Symbolic Execution with Interval Solving and Meta-heuristic Search

[†]Mateus Borges, [†]Marcelo d'Amorim, [♯]Saswat Anand, [‡]David Bushnell, and [‡]Corina S. Păsăreanu
[†]Federal University of Pernambuco, Recife, PE, Brazil
[♯]Georgia Institute of Technology, Atlanta, GA, USA
[‡]SGT and CMU, NASA Ames Research Center, Moffett Field, CA, USA
[†]{mab,damorim}@cin.ufpe.br, [♯]saswat@gatech.edu, [‡]{David.H.Bushnell,corina.s.pasareanu}@nasa.gov

***Abstract*—An important problem in symbolic execution is dealing with complex mathematical constraints; for instance, constraints that include floating-point variables and transcendental functions. We recently showed that the use of meta-heuristic search is effective in solving some of these constraints. This paper extends that work with a constraint solving method that combines meta-heuristic search with a traditional constraint solving technique, namely interval-based solving. We implemented this combination in the context of the CORAL constraint-solving infrastructure and evaluated its effect on publicly available subjects from the aerospace domain. Results indicate that the proposed method can solve significantly more complex mathematical constraints than previous techniques, thus broadening the application of symbolic execution in practice.**

***Keywords*- testing; symbolic execution; constraint solving.**

I. Introduction

Symbolic execution [1] is a technique for systematic test-input generation that has gained significant attention in recent years. The technique takes as input a parameterized procedure P and generates values for P's parameters that ensure high structural coverage of P. Internally, symbolic execution has two components: (i) path condition generation and (ii) path-condition solving. A path condition is a symbolic boolean expression that encodes the conditions on the inputs to follow one particular path through the program. Path condition solving is used to determine (in)feasibility of program paths and to generate the actual test inputs. Thus, the effectiveness of symbolic execution in generating test-inputs depends on whether solutions to satisfiable path conditions can be found.

A major challenge in symbolic execution is dealing with path-conditions that manipulate constraints from complex theories, say undecidable or intractable theories. In previous work we proposed the constraint solver CORAL [2], [3] for heuristically solving complex mathematical constraints. CORAL reduces the task of solving a conjunction of numeric constraints to a search problem. It uses meta-heuristic search [4], a method that explores the problem search space by applying successive refinements to a set of *candidate solutions* (also called population) based on some measure of quality. In the context of solving path conditions, a candidate solution is an assignment of concrete values to symbolic input variables, and quality is measured in terms of values of a *fitness function* that estimates the proximity of a candidate to a solution of the path condition. The search starts with an *initial population*, typically selected at random and stops when it exceeds resource limits (e.g., time or number of iterations). While the meta-heuristic search approach has been shown effective in practice [3], it is inherently *incomplete*, meaning that it may fail to find a solution even when one exists.

To alleviate this problem, we propose to integrate the meta-heuristic search with a well-known technique for constraint solving, namely interval-based solving. In particular, we propose to use an interval solver to *seed* the population of the meta-heuristic search with candidate solutions drawn from the intervals reported on a given solve request. The goal is to increase the chances of finding a solution within the given resource constraints. Interval solvers take as input a list of equality and inequality numeric constraints involving n variables and report a list of n-dimensional *boxes* on output. A box is an assignment of *intervals* to symbolic variables that do *not* necessarily contain solutions to the constraint problem. Our approach consists of two steps: (1) an interval solver is used to generate a list of intervals that may contain solutions of a given path condition and (2) a meta-heuristic search is used to solve the path condition. However, unlike in our prior work, where the initial population was generated randomly, in our new approach, the initial population is drawn from the intervals reported at step 1.

Our combined approach leverages the power of interval solvers and meta-heuristic search techniques to overcome their individual limitations. Interval solvers can efficiently compute parts of the problem's search space that are likely to contain solutions. However, interval solvers cannot typically generate exact solutions, but our combined technique can. On the other hand, a meta-heuristic search technique can find exact solutions in large search spaces. However, its efficiency depends on the quality of its initial population. Our conjecture (supported by experiments) is that actual solutions are in close proximity to the intervals reported by the interval solver; hence we use these intervals to improve the quality of the initial population.

We have developed a prototype implementation of our proposed approach by integrating the RealPaver interval solver [5] into the CORAL infrastructure. CORAL supports two meta-heuristic search techniques: (1) Particle Swarm Optimization (PSO) [6], which is a global search method, and (2) Alternating Variable Method (AVM) [7], which is a local search

method. The AVM technique was added to CORAL as part of the work reported here. We evaluate the approach on programs from the aerospace domain, that use complex mathematical functions, non-linear operations, and floating-point input values. We use Symbolic Pathfinder [8] to symbolically execute the programs and generate path conditions. Our experiments show that the proposed combination of techniques can solve more constraints than the techniques used separately.

