A Comparative Study of Randomized Constraint Solvers for Random-Symbolic Testing

Mitsuo Takaki [‡], Diego Cavalcanti [†], Rohit Gheyi [†],
Juliano Iyoda [‡], Marcelo d'Amorim [‡], and Ricardo Prudêncio [‡]

[‡] Federal University of Pernambuco, Recife, PE, Brazil
[†] Federal University of Campina Grande, Campina Grande, PB, Brazil
{mt2,jmi,damorim,rbcp}@cin.ufpe.br
{diegot,rohit}@dsc.ufcg.edu.br

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 compare the effectiveness of a symbolic solver (CVC3), a random solver, three hybrid solvers (i.e., mix of random and symbolic), and two heuristic search solvers. We evaluate the solvers on two benchmarks: one consisting of manually generated constraints and another generated with a concolic execution of 8 subjects. In addition to fully decidable constraints, the benchmarks include constraints with non-linear integer arithmetic, integer modulo and division, bitwise arithmetic, and floating-point arithmetic. As expected symbolic solving (in particular, CVC3) subsumes the other solvers for the concolic execution of subjects that only generate decidable constraints. For the remaining subjects the solvers are complementary.

1 Introduction

Software testing is important and expensive [8, 28, 35]. Several techniques have been proposed to reduce this cost. Automation of test data generation, in particular, can improve testing productivity. Random testing [13, 30] and symbolic testing [25] are two widely used techniques with this goal and with well-known limitations. On the one hand, random testing fails to explore a search space in a systematic manner: it can 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 *incapability of solving 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 possible way to combine random and symbolic solvers is to first delegate to the random solver the parts of a constraint that build on theories a symbolic solver does not support. Then use the 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. (For simplicity, we assume the constraint is satisfiable and that the random solver can find a solution.) Important to note is that, as for typical decision procedures in SMT solvers [20, 37], random and symbolic solvers are *not* independent in this combination; they *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.

We define **recall** as the fraction of constraints that a solver can find solutions out of the total number of *satisfiable constraints*. (We classify aproximately a constraint as satisfiable if at least one solver can find solution to it.) This metric quantifies completeness. Our goal is to increase recall. This paper makes the following contributions:

- The proposal and implementation of several randomized constraint solvers. We implemented a plain random solver, three hybrid constraint solvers, and two search-based solvers. We use the random solver and the symbolic solver (in our case, CVC3 [1]) as baselines for comparison;
- Empirical evaluation of solvers with manually constructed constraints and constraints generated with a concolic execution of 8 subjects.

2 Technique: Randomized Solvers

This section presents randomized solvers with common input-output interface. **Input.** All solvers take as input (i) a constraint *pc* (in reference to a *path condition* from a symbolic execution), (ii) a random seed *s*, and (iii) a range of values $[lo, hi]$. An input constraint 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 input constraint. 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 one input constraint. 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 or the flag *empty* otherwise.

Note on implementation. We wrote all solvers in the Java language, used the BCEL library [14] to instrument the bytecode of the experimental subject for concolic execution, and used part of the code from the JPF 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. Figure 1 shows the pseudo-code for a random constraint solver ranSOL. The main loop generates random input vectors 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 simplification, 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. Symbolic constraint solvers are complete for a set of decidable theories. For example, CVC3 [1] supports rational and integer linear arithmetic (among others). However, these solvers are incomplete for solving constraints with non-linear arithmetic, integer division and modulo whose theories are undecidable. We use the label **symSOL** to refer to a symbolic solver. We used CVC3 in our experiments.

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) [24]. Conceptually, these solvers attempt to optimize the random search that ranSOL drives. 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 (i.e., candidate solutions to the search problem) in the search space. The search proceeds by making movements on each individual iteratively with *search operators* until the search meets some stop criteria (e.g., the result is good enough or the search time expired). The decision to move individuals in the search space depends on the evaluation of

```
Input: path condition $pc$
Input: random seed $s$, range $[lo,hi]$
1: while $\neg timeout$ do
2:   $\overrightarrow{iv} \Leftarrow$ random(vars($pc$), range)
3:   if eval($pc$, $\overrightarrow{iv}$) then
4:     return $\overrightarrow{iv}$
5:   end if
6: end while
7: return empty
```

