Grey-box Testing and Verification of Java/JML

Frédéric Dadeau and Fabien Peureux
LIFC - Université de Franche-Comté / INRIA CASSIS
16 route de Gray – 25030 Besançon, France
Email: {frederic.dadeau, fabien.peureux}@lifc.univ-fcomte.fr

***Abstract*—We present in this paper the application of constraint solving techniques to the validation and automated test cases generation for Java programs, annotated with JML specifications. The Java/JML code is translated into a constraint representation based on a subset of the set-theory, which is well-suited for modelling object-oriented programs. Symbolic code execution techniques can then be applied to produce test cases, using classical structural test selection criteria, or to detect possible runtime errors, and non-conformances between the Java code and its embedded JML model.**

***Keywords*-symbolic execution, constraint solving, Java/JML, grey-box testing, set-theory**

```
public class A {

   //@ invariant x > 0;
   short x;
   static byte y;
   A z;

   /*@ requires a != null &&  y > x && a != this;
    *@ assignable \everything;
    *@ ensures \result > 0;
    *@ signals (BarException e) \old(x) == x;
    */
   public int foo(A a) throws BarException { ... }
}
```

Fig. 1. General Form of the Contracts in JML

I. Introduction

Design-by-Contract (DbC) [1] is a lightweight technique that aims at introducing formal methods for non-specialists. It consists of adding contracts to the code of a program for basic modelling purposes, namely, specifying invariants on classes, and pre- and post-conditions on methods. The annotations are written directly inside the code, which simplifies its use by developers, who do not need to learn a whole new modelling language, beside some simple keywords.

Over the past few years, several annotation languages have arisen, such as JML [2] for Java, Spec# [3] for C#, ACSL [4] for C, etc. Annotation languages are very convenient at two levels [5]. At the verification level, it is possible to take the annotations into account to ease the proof of the considered program, and strengthen the properties one wants to specify on the system. At the validation level, the contracts can be used for generating black-box test data, using method preconditions, and to assign the verdict of the test, using the postcondition by runtime assertion checking techniques.

Various approaches use constraint solving techniques together with annotated programs either to produce test cases or to verify program correctness. In most cases, the program is flattened into Single Static Assignment form and a set of constraints is computed as a *path condition* that represents the conjunction of the decision predicates on the path.

We propose here to reinvest constraint solving techniques that have been originally designed for the symbolic animation of B machines [6], in order to provide a flexible and expressive means for both verifying and validating a Java program annotated with JML constructs. This technique is based on a graph representation of the program structure/code. The symbolic evaluation is then done on-the-fly, when exploring the graph to cover a specific path, that can be extracted using classical structural coverage strategies (all-nodes, all-decisions, etc.). We show in this paper how the underlying set-theory of B, tool-supported by a customized constraint solver, can be used to efficiently represent object-oriented programs/models.

The paper is organised as follows. Section II presents an overview of the Java Modeling Language syntax and semantics. Then, Section III illustrates how Java programs (i.e., data model and method codes) can be represented in the set-theory handled by our constraint solver. Section IV presents our grey-box test generation technique, that takes into account both the program and its specification. Then, we present in Sect. V a verification technique, that aims at finding three types of non-conformances between the program and its specification. Finally, Section VI presents the related works, before concluding in Sect. VII.

II. Overview of the JML syntax and semantics

The Java Modeling Language is an annotation language based on Design-By-Contract paradigm, introduced by Eiffel [1]. The contracts are written inside the Java code, which facilitates the use of JML by developers. In this section, we briefly introduce the syntax and the semantics of JML.

Contracts can be defined as illustrated in Fig. 1. Keyword `requires` indicates the precondition, namely the contract that has to be fulfilled by the caller of the method. The `assignable` clause provides the frame condition, namely the set of attributes that can be assigned inside the method body. The `ensures` clause designates the normal postcondition that is supposed to be established when the method terminates normally, i.e. without throwing an exception. Finally, the `signals` clause designates the postcondition that is supposed to be established if the method terminates abnormally, i.e. by throwing an exception. In this case, the specification indicates the exception and its possible postcondition. In the example, the presence of an exception of any type (specified by the generic Exception class) implies that attribute `x` is not modified (its after value, designated by `x`, has to be equal to its before value, designated by `\old(x)`).

Besides pre- and postconditions, JML contracts are completed by class invariants, that apply on class attributes.

