2-2008

Integrating Random Testing with Constraints for Improved Efficiency and Diversity

Yoonsik Cheon
The University of Texas at El Paso, ycheon@utep.edu

Antonio Cortes

Martine Ceberio
The University of Texas at El Paso, mceberio@utep.edu

Gary T. Leavens

Comments:
Technical Report: UTEP-CS-08-07

Integrating Random Testing with Constraints for Improved Efficiency and Diversity

Yoonsik Cheon, Antonio Cortes, Martine Ceberio, and Gary T. Leavens

TR #08-07
February 2008; revised May 2008

Keywords: random testing, constraint solving, test data generator, pre and postconditions, object-oriented programming, JML language.

1998 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — class invariants, formal methods, programming by contract; D.2.5 [*Software Engineering*] Testing and Debugging — testing tools (e.g., data generators, coverage testing); 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; I.2.8 [*Artificial Intelligence*] Problem Solving, Control Methods, and Search — Graph and tree search strategies.

Published in the *Proceedings of SEKE 2008, The 20-th International Conference on Software Engineering and Knowledge Engineering, July 1-3, 2008, San Francisco, CA,* pages 861–866, July 2008.

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

Integrating Random Testing with Constraints for Improved Efficiency and Diversity

Yoonsik Cheon, Antonio Cortes, Martine Ceberio
Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968

Gary T. Leavens
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816

Abstract

Random testing can be fully automated, eliminates subjectiveness in constructing test data, and increases the diversity of test data. However, randomly generated tests may not satisfy program's assumptions such as method preconditions. While constraint solving can satisfy such assumptions, it does not necessarily generate diverse tests and is hard to apply to large programs.

We blend these techniques by extending random testing with constraint solving, improving the efficiency of generating valid test data while preserving diversity. For domains such as objects, we generate input values randomly; however, for values of finite domains such as integers, we represent test data generation as a constraint satisfaction problem by solving constraints extracted from the precondition of the method under test. We also increase the diversity of constraint-based solutions by incorporating randomness into the solver's enumeration process. In our experimental evaluation we observed an average improvement of 80 times without decreasing test data diversity, measured in terms of the time needed to generate a given number of valid test cases.

1 Introduction

A random approach to generating test data has the potential for finding faults that are difficult to find in other ways, because it eliminates subjectiveness in constructing test data and increases the diversity of input values. It also facilitates test automation. Our recent work explored random test data generation to unit testing of Java classes annotated with assertions [6]. A test case for a method is constructed dynamically to ensure that it satisfies the precondition of the method under test. If a test case does not satisfy the precondition, it is inadequate to test the method because the precondition is the client's obligation [5].

However, randomly generated tests may not satisfy the program's assumptions. In our case these assumptions are method preconditions formally written in JML [11], an interface specification language for Java. If the preconditions are not trivial, the chances are very low that randomly-generated test data will satisfy them. In our recent experiment we observed that up to 99% of randomly-generated test cases did not meet the preconditions of the methods under test [6].

In this paper we propose an extension to pure random testing to improve the efficiency of generating a given number of valid test cases that satisfy the precondition of the method under test. The key idea of our extension is to integrate constraint solving with random test data generation. For method parameters of continuous or infinite domains such as objects, we generate test values randomly. However, for parameters of discrete and finite domains such as integers, we represent test data generation as a constraint solving problem, where constraints are the assertions of the method precondition that involve the parameters. This extension is based on our observation that about 10% to 50% of methods have formal parameters of discrete and finite domains and the precondition assertions on these parameters can be efficiently solved by finite-domain constraint solvers.

We evaluated the effectiveness of our approach by implementing a prototype tool based on our own random testing tool called JET [6] and an open-source constraint solver called Cream [14]. In our experiments we observed an average improvement of 80 times over pure random testing measured in the time needed to generate a given number of valid test cases that satisfy a method's precondition (see Section 5).

2 Background

Our long term goal is to fully automate unit testing of Java classes, from test data generation to test execution and test outcome decision. The class under test is assumed to be annotated with a JML specification (see Section 3); formal specifications such as method postconditions are used as test oracles. Each method of the class is tested separately, and thus a *test case* consists of a receiver object and argument values. We generate test cases automatically—the subject of this paper—

and perform test executions by invoking the method under test with the generated test data. We use JML's runtime assertion checker to recognize invalid test cases as well as to decide test outcomes; i.e., we interpret certain types of assertion violations, such as postcondition violations, as test failures [5, 13].

