Constraint Solving for Generating Glass-Box Test Cases

Christoph Lembeck[1], Rafael Caballero[2⋆], Roger A. Müller[1], and Herbert Kuchen[1]

[1] Department of Information Systems, University of Münster, Germany
christoph.lembeck@wi.uni-muenster.de
romu@wi.uni-muenster.de
kuchen@uni-muenster.de

[2] Dpto. de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain
rafa@sip.ucm.es

Abstract Glass-box testing tries to cover paths through the tested code, based on a given criterion such as def-use chain coverage. When generating glass-box test cases manually, the user is likely to overlook some def-use chains. Moreover it is difficult to find suitable test cases which cause certain def-use chains to be passed. We have developed a tool which automatically generates a system of test cases for a piece of Java byte code, which ensures that all def-use chains are covered. The tool consists of a symbolic Java virtual machine (SJVM) and a system of dedicated constraint solvers. The SJVM uses the constraint solvers in order to determine which branches the symbolic execution needs to consider. A backtracking mechanism is applied in case that several branches remain feasible. Thus we have applied implementation techniques known from functional logic and constraint programming to handle the considered applications problems.

1 Introduction

As quality standards for software systems are permanently increasing, software testing becomes a more and more important part of the software development process. Since software testing is time consuming and costly and tests have to be repeated several times, much effort has been invested in the automation of testing. As a result of these endeavors many regression test suites have been implemented. Even though these regression test suits assist a developer performing already known tests on his code, the generation of new test cases remains up to the user.

In the literature, testing is divided into black-box testing and glass-box testing. While black-box testing derives test cases from the specification of the user requirements, glass-box testing is based solely on the code and is hence adapted exactly to the way a given problem will be solved by the program. Here, we will focus on glass-box testing.

Glass-box testing is particularly suited to unit testing and to testing of algorithmically challenging parts of software systems. Unfortunately generating an exhaustive set of test cases based on a given coverage criterion as the def-use chain coverage is then challenging and error-prone, too.

In order to simplify glass-box testing, we have developed a tool that automates the generation of (mostly complete) sets of test cases satisfying a selectable testing criterion. Each test case consists of a set of constraints that describe the values for the input parameters leading to the pass through the test case in a general way, a concrete instance of values satisfying these constraints, a description of the path that will be taken during the runtime of the test, and the symbolic and the real result that will be expected at the end of the run. The testing criterion discussed in this paper will be the def-use chain coverage, which roughly ensures that all values defined somewhere in the program do not cause problems at places, where they are used. However the tool is also able to support other coverage criteria like statement coverage, branch coverage, or condition coverage and offers an open interface for the integration of further coverage criteria. Our tool is implemented in Java.

In order to identify the paths that may be followed by a real Java virtual machine [Su03] as precisely as possible, we have designed our tool as a symbolic Java virtual machine that processes

⋆ Author partially supported by the Spanish CICYT (project TIC2002-01167 'MELODIAS').

Java byte code. The difference of our SJVM to an original Java Virtual Machine [LY99] is that the value of a variable is not just a numerical value but an expression on some input variables. Prior to the symbolic execution, our tool analyzes the byte code in order to determine all def-use chains. During the symbolic execution, the SJVM generates at each branching instruction (e.g. a conditional jump) a constraint corresponding to the branch taken and propagates it to a constraint solver. If the overall system of constraints has no more solutions, or if all definitions and uses in the considered branch have been processed, the SJVM backtracks to the latest branching instruction and continues with an alternative branch, much like the Warren Abstract Machine known from Prolog [Wa83]. When the end of a path has been reached without any conflicts, it is up to the constraint solver to calculate one particular solution of the collected set of constraints, and to generate a corresponding test case. This can be added to a regression test suite.

As already mentioned, the main difficulty of the approach above is checking, whether the collected constraints remain solvable, or if they are already contradicting each other. For this purpose we have implemented a dedicated constraint solver, which is integrated into our SJVM and is connected to the symbolic execution engine by a special constraint solver manager that administrates the gathering of new constraints and facilitates an incremental growth and solving of the constraints (see Fig. 1). The SJVM has been presented in [ML03] and will not be explained in detail here. We will rather focus on the system of constraint solvers.

This paper is structured as follows. The tasks and the functionality of the constraint solver manager are described in Section 2. Section 3 explains how linear constraints will be handled, while Section 4 shows the treatment of non-linear constraints. Extracts of our experimental results and some known limitations of our approach will be discussed in Section 5. Related work and our conclusions can be found in Section 6 and 7, respectively.

2 The Constraint Solver Manager

The constraint solver manager (CSM) acts as an interface between the symbolic execution engine of the SJVM and the different constraint solvers we have implemented to handle different kinds of constraints. Since the constraints produced by the execution engine arrive incrementally at the CSM, it has to store each of them for further calculations. As the backtracking mechanism of our tool also guarantees that the latest constraints added to the system are the first that will be removed again, the CSM needs a constraint stack to maintain them.

Moreover, the CSM analyzes the constraints and transforms them to some kind of normal form. Additionally, it selects the most appropriate constraint solver for each system of constraints and distributes each constraint to the corresponding constraint solver, in case that the overall system of constraints consisted of several independent subsystems. At present only a small set of constraint solving algorithms are implemented, but the CSM has an easily implementable interface for the integration of further, more powerful algorithms.

