Editorial Manager(tm) for Innovations in Systems and Software Engineering
Manuscript Draft

Manuscript Number:

Title: Randomized Constraint Solvers: A Comparative Study

Article Type: SI: NFM 2009

Keywords: constraint solvers; software testing; concolic execution; random solvers

Corresponding Author: Mitsuo Takaki,

Corresponding Author's Institution: UFPE

First Author: Mitsuo Takaki

Order of Authors: Mitsuo Takaki; Diego Cavalcanti; Rohit Gheyi; Juliano Iyoda; Marcelo d'Amorim; Ricardo Prudêncio

Abstract: The complexity of constraints is a major obstacle for constraint-based software verification. Automatic constraint solvers are fundamentally incomplete: input constraints often build on some undecidable theory or some theory the solver does not support. This paper proposes and evaluates several randomized solvers to address this issue.We compared the effectiveness of a symbolic solver (CVC3), a random solver, two heuristic search solvers, and seven hybrid solvers (i.e. mix of random, symbolic and heuristic solvers). We evaluated the solvers on a benchmark generated with a concolic execution of 9 subjects. The performance of each solver was measured by its precision, which is the fraction of constraints that the solver can find solution out of the total number of constraints that some solver can find solution. As expected, symbolic solving subsumes the other approaches for the 4 subjects that only generate decidable constraints. For the remaining 5 subjects, which contain undecidable constraints, the hybrid solvers achieved the highest precision (fraction of constraints that a solver can find a solution out of the total number of satisfiable constraints). We also observed that the solvers were complementary, which suggests that one should alternate their use in iterations of a concolic execution driver.

Innovations in Systems and Software Engineering: A NASA Journal manuscript No.
(will be inserted by the editor)

Randomized Constraint Solvers: A Comparative Study

Mitsuo Takaki[‡] · **Diego Cavalcanti**[†] · **Rohit Gheyi**[†] · **Juliano Iyoda**[‡] · **Marcelo d'Amorim**[‡] · **Ricardo B. C. Prudêncio**[‡]

Received: date / Accepted: date

Abstract The complexity of constraints is a major obstacle for constraint-based software verification. Automatic constraint solvers are fundamentally incomplete: input constraints often build on some undecidable theory or some theory the solver does not support. This paper proposes and evaluates several randomized solvers to address this issue. We compared the effectiveness of a symbolic solver (CVC3), a random solver, two heuristic search solvers, and seven hybrid solvers (i.e. mix of random, symbolic and heuristic solvers). We evaluated the solvers on a benchmark generated with a concolic execution of 9 subjects. The performance of each solver was measured by its precision, which is the fraction of constraints that the solver can find solution out of the total number of constraints that some solver can find solution. As expected, symbolic solving subsumes the other approaches for the 4 subjects that only generate decidable constraints. For the remaining 5 subjects, which contain undecidable constraints, the hybrid solvers achieved the highest precision (fraction of constraints that a solver can find a solution out of the total number of satisfiable constraints). We also observed that the solvers were complementary, which suggests that one should alternate their use in iterations of a concolic execution driver.

1 Introduction

Software testing is important and expensive [8,31,39]. Several techniques have been proposed to reduce this cost. Automation of test data generation, in particular, helps to improve software productivity by transfering a tedious and error-prone search task to the machine [24]. It enables higher system coverage which often translates to higher reliability [8]. Random testing [13,34] and symbolic testing [27] are two widely used techniques with this goal and with well-known limitations. On the one hand, random testing might explore the same program path repeatedly and also fail to explore important paths (i.e., paths to which only a small portion of the space of input data can lead to an execution). On the other hand, pure symbolic testing is problematic for indexing arrays, dealing with native calls, detecting infinite loops and recursion, and, especially, dealing with undecidable constraints. Combined random-symbolic testing [22] has been recently proposed to circumvent these limitations. One important limitation it attempts to address is the *inability to solve general constraints*. This is the focus of this paper. We study the impact of alternative randomization strategies for solving constraints. In this setting, random-symbolic testing reduces to random-symbolic constraint solving.

One can combine random and symbolic solvers by first delegating to the random solver the parts of a constraint that build on theories a symbolic solver does not support. Then use that partial solution to simplify the original constraint and finally combine the random solution with the one obtained from calling the symbolic solver on the simplified constraint. To simplify illustration of algorithms assume constraint are satisfiable; solvers will timeout on unsatisfiable input constraints which often arise in symbolic testing, in general, and specifically in our experiments. Important to note is that, as for typical decision procedures in SMT solvers [20,41], random and symbolic solvers are *not* independent in this combination, i.e., their input constraints are very often related by data dependency.

[‡] Federal University of Pernambuco, Recife, Brazil. E-mail: mt2,jmi,damorim,rbcp@cin.ufpe.br
[†] Federal University of Campina Grande, Campina Grande, Brazil. E-mail: diegot,rohit@dsc.ufcg.edu.br

They should therefore *collaborate*. One practical consequence of this is that the more constraints the symbolic solver rejects the more complex random solving becomes, and conversely. Therefore, random solving is critical for the effectiveness of the combined solver.

In our work, we performed a comparison of 11 different solvers. As a baseline, two solvers were considered: a plain symbolic (in our case, CVC3 [2]) and a purely random search. As an extension of the random solver, we proposed two heuristic search solvers based on Genetic Algorithms (GA) [23] and on Particle Swarm Optimization (PSO) [26]. GA and PSO are two widespread and general search procedures adapted in our work as constraint-based solvers. We can take advantage of their complementary nature with respect to a symbolic solver in the same way a pure random solver does. Finally, seven hybrid solvers were implemented by combining the symbolic, random and heuristic solvers.

For the purpose of comparison, we consider a constraint as satisfiable only if one of our solvers can find a solution to it and define **precision** as the fraction of constraints that a solver can find solutions out of the total number of *satisfiable constraints*. In our experiments, we observed that the symbolic and the hybrid solvers showed very satisfactory results in the subjects that contain decidable constraints, as expected. Considering the other subjects, the hybrid solvers achieved a more consistent performance compared to individual symbolic and random solvers combined.