Previous work has either generated test data randomly (e.g., [6, 7, 8, 12]) or has generated tests purely by solving constraints (e.g., [1, 3]). In *random testing*, a random object of a class C is obtained via a *call sequence*, consisting of one constructor and zero or more method invocations, such as $C\ o$=**new** $C_0(); o.m_1(); o.m_2(); \ldots; o.m_n()$. In such a call sequence, m_1 through m_n mutate the state of o. Methods m_i and their arguments are selected randomly from appropriate methods of C.

In *constraint-based testing*, test cases are generated by solving constraint satisfaction problems. A constraint satisfaction problem consists of a finite set of variables and a set of constraints on those variables. Each variable is associated with a set of possible values, known as its domain. A constraint is simply a relation on some subset of these variables. A solution to a constraint satisfaction problem is an assignment of a value to each variable from its domain, such that all the constraints are satisfied. There are efficient constraint solvers for finite domains. For example, Cream [14] is a Java class library for solving constraints on finite domains. In Cream, the collection of variables, domains, and constraints are called a *constraint network*. Cream provides several built-in strategies, called *solvers*, that enumerate solutions for constraint networks (see Section 3).

3 Illustration

To illustrate our approach, let us consider the `Account` class given in Figure 1. This class is annotated with JML assertions written as special comments. The keyword **`spec_public`** states that the private field `bal` is treated as public for specification purpose; e.g., it can be used in the specifications of public methods. A method specification precedes the declaration of the method and specifies its precondition (**`requires`** clause), its frame condition (**`assignable`** clause), and its postcondition (**`ensures`** clause). The keyword **`\old`** denotes the pre-state value of its expression and is used in the specification of a mutation method such as the `transfer` method that changes the state of an object.

Consider the `transfer` method and the likelihood of randomly generating a valid test case. To test this method, we need a test case consisting of two `Account` objects—one for the receiver and the other for the argument—and an integer. Let p be the probability of generating an `Account` object successfully, i.e., the probability that all the calls in its call sequence terminate normally. Then, the probability of generating a valid test case is p^2q, where q is the probability that the test case satisfies the method precondition,

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

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

  /*@ requires amt > 0 && amt <= acc.bal;
    @ assignable bal, acc.bal;
    @ ensures bal == \old(bal) + amt
    @   && acc.bal == \old(acc.bal - amt); @*/
  public void transfer(int amt, Account acc) {
    acc.withdraw(amt);
    deposit(amt);
  }

  // The rest of the definition including:
  // Account(Account), deposit(int),
  // withdraw(int), and int balance().
}
```

Figure 1. JML-annotated class

`amt > 0 && amt <= acc.bal`. In a purely random approach, q is 0.25 if we conservatively estimate the probability of satisfying each conjunct to be 0.5.

However, we can be clever by selecting an `amt` value such that it automatically satisfies the precondition, thus improving q to 1. To do this, we solve the constraints on `amt` imposed by the precondition; i.e. we represent the problem of test case generation partly as a constraint satisfaction problem. For this particular case, we need to solve the constraints: $x > 0$ and $x \leq B$, where B is `acc.bal`, the balance of the randomly-generated `Account` object. These constraints can be easily translated to the following Cream code.