This paper makes the following contributions.

- A novel approach to solve path conditions generated from the symbolic execution of programs that use mathematical functions, non-linear operations, and floating-point input values. The approach leverages the power of interval constraint solvers and meta-heuristic search techniques to overcome their individual limitations.
- An open-source implementation of the proposed approach. The implementation is publicly available[1] and it has been integrated with the Symbolic Pathfinder [8] symbolic execution system.
- We evaluated our proposal using subjects from the aerospace domain and compared the proposed approach with our previous version of CORAL. We used the output of the RealPaver interval solver to seed a global search (PSO) and a local search method (AVM), which was newly incorporated in CORAL. Results indicate that both proposed combinations lead to a significant increase in the number of constraint solved.

II. Background

This section provides background information for the rest of the paper.

A. Symbolic Execution

1) Path-Condition Generation: Symbolic execution is a program analysis technique that executes a program with symbolic values instead of concrete inputs. It computes the effect of program execution on symbolic states, which map variables to symbolic expressions. When the execution evaluates a branching instruction, it needs to decide which branching choice to select. In a regular execution the evaluation of a boolean expression is either true or false so only one branch of the conditional can be taken. In contrast, in symbolic execution the evaluation of the boolean expression is a symbolic value, so both branches can be taken resulting in different paths explored through the program. Symbolic execution characterizes each path it explores with a *path condition* over the input variables $\overrightarrow{x}$. This condition is defined with a conjunction of boolean expressions $pc(\overrightarrow{x}) = \bigwedge b_i$. Each boolean expression b_i denotes a branching decision made during the execution of a distinct path in the program under test. Symbolic execution terminates when it explores all such paths corresponding to the different combinations of decisions. Programs with loops and recursion may result in an infinite number of paths; in those cases, one needs to put a bound on the number of paths that can be explored with symbolic execution.

[1] pan.cin.ufpe.br/coral

```
if (Math.log(in) > 4.0) do1();
else do2();
```

1.	$log(in_SYM) < CONST_4.0$
2.	$log(in_SYM) == CONST_4.0$
3.	$log(in_SYM) > CONST_4.0$

Fig. 1. Example with Math function and corresponding path conditions.

2) Constraint Solving: Symbolic execution uses constraint solving in two contexts: (i) to check path feasibility and (ii) to generate test inputs. In the first context, symbolic execution checks if the current path is feasible by checking if the corresponding constraint is satisfiable. If the path condition becomes unsatisfiable, symbolic execution does not continue for that path. In the second context, symbolic execution uses a constraint solver to solve constraints associated with complete paths; solutions can be used as inputs to test the program.

3) Symbolic PathFinder (SPF): Symbolic PathFinder (SPF) is a symbolic execution tool for Java bytecodes. SPF is part of the Java PathFinder (JPF) verification tool-set [9], a freely available open-source project. SPF has been used at NASA, in industry, and in various research projects from academia. The symbolic execution of SPF interprets Math functions on the abstract "model" level: whenever the symbolic execution reaches a call to a complex Math function, SPF intercepts that call and builds a new symbolic expression using a symbolic operator, i.e. an uninterpreted function, associated with that function. The path conditions containing such expressions are dispatched to an appropriate constraint solver that can handle complex Math constraints, such as CORAL. SPF uses uninterpreted functions for the following methods from the Java Math library: `acos`, `asin`, `atan`, `atan2`, `cos`, `exp`, `log`, `pow`, `round`, `sin`, `sqrt`, and `tan`. For all the other Math methods, which are much simpler, it provides implementations that are directly interpreted by SPF.

4) Example: Figure 1 shows one example that uses the `log` Math function. Variable `in` stores the symbolic input in_SYM. The symbolic execution of this code produces the three path conditions that appear at the bottom of the figure. As mentioned before SPF does not directly interpret the call to the standard Java library function `Math.log`. Instead, it constructs a symbolic expression `log(in_SYM)` which is then used to build the symbolic constraints. When executing the `if` statement above, SPF creates a 3-choice split point related to the outcomes of the relational expression. Each execution will explore one choice. As execution goes along, more boolean expression are added to the current path, building longer path constraints. The constraints are solved with an appropriate constraint solver; i.e., one that can handle such complex mathematical functions directly.