This paper makes the following contributions:

- The proposal of hybrid solvers combining random and symbolic constraint solving;
- The implementation of existing and proposed solvers;
- An empirical evaluation of solvers over constraints generated from concolic executions of 9 subjects.

The remaining of this paper is organized as follows. Section 2 describes the solvers implemented and evaluated in our work. Section 3 brings the performed experiments and obtained results, followed by Section 4 which presents some related work. Finally, Section 5 concludes the paper.

2 Techniques: Randomized Solvers

In this section, we explain each of the solvers proposed in this work. Before that, we present the common input-output interface for them.

Input. All solvers take as input (i) a constraint system *pc* (in reference to a *path condition* from a symbolic execution), (ii) a random seed s, and (iii) a range of values $[lo, hi]$. A constraint system takes the form $\bigwedge b_i$, where b_i is a boolean expression constructed, in principle, with any logical system. For example, the expression $x > 0 \wedge x > y+1$ illustrates a valid constraint system. We often use the term **constraint** alone or **clause** in reference to a single boolean expression b_i and **constraint system** or **pc** in reference to the conjunction of all constraints.

Output. A **solution** is a vector of variable assignments that satisfies a constraint system. For instance, $\langle x \mapsto 2, y \mapsto 0\rangle$ is a solution to the constraint $x > y+1$ (using integer variables). A solver returns a solution when it finds one, and the flag *empty* otherwise.

Note on implementation. We wrote all solvers in the Java language and used part of the code from the Java Pathfinder symbolic execution [5] for the integration with CVC3.

2.1 Baseline solvers

We use the solvers **ranSOL** and **symSOL** as representatives of plain random and symbolic solvers respectively. In our experiments we use these solvers as baselines for comparison.

Random Solver. The random solver tries to solve a constraint by randomly generating assignments to all variables in a certain amount of time. Figure 1 shows the pseudo-code for the random constraint solver ranSOL. The main loop generates random solutions (the *input vector* $\overrightarrow{iv}$) and selects those that satisfy *pc* (lines 1-6). The expression vars(*pc*) denotes the set of variables that occur in *pc*. Function *random* selects random integer values in the range $[lo, hi]$ and builds assignments to each variable in this set (line 2). (For simplicity, we only show the case for integers.) The function $eval(pc, \overrightarrow{iv})$ checks whether the candidate solution $\overrightarrow{iv}$ models *pc*. This function evaluates the concrete boolean expression that *pc* encodes using the variable assignments in $\overrightarrow{iv}$. ranSOL returns $\overrightarrow{iv}$ at line 4 if it is a solution to *pc*, or returns *empty* on timeout (line 7).

Symbolic Solver. We use a well-known SMT solver to represent our baseline symbolic solver: CVC3 [2]. Symbolic constraint solvers are complete for a given set of supported decidable theories. For example, CVC3 supports rational and integer linear arithmetic (among others). However, these solvers are incomplete for solving constraints with non-linear arithmetic, integer division and modulo. Constraints with these operators are undecidable. Nevertheless it is important to note that CVC3 can still find solutions to special cases of such constraints.

```
Require: path condition pc
Require: random seed s, range [lo, hi]
 1: while ¬timeout do
 2:    iv⃗ ⇐ random(vars(pc), range)
 3:    if eval(pc, iv⃗) then
 4:       return iv⃗
 5:    end if
 6: end while
 7: return empty
```

Fig. 1: Random (ranSOL)

```
Require: path condition pc
Require: random seed s, and range [lo, hi]
 1: (pcgood, pcbad) ⇐ partition(pc)
 2: sols ⇐ eRanSOL.solve(pcbad,seed,range)
 3: for all iv1⃗ in sols do
 4:    newpc ⇐ pcgood\iv1⃗
 5:    iv2⃗ ⇐ symSOL.solve(newpc)
 6:    if iv2⃗ ≠ empty then
 7:       return iv1⃗ + iv2⃗
 8:    end if
 9: end for
10: return empty
```

Fig. 3: Undecidable constraints first (UCF)

```
Require: path condition pc
Require: random seed s, and range [lo, hi]
 1: (pcgood, pcbad) ⇐ partition(pc)
 2: iv1⃗ ⇐ symSOL.solve(pcgood)
 3: if (iv1⃗ = empty) then
 4:    return empty
 5: end if
 6: newpc ⇐ pcbad\iv1⃗
 7: iv2⃗ ⇐ ranSOL.solve(newpc,seed,range)
 8: return iv2⃗ = empty ? empty : iv1⃗ + iv2⃗
```

Fig. 2: Decidable constraints first (DCF)

```
Require: path condition pc
Require: random seed s, and range [lo, hi]
 1: (goodvars, badvars) ⇐ partition(pc)
 2: while ¬timeout do
 3:    iv1⃗ ⇐ random(badvars)
 4:    newpc ⇐ pc\iv1⃗
 5:    iv2⃗ ⇐ symSOL.solve(newpc)
 6:    if iv2⃗ ≠ empty then
 7:       return iv1⃗ + iv2⃗
 8:    end if
 9: end while
```

Fig. 4: Bad variables first (BVF)

2.2 Heuristic search solvers

This section discusses two solvers based on well-known heuristic search techniques: Genetic Algorithms (GA) [23] and Particle Swarm Optimization (PSO) [26]. Conceptually, these solvers attempt to optimize the random search of ranSOL. The basic task of these algorithms is to search a *space* of candidate solutions to identify the best ones in terms of a problem-specific *fitness function*. The search process usually starts with the selection of randomly-chosen individuals in the search space (i.e., candidate solutions in the search problem). The search proceeds by making changes to each individual iteratively with *search operators* until the search meets some stop criteria (e.g., the result is good enough or the search time expires). The decision to change individuals in the search space depends on the evaluation of their current fitness values. The principle of these algorithms is that the new individuals generated across successive iterations will converge to the best solutions in the search space, i.e., each iteration potentially explores better regions in the search space. For our application, an individual is simply a *solution* as defined above: a vector of variable assignments (e.g. $\langle x \mapsto 2, y \mapsto 0 \rangle$).