Figure 1: Random (ranSOL)

```
Input: path condition $pc$
Input: random seed $s$, and range $[lo,hi]$
1: ($pcgood$, $pcbad$) $\Leftarrow$ partition($pc$)
2: $\overrightarrow{iv_1} \Leftarrow$ symSOL.solve($pcgood$)
3: if ($\overrightarrow{iv_1}$ = $empty$) then
4:   return $empty$
5: end if
6: $newpc \Leftarrow pcbad \backslash \overrightarrow{iv_1}$
7: $\overrightarrow{iv_2} \Leftarrow$ ranSOL.solve($newpc$,$seed$,$range$)
8: return $\overrightarrow{iv_2} = empty$ ? $empty$ : $\overrightarrow{iv_1} + \overrightarrow{iv_2}$
```

Figure 2: Good constraints first (GCF)

```
Input: path condition $pc$
Input: random seed $s$, and range $[lo,hi]$
1: ($pcgood$, $pcbad$) $\Leftarrow$ partition($pc$)
2: $sols \Leftarrow$ eRanSOL.solve($pcbad$,$seed$,$range$)
3: for all $\overrightarrow{iv_1}$ in sols do
4:   $newpc \Leftarrow pcgood \backslash \overrightarrow{iv_1}$
5:   $\overrightarrow{iv_2} \Leftarrow$ symSOL.solve($newpc$)
6:   if $\overrightarrow{iv_2} \neq empty$ then
7:     return $\overrightarrow{iv_1} + \overrightarrow{iv_2}$
8:   end if
9: end for
10: return empty
```

Figure 3: Bad constraints first (BCF)

```
Input: path condition $pc$
Input: random seed $s$, and range $[lo,hi]$
1: ($goodvars$, $badvars$) $\Leftarrow$ partition($pc$)
2: while $\neg timeout$ do
3:   $\overrightarrow{iv_1} \Leftarrow$ random($badvars$)
4:   $newpc \Leftarrow pc \backslash \overrightarrow{iv_1}$
5:   $\overrightarrow{iv_2} \Leftarrow$ symSOL.solve($newpc$)
6:   if $\overrightarrow{iv_2} \neq empty$ then
7:     return $\overrightarrow{iv_1} + \overrightarrow{iv_2}$
8:   end if
9: end while
```

Figure 4: Bad variables first (BVF)

their current fitness values. The principle of these algorithms is that the movements across successive iterations will approximate the individuals to the solution space, i.e., each iteration potentially explores better regions in the search space. We discuss next two common aspects to GA and PSO central to our domain of application: (i) the representation of a solution (individual) and (ii) the fitness function. **Representation of a (candidate) solution.** One solution to a constraint solving problem is a mapping of variables in the constraint system to a concrete value from its domain. For instance, $\langle x \mapsto 2, y \mapsto 0 \rangle$ is a solution to $x > y + 1$ (using integer variables). **Fitness function.** The fitness function serves to evaluate the quality of **candidate** solutions. Two functions have been widely used for constraint solving problems: MaxSAT [17, 27, 33] 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. 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 harder-to-solve clauses with the increase of iterations. The solver can use this information to favor individuals (i.e., to reduce movements on those individuals) that are more fit to satisfy harder 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. Each iteration produces a new population with special operators: a *crossover* combines two individuals to produce others and a *mutation* changes one individual. The individuals are probabilistically selected considering their fitness values. 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 contribution 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.

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

Figure 5: Update of velocity and position in Particle Swarm Optimization (PSO).

The vectors v and x store respectively velocities and positions for each particle. We use the label t to refer to one iteration. This label is not the index of the vectors. The coefficient ω, typically called inertia, denotes the fraction of velocity in iteration (instant) 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 informed 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 contribution (velocity) v_{t+1}.

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, $(x > 0 \ \wedge \ x > y+1) \backslash \langle x \mapsto 2 \rangle$ is equivalent to $(2 > 0 \ \wedge \ 2 > y+1)$.