```
Network net = new Network();
IntVariable x = new IntVariable(net);
x.gt(0);
x.le(acc.bal);
Solver solver = new DefaultSolver(net);
Solution solution = solver.findFirst();
int valX = solution.getIntValue(x);
```

In Cream, constraints on variables are expressed using framework methods such as `gt` and `le`. The Cream framework also provides a set of arithmetic methods (e.g., `add` and `multiply`) for writing arithmetic expressions.

Our approach improves the probability of generating valid test cases dramatically. Recall that an object in a test case is represented as a call sequence. Thus a test case is set of such call sequences. We can apply constraint solving to each of the method invocations in the call sequence. For example, doing this for the `transfer` method improves not only q but also p—the probability of generating a valid `Account` object—even more dramatically, which has a greater impact on the overall probability.

4 Our Approach

The problem is to generate test cases that satisfy the precondition of the method under test. The key idea of our approach is to solve constraints for values of finite domains such as integer while generating random values for other types such as objects. There are two main issues. The first is how to identify and extract constraints from method preconditions written in JML; not all precondition assertions are constraints on the parameters of interest. The second issue is how to translate the extracted constraints to constraint solving code.

We address the first issue by first desugaring the executable subset of JML precondition assertions to a single boolean expression and then converting it to a *disjunctive normal form*. As in the predicate-based approach to test data generation [16], we consider each disjunct of the disjunctive normal form as a constraint to solve. For the second issue, we define a translation from disjuncts of the normal form to Cream code.

As in [6], the receiver object of a test case is generated randomly, but the arguments are generated in a combination of a random approach and a constraint satisfaction problem. For this, we classify formal parameters of a method into two categories.

Definition 1 *A formal parameter of a method is a* constrained variable *if its declared type is an integral type such as* `int`*. A formal parameter that is not a constrained variable is called an* unconstrained variable*.*

As outlined below, we first prepare the preconditions for possible constraint solving, and then generate test cases. This preparation involves the following steps.

1. Desugar the method precondition to a single boolean expression (see Section 4.2).
2. Convert the boolean expression to a disjunctive normal form (see Section 4.3).
3. For each disjunct of the normal form that has constraints on any of the constrained variables, translate it to constraint solving code (see Section 4.4).

Test case generation is then done by repeating the following until a fixed number of valid tests are generated. We generate random values for the receiver and all arguments as in our previous work [6]. If the generated values do not satisfy the precondition of the method under test, we then find new values for the constrained variables by solving constraints extracted from the precondition.

1. Generate random values for the receiver and all arguments (see Section 4.1).
2. If the values do not satisfy the precondition, find new values for the constrained variables by invoking the constraint solving code (from Step 3 above).

If no constraints were identified for the constrained variables (see Section 4.4) or no solution found, then we repeat the whole process some fixed number of times.

4.1 Random Value Generation

We use the method of [6] to generate the initial random values for the receiver and arguments; e.g., a random object is constructed as a sequence of mutation method invocations preceded by a constructor invocation. However, one important difference is that the receiver and arguments of each method invocation in the object sequence are also generated by using the new approach, because they also have to satisfy the precondition of the invoked method.

4.2 Precondition Desugaring

JML features a great deal of syntactic sugar to enhance Java expression syntax by introducing a rich set of JML-specific expressions and several specification clauses. We desugar the executable subset of a method precondition to a single boolean expression.[1] The desugaring process consists of two steps: desugaring of method specifications and simplification of boolean connectives.

4.3 Disjunctive Normal Form

We use a disjunctive normal form to identify the set of constraints that can be solved independently to find values for the constrained variables that satisfy the precondition of the method under test. A *disjunctive normal form* (*DNF*) is a standardization or normalization of a logical formula which is a disjunction of conjunctive clauses, e.g., $c_1 \vee \ldots \vee c_n$, where each c_i is of the form $e_1 \wedge \ldots \wedge e_m$.

4.4 Constraint Identification

From a method precondition converted to a DNF, we identify constraints on the constrained variables. We assume all Java/JML boolean connectives are already desugared except for conjunction, disjunction, and negation.

Definition 2 *A constrained variable is* executable *in an expression, e, if it has a free occurrence in e other than in a subexpression of a receiver or an argument to a method or constructor call. A boolean-valued expression that contains no logical connectives is an* executable constraint *if it contains at least one executable constrained variable.*

We often use the term "constraint" as shorthand for "executable constraint." The following gives an equivalent characterization of executable constraints.

[1] The executable subset is JML expressions and assertions that are translated to runtime assertion checking code by the JML compiler (`jmlc`) [4].

Theorem 1 *A boolean-valued expression e that contains no logical connectives is an executable constraint if and only if it is of the form $e_1 \diamond e_2$, where both e_1 and e_2 are of an integral type, $\diamond$ is a relational or equality operator, and either e_1 or e_2 contains an executable constrained variable.*

This follows from our definitions of constrained variables (being of integral types) and constraints (being boolean expressions).

Uses of constrained variables are not executable when they occur as arguments to method calls, because Cream does not understand arbitrary methods, and hence it cannot solve for such occurrences. For example, `Math.abs(x) > 10` is not an executable constraint because Cream cannot handle the call to `abs`. (To make it executable, one has to translate the expression manually to one that can be handled by Cream.) On the other hand, in the expression `Math.abs(x - 10) > y`, although `x` is not executable, `y` is, and thus the entire expression is an executable constraint. Cream can thus try to satisfy this assertion when `x` has a random value, even though it does not control `x`'s value.

Definition 3 *A conjunctive clause of the form, $e_1 \wedge \ldots \wedge e_n$, where e_i's do not use disjunction, is an* executable constraint *if at least one e_i is an executable constraint.*

A conjunctive clause that is not an executable constraint may contain a constrained variable, but not one that is executable. If each e_i of the clause is an executable constraint, the solutions of the whole constraint are in general valid test data; otherwise the validity of the test data depends on the e_i's that are not constraints.

4.5 Constraint Solving Code

Given a DNF $c_1 \vee \cdots \vee c_n$, we consider each conjunct clause c_i independently. If c_i is a constraint, we translate it into Cream constraint solving code. The translated Cream code has the following general structure, where x_i's are the constrained variables appearing in the constraint c_i and y_i's are fresh variables to store a solution.

