data is found. Otherwise, the test data population is enriched by applying genetic operations such as crossover and mutation. A pair of existing test data is selected and combined to create new test data and some test data are mutated. This evolutionary process is repeated until a suitable set of test data is found. The final set of test data is generated in such a way to be exercised by the JUnit testing framework.

### A. Example

In this section we demonstrate our approach by applying it to a small example. Fig. 2 shows a JML specification of class `Account` representing bank accounts. JML assertions are annotated in the source code as special comments, i.e., `//@` and `/*@ ... */`. The invariant clause states that the balance (`bal`) should be always non-negative. Method (and constructor) pre and postconditions precede method (and constructor) body and are specified with the `requires` and `ensures` clauses, respectively. In addition to Java boolean expressions, one can also use in JML assertions JML-specific constructs, such as equivalence (`<==>`) and implication (`==>`). In the postcondition of the `withdraw` method, `\old(bal)` denotes the value of `bal` in the prestate, which may be different from that of the poststate because it may be mutated. The specification of the `withdraw` method states that clients have to call the method with a positive amount, `amt`, and the balance, `bal`, is adjusted appropriately if and only

```
public class Account {
    private /*@ spec_public @*/ int bal;
    /*@ public invariant bal >= 0;

    /*@ requires bal >= 0;
        @ ensures this.bal == bal; @*/
    public Account(int bal) {
        this.bal = bal;
    }

    /*@ requires amt > 0;
        @ ensures
        @ (\result <==> \old(bal) - amt >= 0)
        @ && (\result ==> bal == \old(bal) - amt)
        @ && (!\result ==> bal == \old(bal));
        @*/
    public boolean withdraw(int amt) {
        int newBal = bal - amt;
        if (newBal >= 0) {
            bal = newBal;
        }
        return newBal >= 0;
    }

    // other methods ...
}
```

Fig. 2. Sample JML specification of bank accounts

```
public class Account_JML_Test
    extends junit.framework.TestCase {

    public void oracleWithdraw(
        Account receiver, int amt) {
        try {
            receiver.withdraw(amt);
        } catch (JMLEntryPreconditionError e) {
            // meaningless test
        } catch (JMLAssertionError e) {
            junit.framework.Assert.fail();
        } catch (java.lang.Throwable e) {
            return;
        }
    }

    // other oracle methods ...
}
```

Fig. 3. Part of test oracle class for `Account`

if the request amount is less than or equal to the current balance.

Fig. 3 shows a part of test oracle class generated for the `Account` class. In particular, it shows the oracle method for the `withdraw` method; it is a simplified version of the actual oracle method generated by a proof-of-concept tool (see Section IV). As in [6], the oracle method calls the method under test (`withdraw`) with test data passed as arguments and decides test success or failure based on the assertion error detected by the runtime assertion checker.

```

public class Account_JML_TestData
  extends Account_JML_Test {

  public void testWithdraw1() {
    Account receiver;
    int arg0;
    try {
      receiver = new Account(1);
      arg0 = 1;
    } catch (java.lang.Throwable e) {
      return;
    }
    oracleWithdraw(receiver, arg0);
  }

  // other test methods ...
}

```

Fig. 4. Part of test data class for Account

For a complete automation of testing, our approach also generates test data for each method under test. For the `withdraw` method, for example, the prototype tool generates two test cases, one that makes the return value true and the other that makes the return value false. In this case, each test case is a pair of an `Account` object and an integer value. Each test case becomes a separate JUnit test method, and the first test case is shown in Fig. 4. The JUnit test method calls the corresponding oracle method to perform a test execution and to decide the test result.

The target class and generated oracle and test data classes are compiled and run by the prototype tool, either on the command-line or through a GUI (see Section IV).

### B. Key Research Issues

In addition to the engineering challenge of extending, adapting, and integrating different components, such as JML, JUnit, and genetic algorithms, the key research component of our work is to advance the genetic algorithm techniques to generate test data for object-oriented programs. In particular, our study focuses on answering the following questions.