B. Interval-based solvers

Interval solvers take as input a list of equality and inequality numeric constraints involving n input variables and report as

output a list of n-dimensional *boxes*. A box is a characterization of a subset of the cartesian product of variable domains. For example, for the constraint:

$$(1.5 - (x1 * (1.0 - x2))) = 0.0 \quad (1)$$

RealPaver reports the following box:

```
x in [99.99925650834012, 100]
y in [0.9849998884754217, 0.9850000000000001]
```

The error bound associated with this box was set to 3 decimal digits. The user-specified error bound parameter controls the precision of the result and hence the cost of the search. It is import to remark that one interval solver does *not* respond the question on how to obtain solutions from a box, but the box above contains the following solution to the input constraint:

```
x = 99.99999999999991, y = 0.985
```

1) Branch and Prune: The boxes reported as solutions to the constraint system are obtained from the initial domain of variables (user specified or default) using a branch-and-prune search algorithm [10]. The algorithm starts with one "large" n-dimensional box, corresponding to the domains of the n input variables, and it iteratively performs *branch* and *prune* steps. The branch step divides a large box into smaller ones and the prune step removes inconsistent regions from one box, i.e., regions where all the points violate at least one part of the constraint. Different interval solver implementations are distinguished by the way they implement these two steps. For instance, the pruning step of the Realpaver solver [5] uses advanced constraint-satisfaction techniques [11].

2) Inner and Outer boxes: Interval solvers that use the branch-and-prune approach report *inner* and *outer* boxes [5, § 2, par. 2]. An inner box is guaranteed to contain solutions, while an outer box *may or may not* contain solutions. The solver output is a list of such boxes whose union denotes the solution set. The problem is inconsistent (the input constraint unsatisfiable) if the solver reports no box [5, § 3.2]. The approach does not guarantee which kind of boxes is reported first. RealPaver classified the box above as an outer box, but the box indeed includes a solution in this case.

C. CORAL *solvers*

CORAL is an infrastructure for constraint solving with support for constraint simplification and (simple) inference of variable domains.

1) Meta-heuristic Search: We considered in this work both global and local methods of meta-heuristic search; these methods vary in the scope of the search. The principle is to evaluate how far distant from the boxes that the interval solver reports the search could find solutions. We used PSO [6] for global search and AVM [7] for local search.

PSO, similar to Genetic Algorithms [4], uses an evolutionary approach to search: the population evolves during the search according to some user-defined principle. Each evolution step approximates the candidates to a solution (or local maximum). The search starts with an *initial population*, typically selected at random, and stops when it finds a solution or exhausts resources; say it reaches a timeout or maximum number of iterations.

$$f(\overrightarrow{x}) = \sum_i w_i * g_i(\overrightarrow{x})$$
$$g_i(\overrightarrow{x}) = \max_{1<j<m} 1 - d(b_{ij}, \overrightarrow{x})$$

Fig. 2. Fitness function of CORAL.

AVM is essentially an adaptation of Hill Climbing [4]. It starts the search with *one* vector of assignments to input variables and alternates across different variables during the search. In each step it makes small positive and negative increments to the values associated to the selected variable and re-evaluates fitness to decide whether to go up or down hill. As the mutation is fine-grained, AVM often incorporates random-restarts to escape local maxima.

2) Fitness functions: The role of a fitness ("objective") function is to drive the search towards (fitter) solutions. This function gives a score denoting the quality of an input candidate for solving the problem. CORAL solvers use a variation of the Stepwise Adaptive Weighting (SAW) fitness function [12] that dynamically adjusts the importance of different sub-problems for solving the whole problem. For constraint solving, the problem is to solve the entire path condition $pc(\overrightarrow{x}) = \bigwedge b_i$ and the sub-problems are to solve the clauses $b_i = b_{i1} \vee \cdots \vee b_{im}$ of the input path condition.

Figure 2 shows the fitness function we used. Function f is the weighted sum of $g_i(\overrightarrow{x})$, which denotes the score of candidate $\overrightarrow{x}$ to solve the clause b_i of the path condition. Conceptually, g measures how close a clause b_i is from satisfaction. Function d measures distance and is defined elsewhere [3]. The distance score is given in the continuous interval $[0.0, 1.0]$ with higher values indicating better fitness and lower values indicating worse fitness. The search goal is to maximize the function f, i.e., to find inputs that produce maximal outcomes: high valuations of inputs on this function indicate fitter candidates. The search procedure dynamically increases the weight w_i associated to each clause b_i as that clause remains unsolved for longer than some specified number of times. The use of weights helps the search to positively differentiate candidate solutions that satisfy "difficult" clauses from solutions that satisfy many "easy" clauses. Note that a final solution is only relevant if it satisfies all clauses b_i.

III. Examples

In this section we give examples that illustrate the benefits of combining interval solving with meta-heuristic search for solving complex constraints, see Figure 3.