Good constraints first (GCF). Figure 2 shows the pseudo-code for the **GCF** solver. At line 1, the solver partitions the constraint pc in 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* GCF returns $\overrightarrow{iv_1} + \overrightarrow{iv_2}$ as solution, i.e., variable assignments that the symbolic and random solvers produced, respectively. For illustration, GCF 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.) GCF 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 $b \ \% \ 2 \neq 0$.

Bad constraints first (BCF). Figure 3 shows the pseudo-code for the **BCF** solver. It differs from GCF in the order of randomization: it attempts to solve the undecidable parts first. BCF 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 GCF, BCF calls symSOL once in each iteration. This algorithm corresponds to the one we discussed in Section 1.

Bad variables first (BVF). Figure 4 shows the pseudo-code for the **BVF** solver. It is similar to BCF 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 BCF and GCF randomizes all variables in this case as they appear in a clause involving non-linear arithmetic. BVF is similar to the one 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 $a = b^2 \wedge \ldots \wedge b = a^2$ illustrates one diference between BVF and DART. BVF randomizes variables b and a while DART can avoid the randomization of a as its value depends only on b's value. We did not evaluate DART itself in this paper.

3 Evaluation

We evaluate the proposed solvers with two sets of experiments. The first compares the solvers we proposed and also the symbolic solver CVC3 [1] using a set of constraints written independently by the authors. The second compares the solvers using constraints generated from the concolic execution [36] of data-structures from a variety of sources.

3.1 Microbenchmark

The microbenchmark consists of 51 satisfiable constraints. We included 15 constraints with only linear integer arithmetic, 7 using the absolute value operator (not supported natively on CVC3), 5 using modulo and division (undecidable), 22 using non-linear integer arithmetic (undecidable), and 2 using floating-point arithmetic. Except for CVC3, we run each solver 10 times with different random seeds, using the range of values [-100,100], and a timeout of 1 second. We selected these input parameters arbitrarily. The experiments show that, except for the symbolic solver CVC3, the average recall of each solver was roughly the same: minimum average recall is 0.85 for BVF and maximum average recall is 0.92 for PSO. As expected CVC3 could not solve most of the constraints in this microbenchmark. It solved 21 out of the 52 constraints. But note that it could solve some special cases of undecidable constraints (only 15 decidable constraints in the microbenchmark). For each constraints except two (one involving non-linear integer arithmetic and the other floating-point) there was a solver that can solve it.

3.2 Concolic execution

Subjects and Setup. We used data-structure from a variety of sources. *bst* (P1) is an implementation of a binary search tree from Korat [11]. *filesystem* (P2) is a simplification of the Daisy file system [32]. *treemap* (P3) is a jdk1.4 implementation (`java.util.TreeMap`) of red-black trees. *switch* (P4) refers to one example program from the jCUTE distribution [36]. *ratpoly* (P5) is an implementation of rational polynomial operations from the Randoop distribution [30]. *rationalscalar* (P6) is another implementation of rational polynomials from the ojAlgo library [3]. *newton* (P7) is an implementation of the newton's method to iteratively compute the square root of a number [2]. *hashmap* (P8) is a jdk1.4 implementation (`java.util.HashMap`) of a map that uses hash values as keys. This experiment uses a concolic (concrete and symbolic) execution [36] to generate constraints for the subject programs described above. A concolic execution interprets the program simultaneously in a concrete and symbolic domain. On the one hand, the use of a concrete state enables a concolic execution to evaluate *deterministically* any program expression. This provides a means to handle infinite loops

```
Input: parameterized test $ptest$
Input: random seed $s$, range $[lo, hi]$
1: $\overrightarrow{iv} \Leftarrow$ random(vars($ptest$), range)
2: $result \Leftarrow \{\overrightarrow{iv}\}$
3: $pcs \Leftarrow pcs$ + run($ptest$, $\overrightarrow{iv}$)
4: while size($pcs$) $> 0$ do
5:     $\overrightarrow{iv} \Leftarrow$ solve(pickOne($pcs$), $s$, $range$)
6:     if $\overrightarrow{iv} \neq empty$ then
7:         $result \Leftarrow result \cup \{\overrightarrow{iv}\}$
8:         $pcs \Leftarrow pcs$ + run($ptest$, $\overrightarrow{iv}$)
9:     end if
10: end while
11: return $result$
```