```
Network net = new Network();
IntVariable x_1 = new IntVariable(net);
...
IntVariable x_m = new IntVariable(net);
⟨Translated constraints of c_i⟩
Solver solver = new DefaultSolver(net);
Solution solution = solver.findFirst();
int y_1 = solution.getIntValue(x_1);
...
int y_m = solution.getIntValue(x_m);
```

This skeletal Cream code has three parts. It first creates a new constraint network and adds the constrained variables of c_i to the network. It then specify the constraints of c_i using the added constrained variables (see Section 4.6 below). It finally solves the constraints and retrieves a solution.

JML Expression		*Cream Constraint Code*
$x \geq 10$	x	`IntVariable v1 = null;` `v1 = x;`

	10	`int i1 = 0;`
		`i1 = 10;`
		`IntVariable v2 = new IntVariable(net);`
		`v2.equals(i1);`

	$\geq$	`v1.ge(v2);`

$x + y$ < size()	$x + y$	`IntVariable v3 = null;`
		`IntVariable v5 = null;`
		`v5 = x;`
		`IntVariable v6 = null;`
		`v6 = y;`
		`v3 = v5.add(v6);`

	size()	`int i2 = 0;`
		`i2 = size();`
		`IntVariable v4 = new IntVariable(net);`
		`v4.equals(i2);`

	$<$	`v3.lt(v4);`

Figure 2. Sample translation of a conjunct $x \geq 10 \ \wedge \ x + y <$ size(), where x and y are constrained variables.

4.6 Constraint Translation

Given a conjunctive clause of the form $e_1 \wedge \ldots \wedge e_n$, we translate each e_i into Cream if it is a constraint; otherwise, we ignore it because it does not constrain the parameters of interest or the constraint cannot be handled in Cream. For this, we defined a set of translation rules and the rules systematically translate JML expressions to Cream code by converting Java/JML operators to Cream framework methods and by introducing temporary variables as necessary. As an example, consider a conjunctive clause $x \geq 10 \ \wedge \ x + y <$ size(), where x and y are constrained variables. It is translated to the Cream constraint code shown in Figure 2.

5 Evaluation

We performed several experiments semi-automatically to evaluate the effectiveness and efficiency of our approach. One challenge for our experiments was that the constraint solving code should run in the same environment as that of the method under test because the constraints are written in terms of the names available to the method (e.g., formal parameters, fields of the receiver, and other methods of the class) and it should handle JML-extensions to Java (e.g., specification-only variables). Our solution was to manually inject constraint-solving code to the instrumented source code produced by the JML compiler.[2] We also extended both JET [6] and Cream [14]. JET is an automated unit testing tool that generates test cases randomly, and our extension was to implement the algorithm sketched in Section 4 as a new test data generation strategy,

[2]The JML compiler (`jmlc`) has an option (`--print`) to produce the instrumented source code before compiling it to bytecode.

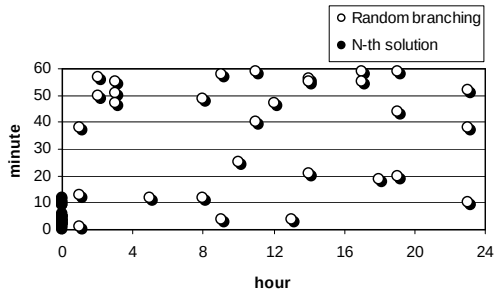


Figure 3. Diversity of solutions found for the constraint $0 \leq \text{hour} < 24 \wedge 0 \leq \text{minute} < 60$**.**

which essentially calls the injected constraint solving code as necessary.

The Cream extension was made to increase the diversity of generated test cases. Our initial experiments produced many duplicate or redundant test cases, and we shortly learned that this was caused by Cream's deterministic algorithm for enumerating solutions. If there are multiple solutions, Cream enumerates them in increasing order by always returning the smallest solution first. To remedy this problem, we introduced two techniques. The first technique, implemented without modifying the Cream framework, was to find and use the n-th solution. The second technique called a *random branching* introduced randomness in finding a solution by modifying the Cream framework. This modification explores the search space by bisecting the domains of variables. The original Cream explores by, at each step, selecting one variable, then exploring the rest of the search space by using only the first half of the domain of this variable; later, it looks for solutions in the other half. We changed this deterministic behavior by randomly choosing the half to explore first. This small modification greatly increased the diversity of the generated test cases, as shown in Figure 3, and improved the effectiveness of our approach.