A. Improving Interval Solving

Consider the constraint 1 from Figure 3. Even though the expression $sin(x1) - cos(x2)$ evaluates to a value very close to 0.0 for any assignment of $x1$ and $x2$ within the box that RealPaver reports for constraint 1 (see row "interval"), a floating-point solution does not appear to exist there. To

1	**Constraint:** $(sin(x1) - cos(x2)) = 0.0 \wedge x1 >= -1 \wedge x2 <= 1$ **Intervals:** x1=[0.9640858380445663, 0.9646745947979339], x2=[0.6061217319969636, 0.6067104887503316] **Solution:** x1=0.5734475041703869, x2=-0.9973488226245096
2	**Constraint:** $0.0 == pow(((x1 * sin(((c1 * x2) - (c1 * x3)))) - (0.0 * x4)), 2.0) + pow((x1 * cos((((c1 * x2) - (c1 * x3)) + 0.0))), 2.0)$ & $x5 \neq 0$ & $c1 = 0.017453292519943295$ **Intervals:** x1=[-0.0,-0.0], x2=[33.333...,100.0], x3=[33.333...,100.0], x4=[-100.0,100.0] **Solution:** x1=-0.0, x2=40.76274086185327, x3=95.33728666158905, x4=92.99285811089572, x5=51
3	**Constraint:** $sqrt(pow(((x1 + (e1 * (cos(x4) - cos((x4 + (((1.0 * (((c1 * x5) * (e2/c2))/x6)) * x2)/e1)))))) - (((e2/c2)) * (1.0 - cos((c1 * x5))))), 2.0)) > 999.0$ & $(c1 * x5) > 0.0$ & $x3 > 0.0$ & $x6 > 0.0$ & $c1 = 0.017453292519943295$ &$c2 = 68443.0$ & $e1 = ((pow(x2, 2.0)/tan((c1 * x3)))/c2)$ & $e2 = pow(x6, 2.0)/tan((c1 * x3))$ **Intervals:** x1=[99.99962366471537,100.0], x2=[99.99962366471537,100.0], x3=[89.99999999999997,90.00000000000003], x4=[99.99962366471537,100.0], x5=[99.99943549707307,100.0], x6=[99.99943549707307,100.0] **Solution:** x1=100.0, x2=98.4818097287292, x3=3.0825564323322396E-11, x4=83.0313044811115, x5=73.32021467962014, x6=41.927226898838214

Fig. 3. Meta-heuristic solving can solve constraint 1 but not constraint 2. Interval solving (using random search on reported intervals) can solve constraint 2 but not constraint 1. Only the combination can solve constraint 3.

confirm this we increased the precision of RealPaver to 16 digits and obtained the box below, which admits only 4 pairs of concrete assignments. Fitness values is very close to 100% but still not a hit. Increasing the precision bound does not change results. Note that the interval boundaries are characterized with floating-point numbers and their precisions are at the limit.

```
x1 in [0.9642330272329055, 0.9642330272329056]
x2 in [0.6065632995619922, 0.6065632995619923]
```

We evaluated the constraint 1 in Java with each of the 4 possible assignments included in the box above and could not find a floating-point solution. In contrast, CORAL (with or without seeding) finds one solution. Important to note that RealPaver reports many other boxes, we only inspected the first box reported.

B. Improving Meta-heuristic Search

Consider the constraint 2 from Figure 3. To make the original constraint shorter (for presentation) we replaced multiple occurrences of the constant 0.017453292519943295 with uses of the (new) variable $c1$ that denotes this constant. CORAL alone could not find a solution to this constraint; recall that the approach of meta-heuristic solving is fundamentally incomplete. The fitness function of CORAL did not help much to guide the search in this case mainly because the constraint contains only two conjuncts (ignoring the clause with the assignment to $c1$). As a result, the search got stuck at a local maximum with a fitness value close to 100%.

The figure also shows the intervals that RealPaver reports for this constraint. Note that it only reports intervals for the first four variables. This is because it does not support "!=", negation, or disjunction; hence we make a request for RealPaver to solve a simplified version of the original constraint with the conjunct $x5 \neq 0$ removed. Note that the interval for variable $x1$ admits only one value: 0.0. For this case, the assignment $x1 = 0$ makes the search for solutions trivial: the assignment to all other variables becomes unconstrained. Any random assignment of values to variables across these intervals will satisfy the input constraint.

C. Improving Both Interval Solving and Meta-heuristic Search

The two previous examples highlight limitations of the individual techniques we use: interval solving and meta-heuristic search. The first example constraint was a bad match to RealPaver but a good match to CORAL. In contrast, the second example constraint was a good match to RealPaver but a bad match to CORAL. In both of these cases, one solver was able to overcome the limitation of the other solver. In practice, we observed more interesting scenarios where the interaction between the two approaches – interval solving and meta-heuristic search – was beneficial to both. Consider the constraint 3 from Figure 3. This constraint is a fragment of a path condition generated with the symbolic execution of the "TSAFE:Conflict Probe" subject (see Section V). As before we use additional symbols to denote constants. The constraint could not be solved either with PSO alone or with a Random search on the intervals reported by RealPaver. A random search would likely be able to find solutions only if the search space associated with reported intervals were small (not the case here). Our proposed combination was able to find a solution to this constraint. Note that the combined solver has found a solution that is close to the intervals for the variables $x1$, $x2$, and $x4$ but not so for the remaining variables.