One step of the normalization of constraints is that inequalities of the form $exp_1 \neq exp_2$ are transformed to an easier manageable system of inequalities $exp_1 < exp_2 \vee exp_1 > exp_2$. Moreover, fractions are removed by expanding and extending the constraints as shown in Figure2. Here it is important to notice that for each denominator it has to be examined, in which situations it will have the value 0, because then the occurrence of an `ArithmeticExcpetion` has to be considered (in Figure 2 this is represented by constraint (1)). Since multiplying inequalities with negative factors lets the comparison operator switch its direction, the original constraint has to be split into two new systems of constraints (line (2) and (3) of the example; all connected by $\vee$). These three systems

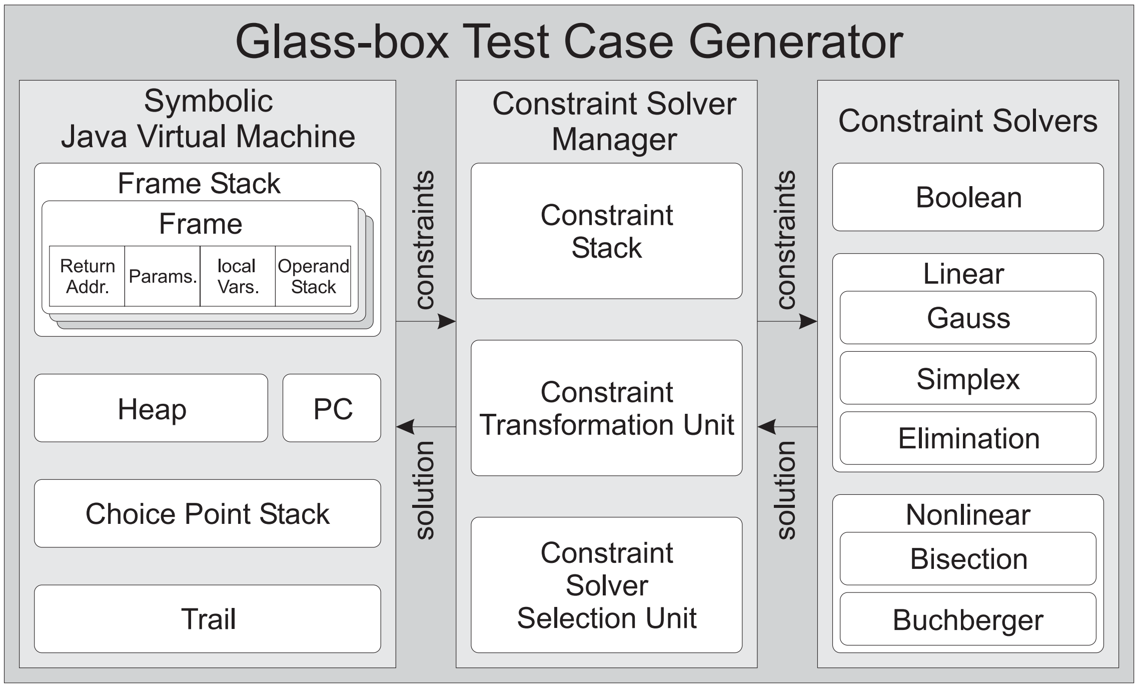

Figure 1. Symbolic Java Virtual Machine

$$\frac{1}{1+x} \geq a \quad \rightsquigarrow \quad \bigvee \begin{cases} 1+x=0 & (1) \\ 1 \geq a+ax \wedge 1+x>0 & (2) \\ 1 \leq a+ax \wedge 1+x<0 & (3) \end{cases}$$

Figure 2. Removal of fractions for $\frac{1}{1+x} \geq a$

will be successively considered using a backtracking mechanism. Thus, one system at a time will be passed on to an appropriate constraint solver.

Removing divisions also implies a consideration of the data types of the involved variables. While the types `float` and `double` introduce few difficulties, integer divisions (for the Java types `byte`, `char`, `short`, `int`, and `long`) have to be replaced in a special way. For example the equation $a/5 = 4$ containing the `double` variable a will be transformed to $a = 20.0$, while the same equation with an integer division has to be transformed to the set of constraints $a = 20 + r \wedge r \geq 0 \wedge r \leq 4$. Possible values for a will then be 20, 21, 22, 23, and 24.

The resulting system of purely polynomial equations and inequalities is then analyzed, in order to check whether they can be broken up into smaller, independently solvable systems. This is an interesting question, because maybe some parts of the system then can be handled by the more efficient linear constraint solvers, while only those parts that are really depending on nonlinear equations are passed on to the more complex nonlinear solvers. The next benefit of this approach concerns the ability of handling incrementally growing systems of constraints. When a new equation or inequality arrives, it has to be checked, which constraints are depending on the variables contained in the new equation or inequality and only for these constraints a new solution has to be computed. All the other constraints that are not depending on the variables of the new equation are left untouched and their solution calculated before can be reused. Because calculating solutions for complex systems of equations and inequalities can be very time consuming, the CSM firstly verifies if the existing solution of the previous calculations satisfies the new equation too. Only if the old solution does not fit any longer, the computation of a new tuple of values will be initiated. Nevertheless the new constraint has to be added to the constraint stack in both cases.

The next and maybe most important job of the CSM is the analysis of the constraints and the choice of the best constraint solver for the given problem. For this purpose the constraints are divided up into several categories, each having a dedicated constraint solver.