Two fitness functions have been widely used for constraint solving problems: MaxSAT [17,29,37] and Stepwise Adaptation of Weights (SAW) [6,16]. MaxSAT is a simple heuristic that counts the number of clauses that can be satisfied by a solution. Maximum fitness is obtained when the solution satisfies all clauses (boolean expressions) in a constraint system (conjunction of clauses). The main issue with MaxSAT is that the solver can sometimes favor solutions that satisfy several easy-to-solve constraints at the expense of solutions that satisfy only a few hard-to-solve constraints. Bäck et al. proposed SAW [6] to reduce the impact of this issue. SAW associates a weight to each clause in a constraint. Each weight is updated with each iteration when it is not satisfied. The use of SAW helps to identify hard-to-solve clauses with the progress of search iterations. The solver can use this information to favor individuals that are more fit to satisfy hard-to-solve clauses. We used SAW to evaluate fitness in our GA and PSO implementations.

Summary of GA and PSO. A GA search starts with a population of individuals randomly selected from the search space. The GA algorithm evaluates each individual at each iteration in order to select and combine them in pairs. The worst fitted are removed from the population. Each iteration produces a new population with special operators: a *crossover* combines two individuals to produce others and a *mutation* changes one individual. The crossover operator splits two individuals (two solution vectors) in two parts by setting, randomly, a cut-point, and produces two new subjects that are the combination of both solutions. The mutation operator randomly selects a variable of the solution vector and changes its value.

Similar to GA, PSO operates with an initial random population of candidate solutions called *particles*. The interactive collaboration of particles to compute a solution is the main difference between GA and PSO. Each particle has a *position* in the search space and a contributing factor to the population, typically called *velocity*, which PSO uses to update the next position of each particle. A typical PSO iteration updates the velocity of a particle according to global best and local best solutions. The next position of a particle depends on the old position and the new computed velocity. The mutually-recursive equations below govern the update of velocity and position across successive iterations t.

$$v_{t+1} = \omega * v_t + r_1 * c_1 * (best_{part} - x_t) + \\ r_2 * c_2 * (best_{pop} - x_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

The vectors v and x store respectively velocities and positions for each particle. The label t refers to one iteration. The coefficient ω, called inertia, denotes the fraction of velocity in iteration t that the particle will inherit in iteration $t+1$. Coefficients r_1 and r_2 are numbers within the range [0,1] randomly generated according to some probability distribution (in our implementation, the uniform distribution). The vector $best_{part}$ stores the best solution each particle visited and c_1 indicates the confidence level to local solutions (i.e., to one individual particle). The term $best_{pop}$ indicates the best solution in the population and c_2 indicates the confidence level to global solutions. Note that the position of a particle at instant $t+1$ is computed by simply adding the velocity v_{t+1}.

Discussion. Although the algorithms have a similar behavior, each one has a different computational cost. At each iteration GA needs to eliminate less fitted individuals, add new ones with crossover, and modify existing with mutation. The PSO algorithm updates the search state more efficiently; it essentially only use matrix arithmetic. Note also that both approaches evaluate each solution in every iteration. It is also important to highlight that there is a trade-off associated to the choice of the population size. Too large values for this parameter will slow the convergence of the heuristic search (in this case the heuristic search will be more similar to a purely random search). On the other hand, too small values for population size may be not adequate to provide an effective space exploration, specially considering large search spaces.

2.3 Hybrid solvers

This section describes solvers that conceptually combine ranSOL and symSOL. These hybrid solvers make different decisions in (i) what to randomize and in (ii) which order.

Note on terminology. We use the term **eRanSOL** in reference to an extension of ranSOL that can return many solutions. We use the term $pc\backslash\overrightarrow{iv}$ to denote a substitution of variables in pc with their concrete values in $\overrightarrow{iv}$. For example, $(x > 0 \wedge x > y+1)\backslash\langle x \mapsto 2\rangle$ reduces to $(2 > 0 \ \wedge \ 2 > y + 1)$.

2.3.1 Decidable constraints first (DCF)

Figure 2 shows the pseudo-code for the **DCF** solver. At line 1, the solver partitions the constraint *pc* into two: the first, named *pcgood*, contains decidable constraints. The second, *pcbad*, complements the first with undecidable constraints. Recall that *pc* consists of a conjunction of boolean expressions. The algorithm reduces to plain random solving if *pcgood* is empty and to plain symbolic solving if *pcbad* is empty. (We omit these checks for simplicity.) When both parts are non-empty, the combined solver uses the symbolic solver to first find a solution to *pcgood* (line 2). As *pcgood* only contains decidable constraints, an empty answer from symSOL indicates that *pcgood* is unsatisfiable (lines 3-5). Consequently, *pc* is also unsatisfiable since $\neg pcgood$ implies $\neg pc$ (from the partition function). In case symSOL finds a solution, the solver produces the constraint *newpc* with the substitution $pcbad\backslash\overrightarrow{iv_1}$. If the random solver can find one solution to *newpc*, then DCF returns $\overrightarrow{iv_1} + \overrightarrow{iv_2}$ as solution, i.e. variable assignments produced by the symbolic and the random solvers, respectively. For illustration, DCF partitions the constraint $b \ \% \ a \neq 0 \wedge a > 0$ in two: $pcgood = a > 0$ and $pcbad = b \ \% \ a \neq 0$. (The modulo operator makes the constraint undecidable.) DCF passes *pcgood* to the symbolic solver, and uses the solution, say $\langle x \mapsto 2\rangle$, to simplify *pcbad* and finally call the random solver on $b \ \% \ 2 \neq 0$, which can be easily solved by it.

2.3.2 Undecidable constraints first (UCF)

Figure 3 shows the pseudo-code for the **UCF** solver. It differs from DCF in the order of randomization: it attempts to solve the undecidable parts first. UCF uses eRanSOL to find many solutions to *pcbad*. The main loop checks for each solution $\overrightarrow{iv_1}$ whether symSOL can find a solution to $pcgood\backslash\overrightarrow{iv_1}$ (lines 3-9) . Note that, differently from DCF, UCF calls symSOL once in each iteration. This algorithm corresponds to the one we discussed in Section 1.