The JML Runtime Assertion Checker (RAC) instruments the Java code (without modifying its behaviour) to performed on-the-fly verification that will raise a specific set of exceptions in case of verification failure. The verifications performed are the following:

(i) at the beginning of the method, the class invariant is checked, along with the precondition. All `\old` expressions used in the postconditions are evaluated.

(ii) at the end of the method body execution, the normal postcondition is checked, replacing the `\old` expressions with their previously-stored values.

(iii) if the method body throws an exception, then there are two cases: either it is a JML-specific exception, raised inside the body of the method and it is thus propagated, or it corresponds to an exception of the program, and thus the corresponding exceptional postcondition (`signals` clause) is checked (if it exists), before propagating the exception.

When executing tests, the absence of JML-specific exceptions means that the test passed, otherwise it failed and the corresponding assertion can be retrieved to help the debugging.

III. Expressing Java Programs in Set-Theory

In order to be able to apply a constraint-based verification and validation techniques on Java programs, it is mandatory to provide a representation of the target language in constraints. We start this section by the description of our underlying constraint solver, namely CLPS-BZ. We then explain how to use a set-theoretical representation of object-oriented programs, inspired by the one used in a previous work [7]. We then introduce a graph representation of the program code, that can be explored by our symbolic animation engine in order to perform symbolic execution of the code.

A. Using Constraint Solving to evaluate Java/JML Programs

The evaluation of Java/JML programs is based on the constraint logic programming technology called CLPS-BZ [6]. CLPS-BZ is a constraint solver implemented in SICStus Prolog [8], and originally designed to evaluate and to generate functional test cases from B and Z formal models [9]. Because of the set oriented features of both B and Z notations, CLPS-BZ is mainly a set constraints solver.

Its constraints domain is the hereditarily finite sets with nested pairs that makes it possible to manage relations and functions, but CLPS-BZ also supports constraints over integers, atoms, pairs, sets, relations and functions, including equality, inequality, membership, subset, range, domain constraints, etc.

The constraints solving techniques are based, on the one hand, on set interval propagation [10], and, on the other hand, on specific propagation rules in order to compute variable instantiations. For basic arithmetic operations and constraints solving on set cardinality, there is a cooperation between CLPS-BZ constraints propagation and the integer finite domains solver CLP(FD) of SICStus Prolog. CLPS-BZ involves partial consistency techniques [11] during constraints propagation; the completeness is ensured by classical labelling based on AC-3 algorithm [12].

More precisely, CLPS-BZ is composed of four subsystems: an executer, a reducer, a constraint solver and a constraint store. The executer manages the symbolic execution of the Java/JML programs: it translates this source code into a labelled transitions system representing its control flow graph, and passes this system to the reducer. The reducer then achieves the effect of the execution by translating each transition label expression into lower-level constraints that can be recursively acquired on-the-fly by the constraint solver. Finally, the constraint store module allows recording the (possibly symbolic) current state during all the execution. After all constraints have been added, if a unique solution is not obtained, the executer applies a labelling procedure to calculate a solution to the constraint system, and providing one instantiation to cover the original path of the Java/JML program given as input.

We now describe the rewriting rules applied to translate Java/JML expressions into CLPS-BZ constraints, and define the approach, based on CLPS-BZ, to generate test data for Java/JML methods using a control flow graph representation.

B. Expressing Data Model with Sets and Functions

a) Instances: We consider the set of existing objects to be a set of atoms, that are named constants. Each value designates an address of the object in the memory heap. Class instances are sets of atoms, subsets of this superset. Class instances are denoted by $\mathcal{I}_C$ where C is the class name. Inheritance relationships make it possible to partition the sets of objects, such that: if class B and class C both directly extend class A, then

$$\mathcal{I}_B \subseteq \mathcal{I}_A \ \wedge\ \mathcal{I}_C \subseteq \mathcal{I}_A \ \wedge\ \mathcal{I}_B \cap \mathcal{I}_C = \emptyset$$

Due to finite domains constraints solving, the set of existing objects has to be included in a superset of all possible instances, (similar to a set of possible object addresses), that are (or not) assigned to objects, and denoted $\mathcal{I}$.

b) Object attributes: Instance attributes are represented as total functions mapping existing instances to their values. Static attributes (whose values are shared by all instances) are directly valued. Values can be either a range of integers (for primitive integer-like types: byte, short, int[1], characters, booleans), or a class. In this case, the domain is defined by the set of existing instances of the corresponding class augmented by a special value *null*, designating a null reference. Each aforementioned attributes value is named by a unique identifier, concatenation of its originating class and attribute.