Pure boolean constraints consisting only of the boolean constants `true` and `false`, boolean variables and the boolean Java operations `&` (and), `|` (or), `!` (not), `^` (xor), and the comparison operators `==` (equal) and `!=` (not equal) are solved by a simple backtracking mechanism known from (functional) logic programming languages like Prolog [SS94] or Curry [HK95].

The most important decision criterion of the arithmetic constraints is the linearity or nonlinearity of the system. Further criteria are the existence of weak ($\leq$, $\geq$) or strict ($<$, $>$) inequalities or of variables with integer data types (`byte`, `char`, `short`, `int`, or `long`).

Linear equations consisting only of floating-point data types (`float` and `double`) can easily be solved using Gaussian elimination, while inequalities can be solved using a variable elimination solver, parts of the simplex algorithm [BJ90] or a combination of them. The simplex algorithm can be enhanced by a branch-and-bound add-on or total enumeration, which enables the solution of linear (mixed) integer problems [BJ90,GN72]. These solvers are already included in the current version of our test tool and have proven to be reliable.

To explain the functionality of the whole test tool and especially of some of the mentioned solvers, the analysis of the short Java method `gcd` that calculates the greatest common divisor of two positive integer numbers will be used as a running example. The source code of the Java method is shown in Figure 3.

```
public static double gcd (double a, double b){
  while (a > 0){
    if (a < b){
      double h = a;
      a = b;
      b = h;
    }
    a = a % b;
  }
  return b;
}
```

Figure 3. Java source code of the greatest common divisor method

Since our tool operates on the byte code due to several reasons (e.g. efficiency and support of different programming languages (Java 1.4, Java 1.5, Pizza)) the source code of Figure3 has to be compiled to the form presented in Figure 4. Starting from that byte code our test tool offers the developer the choice of different testing criteria. In our example we decided to select the def-use chain coverage. Def-use chains [Be90] are defined as follows:

$$def(S) := \{X| \text{ instruction } S \text{ (re)defines } X\}$$
$$use(S) := \{X| \text{ instruction } S \text{ uses } X\}$$

Then $[X, S, S']$ is a *def-use chain* for variable X, if $X \in def(S) \cap use(S')$ but $X \notin def(S'')$ for all instructions S'' passed on some path from S to S'. We will indicate S and S' by the corresponding

```
 0 goto 15                   11 dload_0   // a
 1 dload_0   // a            12 dload_1   // b
 2 dload_1   // b            13 drem      // a%b
 3 dcmpg                     14 dstore_0 // a=a%b
 4 ifge 11   // a<b?         15 dload_0  // a
 5 dload_0   // a            16 dconst_0 // 0
 6 dstore_2  // h=a;         17 dcmpl
 7 dload_1   // b            18 ifgt 1   // a>0?
 8 dstore_0  // a=b;         19 dload_1  // b
 9 dload_2   // h            20 dreturn  // return b
10 dsotre_1  // b=h;
```

Figure 4. Java byte code of the greatest common divisor method

byte code line numbers. Taking into account that according to the Virtual Machine Specification [LY99] the SJVM puts the input parameters of the `gcd`-method on top of the methods local operand stack as variables 0 and 1 (which is annotated with VM in the def-use chains), the method contains the following set of def-use chains: [a,VM,1], [a,VM,5], [a,VM,11], [a,VM,15], [a,8,11], [a,14,1], [a,14,5], [a,14,11], [a,14,15], [b,VM,2], [b,VM,7], [b,VM,12], [b,VM,19], [b,10,2], [b,10,7], [b,10,12], [b,10,19], and [h,6,9].

To cover all these def-use chains several test cases have to be generated, each causing a different path through the code to be taken and requiring special values as input for the method. The computation of these values will be discussed in the following sections.

3 Solving Linear Constraints

Even though solving linear constraints seems to be one of the easier manageable problems during the generation of test cases and lots of algorithms concerning this subject have been discussed in the literature, many of the known algorithms may be applied just to special subsets of the problems that appear in practice. While Gaussian elimination is working fine on systems of linear equations, the first phase of the Two-Phase Simplex Algorithm (leading to an initial basic feasible solution) [BJ90] are utilized for systems of linear equations and weak inequalities ($\leq$, $\geq$). Although some variations of the simplex algorithm exists that allow e.g. the additional handling of negative variables, the treatment of strict inequalities ($<$, $>$) is not sufficiently supported. For this reason we added an additional, more flexible algorithm to our test tool based on the Fourier-Motzkin elimination procedure [DE73,Ap03]. This elimination procedure iteratively removes a selected variable by isolating it in each inequality, combining these new inequalities and replacing the whole system of inequalities by a new system that does not contain the selected variable. It has been shown that the new system has a solution if and only if the original system has a solution, too (the two systems are also called equisatisfiable). This approach will be repeated until a system of inequalities remains that contains a set of restrictions to only one variable, for which a solution can easily be found. Substituting the solution for the last variable into the previously build systems leads then iteratively to a complete solution of the whole original problem.

More detailed the approach works as follows. Firstly all equations have to be removed from the system of equations and inequalities. This can be done by isolating an arbitrary variable of the first equation that is contained in the system of constraints. This variable can now be substituted in all equations and inequations by the right hand side of the transformed equation without varying the solution space of the remaining variables. This step will be repeated until the constraints are free from any equations. If the original equations contain both integer and noninteger variables it is recommendable to remove the noninteger variables first.