2.3.3 Bad variables first (BVF)

Figure 4 shows the pseudo-code for the **BVF** solver. It is similar to UCF in the order of calls to random and symbolic solvers. However, it partitions the problem differently. While the previous hybrid solvers partition the set of clauses from one input constraint, BVF *partitions the set of variables* that occur in that constraint. For example, BVF randomizes only the variable b to solve the constraint $a = b^2 + c$, while UCF and DCF randomizes all variables in this case as they appear in a clause involving non-linear arithmetic.

BVF is similar to the algorithm proposed in DART [22] as it randomizes a selection of variables for making the constraint decidable. DART, however, randomizes variables incrementally from left to right in the order they appear in the constraint. The constraint $a = b^2 \wedge ... \wedge b = a^2$ illustrates one diference between BVF and DART. BVF randomizes variables b and a while DART can avoid the randomization of a as its value is determined after b's value selection. We did not evaluate DART itself in this paper but we intend to evaluate it as a future work.

2.3.4 Other combinations

Note that DCF and UCF use a random solver. We also want to evaluate whether these solvers improve precision with the use of a guided random search. To that end, we combine DCF and UCF with GA and PSO leading to 4 new solvers. It remains to be investigated how UCF can be combined with heuristic search solvers.

3 Evaluation

This section reports the evaluation of the proposed solvers with the constraints that a concolic execution [40] generates. To that end we use data-structures from a variety of open source programs.

Concolic execution. A concolic execution interprets the program simultaneously in a concrete and symbolic domain. The use of a concrete state enables a concolic execution to evaluate *deterministically* any program expression. This provides a way to handle typical limitations of a pure symbolic execution such as infinite loops and recursion, exploration of infeasible paths, and array indexing. The use of a symbolic state (which the concrete state is an instance of) enables a concolic execution to collect constraints that lead to non-visited paths along the execution of one concrete path.

Figure 5 shows the pseudo-code for a concolic execution test driver. The driver's goal is to produce inputs to a procedure *ptest*. It takes as parameter the procedure *ptest* for which we want to generate input, a random seed s and the range of values $[lo, hi]$. The driver reports as output a set of input vectors to *ptest*, where each input leads *ptest*'s execution to a different program path. One iteration of the main loop explores one concrete path and produces several path constraints (corresponding to non-visited paths along that concrete path). A solution to a constraint, when found, drives the next concolic execution of *ptest* (line 8).

```
Require: parameterized test ptest
Require: random seed s, range [lo, hi]
 1: iv⃗ ⇐ random(vars(ptest), range)
 2: result ⇐ {iv⃗}
 3: pcs ⇐ pcs + run(ptest, iv⃗)
 4: while size(pcs) > 0 do
 5:    iv⃗ ⇐ solve(pickOne(pcs),s,range)
 6:    if iv⃗ ≠ empty then
 7:       result ⇐ result ∪ {iv⃗}
 8:       pcs ⇐ pcs + run(ptest, iv⃗)
 9:    end if
10: end while
11: return result
```

Fig. 5: Concolic Execution Driver

Subjects. We used data-structure from a variety of sources. *BST* is an implementation of a binary search tree from Korat [11]. *FileSystem* is a simplification of the Daisy file system [35]. *TreeMap* is a jdk1.4 implementation (`java.util.TreeMap`) of red-black trees. *Switch* refers to one example program from the jCUTE distribution [40]. *RatPoly* is an implementation of rational polynomial operations from the Randoop distribution [34]. *RationalScalar* is another implementation of rational polynomials from the ojAlgo library [3]. *Newton* is an implementation of the newton's method to iteratively compute the square root of a number [44]. *HashMap* is a jdk1.4 implementation (`java.util.HashMap`) of a map that uses hash values as keys. *Colt* [1] is an open source Java library to support scientific computing and data analysis. It includes data structures and algorithms for linear algebra, statistics, and Monte Carlo simulation. From this library, we have chosen the `MersenneTwister` class to be tested. This class is a random number generator that follows a normal distribution [30]. It contains large non-linear constraints.

We decided to extract our benchmark from real software as our main motivation is software testing. Surely benchmarks like SMT-LIB [36] could also be good source of constraints provided that they contain constraints that occur in software (in contrast to constraints from hardware or hybrid systems).

Table 1 shows the characteristics of each concolic execution in terms of the size, used domains, and used

Subject	Literal Type	Avg. Number of Literals	Avg. Clause Size	Avg. Number of Clauses	Operators
BST	int	12.13	2	95.35	$> \geq < \leq$
FileSystem	int	3.54	2	41.67	$\geq < = \neq$
TreeMap	int	12.67	2	154.14	$\geq < = \neq$
Switch	int	10.95	2.55	106.93	$+ * > \leq = \neq$
RatPoly	int	3.89	8.11	60.36	$+ - * / \% > \geq < \leq = \neq$
RationalScalar	int	3.43	36.72	25.7	$* / \% = \neq$
Newton	float	1.56	4818.02	27.48	$+ - * / > \leq$
HashMap	int	17.73	63	78.31	$+ \ll \gg \ggg \hat{} =$
Colt	int	1	1390.82	16.81	$+ * \gg \ggg \& \mid \hat{} = \neq$

Table 1: Experiments features.

	ranSOL	GA	PSO	**CVC3**	DCF	UCF	BVF	DCF + GA	UCF + GA	DCF + PSO	UCF + PSO	BEST
BST	0.015	0.010	0.022	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
FileSystem	0.060	0.00	0.060	0.999	0.999	0.999	0.999	1.000	0.999	1.000	0.999	1.000
TreeMap	0.021	0.013	0.020	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Switch	0.484	0.000	0.349	0.994	0.997	0.995	0.996	0.079	0.994	0.074	0.993	0.997
avg.	0.145	0.005	0.112	0.998	**0.999**	0.999	0.999	0.769	0.998	0.768	0.998	-
RatPoly	0.119	0.011	0.100	0.092	0.855	0.271	0.173	0.855	0.129	0.855	0.192	0.855
RationalScalar	0.442	0.010	0.370	0.040	0.428	0.180	0.100	0.220	0.040	0.500	0.144	0.500
Newton	0.743	0.143	0.571	0.000	0.486	0.314	0.000	0.143	0.000	0.286	0.171	0.743
HashMap	0.023	0.000	0.632	0.000	0.023	0.023	0.000	0.000	0.000	0.643	0.652	0.652
Colt	0.856	0.320	0.351	0.000	0.930	0.931	0.000	0.321	0.320	0.363	0.366	0.931
avg.	0.436	0.096	0.404	0.026	**0.544**	0.344	0.055	0.308	0.098	0.529	0.305	-