IV. Approach

We now describe our approach to combine interval solving with meta-heuristic search. Let us first remark that, when using interval solving alone, one may fail to find a solution within one box for the following reasons:

- A floating-point solution does not exist in the interval. The solver operates on reals and may report intervals that do not admit a floating-point solution (see Section III);
- A real solution does not exist in the interval. It is possible that the box reported by the interval solver does not contain a solution.

The (inn)accuracy of the box that the interval solver reports is determined by the following factors:

- User-specified precision;

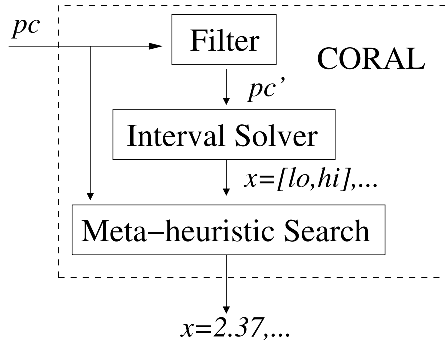

Fig. 4. Organization of CORAL with interval solver.

- Removed clauses from the input constraints containing unsupported operators. Reported boxes therefore over-approximate the solution space of the original problem;
- The effect of the locality problem [5, § 2.5, Figure 3]. This scenario can manifest itself with specific input constraints. The effect on the precision of the reported boxes depends on the supporting algorithms used to deal with it, which are often expensive.

Despite all these practical limitations, the boxes reported could still help CORAL to guide the search for solutions: it is possible that solutions do exist within the reported box or solutions exist close to the box. Our approach builds on this conjecture to improve constraint solving for symbolic execution, where exact solutions are important.

A. Combining interval solving with meta-heuristic search

Figure 4 shows the organization of CORAL as a pipeline with three distinct processing steps that we describe below.

- In the first step, a filter component derives a path condition pc' with some of the conjuncts removed from constraint pc. We note that CORAL applies some simplifications to the input constraint, e.g., it attempts to remove variables whose values fully depend on the solution of other variables. Therefore, the constraint pc in Figure 4, has already been simplified by CORAL. The filter eliminates clauses that contain operations not handled by the underlying constraint solver. For example, RealPaver does not support type casting expressions (e.g., asint(x), where x is a symbolic variable of type double), negation, difference, and disjunction. We note that SPF can avoid the addition of negated constraints to path conditions, by introducing extra non-determinism in the path exploration. In result of this filtering, a solution box reported for pc' could, in principle, include a concrete solution to pc' that is not a solution to pc.
- In the second step, the filter passes the modified path condition pc' to the interval solver for processing. The interval solver forwards to CORAL the first solution box reported within a user-specified timeout. If it cannot find one, CORAL behaves as usual.
- Finally, the third step in the figure is responsible for invoking our search infrastructure to solve the constraint.

We made two changes in CORAL to enable this integration. We made CORAL use the intervals reported by the interval solver instead of the intervals it infers for particular cases. For example, original CORAL can infer intervals from expressions of the form $x \leq 3.1415..$ and from the domains of math functions. We also made CORAL check if the interval associated with a variable admits only one value (i.e., $[c, c]$). In that case, the constraint passed to CORAL is simplified by replacing the variable with its value.

In summary, the constraints that the interval solver and CORAL operate on are similar but not necessarily the same. A concrete solution to pc' may not solve pc. However, the meta-heuristic searches (both PSO and AVM) are not confined to the box reported by the interval solver. Our hypothesis, substantiated with experimental results, is that CORAL can often find concrete solutions to the input path condition (pc) close to the box but not necessarily within it; the solution box can therefore improve CORAL's effectiveness.

B. Implementation

For our implementation, we evaluated two interval solvers: RealPaver [5] and ICOS [13]. To support our decision of which one to select, we considered the robustness of the tools (i.e. whether they crash), the supported operators that are relevant to our benchmarks, and the quality of results, including time efficiency and precision of results. RealPaver performed better overall for the experiments we considered. For this reason we decided to report the results for this solver. We encapsulated RealPaver inside CORAL; the interaction between these solvers is as defined in Figure 4. In each call, the combined solver translates the input format of CORAL to the input format of RealPaver, AMPL (A Mathematical Programming Language) [14]. We used the default configurations of RealPaver except that we set the precision of error bounds to 3 decimal places, the timeout to 2s, and also the domain of variables to $[-100, 100]$ (this was necessary to force RealPaver report useful first boxes). This integration is part of the recent release of CORAL, which is publicly available.