- *How to encode objects and values as chromosomes and genes?* An efficient way need be defined to represent test data as chromosomes. A chromosome is a sequence of genes—e.g., a receiver and arguments for a method call. In object-oriented programs, a gene may be an object or a primitive value. For an object gene, its state may be described as a sequence of statements that need be executed to bring the object to the state of interest. However, the fact that object’s state is encapsulated creates a fundamental difficulty. It is not trivial and often impossible to automatically create an object of the desired state.
- *What genetic operations?* We need to define genetic

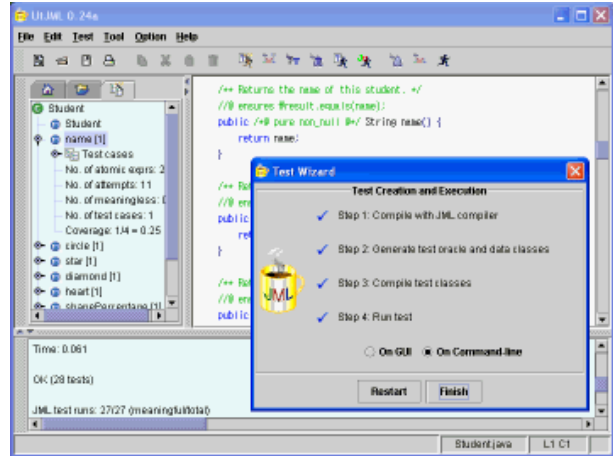


Fig. 5. Sample screen of a proof-of-concept tool

operations and an evolutionary process to create a better set of chromosomes. The well-known genetic operations such as cross-over and mutation may be employed, but care must be taken to ensure that the test data matches the signature of the method being tested. It is not clear, however, whether manipulating objects as bit patterns would be effective.

- *How to determine fitness of test data?* We need to define the fitness of individual test data and the test set as a whole. Our plan is to study both the specification-based coverage and the program-based coverage. In particular, condition coverage on the postcondition seems promising, though we are still evaluating its effectiveness. In terms of tool support, we need to generate coverage information from specifications or source code, by extending the JML’s runtime assertion checker or instrumenting source code.

## IV. CURRENT STATUS

We have implemented a proof-of-concept tool that completely automates unit testing of Java programs (see Fig. 5). The tool integrates several JML tools, JUnit, and a new test case generation tool. Upon a click of mouse, the tool generates test data, performs test execution, and decides test success or failure.

The JML compiler [6] was extended to make compiled bytecode produce test coverage information. Currently we only support condition coverage on postconditions to determine the coverage of generated test suite. The idea of condition coverage on postconditions seems to be new. We can view postconditions as boolean expressions and apply the condition coverage criteria to them. That is, for each test data we check whether it covers a new combination of atomic boolean expressions. An atomic boolean expression of a postcondition is a boolean

expression that can not be further divided into smaller boolean expressions. We are currently evaluating the effectiveness of this notion of condition coverage.

The new test case generation tool extends the work of one of the authors [6]. The tool generates not only the test oracle class but also test data class that contains a set of test data. Currently the tool generates test data randomly; however, our plan is to implement genetic algorithms to generate test data intelligently and efficiently, as discussed in Section III. The process of test data generation is iterative, and the condition coverage is used to decide whether newly generated test data should be added to the test suite or not. The process ends if no more test data is found that improves the coverage of the test suite.

The generated test oracle and test data classes are compiled and run by the tool, and test success or failure are determined by observing assertion violations at run-time [7].

## V. RELATED WORK

The work most related is the use of specifications as test oracles and the applications of genetic algorithms to automatic test data generation.

Peters and Parnas developed a tool that generates C++ test oracle procedures from relational program specifications [14]. The generated oracle procedure checks if a given input and output pair satisfies the relation described by the specification. Cheon and Leavens employed a runtime assertion checker as a test oracle engine [7]. The runtime assertion checker supports a significant subset of JML, including pre and postconditions, class invariants, intraconditions, abstract value manipulation [15], and specification inheritance. Both approaches are semi-automatic in that the user has to provide test data. Our work extends the approach of Cheon and Leavens to generate test data and perform test execution automatically.

A substantial amount of research work can be found in the literature on automatic test data generation. The work can be classified into two categories: specification-based and program-based. The *specification-based approach* derives test data from formal specifications such as postconditions by employing a constraint solver or a theorem prover (e.g., [16]). In spite of much research, the approach has seldom been applied in practice, perhaps due to the use of formal specifications or the lack of complete automation, especially by the underlying constraint solver or theorem prover. Our approach also uses specifications, but the specifications are written in boolean expressions of programming languages, and it does not rely on a constraint solver or a theorem prover. The *program-based approach*, such as statement testing, branch testing, condition testing and path testing,

generates test data by analyzing the source program to be tested (e.g., [17]). This approach is practical and supported by several commercial tools; however, it requires separate test oracle code to be written. Our approach uses genetic algorithms to generate test data and supports both the program-based and the specification-based approaches.