Table 2: Average precision observed for each pair of subject (row) and solver (column).

	ranSOL	GA	PSO	**CVC3**	DCF	UCF	BVF	DCF + GA	UCF + GA	DCF + PSO	UCF + PSO
BST	0.000	0.000	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
FileSystem	0.000	0.000	0.000	0.002	0.002	0.002	0.000	0.002	0.000	0.002	0.000
TreeMap	0.000	0.003	0.003	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Switch	0.004	0.000	0.034	0.008	0.004	0.006	0.010	0.045	0.013	0.033	0.011
RatPoly	0.012	0.006	0.002	0.000	0.000	0.119	0.078	0.000	0.023	0.000	0.010
RationalScalar	0.033	0.017	0.014	0.000	0.052	0.025	0.000	0.000	0.000	0.000	0.008
Newton	0.060	0.000	0.000	0.000	0.100	0.259	0.000	0.000	0.000	0.000	0.148
HashMap	0.000	0.000	0.153	0.000	0.000	0.000	0.000	0.000	0.000	0.198	0.178
Colt	0.019	0.020	0.019	0.000	0.013	0.014	0.000	0.020	0.021	0.027	0.021

Table 3: Standard deviation observed for each pair of subject (row) and solver (column).

operators. The operators $\ll$, $\gg$, $\ggg$ &, | and $\hat{}$ are the bitwise operators: shift left, shift right, unsigned shift right, "and", "or" and exclusive "or", respectively. The others correspond to typical arithmetic, relational and boolean operators. Although one cannot infer the problem complexity based on such data (as it depends on the unknown size of the solution space), it is still useful to understand the performance and the behavior of the discussed solvers. The number of dimensions of the search space is called *dimensionality* and, on the random and search-based algorithms, each literal is mapped as a search space dimension.

General Results. Table 2 shows the average precision obtained by the implemented solvers (each solver was executed 10 times with different random seeds). Table 3 in turn presents the standard deviation of precision observed in these experiments. In each run, we used a 1s timeout associated to each solver call and a 180 minutes timeout per subject. The first group of 4 subjects on top of the tables 2 and 3 (namely, *BST*, *FileSystem*, *TreeMap* and *Switch*) produces only decidable constraints. For this group, the symbolic solver and, consequently, all hybrid solvers showed roughly the same average precision and low standard deviation values. The second group of 5 subjects at the bottom of tables 2 and 3 (namely, *RatPoly*, *RationalScalar*, *Newton*,

HashMap, and *Colt*) produces only undecidable constraints. For this group, CVC3 rarely finds a solution. The best results in turn were obtained by using hybrid solvers: the DCF solver had the best performance (0.544 of precision in average), followed by DCF+PSO (0.529 of precision in average). The results observed in these subjects presented in general higher variation compared to the subjects which contain only decidable constraints.

Comparing the random and the search-based algorithms applied to the bottom 5 subjects on Table 2, we notice that the random solver was more consistent: it finds more solutions in 4 out of 5 subjects, and presents higher average precisions particularly when compared to the GA solver. We highlight that the search-based solvers are affected by its computational complexity. In fact, a candidate solution must be evaluated on each iteration and a set of new solutions must be generated through the application of search operators. This is more drastic for the GA algorithm, since it has more complex search operators compared to PSO. ranSOL, due to its lower complexity, traverses the search space swiftly and is able to find solutions for more than half of the satisfiable constraints.

It is important to note that one solver often found solutions that another solver missed, particularly when the subjects contain undecidable constraints. In order to illustrate this feature, Figure 6 shows a pairwise comparison of the solvers observed in an experiment for the subject *Colt*. A cell on line i and column j indicates the number of constraints that solver i solved and solver j missed. As we can see, the behavior of the solvers varied significantly in the set of constraints they solved. No single solver was able to find solutions to all constraints in this subject. By comparing PSO and GA, for instance, we can see that PSO is able to solve 79 constraints not solved by GA. Conversely, the GA solver finds solutions to 39 constraints that PSO is not able to solve. This result shows a complementary behavior between the two techniques.

Impact of timeout on precision. In order to observe the impact of timeout on precision, we run each solver with increasing timeout values: 100, 300, 500 and 700ms. In general, the impact of the timeout on precision was not significant. This behavior is illustrated by the precision of the solvers for *BST*, which is presented in Figure 7 as stacked bars. The precision is aggregated at each timeout increase. The subjects that generate linear-integer constraints are solved more efficiently by CVC3, and no significant gain is observed with a timeout increment. The only exception was *Switch* (Figure 8), in which the timeout affected the CVC3 performance in such a way that it was no better than the other solvers. This is, to some extent, unexpected as the constraints generated by Switch are supported by CVC3, which should therefore outperform the others.

Just like *Switch*, the solver precision for *Colt* also increased as the timeout increased (see Figure 9). As *Colt* generates constraints that CVC3 cannot solve, we did not include CVC3 in Figure 9.[1]

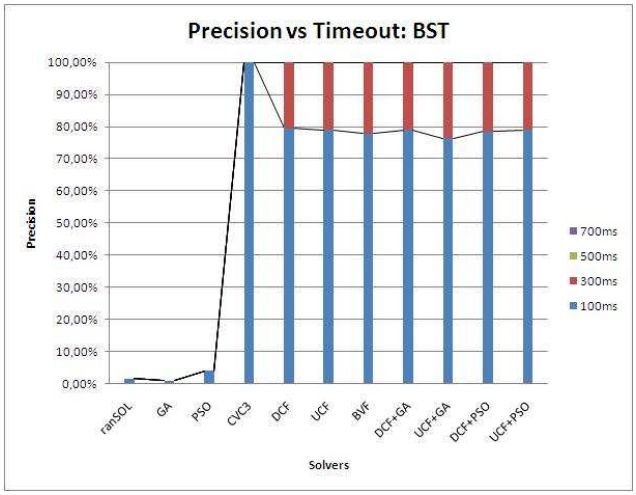

Fig. 7: Solvers precision on *BST* for different timeout values.