Figure 6: Concolic Execution Driver

	S1	S2	S3	**S4**	S5	S6	S7
P1	0.17	0.16	0.28	0.98	0.98	0.98	0.98
P2	0.2	0.03	0.21	1.00	1.00	1.00	1.00
P3	0.25	0.57	0.71	1.00	1.00	1.00	1.00
P4	0.48	0.25	0.48	1.00	0.04	0.25	0.15
avg.	0.28	0.25	0.42	**0.99**	0.71	0.77	0.74
P5	0.55	0.00	0.45	0.00	1.00	1.00	0.00
P6	0.92	0.02	0.98	0.00	0.00	0.92	0.18
P7	0.82	0.11	0.89	0.00	0.82	0.82	0.11
P8	0.36	0.64	0.86	0.00	0.36	0.36	0.00
avg.	0.73	0.08	**0.78**	0.00	0.50	**0.88**	0.09

Figure 7: Cell shows recall for each pair subject (row) and solver (column). S1 and S4 correspond to our baseline solvers.

and recursion, exploration of infeasible paths, and array indexing; which are typical limitations of a pure symbolic execution. On the other hand, 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 6 shows the pseudo-code of a test driver for a concolic execution. The driver takes as input (in addition to random seed and range of values) any procedure with parameters *ptest* and outputs inputs to *ptest* (that will lead execution to its different program paths). 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, will drive the next concolic execution of *ptest* (line 8). The operation solve at line 5 calls each solver with a 300 milliseconds timeout (based on average time from the microbenchmark). We set the overall timeout to 30 minutes. The concolic execution of the first four subjects only generates integer linear constraint, while the others construct non-linear constraints and unsupported constraints to CVC3.

Discussion. We use the following identifiers to label solvers: S1=ranSOL, S2=GA, S3=PSO, S4=CVC3, S5=GCF, S6=BCF and S7=BVF. Figure 7 shows a summary of the results. For the first 3 subjects the symbolic solver (S4) and consequently all hybrid solvers showed roughly the same average recall. Note that all constraints passed to the solver in this case are decidable. For switch (P4) which also builds decidable constraints, S4 timeouts often. For the last 4 subjects, S4 can rarely find a solution. In these cases, the search-based algorithms performed better on average. However, we observed that often one solver find solutions when the other misses. Figure 8 makes a pairwise comparison of the solvers. Line and row denote identifiers of solvers. A cell on line *i* and column *j* indicates that solver *i* solves a constraint that *j* misses. Note that, for the 4 experiments at the bottom and switch, the solvers vary significantly in the set of constraints they can solve. These results confirm our expectations that the solvers are complementary. 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.

Impact of timeout in recall. Efficiency is important to enable symbolic testing: the number of queries submitted to the solver can be very high. One way to deal with this issue is to reduce the alloted time for constraint solving. However timeout reduction can reduce recall. To observe the impact of timeout in recall, we run each concolic execution experiment using timeouts from 100 to 500ms. We observed that

BST							
	S1	S2	S3	S4	S5	S6	S7
S1	-	5	2	0	0	0	0
S2	4	-	4	0	0	0	0
S3	13	16	-	2	2	2	2
S4	80	81	71	-	0	0	0
S5	80	81	71	0	-	0	0
S6	80	81	71	0	0	-	0
S7	80	81	71	0	0	0	-

Summary: 99 SAT, 354 UNK.
S1:17, S2:16, S3:28, S4:97, S5:97, S6:97,S7:97

FileSystem							
	S1	S2	S3	S4	S5	S6	S7
S1	-	12	0	0	0	0	0
S2	2	-	1	0	0	0	0
S3	1	12	-	0	0	0	0
S4	49	59	48	-	0	0	0
S5	49	59	48	0	-	0	0
S6	49	59	48	0	0	-	0
S7	49	59	48	0	0	0	-

Summary: 61 SAT, 475 UNK.
S1:12, S2:2, S3:13, S4:61, S5:61, S6:61,S7:61

TreeMap							
	S1	S2	S3	S4	S5	S6	S7
S1	-	1	1	0	0	0	0
S2	22	-	11	0	0	0	0
S3	31	20	-	0	0	0	0
S4	49	28	19	-	0	0	0
S5	49	28	19	0	-	0	0
S6	49	28	19	0	0	-	0
S7	49	28	19	0	0	0	-