Then the m inequalities with n variables have to be transformed to a system of inequalities of the form shown in (1). The operator $\odot$ stands for one of the operators $\leq$ and $<$ in the following:

$$\begin{array}{ccc} a_{11}x_1 + \cdots + & a_{1n}x_n \odot b_1 \\ \vdots & \vdots \\ a_{m1}x_1 + \cdots + & a_{mn}x_n \odot b_m \end{array} \tag{1}$$

Choosing x_1 as the first variable to eliminate lets us split the set $I = \{1, ..., m\}$ of indices for the inequalities into $I = I_- \cup I_0 \cup I_+$:

$$\begin{array}{ll} I_- & = \{i \in I | a_{i1} < 0\} \\ I_0 & = \{j \in I | a_{j1} = 0\} \\ I_+ & = \{k \in I | a_{k1} > 0\} \end{array} \tag{2}$$

The classification of the inequalities into the sets I_- and I_+ assists us to transform the inequalities of system (1) into the form (3):

$$\begin{array}{ll} a_{j1}^{-1}b_j - a_{j1}^{-1}a_{j2}x_2 - \cdots - a_{j1}^{-1}a_{jn}x_n \odot x_1 & \forall j \in I_- \\ x_1 \odot a_{i1}^{-1}b_i - a_{i1}^{-1}a_{i2}x_2 - \cdots - a_{i1}^{-1}a_{in}x_n & \forall i \in I_+ \end{array} \tag{3}$$

Combining each of the inequalities with the indices in I_- with each of the inequalities with indices in I_+ by the isolated variable x_1 using the transitivity of the $\leq$ and $<$ relations leads to the new system (4):

$$\begin{array}{ll} \sum_{j=2}^{n}(a_{k1}^{-1}a_{kj} - a_{l1}^{-1}a_{lj})x_j \odot a_{k1}^{-1}b_k - a_{l1}^{-1}b_l & \forall k \in I_+, l \in I_- \\ \sum_{j=2}^{n} a_{ij}x_j \odot b_i & \forall i \in I_0 \end{array} \tag{4}$$

It is obvious that the new system consists only of $n-1$ variables and $|I_0| + |I_-| \cdot |I_+|$ inequalities. This step has to be repeated until only one variable is left or some constraint becomes obviously contradictory. Here it is also recommendable to begin eliminating the noninteger variables so that after some eliminations a system of inequalities remains that does only contain integer variables[1]. Together with the integer equations noted before, the generated weak inequalities over integer variables may now be solved by the branch and bound extended simplex algorithm or total enumeration.

Although the number of inequalities grows in the worst case exponentially in the number of variables, in our application this is no problem, since it is usually no problem to find a variable for which one of the sets I_+ or I_- and thus the total number of inequalities in the next step become relatively small. Note that if I_- or I_+ are empty, it is trivial to find a solution. Additionally the number of input parameters of typical Java methods tends to be small.

Taking our example of the `gcd`-method described earlier, the Fourier-Motzkin elimination will be used by the CSM for situations, where the SJVM wants to know, what branches of the `if`-instruction in the byte code on line 4 are still reachable, after the ”true branch” of the conditional jump instruction on line 18 has been taken positive. Here the simple inequalities $a > 0 \wedge a < b$ will have to be handled. Since this is a trivial problem for the elimination algorithm, we make the example a little bit more interesting by adding the conditions $a \leq 100$ and $b \leq 100$.

[1] This mixture of possibly strict and weak inequalities can then easily be transformed into a system of weak inequalities by subtracting the greatest common divisor of all coefficients and the right hand side of each inequality from their right hand sides.

The set of inequalities gathered so far by the CSM will then be:

$$\begin{array}{lcrl} a > 0 & \leadsto & -a < 0 & (1) \\ a < b & \leadsto & a - b < 0 & (2) \\ a \leq 100 & \leadsto & a \leq 100 & (3) \\ b \leq 100 & \leadsto & b \leq 100 & (4) \end{array}$$

Selecting a as the next variable to eliminate lets us build the three sets of indices $L = \{1\}$, $I_0 = \{4\}$, and $I_+ = \{2, 3\}$. Preparing the inequalities (1), (2), and (3) for the later elimination of a leads to:

$$\begin{array}{ll} 0 < a & \\ & a < b \\ & a \leq 100 \end{array}$$

Combining them pairwise and adding the inequalities of I_0 produces the $1+1\cdot 2 = 3$ inequalities with b as the only contained variable:

$$0 < b \quad 0 < 100 \quad b \leq 100$$

The constraint solver is now able to detect that there are no conflicting constraints and that the value of the variable b has to be selected from the interval $]0; 100]$. Selecting $b = 50$ randomly and inserting this value into the original system results in:

$$\begin{array}{rr} -a < 0 & a < 50 \\ a \leq 100 & 50 \leq 100 \end{array}$$

Thus the interval for the variable a is $]0; 50[$. The information, the CSM can pass on to the symbolic execution engine of the SJVM is that the considered path of the symbolic execution can still be reached and possible values for a test case leading to that path can e.g. be $a = 40$ and $b = 50$.

4 The Nonlinear Constraint Solver

For nonlinear constraints we offer a symbolic approach based on the Buchberger algorithm [BW93] [ML03] and a numerical approach. The Buchberger algorithm has the disadvantage that it is doubly exponential and that there is no systematic approach to find one concrete solution which could be used to generate a test case in case that there are infinitely many solutions. In the present paper we will focus on our numeric approach, a bisection algorithm, which successively divides the search space looking for solutions of the considered system of polynomial equations and inequalities.