We selected three classes for our experiments. As our approach offers benefits to methods with integral parameters, we selected classes of this characteristic.

1. `PINChecker`: This class stores, resets, and checks the validity of a personal identification number (PIN). The method parameters are a combination of objects and primitive data types.

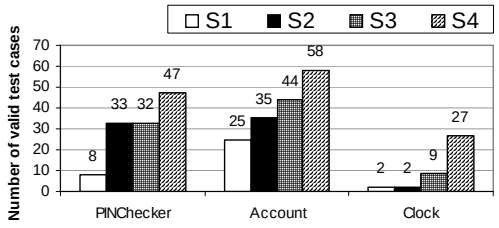
2. `Account`: This class represents a bank account and has methods such as `deposit`, `withdraw`, and `transfer` (see Figure 1). Most parameters are of integer type with non-negativeness constraints. Interestingly, using our approach we discovered an error in the `transfer` method—an overflow caused by adding a large number.

3. `Clock`: This class has a single method with a constraint like `0 <= hour && hour < 24`.

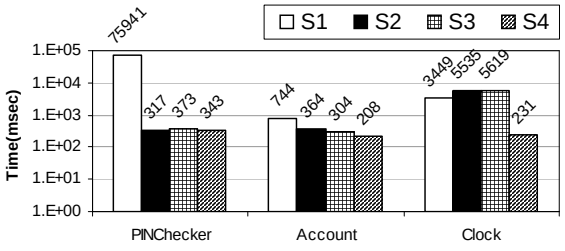
We evaluated the performance of four test generation strategies: the random strategy presented in [6] and three variations of our new approach (see Figure 4(a)). The variations correspond to the ways we used or modified the Cream framework. We first measured the number of non-duplicate, valid test cases generated by each strategy for each of the selected classes (see Figures 4(b)). As expected, constraint solving improved the effectiveness of random testing though the exact improvement varied widely depending on the characteristics of the classes. In particular, random branching is the most effective; for classes with non-trivial preconditions such as `PINChecker` and `Clock`, we noticed huge improvements in the numbers of generated test cases (i.e., 592% and 1331%, respectively). We next measured the time needed to generate a fixed number of non-duplicate, valid test cases (see Figure 4(c)). Again the improvements varied widely. The constraint strategy with random branching, for example, showed 22140%, 358%, and 1493% improvements for the three classes over the pure random strategy, giving an average of 7997% improvement. We ran the experiments on an AMD Turion™ 64X2 1.80 GHz with 2 GB of main memory.

Strategy	*Description*
S1	Pure random
S2	Constraint with first solution
S3	Constraint with n-th solution
S4	Constraint with random branching

(a) Test data generation strategies



(b) Average numbers of valid test cases generated for 100 attempts per method.



(c) Average time needed to generate 100 valid test cases per method.

Figure 4. Experimental results

6 Related Work

Previous work focused either on random testing (e.g., [6, 7, 8, 12]) or constraint solving (e.g., [1, 3]) in isolation, without taking an advantage of synergistic effects of both approaches. Some work also used meta-heuristic information to guide the search for valid test data (e.g., [9]); e.g., in genetic algorithms, call sequences that are likely to produce valid test data are selected and then made to evolve by applying genetic operations such as mutation and crossover [15].

The most closely related work is JML-TT [2], a specification animator for JML based on the constraint logic programming. It can execute methods leaving primitive parameters undefined or with ranges of values specified for them, showing a counter-example upon a specification violation. JML-TT can also generate test cases with boundary values. For this, it first extracts boundaries from preconditions and invariants. For example, a boundary test case for the `transfer` method of class `Account` could be `a1.transfer(1, a2)`, where `a1` and `a2` are objects of class `Account`, with balances zero and `Integer.MAX_VALUE`, respectively. Once a boundary test case is identified, it constructs needed objects (e.g., `a1` and `a2`) using the animator. However, this step is undecidable and thus may require a human assistance.