[1] with a restriction on this data type since SICStus Prolog does not represent integers on 32 bits, but on 24 bits

Example 1 – Attributes naming and typing. *Considering the example given in Fig. 1, attributes* x, y *and* z *will be named and typed as follows:*

$$A.x \in \mathcal{I}_A \rightarrow -32768..32767, \quad A.y \in -128..127$$
$$and \;\; A.z \in \mathcal{I}_A \rightarrow \mathcal{I}_A \cup \{null\}$$

where $\rightarrow$ *designates a total function.*

c) Arrays: In Java, arrays are represented as objects of a specific type. We thus represent existing arrays similarly to existing objects using a set of dedicated instances, depending on the data type contained in the array. For example, $\mathcal{I}_{ashort}$ represents the set of arrays containing short integers. Array contents are described in dedicated variables, depending on the possible content type. Arrays are typed as a set of triplets (a, i, v) where v is the content of array a at index i. For example, the set of arrays containing short integers is typed as: $arrayContent_{ashort} \in \mathcal{I}_{ashort} \times 0..MAXINT \times -32768..32767$. Notice that, when executing Java code, we assume that the sizes of the considered arrays are always known.

d) System States: Since we will consider the symbolic execution of the code, we need to be able to represent system states and their evolutions. Systems states are stored by the CLPS-BZ constraint store module in a map associating to the attribute name its value. This map is updated when an attribute is assigned, and referred when a value is needed. For example, knowing instance a of class A, the value of $a.x$ is given by the image of a by the function representing the attribute x, i.e. $A.x(a)$.

C. Expressing Method Code with Graphs

Inspired of our previous work on the symbolic animation of B machines, we have decided to represent the method code as labelled transitions systems encoding the control flow graph of the operation.

Symbolic execution of a method consists in simulating the execution of the code on the constraint representation of the program. To achieve that, we follow the execution paths of the code, guided by the control-flow graph. For each label on the path, we perform the following actions:

- if the label designates a condition, it is evaluated w.r.t. the current values of the state variables, namely, the references to the data are replaced by the (symbolic) values provided by the system states and a constraint is added to the constraint store. When this latter becomes inconsistent, the considered path prefix is declared infeasible.
- if the label designates a statement, it is executed and the system state is updated. In a large majority of cases, the statements are in fact assignments, for which the assigned expression is first evaluated, and the corresponding data in the system state is then updated with the (possibly symbolic) value resulting of this evaluation.

```
short gcd(short x, short y) {

   int r;

   if (x < y) {
      r = x;
      x = y;
      y = r;
   }

   while (x % y != 0) {
      r = x % y;
      x = y;
      y = r;
   }

   return y;
}
```

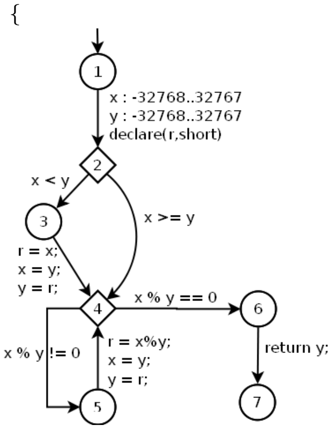

Fig. 2. The GCD example

Example 2 – Symbolic execution of Java code. *Suppose the Java code shown in Fig. 2 implementing Euclide's algorithm for computing the greatest common divisor of two positive integers arguments* x *and* y. *We provide its associated control flow graph representation.*

Considering path [1, 2, 4, 5, 4, 6, 7]. We start with an initial state: $x = X_1, y = Y_1$, *with associated constraints* $X_1 \in -32768..32767, Y_1 \in -32768..32767$. *We then perform the interpretation of the labels that change the system state:*

- *after 1→2:* $x = X_1, y = Y_1, r = undef$
- *after 2→4:* $x = X_1, y = Y_1, r = undef$ *and constraint* $X_1 \geq Y_1$ *is added to the constraint store*
- *after 4→5:* $x = X_1, y = Y_1, r = undef$ *and constraint* $X_1 \; mod \; Y_1 \neq 0$ *is added to the constraint store*
- *after 5→4:* $x = Y_1, y = R_1, r = R_1$ *and constraint* $R_1 = X_1 \; mod \; Y_1$ *is added to the constraint store*
- *after 4→6:* $x = Y_1, y = R_1, r = R_1$ *and constraint* $Y_1 \; mod \; R_1 = 0$ *is added to the constraint store*
- *after 6→7: no more constraint is added.*

Finally, an instantiation of the constraint store provides initial values for the parameters; here, x$= -16384$ *and* y$= -32768$.