Returning to our running example, the `gcd`-method, let us consider the constraints gathered when for the first three if-instructions of the byte code the true-branch was selected and for the following two if-instructions the conditions were assumed to be false. This leads to a test case covering the def-use chains [a,VM,1], [a,VM,11], [a,VM,15], [a,8,11], [a,14,1], [a,14,5], [a,14,15], [b,VM,2], [b,VM,7], [b,VM,12], [b,10,12], [b,10,19], and [t,6,9]. The result of the symbolic execution in that case is ($a \bmod b$) and the gathered constraints are:

$$\begin{array}{lll} a > 0 & a \bmod b > 0 & b \bmod (a \bmod b) \leq 0 \\ a \geq b & a \bmod b < b & \end{array}$$

Because many algorithms can not handle constraints containing modulo operations directly, the disturbing $\bmod$-operations will be eliminated first by inserting additional variables:

$$\begin{array}{rll} a > 0 & x > 0 & y \leq 0 \\ a \geq b & a = mb + x & b = nx + y \\ m \geq 0 & x \geq 0 & x \leq b - 1 \\ n \geq 0 & y \geq 0 & y \leq x - 1 \end{array}$$

Here it is easy to see, that y has to be zero and that the whole problem is obviously nonlinear. Substituting b, the equation for the variable a will be transformed to $a = m \cdot n \cdot x + m \cdot y + x$, which is a polynomial equation of third degree and thus finding solutions for it is a nontrivial problem.

The constraints in our setting are conjunctions of *atomic polynomial constraints*. Disjunctions are not considered because the solver manager handles them via backtracking. Each atomic polynomial constraint has the form $p \lozenge 0$, where p is a polynomial and $\lozenge \in \{=, <\}$. Notice that for the further reasoning it is not necessary to include weak inequalities, since they can be decomposed in the disjunction of an equality and a strict inequality, and that constraints of the form $p > 0$ can be replaced by $-p < 0$.

After transforming the example we get:

$$\begin{array}{rrr} -a < 0 & -n < 0 & b - nx = 0 \\ b - a < 0 & a - mb - x = 0 & x - b + 1 < 0 \\ -m < 0 & -x < 0 & 1 - x < 0 \end{array}$$

Each variable in the constraint has some predefined domain, represented by the different types used in Java for representing numbers (`byte`, `char`, `short`, `int`, `long`, `float`, `double`). Given a variable x we represent the set of values of its domain by $dom(x)$. In our example we would expect a, b, and x to be `double` values and, m and n `int` values.

Let $\overline{x}_n$ be the number of different variables in a constraint φ, $\varepsilon \in \mathbb{R}_+$, and $\overline{c}_n \in \mathbb{R}^n$. Then we say:

- $\overline{c}_n$ *ε-satisfies* an atomic constraint of the form $p < 0$ if $p(\overline{c}_n) < -\varepsilon$ and each $c_i \in dom(x_i)$ for $i = 1 \ldots n$.
- $\overline{c}_n$ *ε-satisfies* an atomic constraint of the form $p = 0$ if $-\varepsilon \leq p(\overline{c}_n) \leq \varepsilon$ and each $c_i \in dom(x_i)$ for $i = 1 \ldots n$.
- $\overline{c}_n$ *ε-satisfies* a constraint φ if it ε-satisfies all the atomic constraints in φ.
- $\overline{c}_n$ *satisfies* a constraint φ if it 0-satisfies all the atomic constraints in φ.

The constraint solver aims at finding some values $\overline{c}_n$ ε-satisfying the constraint φ for some small enough ε, $0 < \varepsilon < 1$. However, this problem is quite involved.

Fortunately, we are not interested in finding *all* the solutions of the system. For our purposes it is enough to find one solution, if such a solution exists, to obtain a test case covering the considered def-use chains. With this aim our solver uses a numerical technique based on a generalized *bisection algorithm* [KS99], which has proved to be reliable for isolating real solutions of multivariate polynomial systems. The algorithm, adapted to our requirements, considers some initial n-dimensional rectangle D, where the number n of variables in the given constraint φ, and some error bound ε, and checks if there is any point $\overline{q}_n \in D$ which ε-satisfies φ. It can be summarized as follows:

```
Solve(D,$\varphi$,$\varepsilon$):

I = {D}
While I is not empty and no solution has been found
    Select a rectangle S of I.
    I = I - $\{S\}$
    if there exists $\overline{q}_n \in S$ such that $q_i \in dom(x_i)$ for $i = 1 \ldots n$ then
        if $\overline{q}_n$ $\varepsilon$-satisfies $\varphi$ then  a solution has been found ($\overline{q}_n$)
        else if mightBeSatisfied($S, \varphi, \varepsilon$) then
                split $S$ in smaller rectangles $\{S_1, \ldots, S_k\}$
                I = I $\cup \{S_1, \ldots, S_k\}$
```