The `jmle` tool [10] is another specification animator for JML. It translates a JML specification to an executable Java implementation. The generated code relies on a constraint solver to simulate the specified behavior; i.e., the tool transforms a JML specification into a constraint satisfaction problem. For the approach to work, the specification should be detailed enough so that the constraint solver can reach the postcondition from the precondition. While `jmle` does not generate test cases, it would be possible to use some of its techniques to interpret a fixed set of method calls.

Jartege [12] is similar to JET in that it generates test data randomly and uses the runtime assertion checker as a test oracle procedure. There are also assertion-based random testing tools for other languages such as Eiffel [7].

7 Conclusion

We combined random testing with constraint solving to generate valid test data—test data that satisfies the precondition of the method under test. The key idea of our approach is first to generate random test data and then, if the generated test data does not satisfy the precondition, to solve constraints extracted from the precondition for parameters of finite domains such as integers. Our approach improves both the effectiveness of random testing from 6 to 13 times measured in the number of valid test cases generated and the efficiency from 4 to 221 times measured in the time needed to generate a given number of valid test cases.

Acknowledgment

Cheon's work was supported in part by NSF grants CNS-0509299 and CNS-0707874 and by the Department of Defense. Leavens's work was supported in part by NSF grant CNS-0808913.

References

[1] B. K. Aichernig and P. A. P. Salas. Test case generation by OCL mutation and constraint solving. In *Proc. of QSIC, Melbourne, Australia, September 19-20, 2005*, pages 64–71, 2005.

[2] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. In *ICFM*, volume 3582 of *LNCS*, pages 75–90. Springer-Verlag, July 2005.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ACM SIGSOFT ISSTA, Rome, Italy*, pages 123–133, July 2002.

[4] 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*, 7(3):212–232, June 2005.

[5] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP*, volume 2374 of *LNCS*, pages 231–255. Springer-Verlag, June 2002.

[6] Y. Cheon and C. E. Rubio-Medrano. Random test data generation for Java classes annotated with JML specifications. In *SERP, Volume II, June 25–28, 2007, Las Vegas, Nevada*, pages 385–392, June 2007.

[7] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *ACM SIGSOFT ISSTA*, pages 84–94. ACM, 2007.

[8] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software—Practice and Experience*, 34(11):1025–1050, Sept. 2004.

[9] M. Harman and B. F. Jones. Search-based software engineering. *Info. & Software Technology*, 43(14):833–839, 2001.

[10] B. Krause and T. Wahls. jmle: A tool for executing JML specifications via constraint programming. In *FMICS*, volume 4346 of *LNCS*, pages 293–296. Springer-Verlag, 2006.

[11] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Soft. Eng. Notes*, 31(3):1–38, Mar. 2006.

[12] C. Oriat. Jartege: A tool for random generation of unit tests for Java classes. In *ICQSA*, volume 3712 of *LNCS*, pages 242–256. Springer-Verlag, Sept. 2005.

[13] D. K. Peters and D. L. Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, Mar. 1998.

[14] N. Tamura. Cream: Class library for constraint programming in Java. Available from `http://bach.istc.kobe-u.ac.jp/cream/`, as of January 2008.

[15] P. Tonella. Evolutionary testing of classes. In *Proc. of the ACM SIGSOFT ISSTA, Boston, MA*, pages 119–128, July 2004.

[16] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.