Though not much work is found in the literature, genetic algorithms have been applied to testing—in particular to generate test data automatically [10] [11]. However, previous work has been limited to simple data types, such as integers, and relatively little attention has been given to object-orientation [12]. Our work extends genetic algorithms to object-oriented programs and integrates with formal specifications and automatic test execution to achieve a complete automation of unit testing.

## VI. DISCUSSION

Our proof-of-concept tool shows that a complete automation is possible for unit testing object-oriented programs, and we believe that such an automation is practical and effective. The reason behind this belief is that, in well-written object-oriented programs, methods tend to be small (to be reusable), and such small methods are amenable to automatic testing, e.g., higher coverage can be achieved with automatically generated test data. We also believe that automated testing techniques such as ours can complement manual testing; manual testing can focus more interesting problems, such as inter-class testing.

Our approach advances the current state of the art of object-oriented unit testing. It contributes techniques of applying genetic algorithms to automatic generation of object-oriented test data. The integrated approach of combining JML, evolutionary techniques, and JUnit has the potential to automate unit testing of Java classes and interfaces and, thus, to perform continuous testing. This is significant because complete automation will reduce the cost of software testing dramatically and also facilitate continuous testing. It is reported that at least 50% of the total software development costs is due to testing [1], and 10–15% of development time is wasted due to frequent stops for regression testing [18]. Automation will also help get rid of cognitive biases that have been found in human testers [19]. Our approach also provides an effective tool for finding errors in assertions [7].

An earlier proof-of-concept implementation of our approach is available from <http://www.cs.utep.edu/~cheon/download>.

## ACKNOWLEDGEMENT

The work of the authors was supported in part by The University of Texas at El Paso through URI grant 14-

## REFERENCES

- [1] B. Beizer, *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, 1990.
- [2] M. A. Cusumano and R. W. Selby, "How Microsoft builds software," *CACM*, vol. 40, no. 6, pp. 52–61, June 1997.
- [3] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: A notation for detailed design," in *Behavioral Specifications of Businesses and Systems*, H. Kilov, B. Rumpe, and I. Simmonds, Eds. Boston: Kluwer Academic Publishers, 1999, pp. 175–188.
- [4] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification," *Science of Computer Programming*, vol. 55, no. 1-3, pp. 185–208, Mar. 2005.
- [5] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, Feb. 2005, to appear.
- [6] Y. Cheon and G. T. Leavens, "A runtime assertion checker for the Java Modeling Language," in *Proceedings of the International Conference on Software Engineering Research and Practice, Las Vegas, Nevada*, June 2002, pp. 322–328.
- [7] —, "A simple and practical approach to unit testing: The JML and JUnit way," in *Proceedings of European Conference on Object-Oriented Programming, Málaga, Spain*, ser. Lecture Notes in Computer Science, vol. 2374. Springer-Verlag, June 2002, pp. 231–255.
- [8] J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
- [9] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *IEEE Computer*, vol. 27, no. 6, pp. 17–26, June 1994.
- [10] R. Pargas, M. J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Journal of Software Testing, Verification and Reliability*, vol. 9, no. 3, pp. 263–282, Sept. 1999.
- [11] P. McMinn, "Search-based software test data generation: A survey," *Journal of Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, June 2004.
- [12] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, MA*, July 2004, pp. 119–128.
- [13] K. Beck and E. Gamma, "Test infected: Programmers love writing tests," *Java Report*, vol. 3, no. 7, pp. 37–50, 1998. [Online]. Available: <http://junit.sourceforge.net/doc/testinfected/testing.htm>
- [14] D. Peters and D. L. Parnas, "Generating a test oracle from program documentation," in *Proceedings of the ACM SIGSOFT International Symposium on Testing and Analysis, Seattle, Washington*, Aug. 1994, pp. 58–65.
- [15] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards, "Model variables: Cleanly supporting abstraction in design by contract," *Software—Practice and Experience*, vol. 35, no. 6, pp. 583–599, May 2005.
- [16] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Roma, Italy*, July 2002, pp. 123–133.
- [17] N. Gupta, A. Mathur, and M. L. Sofia, "Generating test data for branch coverage," in *Proceedings of 15th IEEE International Conference on Automated Software Engineering, Grenoble, France*, Sept. 2000, pp. 219–228.
- [18] D. Saff and M. D. Ernst, "An experimental evaluation of continuous testing during development," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, MA*, July 2004, pp. 76–85.
- [19] W. Stacy and J. MacMillan, "Cognitive bias in software engineering," *Communications of the ACM*, vol. 38, no. 6, pp. 57–63, June 1995.