As it can be seen in this description, the algorithm first checks whether there is any point $\overline{q}_n$ in the rectangle such that $q_i \in dom(x_i)$ for all $i = 1 \ldots n$. If such a point does not exist then the rectangle can be discarded. Otherwise one of these points is singled out to check if it ε-satisfies the constraint. In this case we have obviously found a solution. Otherwise the algorithm uses a test, represented by the function *mightBeSatisfied* to determine if the rectangle could still contain some point ε-satisfying φ. If this happens the rectangle is split into k smaller rectangles (in our current implementation $k = 3$), which will be considered in turn. The process continues until some solution is found or the set of rectangles is empty, meaning that no solution has been found. Both the termination property and the correctness of the algorithm depend on the test *mightBeSatisfied*, which:

1. Returns *no* if it can be ensured that actually there is no point in the rectangle that can ε-satisfy the constraint. In this way the correctness is ensured, since no solution can be skipped.
2. Returns *no* too, if the size of the rectangle S is smaller than our predefined ε in each dimension and if any $\overline{q}_n \in S$ with $q_i \in dom(x_i)$ for $i = 1 \ldots n$ does not ε-satisfy φ. This ensures the termination of the algorithm if no valid solution will be found within the given precision.

To meet the first requirement we must devise some test *cannotBeSatisfied(S,φ,ε)* that can ensure, in certain cases, that the constraint cannot be ε-satisfied in some rectangle S. Then obviously *mightBe-Satisfied* can be defined as:

```
mightBeSatisfied(S,$\varphi$,$\varepsilon$):

if cannotBeSatisfied(S,$\varphi$,$\varepsilon$) $\Rightarrow$ then return no
else return yes
```

A constraint φ cannot be ε-satisfied if some of its atomic constraints $p \Diamond 0$ cannot be ε-satisfied. And this can be checked since p is a polynomial and hence a continuous function:

```
cannotBeSatisfied(S,$p\Diamond 0$,$\varepsilon$):

Let $m, M \in \mathbb{R}^n$ be two numbers such that $m < p(\overline{q}_n) < M$ for all $\overline{q}_n \in S$
if $\Diamond$ is '=' and $m \cdot M > 0$ and $|m|, |M| > \varepsilon$ then return yes
else ($\Diamond$ is $' <'$)
    if $m > -\varepsilon$ then return yes
        else return no
```

i.e. a constraint of the form $p = 0$ cannot be ε-satisfied in S when the upper and the lower bounds of the polynomial have the same sign (condition $m \cdot M > 0$) and both are greater that ε in absolute value. The same occurs if the constraint is of the form $p < 0$ and the lower bound is greater than $-\varepsilon$. If neither of these cases occurs then the constraint still might be ε-satisfied in S. To determine the upper and the lower bounds of a polynomial p in the rectangle S we use the *Bernstein expansion* of the polynomial since this representation has the following well-known property [GG99]:

Let p be a polynomial with n variables and p_B its Bernstein expansion. Let m, M be the minimum and the maximum respectively of the coefficients of p_B. Then $m < p(\overline{q}_n) < M$ for all $\overline{q}_m \in [0,1]^n$, where $[0,1]^n$ represents the n-dimensional cube $\underbrace{[0,1] \times \cdots \times [0,1]}_{n}$

Then the steps to find the values m, M used in the definition of the function $cannotBeSatisfied(S, p\Diamond 0, \varepsilon)$ are the following:

1. Define a n-dimensional homotecy $h : [0,1]^n \to S$ converting $[0,1]^n$ into S.
2. Compute the Bernstein expansion p_B of the composition $p \cdot h$, which is also a polynomial.
3. Obtain the maximum coefficient of p_B as value M and the minimum coefficient as value m.

Checking that the lower (respectively the upper) bounds of $p \circ h$ in $[0,1]^n$ is the lower (respectively the upper) bounds of p in S is straightforward. Moreover this technique has the following nice property:

Let p be a polynomial with n variables and $\varepsilon \in \mathbb{R}$, $0 < \varepsilon < 1$. Then for small rectangles S, the composition $p \circ h$, with h an homotecy such that $h : [0,1]^n \to S$, verifies that the minimum m and the maximum M coefficients of its Bernstein expansion $(p \circ h)_B$ are such that $M - m < \varepsilon$.

This ensures the termination property of the algorithm, since every rectangle such that $M - m < \varepsilon$ either contains a solution or can be discarded. However our current implementation also includes a time-out to ensure that reaching this limit is not too costly. Of course if the time-out is triggered then the technique cannot ensure if there is any solution to the system. But in practice this occurs only with very complex constraints (i.e. polynomials of very high degree). For instance given the system at the beginning of the section, the solver gets a solution after a few milliseconds.

5 Experimental Results and Restrictions

As we have seen in the previous sections, our constraint solvers are able to handle linear and nonlinear constraints and combinations of them. Interestingly, many of the common algorithms and data structures known from the literature get by with simple boolean and linear constraint solvers and the nonlinear solvers will rarely be used.