V. Evaluation

This section presents our experimental results. In Section V-A we first outline the research questions that we want to answer with the experiments. In Section V-B we describe the subject programs that we analyzed with symbolic execution, and in Section V-C we describe our experimental setup. The remaining sections present and discuss our results.

A. Research questions

The list below shows the main research questions (*RQ*) that we want to answer with our experiments:

RQ-1. Is it possible that intervals reported by the subordinate interval solver are already tight (i.e., the difference between the limits of each interval is very small)? If the answer is positive, any search based on these intervals would perform equally well.

RQ-2. How effective are the proposed combined solvers for dealing with constraints that could not be handled by either original CORAL or by a random search over the intervals that RealPaver reports?

RQ-3. Is it possible that the result of a combined solver outperforms original CORAL due to coincidence? Is it possible that a positive result for a combination solver is the effect of an (un)fortunate selection of random seeds.

B. Subject Programs

The list below describes the subjects we used.

- **Apollo.** The Apollo Lunar Autopilot is a Simulink model that was automatically translated to Java using the Vanderbilt tool-set [15]. This 2.6KLOC subject is deployed in a single package with 54 classes. (Numbers computed with the JavaNCSS tool [16].) The Simulink model was created by one of the engineers who worked on the Apollo Lunar Module digital autopilot design team to see how he would have done it using Simulink if it had been available in 1961. The model is available from MathWorks[2]. It contains both Simulink blocks and Stateflow diagrams and makes use of complex Math functions (e.g. `Math.sqrt`).
- **Collision Detector (CDx).** The CDx system is a discrete-time simulator for collision detection of aircraft in flight [17]. The input is an array of pairs (p_i,tr_i), where p_i describes the position of aircraft i and tr_i its trajectory as function of time, (e.g., $2 \times cos(t)$). The output is a report describing whether or not a collision was detected. At every simulation step of a regular execution (simulating a progression of radar frames), the simulator updates the position of every aircraft according to the current time and trajectory functions, checks for aircraft collisions, and then increments the clock of the simulation. We modified the code to take the simulated variable time as a parameter. This enabled our test drivers to pass a symbolic variable for the time. Symbolic execution conceptually makes jumps along the simulation timeline corresponding to the branching choices it makes during the state-space exploration. It produces path conditions that manipulate the time variables and include the complex math functions that arise from the movements of aircraft across the simulation space. Path conditions for this subject include the math functions `sin`, `cos`, and `pow`.
- **TSAFE.** Because air traffic in the United States is expected to grow dramatically in the coming decades, NASA and the FAA are developing a new, more automated air traffic control system called NextGen. NextGen has several components, including one called TSAFE (Tactical Separation Assisted Flight Environment), which is designed to prevent near misses and collisions that are predicted to happen in the near future (within thirty seconds to three minutes). TSAFE is still under development by NASA and the FAA, but it includes algorithms for separation assurance during level flight, ascent and descent, and in terminal airspace surrounding airports. The **Conflict Probe** module of TSAFE tests for conflicts between a pair of aircraft within the TSAFE time horizon. The two aircraft may either be flying level or engaged in turns of constant radius. The Conflict Probe module is self contained, but it uses transcendental functions and contains loops, so it represents a significant floating point calculation. The following math functions appear in the path conditions of this subject: `cos`, `pow`, `sin`, `sqrt`, and `tan`. The **Turn Logic** module of TSAFE computes the change in heading required once an impending loss of separation between two aircraft is detected. It assumes a constant turning radius for the aircraft making the maneuver. The following math functions appear in the path conditions of this subject: `atan2`.

[2] http://www.mathworks.com/products/simulink/demos.html

C. Experimental Setup

Constraints considered. The set of constraints we analyzed originate from symbolic executions obtained with Symbolic Pathfinder. For large programs such as Apollo and CDx we symbolically execute the program once for each solver and collect 100 constraints that the solver can solve and 100 constraints it cannot. We selected the longer constraints generated (denoting deeper paths). Then, we compare the solvers with respect to their capability of solving the constraints from the set containing all constraints combined. This approach enables each solver to try to solve constraints generated with the symbolic execution of another solver. For smaller subjects, we run symbolic execution with a wrapper solver that submits the input constraint to each solver that we want to compare and counts the difference between the number of cases one solver can solve and the other cannot. We use Venn-diagrams to highlight the distinct sets of constraints that each solver can solve (in response to RQ-2).

Solvers considered. We used RealPaver [5] as our interval constraint solver, as discussed on sections II-B and IV-B. We describe below the solvers that we used as baselines for comparison:

- **pso** is the solver that uses CORAL with Particle Swarm Optimization search (PSO) [6]. We previously found [3] that CORAL with PSO search performed the best compared to other search strategies such as random search and genetic algorithms. *This is the baseline solver for CORAL.*
- **rp+ran** is the combination that uses interval solving combined with random search. At each iteration, random search chooses one random input assignment from the intervals and computes the fitness values associated with it. *This is the baseline solver for RealPaver*. The principle is that random search should find solutions in case RealPaver reported boxes with solutions and the intervals are very tight (see Section IV). This solver therefore serves to evaluate the "tight intervals" effect (in response to RQ-1).