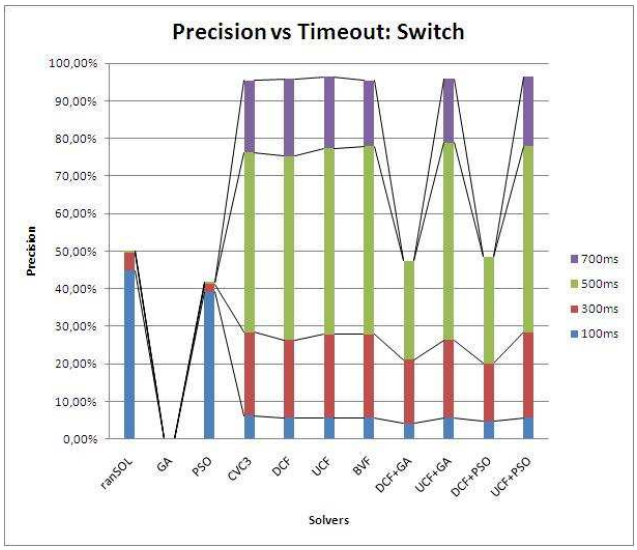

Fig. 8: Solvers precision on *Switch* for different timeout values.

Impact of the search parameters. As shown in Table 2, the random solver outperformed PSO and GA. As mentioned before, the complexity of the algorithms

[1] The complete set of data for these experiments are available at `http://www.cin.ufpe.br/~jmi/isse`.

Colt											
	ranSOL	GA	PSO	CVC3	DCF	UCF	BVF	DCF + GA	UCF + GA	DCF + PSO	UCF + PSO
ranSOL	-	226	181	311	0	0	311	226	226	184	180
GA	10	-	39	95	7	7	95	0	0	48	43
PSO	5	79	-	135	4	4	135	79	79	26	23
CVC3	0	0	0	-	0	0	0	0	0	0	0
DCF	33	256	213	344	-	1	344	256	256	216	213
UCF	34	257	214	345	2	-	345	257	257	217	214
BVF	0	0	0	0	0	0	-	0	0	0	0
DCF+GA	10	0	39	95	7	7	95	-	0	48	43
UCF+GA	10	0	39	95	7	7	95	0	-	48	43
DCF+PSO	4	84	22	131	3	3	131	84	84	-	24
UCF+PSO	3	82	22	134	3	3	134	82	82	27	-

Summary: 358 SAT, 323 UNK.
ranSOL:311, GA:95, PSO:135, CVC3:0, DCF:344, UCF:345
BVF:0, DCF+GA:95, UCF+GA:95, DCF+PSO:131, UCF+PSO:134

Fig. 6: Results of various solvers for *Colt*. Column and row show solver identifiers. A cell denotes the difference of constraints that a solver (from row) can solve and another (from column) cannot. The bottom line summarizes the results.

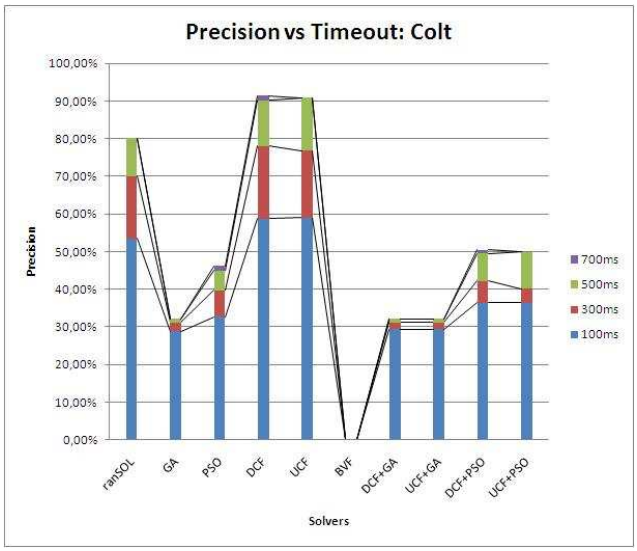

Fig. 9: Solvers precision on *Colt* for different timeout values.

of the heuristic solvers impacts significantly their performance when a short timeout is set.

Another aspect that can have an impact on the heuristic solver performance is the choice of the search parameters. In our experiments, we have used default values for GA and PSO parameters, adopted in other domains of application. However, the best values for the search parameters are dependent on the subject at hand. For instance, the ranSOL solver obtained a better performance in comparison to both GA and PSO for the *Colt* subject (Table 2). In a new experiment, we adopted a high value of 30 individuals in the population size, for which it is expected that both GA and PSO present a more similar behavior compared to a random search. In this experiment, the precision rates achieved by GA and PSO were respectively 0.52 and 0.49, which are still lower than the ranSOL precision (0.85), but higher than the precision rates observed by the GA and the PSO default parameters (0.32 and 0.35 respectively).

The above experiment indicates that an improvement in performance of the heuristic solvers could be achieved by an adequate choice of the search parameters. Obviously, a trial-and-error procedure to choose the parameters is not feasible in practice. In future, we intend to investigate strategies to define the search parameters taking into account the features of the subject being tackled.

Discussion. On linear-integer subjects, except for *Switch*, the expected behavior was observed: the symbolic solver performed well. *Switch* was an exception as the symbolic solver ran out of time constantly when solving its constraints. For the second group of subjects, which produces undecidable constraints, no solver came out as the most appropriate. Each solver seems to be useful for different problems. On average, DCF and DCF+PSO have shown an equivalent performance with the highest precision values. BVF was observed as the less appropriate for those subjects.

GA and its hybrids did not performed well compared to the other search solvers. This can be explained by its higher complexity in comparison to the ranSOL and the PSO algorithms. Its operators (crossover and mutation) have a computational cost that makes it inappropriate for symbolic testing. Nevertheless, a hybrid algorithm combining GA and PSO could be useful to provide a further space traversing technique since GA was able to solve constraints not solved by PSO (as it was mentioned in the *Colt* experiment).