Algorithm	Characteristics of constraints			
	linear	nonlinear	floating-point	integer
Ackermann	$\surd$			$\surd$
Binary search	$\surd$			$\surd$
Bubble Sort	$\surd$			$\surd$
Bresenham	$\surd$			$\surd$
Factorial	$\surd$			$\surd$
Gaussian Elimination	$\surd$			$\surd$
Greatest Common Divisor	$\surd$	$\surd$		$\surd$
Histogram	$\surd$			$\surd$
Dijkstra Shortest Path	$\surd$			$\surd$
Logarithm		$\surd$	$\surd$	
Matrix Multiplication	$\surd$			$\surd$
McCarthy	$\surd$			$\surd$
Pattern Matching	$\surd$			$\surd$
Sin		$\surd$	$\surd$	
Sqrt		$\surd$	$\surd$	
StoogeSort	$\surd$			$\surd$

Table 1. Characteristics of constraints for several algorithms

This phenomenon is caused in the fact that although the results of many algorithms can involve rather complex calculations, the decisions, which guide the control flow, are mostly very simple. Taking for example factorial: the results for the factorials of the first natural numbers are $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$, and $5! = 120$. Admittedly the corresponding calculations are surely nonlinear, but a look at the definition $factorial(n) = \prod_{i=1}^{n} i$ shows that a simple loop over the natural numbers from 1 to n is sufficient. The constraints the CSM has to solve during test case generation for the factorial method are $1 \leq n$, $2 \leq n$, $3 \leq n$, etc., because only the value of the control variable i and the value of the input variable n are responsible for the termination of the loop, whereas the value of the variable i is known in each iteration.

A similar behavior can be observed for every algorithm that is based on simple loops or even several nested loops having dependent or independent termination conditions. Algorithms in this category are vector and matrix multiplication, several sorting algorithms such as bubble sort, naive string pattern matching algorithms, and even the recursive Ackermann function.

The constraints become a little bit more complicated, if methods are performing nonlinear computations and their results are used to decide on the termination of the algorithm. Then the nonlinear computations will become part of the constraints. This problem can often be found in numerical algorithms like the computation of different Taylor series to approximate trigonometric functions, logarithms, roots, or the exponential function. Here often the value of a previously calculated estimation is compared to the current result to check whether more approximations are needed.

At a first glance this may quickly lead to a big problem, since in some calculations the exponents of the symbolically calculated intermediate results grow by each iteration of the calculation. But in order to generate complete sets of test cases for simple testing criteria like the commonly used def-use chain coverage we typically require only a small number of passes through loops. Thus the complexity of the constraints we gathered was mostly relatively small and hence manageable in a short amount of time. In particular our tool is able to generate the complete sets of test cases regarding the def-use chain coverage for the algorithms mentioned in Table1 in a few seconds. This table shows what kinds of constraints occur in the example applications we have considered.

Although the prototype of our test case generator has proven to be applicable to many algorithms, there are still a few problems we have to solve. For instance, our tool does not yet offer a constraint solver that can manage constraints containing bit operations corresponding to the byte code instructions `ishl`, `ishr`, `iushr`, `lshl`, `lshr`, and `lushr`, which are produced by Java's shift operators `<<`, `>>`, and `<<<` on expressions of the types `int` and `long`. Since these operations are lacking in the usual algebra, workarounds have to be invented to deal with them. Fortunately, they rarely occur in application software and are typically only found in system software such as device drivers.

As mentioned already, very few iterations of each loop are typically sufficient to cover all def-use-chains. In cases where the number of iterations or recursive calls is very large, e.g. since it has been fixed by a large constant (note that this is bad programming style), the symbolic computation might take too long. Pragmatically, our tool will stop with a corresponding error message in that case. Note that due to the halting problem, it is in general impossible to determine the number of required iterations.

On the other hand our approach is vulnerable to rounding errors that come along with calculations on floating point values. Since our computations are not identical to that of the concrete JVM, the rounding errors in our tool and in the concrete JVM computation may differ. Note that it is equally bad to be less precise that the concrete JVM computation than to be more precise. The different rounding behavior may cause the symbolic and concrete computation to diverge, which would affect the correctness of our approach. Thus after generating the test cases, we will compare the paths taken by the symbolic and concrete computations. If they diverge, we will (try to) adapt the responsible input parameter, say from 3.9999 to 4.0. In case, where this does not work, our tool has to give up with a corresponding error message. After all, our tool tries to be helpful in almost all cases. In the few remaining cases, the user has to find suitable test cases on his own. In particular, this might happen in numerically instable calculations, which should be avoided anyway. In any case, we can guarantee that only correct test cases are generated. To summarize, our tool is correct but not necessarily complete (among others due to the halting problem).

As we already mentioned, we actually have only a prototypically implementation of our test tool and some features are not entirely realized. E.g. the symbolic execution engine of our tool is not yet able to handle all of the features of the Java language. First of all we have to consider the multithreading capability in this context, which may cause a nondeterministic behavior of Java programs and thus makes it hard to generate test cases that satisfy a certain coverage criterion in a real virtual machine. Secondly it is allowed to integrate functions and applications into Java programs that are not written in the Java language by using the Java Native Interface [Li99]. Java methods based on native method calls or methods invoking those native methods are at the moment not analyzable by our test tool.

6 Related Work

Many approaches to software testing have been proposed (see e.g. [Ed99] for an overview), but only a few can directly be related to our approach.

Gupta, Mathur and Soffa [GM00] have proposed an approach that uses a branch selection algorithm that generates input data which exercises a selected branch in a program. However their approach features no virtual machine and is strictly numerical. Gotlieb, Botella and Rueher [GB98] have proposed a limited constraint-based approach for a sub-set of the C-language that can identify paths that cover every instruction in the program. Their approach featured neither an exchangeable testing criterion, nor did the test tool contain several, automatically chosen constraint solvers. Korel [Ko96] has presented an approach to generate test data by performing a data dependence analysis in Turbo Pascal programs and using minimization techniques to discover suitable input values. The constraint-based approach by Offutt and DeMillo [DO91] is based on their unusual mutation analysis test criterion, but comprehends just a naive constraint solver. An early approach to test-data generation was proposed by Ramamoorthy, Ho, and Chen [RH76], who transformed Fortran-code to a basic form and tried to solve the constraints describing the sought paths through the program by forward substitution. Finally, Lapierre et al. [LM99] implemented a tool that tried to solve the path-describing constraints by using mixed-integer linear programming for C programs.