Notice that the first part of the path contains conditions that represent explicit typing informations on the parameters, and on the instance on which the method is invoked.

Considered subset of Java/JML. As explained, our representation makes it possible to express the classes, their instances, and arrays. At the moment, integers are restricted to those supported by SICStus Prolog (i.e. 24-bits integers), and reals (float and double in Java) are not supported. In addition, arrays are (preferably) of one dimension.

IV. Grey-Box Test Generation

Our grey-box testing approach considers both the Java code for extracting structural test cases, and the JML specification, in order to provide the preconditions and the class invariant, that represent the invocation context of the methods.

We propose to apply classical structural coverage criteria on the graph in order to extract paths. These paths are provided as an input to our process. If the path is infeasible, it will be discarded. Otherwise, the process produces test data for input parameters and user-defined values.

The code coverage criteria that we are able to apply are the classical ones from the literature (as listed by Myers [13]): all-statements, all-decisions, all-k-paths. In addition, condition coverage criteria can be applied to cover the disjunctive predicates in the decisions (e.g. MC/DC, MCC, etc.) [14].

Example 3 – Grey-box test generation. *Consider again the* `gcd` *example introduced previously. We add the following contract.*

```
/*@ requires x >= 0;
  @ assignable \everything;
  @ ensures \old(x) % \result == 0 &&
  @         \old(y) % \result == 0;
  @*/
  short gcd(short x, short y) { ... }
```

Using the test data produced initially (`x`$= -16384$*,* `y`$= -32768$*), and running the tests on an JML-instrumented program raises a* PreconditionFailedException *indicating an invocation of the method without satisfying the precondition contract. In JML, a direct invocation of the method out of its precondition is not considered as a failed test (but it is when the method is called out of the precondition from another method, raising in this case an* InternalPreconditionFailedException*). If we take the precondition into account, we produce the following test data for path [1,2,4,5,4,6,7]:* `x`$= 3$*,* `y`$= 2$*.*

The instantiation of the constraint system not only provides the values of the parameters but also an invocation context that matches the contract associated to the considered method under test. From the abstract system state representation provided with instantiated sets and functions, one can easily build JMLUnit test cases that will be run on the concrete system under test. Indeed, the Java reflection API makes it possible to access (and directly assign) object attributes, as shown in the example hereafter.

Example 4 – Test case concretization. *Suppose one wants to invoke method* `foo(A)` *whose contract is shown in Fig. 1. The set of constraints associated to the object hierarchy and the contract (precondition + class invariant) is as follows.*

$\mathcal{I} = \{a_0, a_1, a_2, a_3, ...\}$, *// bounded set of existing objects*

$\mathcal{I}_A \subseteq \mathcal{I}$, *// structural constraint for class hierarchy*

$A.x \in \mathcal{I}_A \rightarrow -32768..32767$, *// typing of attribute* `x`

$A.y \in -128..127$, *// typing of static attribute* `y`

$A.z \in \mathcal{I}_A \rightarrow \mathcal{I}_A \cup \{null\}$, *// typing of attribute* `z`

$\forall p \in \mathcal{I}_A \Rightarrow A.x(p) > 0$ *// invariant of class* `A`

$this \in \mathcal{I}_A, this \neq null$, *// existence of* `this`

$a \in \mathcal{I}_A \cup \{null\}$, *// typing of parameter* `a`

$a \neq null$, *// precondition of* `foo`

$A.y(this) > A.x(this)$, *// precondition of* `foo`

$a \neq this$ *// precondition of* `foo`

Notice that this example uses native operators of the CLPS-BZ solver (∈, ∀, ⊆, etc.). This set of constraints can be instantiated by the following (minimal) solution:

$$\mathcal{I} = \mathcal{I}_A = \{a_0, a_1\}, A.x = \{a_0 \mapsto 1, a_1 \mapsto 1\},$$
$$A.y = \{a_0 \mapsto 2, a_1 \mapsto -128\},$$
$$A.z = \{a_0 \mapsto null, a_1 \mapsto null\}, this = a_0, a = a_1$$