The ranSOL solver and its hybrids came out as the most adaptable solvers, with better results in comparision to the solvers based on GA and PSO. In fact, its low computational cost allowed it to find solutions without timing out.

The results indicate that the random and the hybrid solvers are effective to solve undecidable constraints. However, one can not predict which solver fits best for a particular subject since the solvers typically solve different sets of constraints. The results suggest that one should alternate the use of different solvers in successive concolic execution iterations. The solvers could also run in parallel and take advantage of multi-core processors and larger memories (this trend has already been investigated by Holzmann et al. [25] for the model-checking domain).

Regarding the domain size, the choice of domain size is driven by a tradeoff between efficiency and coverage: random solver performs better with the smallest domain that includes all solutions. To the best of our knowledge it is not possible to statically infer such ideal domain. Although we have not carried out experiments with different domain sizes, we believe that random solvers perform consistently for different domain sizes.

Concerning different data-types, we understand that a fuzzing of programs with String and reference types can be adjusted to only generate integer constraints. One can encode a string object with an array of characters (integers), and object references as integer symbolic variables.

4 Related Work

Random-symbolic testing has been widely investigated recently to automate test input generation [21,22,28]. It alternates concrete and symbolic execution to alleviate their main limitations. It is important to note that random-symbolic testing provides two orthogonal contributions: (i) constraint generation and (ii) constraint solving. Our goal is to improve constraint solving. In this context, DART [22] conceptually uses a random solver to simplify symbolic solving. We plan to evaluate the solvers we proposed with a DART solver as discussed in Section 2. Another approach to automate test input generation is random testing [10,18,32,33]. The ranSOL solver differs from random testing in two important ways: (a) random testing generates inputs for program parameters; a classification of good input depends on the result of an actual execution, and (b) random testing typically generates test sequence and data simultaneously. We plan to combine random sequence generation together with random-symbolic input generation to automate testing.

We used the Satisfiability Modulo Theories (SMT) [12,20,41] solver CVC3 [2], which uses built-in theories for rationals and integer linear arithmetic (with some support to non-linear arithmetic). SAT solving research of undecidable theories has focused on the analysis of hybrid and control systems, as recently evidenced by the iSAT [19] and the ABSolver [7] systems. The first integrates the power of SMT solvers to solve boolean constraints with the capability of Interval Constraint Propagation (ICP) [9] to deal with non-linear constraint systems, while the second uses a DPLL-based [14] algorithm to perform the search and defers theory problems to subordinate solvers. As in hybrid and control systems, undecidable theories also arise in the domain of software systems. This paper shows simple algorithms that can be effective to solve both decidable and undecidable fragments of constraints that a concolic *program* execution generates. Another distinguishing feature of our solvers is that, in contrast to a DPLL(T) [20] solver, they are not dependent on a background theory T. One can use the solvers this paper describes in combination with any theory-specific solver to benefit from their complementary nature.

There are variations to the search-based solvers presented in Section 2 which we plan to investigate. Ru and Jianhua propose a hybrid technique which combines GA and PSO by creating individuals in a new generation by crossover and mutation operations [38]. Instinct-based PSO adds another criterion (the instinct) to influence a particle's behavior [4]. The instinct represents the intrinsic "goodness" of each variable of a particle's candidate solution. We also plan to analyze how test inputs generated from our solvers compare to those generated directly with a PSO algorithm whose fitness function is based on coverage [42].

Dwyer et al. [15] propose a technique, called Parallel Randomized State-space Search, that runs multiple parallel randomized state-space searches, and terminates all searches when the first one finds an error. They aim at detecting hard to find errors in concurrent programs. Holzmann et al. [25] propose a new approach that allows to perform verifications within strict time bounds. Their tool uses parallelism and search diversity to optimize coverage. They focus on improving performance of a number of model checking tasks using parallelization. We propose randomized solvers that can be improved using parallelization.

Wintersteiger et al. [43] implement a parallel SMT solver. They parallelize a sequential solver by running multiple solvers, each configured to use different heuristics. Our results show that it is difficult to predict the

best heuristic that will fit for a particular subject. As a future work, we intend to run our solvers in parallel. An approach can incorporate our solvers in their infrastructure [43]. We aim at obtaining a solver with results that are closer to our "BEST" column in Table 2. In other contexts that are sensitive to the range, we can parallelize our random solvers using different ranges.

5 Conclusions

This paper proposes and implements a plain random solver, seven hybrid solvers combining random, search-based and symbolic solvers, and two heuristic search-based solvers. We use a random solver and a symbolic solver (CVC3) as baselines for comparison. We evaluate the solvers with constraints that a concolic execution generates on 9 subjects. For the concolic execution on subjects that generated only decidable constraints the experiments revealed, as expected, that CVC3 is superior in all but 2 cases. CVC3 timed out in these cases. For solving undecidable constraints, no solver subsumes another. It suggests that one may not be able to predict the heuristic that will fit best for a particular subject; it is preferable to run them all in parallel.

Next we want to analyse several open source projects to quantify the number of constraints that would produce undecidable constraints. We believe this is a necessary step to provide evidence for the practical relevance of this research.

Acknowledgements. We are grateful to Augusto Sampaio, Alexandre Mota, Leopoldo Teixeira and the anonymous reviewers for the comments on this work. This work was partially supported by the CNPQ grant 136172/2008.3 and FACEPE grants APQ-0093-1.03/07 and APQ-0074-1.03/07.