We describe below the combined solvers that we propose:

- **rp+pso** is the combination that uses interval solving combined with PSO search. It uses the intervals from the first reported box from RealPaver to seed the initial population

Fig. 5. Differences of constraints solved by different solvers for larger subjects: 5(a) Apollo (417 in total) and for 5(b) CDx (452 in total).

of the PSO search. The search is not constrained within the box used.

- **rp+avm** is the combination that uses interval solving combined with AVM search [7]. In contrast to PSO, which is a population-based search, AVM keeps only one candidate assignment and explores the search space close to that assignment, making incremental changes to the candidate. We translated to Java the C# publicly-available AVM implementation used in `FloPSy` [18]. We used the intervals to reset the search at random restarts of the algorithms, which frequently occur and serve to workaround local maximums.

PSO, as a global search, looks for solutions across the entire state space. If the use of one box does not result in good fitness the search can try longer jumps using the many candidate solutions in the search population. AVM, as a local search, looks for solutions close to the boxes. It performs incremental and systematic changes to the one solution candidate it uses across the search. At random restarts the search makes jumps where the ending point of the jump is within the reference box. During each climbing phase (between random restarts), AVM makes a bounded-depth search rooted at some point within the box. Therefore, the search effectively takes place in close proximity to the box. Conceptually, the local and global modes of exploration speculate optimistically (local) and not so optimistically (global) about the quality of the boxes used. We want to evaluate how they compare. Both approaches have been incorporated in the `CORAL` infrastructure.

D. Differences between solvers

This section presents the results on the differences between our solvers. We show and discuss the differences between the sets of constraints each solver is able to solve. We use Venn diagrams to show this difference; each set in the diagram corresponds to the constraints each solver can solve. Figures 5 and 6 present the diagrams for each subject. Each diagram shows, in the top right corner, the total number of constraints solved and, close to the name of the solver and inside parentheses, the total number of constraints that particular solver can solve for that subject case. We refer to a *distinct solution* as a solution that only one solver can find. Finally, we discuss results for each subject.

1) Apollo: Figure 5(a) shows a Venn diagram comparing the effectiveness of each solver in solving the path conditions from the Apollo subject. We make the following observations:

- Solvers pso and rp+ran solved a similar number of constraints: 102 and 111, respectively. Note, however, that the intersection across their corresponding sets is small: rp+ran cannot solve most of the constraints that pso can, and conversely.
- The rp+pso solver outperforms pso (baseline for `CORAL`) solver: rp+pso missed only 3 constraints from the pso set and found solutions for many constraints that pso missed. On the other hand, rp+pso missed a total of 56 constraints that rp+ran (baseline for RealPaver) found and 66 constraint that rp+avm found.
- The sets of constraints that rp+ran and rp+avm distinctly solve are disjoint. We inspected these cases and observed that rp+avm consistently increments fitness in very small amounts, but it isn't able to make bigger moves without losing fitness values: since fitness increases consistently (at a very small rate) random restarts rarely occur.
- As expected, solvers are complementary. Solver pso did not show significant distinction: it found only 1 constraint that no other solver could find. For this subject, rp+pso found many more solutions than any other solver (292, of which 108 are distinct). However, both rp+ran and rp+avm found a significant number of distinct solutions (56 and 66, respectively).

2) Collision Detector (CDx): Figure 5(b) summarizes results for CDx. We make the following observations:

- The rp+avm solver found a very significant portion of the total number of constraints solved for this subject (86.3%=390/452). Also, the rp+avm solver found *all* solutions that rp+pso found. One of the reasons for this superiority is that while our PSO implementation sets a minimum and maximum limit for the domain of each variable, AVM has no such limitation. PSO needs to limit the scope of its global search so to converge more rapidly. The regions associated to these limits include, by construction, the box used to seed the initial population of PSO. For that reason the AVM combination could find distinct solutions in 57 cases due to its capability of escaping these limits.
- Compared to the case of Apollo, for this subject the pso solver found many more distinct solutions (50 in CDx as opposed to 1 in Apollo). That happens because the rp+pso combination ignores the intervals inferred by `CORAL` from the input constraints. Instead, it uses the intervals reported by RealPaver. In this particular case, the intervals reported by `CORAL` worked better in some cases than those reported by RealPaver. `CORAL` detects that one variable is used in the context of a trigonometric function and associates to that variable the circular interval $[0, 2\pi]$ (with the ending points identified) as its domain. The search draws assignments to variables from these intervals.