Summary: 65 SAT, 470 UNK.
S1:16, S2:37, S3:46, S4:65, S5:65, S6:65,S7:65

Switch							
	S1	S2	S3	S4	S5	S6	S7
S1	-	30	0	0	43	32	38
S2	8	-	8	0	20	17	19
S3	0	30	-	0	43	32	38
S4	49	71	49	-	91	71	81
S5	1	0	1	0	-	3	3
S6	10	17	10	0	23	-	10
S7	6	9	6	0	13	0	-

Summary: 95 SAT, 235 UNK.
S1:46, S2:24, S3:46, S4:95, S5:4, S6:24,S7:14

RatPoly							
	S1	S2	S3	S4	S5	S6	S7
S1	-	27	5	27	0	0	27
S2	0	-	0	0	0	0	0
S3	0	22	-	22	0	0	22
S4	0	0	0	-	0	0	0
S5	22	49	27	49	-	0	49
S6	22	49	27	49	0	-	49
S7	0	0	0	0	0	0	-

Summary: 49 SAT, 295 UNK.
S1:27, S2:0, S3:22, S4:0, S5:49, S6:49,S7:0

RationalScalar							
	S1	S2	S3	S4	S5	S6	S7
S1	-	56	1	57	57	0	46
S2	0	-	0	1	1	0	0
S3	5	60	-	61	61	5	50
S4	0	0	0	-	0	0	0
S5	0	0	0	0	-	0	0
S6	0	56	1	57	57	-	46
S7	0	10	0	11	11	0	-

Summary: 62 SAT, 296 UNK.
S1:57, S2:1, S3:61, S4:0, S5:0, S6:57,S7:11

Newton							
	S1	S2	S3	S4	S5	S6	S7
S1	-	20	3	23	0	0	20
S2	0	-	0	3	0	0	0
S3	5	22	-	25	5	5	22
S4	0	0	0	-	0	0	0
S5	0	20	3	23	-	0	20
S6	0	20	3	23	0	-	20
S7	0	0	0	3	0	0	-

Summary: 28 SAT, 305 UNK.
S1:23, S2:3, S3:25, S4:0, S5:23, S6:23,S7:3

HashMap							
	S1	S2	S3	S4	S5	S6	S7
S1	-	0	0	5	0	0	5
S2	4	-	2	9	4	4	9
S3	7	5	-	12	7	7	12
S4	0	0	0	-	0	0	0
S5	0	0	0	5	-	0	5
S6	0	0	0	5	0	-	5
S7	0	0	0	0	0	0	-

Summary: 14 SAT, 379 UNK.
S1:5, S2:9, S3:12, S4:0, S5:5, S6:5,S7:0

Figure 8: Results of various solvers for constraints that concolic execution generates. 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.

CVC3 typically finds a solution for decidable constraints in less than 100ms. However, for the Switch experiment the recall of CVC3 was 0.25 for 100ms, 0.75 for 200ms and 0.95 from 300 to 500ms. We could not determine the reason for this. In particular, we did not find a strong correlation between the size of the constraints or the number of variables in it and higher impact of time. We also observed a significant variation in recall on PSO for Newton and HashMap and on GA for TreeMap and HashMap. For these cases, we conjecture that the impact of time relates to the complexity of the search problem, i.e., the relative small size of the solution space compared to that of the search space.

4 Related Work

Random-symbolic testing has been widely investigated recently to automate test input generation [22, 26, 21]. 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, 31, 29]. 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) [37, 20, 12] solver CVC3 [1], 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 [15] 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 to any theory-specific solver to fully 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 [34]. 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 [38].

5 Conclusions

This paper proposes and implements a plain random solver, three hybrid solvers combining random and symbolic solvers, and two heuristic search solvers. We use a random solver and a symbolic solver (CVC3) as baselines for comparison. We evaluate the solvers on two benchmarks. One with constraints the authors constructed and the other with constraints that a concolic execution generates on 8 subjects. For the concolic execution on subjects that generated only decidable constraints the the experiments reveal 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.