References

1. Colt webpage. http://acs.lbl.gov/~hoschek/colt
2. CVC3 webpage. http://www.cs.nyu.edu/acsys/cvc3/
3. ojAlgo webpage. http://ojalgo.org
4. Abdelbar, A., Abdelshahid, S.: Instinct-based PSO with local search applied to satisfiability. In: IEEE International Joint Conference on Neural Networks, pp. 2291–2295 (2004)
5. Anand, S., Pasareanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to Java PathFinder. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 134–138 (2007)
6. Bäck, T., Eiben, A.E., Vink, M.E.: A superior evolutionary algorithm for 3-SAT. In: 7th Conference on Evolutionary Programming, pp. 125–136. Springer-Verlag, UK (1998)
7. Bauer, A., Pister, M., Tautschnig, M.: Tool-support for the analysis of hybrid systems and models. In: R. Lauwereins, J. Madsen (eds.) Design, Automation and Test in Europe Conference and Exposition (DATE), pp. 924–929. ACM, Nice, France (2007). URL http://doi.acm.org/10.1145/1266366.1266565
8. Beizer, B.: Software Testing Techniques. International Thomson Computer Press (1990)
9. Benhamou, F., Granvilliers, L.: Continuous and interval constraints. In: F. Rossi, P. van Beek, T. Walsh (eds.) Handbook of Constraint Programming, Foundations of Artificial Intelligence, chap. 16. Elsevier Science Publishers, Amsterdam, The Netherlands (2006). URL http://www.lina.sciences.univ-nantes.fr/Publications/2006/BG06
10. Bird, D.L., Munoz, C.U.: Automatic generation of random self-checking test cases. IBM Systems Journal **23**(3), 228–245 (1983)
11. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 123–133 (2002). DOI http://doi.acm.org/10.1145/566172.566191
12. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., Sebastiani, R.: An incremental and layered procedure for the satisfiability of linear arithmetic logic. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 317–333 (2005)
13. Csallner, C., Smaragdakis, Y.: JCrasher: An automatic robustness tester for Java. Software - Practice and Experience **34**, 1025–1050 (2004)
14. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of ACM **5**(7), 394–397 (1962). DOI http://doi.acm.org/10.1145/368273.368557
15. Dwyer, M.B., Elbaum, S., Person, S., Purandare, R.: Parallel randomized state-space search. In: ICSE '07: Proceedings of the 29th international conference on Software Engineering, pp. 3–12. IEEE Computer Society, Washington, DC, USA (2007)
16. Eiben, A., van der Hauw, J.: Solving 3-SAT by GAs adapting constraint weights. Evolutionary Computation pp. 81–86 (1997). DOI 10.1109/ICEC.1997.592273
17. Folino, G., Pizzuti, C., Spezzano, O.: Combining cellular genetic algorithms and local search for solving satisfiability problems. In: IEEE Conference on Tools with Artificial Intelligence, pp. 192–198 (1998)
18. Forrester, J., Miller, B.: An empirical study of the robustness of windows NT applications using random testing. In: USENIX Windows Systems Symposium, pp. 59–68 (2000)
19. Franzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. Journal on Satisfiability, Boolean Modeling and Computation **1**(?), 209–236 (2007)
20. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: R. Alur, D. Peled (eds.) Computer aided verification : 16th International Conference, CAV 2004, *Lecture Notes in Computer Science*, vol. 3114, pp. 175–188. Springer, Boston, Massachusetts (2004)
21. Godefroid, P.: Compositional dynamic test generation. In: 34th Symposium on Principles of Programming Languages (POPL), pp. 47–54. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1190216.1190226
22. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Procedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), vol. 40, pp. 213–223. ACM Press, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1064978.1065036

23. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1989)
24. Harman, M.: Automated test data generation using search based software engineering. In: AST '07: Proceedings of the Second International Workshop on Automation of Software Test, p. 2. IEEE Computer Society, Washington, DC, USA (2007). DOI http://dx.doi.org/10.1109/AST.2007.4
25. Holzmann, G.J., Joshi, R., Groce, A.: Swarm verification. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 1–6. IEEE (2008)
26. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: IEEE Neural Networks, pp. 1942–1948 (1995)
27. King, J.C.: Symbolic execution and program testing. Communications of ACM **19**(7), 385–394 (1976)
28. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 416–426 (2007). URL http://doi.ieeecomputersociety.org/10.1109/ICSE.2007.41
29. Marchiori, E., Rossi, C.: A flipping genetic algorithm for hard 3-SAT problems. In: Genetic and Evolutionary Computation Conference, vol. 1, pp. 393–400. Morgan Kaufmann, Orlando, Florida, USA (1999)
30. Matsumoto, M., Nishimura, T.: Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation **8**(1), 3–30 (1998). DOI http://doi.acm.org/10.1145/272991.272995
31. National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3 (2002)
32. Pacheco, C., Ernst, M.: Randoop: feedback-directed random testing for Java. In: Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, pp. 815–816 (2007)
33. Pacheco, C., Ernst, M.D.: Eclat documents. Online manual (2004). http://people.csail.mit.edu/people/cpacheco/eclat/
34. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 75–84. IEEE Computer Society, Washington, DC, USA (2007). DOI http://dx.doi.org/10.1109/ICSE.2007.37
35. Qadeer, S.: Daisy File System. Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software. 2004
36. Ranise, S., Tinelli, C.: The SMT-LIB standard: Version 1.2. Tech. rep. (2006)
37. Rossi, C., Marchiori, E., Kok, J.N.: An adaptive evolutionary algorithm for the satisfiability problem. In: Symposium on Applied Computing (SAC), pp. 463–469. Como, Italy (2000)
38. Ru, N., Jianhua, Y.: A GA and particle swarm optimization based hybrid algorithm. In: IEEE World Congress on Computational Intelligence. Hong Kong (2008)
39. Santhanam, P., Hailpern, B.: Software debugging, testing, and verification. IBM Systems Journal **41**, 4–12 (2002)
40. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: CAV, pp. 419–423 (2006)
41. Tinelli, C.: A DPLL-based calculus for ground satisfiability modulo theories. In: Proceedings of the European Conference on Logics in Artificial Intelligence, pp. 308–319. Springer-Verlag, London, UK (2002)
42. Windisch, A., Wappler, S., Wegener, J.: Applying particle swarm optimization to software testing. In: H. Lipson (ed.) Genetic and Evolutionary Computation Conference, pp. 1121–1128 (2007)
43. Wintersteiger, C., Hamadi, Y., de Moura, L.: A concurrent portfolio approach to SMT solving. pp. 715–720 (2009)
44. Ypma, T.J.: Historical development of the Newton-Raphson method. SIAM Review **37**(4), 531–551 (1995)