This state is then used to produce the following J(ML)Unit test case:

```
public void testFoo() {
   // construction of existing objects:
   A a0 = new A();
   A a1 = new A();
   boolean b;
   // use of reflection to set protected field a0.x:
   b = A.class.getDeclaredField("x").isAccessible();
   A.class.getDeclaredField("x").setAccessible(true);
   A.class.getDeclaredField("x").set(a0, (short)1);
   A.class.getDeclaredField("x").setAccessible(b);
   // same treatment for the other attributes
   ...
   // invocation of the method under test:
   a0.foo(a1);
}
```

V. Detection of Errors and Non-Conformances

We present in this section the verifications we aim at performing using our approach. These verifications are classified into two categories: runtime errors that only focus on the Java code, and non-conformances w.r.t. the JML specification.

A. Runtime Errors

The runtime errors represent classical Java errors that may happen at runtime. Usually, such errors raise an exception that is a subclass of *RuntimeException*. We will focus on the most frequent ones. The presence of these runtime errors can be checked by the satisfiability of an additional constraint involving that represents a condition that has to be fulfilled for the exception to happen.

We give here the list of considered runtime errors, for which we provide the context of the error detection, and the considered additional constraint that represents the condition under which the exception will be raised.

Error	Context	Constraint
Null pointer deref.	$x.y$ is evaluated	$x == null$
Index out of bounds	$t[i]$ is accessed	$i > t.length$
Division by zero	x/y or $x\%y$	$y == 0$

When the additional constraint is acquired, a valuation is immediately asked to the solver to check the overall consistency of the constraint store. If the valuation succeeds then it provides a counter-example that shows the potential presence of a runtime error.

Example 5 – Runtime error detection. *Consider the example of the* `gcd` *method introduced before, when performing the exploration of the control flow graph. When encountering the condition labeling transition 4→5, the additional constraint* $y == 0$ *is stored and a labeling is performed. In this case, the constraint system produces an example of test data,* `x`$= 0$ *and* `y`$= 0$*, that leads to an error of division by zero.*

Notice that, as a side effect, our methodology makes it possible to detect infeasible paths. However, in a large majority of cases, the paths are extracted blindly from the graph structure and are, by definition, not necessarily feasible. Nevertheless, detecting loop-free prefixes that are not feasible indicates dead code.

Example 6 – Dead code detection. *Suppose we enrich the precondition of the* `gcd` *example to add the following information:*

```
/*@ requires y > 0 && x > y;
  @ ...
  @*/
  short gcd(short x, short y) { ... }
```

In this case, we are able to detect that all paths beginning with [1,2,3] are unfeasible since the condition on the label of transition 2→3 is contradictory with the JML precondition.

B. Non-Conformances w.r.t. the JML Specification

The second kind of verifications we consider involves the verification of the contracts that makes it possible to detect non-conformances between the program and its specification. Namely, we will consider three kinds of non-conformances that can be checked at runtime. We briefly detail these non-conformances and informally indicate how these verifications can be performed.

- *Provider* non-conformance: when the method terminates, it does not establish its contractual postcondition. Such a non-conformance is detected by negating the postcondition at the end of the method's execution, and verifying the satisfiability of this negation.
- *Client* non-conformance: when the method invokes another internal method, the precondition of this latter is not fulfilled. Such a non-conformance is detected by negating the precondition of the invoked method, and verifying the satisfiability of this negation.
- *Integrity* non-conformance: after the execution of the method, the class invariant is not satisfied. Such a non-conformance is detected by negating the invariant at the end of the method's execution.

Since these verifications can not be done for all the possible paths of the methods code, we propose to apply them on demand, on a relevant subset of the paths that is computed using the classical code coverage criteria (all-statements, all-decisions, etc.).

Example 7 – Provider Non-Conformance. *Consider again the example of the GCD algorithm presented in Fig 2. Suppose one wants to check that path* $[1, 2, 4, 5, 4, 6, 7]$ *does not present a provider non-conformance. Considering the state resulting of the execution of this path:* $\mathrm{x} = Y_1$, $\mathrm{y} = R_1$ *with associated constraints*

$$X_1 \in -32768..32767, Y_1 \in -32768..32767,$$
$$Y_1 > 0, X_1 > Y_1, X_1 \ mod \ Y_1 \neq 0,$$
$$R_1 = X_1 \ mod \ Y_1, Y_1 \ mod \ R_1 = 0$$

The negation of the postcondition provides the additional constraint:

$$X_1 \ mod \ R_1 \neq 0 \ \ or \ \ Y_1 \ mod \ R_1 \neq 0$$

Each of these literals is added independently to the previous store and the satisfiability of the resulting constraint system is checked, to look for counter-example. In both cases, the solver concludes that each of these new constraints breaks the consistency of the constraint store. It is thus possible to conclude that the considered path does contain a non-conformance.