Acknowledgments. We are grateful to Augusto Sampaio, 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] CVC3 webpage. `http://www.cs.nyu.edu/acsys/cvc3/`.

[2] Newton's square root webpage. `https://trac.videolan.org/skin-designer/browser/trunk/src`.

[3] ojAlgo webpage. `http://ojalgo.org`.

[4] Ashraf Abdelbar and Suzan Abdelshahid. Instinct-based PSO with local search applied to satisfiability. In *IEEE International Joint Conference on Neural Networks*, pages 2291–2295, July 2004.

[5] Saswat Anand, Corina S. Pasareanu, and Willem Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *TACAS*, pages 134–138, 2007.

[6] Thomas Bäck, A. E. Eiben, and Marco E. Vink. A superior evolutionary algorithm for 3-sat. In *7th Conference on Evolutionary Programming VII*, pages 125–136, UK, 1998. Springer-Verlag.

[7] Andreas Bauer, Markus Pister, and Michael Tautschnig. Tool-support for the analysis of hybrid systems and models. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 924–929, 2007.

[8] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[9] F. Benhamou and L. Granvilliers. Continuous and interval constraints. In *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 16. Elsevier, 2006.

[10] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 23(3):228–245, 1983.

[11] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, 2002.

[12] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisf. of linear arith. logic. In *TACAS*, pages 317–333, 2005.

[13] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Soft. Practice and Experience*, 34:1025–1050, 2004.

[14] Markus Dahm and Jason van Zyl. Byte Code Engineering Library, April 2003. `http://jakarta.apache.org/bcel/`.

[15] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of ACM*, 5(7):394–397, 1962.

[16] A.E. Eiben and J.K. van der Hauw. Solving 3-sat by gas adapting constraint weights. *Evolutionary Computation, 1997., IEEE International Conference on*, pages 81–86, Apr 1997.

[17] G. Folino, C. Pizzuti, and O. Spezzano. Combining cellular genetic algorithms and local search for solving satisfiability problems. In *IEEE Conference on Tools with Artificial Intelligence*, pages 192–198, 1998.

[18] J. Forrester and B. Miller. An empirical study of the robustness of windows NT applications using random testing. In *USENIX Windows Systems Symposium*, pages 59–68, 2000.

[19] M. Franzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. 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] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Computer aided verification*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[21] Patrice Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Programming Language Design and Implementation*, volume 40, pages 213–223, 2005.

[23] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[24] J. Kennedy and R. Eberhart. Particle swarm optimization. In *IEEE Neural Networks*, pages 1942–1948, 1995.

[25] James C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7):385–394, 1976.

[26] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE*, pages 416–426, 2007.

[27] E. Marchiori and C. Rossi. A flipping genetic algorithm for hard 3-SAT problems. In *Genetic and Evolutionary Computation Conference*, volume 1, pages 393–400, Orlando, Florida, USA, 1999. Morgan Kaufmann.

[28] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, May 2002.

[29] C. Pacheco and M. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA Companion*, pages 815–816, 2007.

[30] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

[31] Carlos Pacheco and Michael D. Ernst. Eclat documents. Online manual, Oct. 2004. `http://people.csail.mit.edu/people/cpacheco/eclat/`.

[32] Shaz Qadeer. Daisy File System. Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software. 2004.

[33] Claudio Rossi, Elena Marchiori, and Joost N. Kok. An adaptive evolutionary algorithm for the satisfiability problem. In *SAC (1)*, pages 463–469, 2000.

[34] Nie Ru and Yue Jianhua. A GA and particle swarm optimization based hybrid algorithm. In *IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008.

[35] P. Santhanam and B. Hailpern. Software debugging, testing, and verification. *IBM Systems Journal*, 41:4–12, 2002.

[36] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

[37] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proceedings of the European Conference on Logics in Artificial Intelligence*, pages 308–319, London, UK, 2002. Springer-Verlag.

[38] Andreas Windisch, Stefan Wappler, and Joachim Wegener. Applying particle swarm optimization to software testing. In Hod Lipson, editor, *Genetic and Evolutionary Computation Conference*, pages 1121–1128, 2007.