7 Conclusions and Future Work

In this paper we have presented a tool for the automatized generation of glass-box test cases for Java methods using a combination of a symbolic Java virtual machine and a dedicated constraint solver. We pointed out that a simple usage of already known constraint solvers or constraint solving algorithms is not sufficient for the considered application, since the constraints that have to be managed occur not in normalized forms but have to be transformed by a dedicated constraint solver manager before passing them to the various solvers. We also discussed the influence of the different Java primitive types on the way, the constraints will be treated. We have additionally shown how the constraint solver manager handles incrementally arriving constraints.

Our system of constraint solvers contains in particular a solver using the simplex algorithm, a Fourier-Motzkin elimination solver for linear constraints, and a bisection solver for nonlinear ones. We are not aware of any other test case generator, which provides this powerful combination of solvers.

The experiences with our prototype have shown a few possible enhancements and some topics that will have to be improved in future releases, e.g. the handling of threads and the integration of native method calls.

References

Ap03. K.R. Apt. Principle of Constraint Programming. Cambridge University Press, Cambridge, UK, 2003.

Be90. B. Beizer. Software Testing Techniques. Van Nostr. Reinhold, 1990.

BJ90. M.S. Bazaraa, J.J. Jarvis, H.D. Sherali. Linear Programming and Network Flows. John Wiley & Sons, New York, NY, 1990.

BS90. M.S. Bazaraa, H.D. Sherali, C.M. Shetty. Nonlinear Programming, Theory and Algorithms. John Wiley & Sons, New York, NY, 1993.

Bu85. B. Buchberger. Grobner Bases: An Algorithmic Method in Polynomial Ideal Theory. *In Multidimensional Systems Theory*, D. Reidel Publishing Company, Dordrecht, Holland, 1985.

BW93. T. Becker, V. Weispfenning. Gröbner Bases, A Computational Approach to Computer Algebra. Springer, New York, NY, 1993.

DE73. G.D. Danzig and B.C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288-297, 1973.

De03. R. Dechter. Constraint Processing. Morgan Kaufmann Publishers, San Francisco, 2003.

DO91. R.A. DeMillo and A.J. Offutt. Constraint-Based Automatic Test Data Generation. *In IEEE Transactions on Software Engineering, Vol.17, No.9*, September 1991.

Ed99. J. Edvardsson. A survey on Automatic Test Data Generation. *In Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21-28. ECSEL, October 1999.

GB98. A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation using Constraint Solving Techniques. *In ISSTA '98, Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*, March 1998.

GG99. J. Garloff and B. Graf. Solving Strict Polynomial Inequalities by Bernstein Expansion *I*n The Use of Symbolic Methods in Control System Analysis and Design, N. Munro, Ed., The Institution of Electrical Engineers (IEE), London, pp. 339-352 (1999).

GM00. N. Gupta, A.P. Mathur, and M.L.Soffa. Generating Test Data for Branch Coverage. 15th IEEE International Conference on Automated Software Engineering (ASE'00), 2000.

GN72. R.S. Garfinkel, G.L. Nehmhauser. Integer Programming. John Wiley & Sons, New York, NY, 1972.

HK95. M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. *In Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, 1995.

Ko96. B. Korel. Automated Test Data Generation for Programs with Procedures. *In Proceedings of the 1996 International Symposium on Software Testing and Analysis, San Diego, CA, USA. Software Engineering Notes 21(3)*, May 1996.

KS99. T. Kutsia and J. Schicho. Numerical Solving of Constraints of Multivariate Polynomial Strict Inequalities. RISC technical report 99-31. Available at: *ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1999/99-31.ps.gz*.

Li99. S. Liang. The Java Native Interface, Programmer's Guide and Specification. Addison-Wesley, Reading, Massachusetts, 1999.

LG97. Q. Li, Y. Guo, T. Ida, J. Darlington. The minimised geometric Buchberger algorithm: an optimal algebraic algorithm for integer programming. *International Conference on Symbolic and Algebraic Computation.* 331 - 338, 1997.

LM99. S. Lapierre, E. Merlo, G. Savard, G. Antoniol, R. Fiutem, and P. Tonella. Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees. Int. Conf. on Software Maintenance 1999.

LY99. T. Lindholm and F. Yellin. The Java(TM) Virtual Machine Specification, Second Edition. Addison-Wesley, 1999.

ML03. R.A. Müller, C. Lembeck, and H. Kuchen. GlassTT - A Symbolic Java Virtual Machine using Constraint Solving Techniques for Glass-Box Test Case Generation, Technical Report 102, University of Münster, 2003.

RH76. C.V. Ramamoorthy, S.-B.F. Ho, and W.T. Chen. On the Automated Generation of Program Test Data. *In IEEE Transactions on Software Engineering, Vol. SE-2, No.4*, December 1976.

SS94. L. Sterling and E. Shapiro. The Art of Prolog, Second Edition. The MIT Press, 1994.

Su03. Sun Microsystems. Java 2 Platform, 2003. http://java.sun.com/j2se/.

Wa83. D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.