VI. Related Works

The present work is an extension of an original work on test generation from JML specifications [7], for which the same data model representation was used. The main difference is the use of the Java code, that was not considered previously.

Many works deal with the verification and validation of Java programs possibly using the JML specifications. JMLUnit [15] and Jartege [16] use random test data generators to respectively build unit test cases and test sequences. In both cases, the JML specification is used to filter irrelevant test data that may be produced, and to assign the test verdict by runtime assertion checking [5].

(j)Cute [17], DART [18], EXE [19], Pathcrawler [20] use symbolic execution for the automated structural test generation using branch coverage strategies. Some of these tool performs both symbolic and concrete execution of the code (also known as *concolic*) so as to generate test data and check the coverage of the corresponding control flow graph. Recently Pathcrawler and Frama-C[2] have been coupled by the SANTE tool [21] to combine test generation and verification of C programs.

In the context of verification of JML/Java programs, ESC/Java [22] uses theorem provers to check verification conditions extracted from Java/JML program. The KeY project [23] aims at similar purposes. Bogor [24] uses a model checking approach to explore Java system states and check the JML assertions.

Our approach is inspired by the InKA approach [25], a goal oriented structural test generator using constraint solving. More recently, Euclide [26] acts in a similar way, taking into account C programs annotated with ACSL contracts. This work represents a continuation of the work done in the DANOCOPS project[3]. A similar approach has been proposed by Collavizza et al. [27], using a bounded model-checking approach to generate counter-examples of program properties using the Comet platform.

VII. Conclusion and Future Works

We have presented in this paper the use of a set-theoretical representation to express Java programs annotated with JML contracts. This representation makes it possible to perform symbolic execution of the Java code of the methods, under the

[2] http://frama-c.cea.fr

[3] http://lifc.univ-fcomte.fr/DANOCOPS

assumptions of the JML preconditions, which can be used for both generating structural test data, or performing code-based verification of assertions, by looking for non-conformances between the program and its specification.

Our future works will consider the improvement of our constraint solving techniques in two directions. First, we have noticed that, in the absence of errors, the AC-3 consistency algorithm used to store the constraints requires the constraints system to be eventually instantiated. In these cases, it is important to notice that a given number of symmetric solutions are enumerated, slowing down the verification process, without changing the result of the computation. The second lead to investigate is to remove the finite domains assumption concerning the set of existing objects. On one hand, the overall set of objects has to be finite to guarantee the termination of the consistency checks, that requires an enumeration. On the other hand, the selected bound might be insufficient to dispose of enough objects for falsifying a verification condition. We are now investigating a means to address these two issues together, in order to improve the execution time and the correctness of the verification and test generation processes. Another option to solve these problems would be to consider another constraint solver that would also handle reals, such as Choco in Java [28]. Nevertheless, it would be necessary to also handle functions and relations, so as to be able to conserve this convenient representation of object-oriented data model.

Finally, we are currently implementing a fully automated tool-chain of this test generation process (especially to automate the translation of Java/JML programs into labelled transition systems representation). This tool-chain will take Java/JML code as input, and automatically derive executable JMLUnit test cases. This achievement will make it possible to more intensively carry out various experiments and get relevant feedback on the scalability of our approach.

References

[1] B. Meyer, *Object-oriented software construction (2nd ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[2] G. Leavens, A. Baker, and C. Ruby, "Preliminary Design of JML: a Behavioral Interface Specification Language for Java," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 1–38, May 2006.

[3] M. Barnett, K. Leino, and W. Schulte, "The Spec# Programming System: An Overview," in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, ser. Lecture Notes in Computer Science, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds. Springer Berlin / Heidelberg, 2005, vol. 3362, pp. 49–69.

[4] P. Baudin, J.-C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto, *ACSL: ANSI C Specification Language (preliminary design V1.2)*, May 2008. [Online]. Available: http://www.frama-c.cea.fr/acsl.html

[5] 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," *Sci. Comput. Program.*, vol. 55, pp. 185–208, March 2005.

[6] F. Bouquet, B. Legeard, and F. Peureux, "CLPS-B: A Constraint Solver to animate a B Specification," *International Journal on Software Tools for Technology Transfer, STTT*, vol. 6, no. 2, pp. 143–157, Aug. 2004.

[7] "SICStus Prolog," Swedish Institute of Computer Sciences, http://www.sics.se/sicstus.html.

[8] F. Bouquet, F. Dadeau, and B. Legeard, "Automated Boundary Test Generation from JML Specifications," in *FM'06, 14th Int. Conf. on Formal Methods*, ser. LNCS, T. Nipkow and J. Misra, Eds., vol. 4085. Hamilton, Canada: Springer-Verlag, Aug. 2006, pp. 428–443.

[9] B. Legeard, F. Peureux, and M. Utting, "Automated Boundary Testing from Z and B," in *Proceedings of the International Conference on Formal Methods Europe (FME'02)*, ser. LNCS, vol. 2391. Copenhagen, Denmark: Springer Verlag, July 2002, pp. 21–40.

[10] C. Gervet, "Interval Propagation to reason about Sets: Definition and Implementation of a Practical Language," *Constraints*, vol. 1, no. 2, pp. 191–246, 1997.

[11] J. Jaffar and M. Maher, "Constraint Logic Programming : A survey," *Journal of Logic Programming*, vol. 19/20, pp. 503–582, May/July 1994.

[12] E. Tsang, *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[13] G. J. Myers, *Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 1979.

[14] S. Vilkomir and J. Bowen, "Formalization of software testing criteria using the Z notation," in *Proceedings of the 25^{th} International Conference on Computer Software and Applications (COMPSAC'01)*. Chicago, USA: IEEE Computer Society Press, October 2001.

[15] G. Cheon, Y.and Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way," in *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*. London, UK: Springer-Verlag, 2002, pp. 231–255.

[16] C. Oriat, "Jartege: A Tool for Random Generation of Unit Tests for Java Classes," in *2nd International Workshop on Software Quality (SOQUA 2005)*. Erfurt, Germany: LNCS 3712, Springer, September 2005, pp. 242–256.

[17] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification, 18th International Conference, CAV 2006*, ser. LNCS, T. Ball and R. Jones, Eds., vol. 4144. Springer, 2006, pp. 419–423.

[18] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*. New York, NY, USA: ACM, 2005, pp. 213–223.

[19] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*. New York, NY, USA: ACM Press, 2006, pp. 322–335.

[20] N. Williams, B. Marre, P. Mouy, and M. Roger, *PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis*, ser. LNCS. Springer Berlin, March 2005, vol. 3463/2005, pp. 281–292.

[21] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand, "Combining Static Analysis and Test Generation for C Program Debugging," in *TAP'10, 4th Int. Conf. on Tests and Proofs*, ser. LNCS, G. Fraser and A. Gargantini, Eds., vol. 6143, Malaga, Spain, Jul. 2010, pp. 94–100.

[22] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata, "Extended Static Checking for Java," *SIGPLAN Not.*, vol. 37, pp. 234–245, May 2002.

[23] B. Beckert, R. Hähnle, and P. H. Schmitt, Eds., *Verification of Object-Oriented Software: The KeY Approach*, ser. LNCS 4334. Springer-Verlag, 2007.

[24] Robby, E. Rodríguez, M. Dwyer, and J. Hatcliff, "Checking JML Specifications using an Extensible Software Model Checking Framework," *Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 280–299, 2006.

[25] A. Gotlieb, B. Botella, and M. Rueher, "Automatic Test Data Generation using Constraint Solving Techniques," in *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '98. ACM, 1998, pp. 53–62.

[26] A. Gotlieb, "Euclide: A Constraint-Based Testing Framework for Critical C Programs," *International Conference on Software Testing, Verification, and Validation*, vol. 0, pp. 151–160, 2009.

[27] H. Collavizza, N. Le Vinh, M. Rueher, S. Devulder, and S. Gueguen, "A Dynamic Constraint-Based BMC Strategy For Generating Counterexamples," in *SAC 2011, Software Verification and Testing*, 2011, to appear.

[28] CHOCO Team, "choco: an Open Source Java Constraint Programming Library," Ecole des Mines de Nantes, Research report 10-02-INFO, 2010.