3) TSAFE: Conflict Probe: Figure 6(a) shows the differences in the number of path constraints solved by different

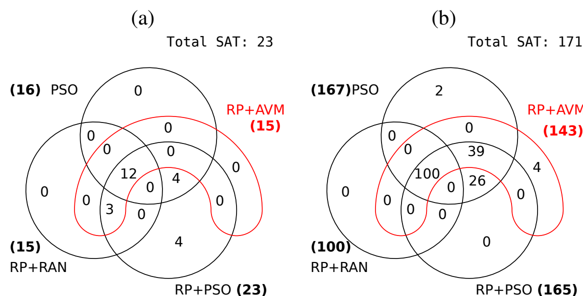

Fig. 6. Differences of constraints solved by different solvers for TSAFE: (a) Conflict Probe (23 in total) and (b) Turn Logic (171 in total).

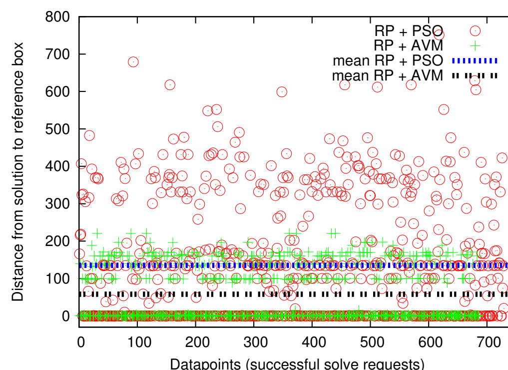

Fig. 7. Distance from solutions to the frontier of the reference box. Only considering successful cases of rp+pso and rp+avm.

solvers. Dynamic symbolic execution of the Conflict Probe subject is expected to produce a small number of feasible paths. This module consists of one small function. Overall, we noticed that the rp+pso combination solved all constraint that other solvers could solve and 4 additional constraints. For this case, rp+pso subsumed the other solvers.

4) TSAFE: Turn Logic: Figure 6(b) shows the differences in the number of path constraints solved by different solvers for Turn Logic. In contrast to the other subjects, the use of interval solvers was not as helpful: RealPaver reported empty boxes on 73% (241/329) of the cases. The reason for this behavior is that RealPaver doesn't support the function $atan2(y, x)$, so we translate it to $atan(y/x)$. Unfortunately, this translation does not cover all the edge cases present in the original function. For example, when the first argument is positive, and the second is zero, the result is the closest value to $pi/2$. However, note that even then our results were not significantly worse.

In summary, we observed the following:

> For three of the cases (Apollo, Conflict Probe, and Turn Logic) the rp+pso combination solved more constraints and more distinct ones. For the CDx subject the rp+avm combination solved more constraints than any other and significantly more distinct ones. Overall, there was no clear winner: in at least one case, each solver considered was subsumed by some other. However the results suggest that the collaboration between a meta-heuristic search and an interval solver was highly effective.

E. Distance to the box

Figure 7 shows the distance of each successful search of rp+pso and rp+avm to the box reported by RealPaver. A circle corresponds to a solution found with rp+pso and a plus symbol corresponds to a solution found with rp+avm. We used the distance of the point $\langle x, y, \dots \rangle$ to the frontier of the box $\langle [x_{lo}, x_{hi}], [y_{lo}, y_{hi}], \dots \rangle$; this distance is given by $d = \sqrt{(x - x_{lo|hi})^2 + (y - y_{lo|hi})^2 + \dots}$. We use $lo|hi$ subscripts to indicate that we choose, in a particular dimension, between the low and high frontiers of the interval to decide which one is the closest to the point.

The data reported indicates that even though many solutions are found within the reference boxes, both PSO and AVM found many solutions *outside* the box. The cases where distances are higher correspond to constraints with more input variables (dimensions). Note that the PSO search on average finds more solutions far from the box than the AVM search, which applies random restarts when fitness function does not improve after some defined number of iterations. In that case, the search resets the candidate solution to within the reference box. We did not include in the plot 14 outliers for the AVM search (distance $> 10^7$); these outliers originated from the Turnlogic subject. For the PSO combination, the mean distance from a solution to the reference box is 135.27 (highlighted with an horizontal line) while the mean distance for the AVM combination is 58.2.

F. Effect of random seeds

In response to research question RQ-3, we evaluated the impact of using different random seeds on effectiveness of constraint solving. Figure 8 shows, in boxplot notation, the distributions of number of constraints solved per solver for 10 different random seeds. We considered pso, rp+pso, avm, and rp+avm solvers in this experiment. The dotted vertical lines highlight the difference between the maximum point in the distribution of the solvers that do not use interval solving and the minimum point in the distribution of the corresponding solver that uses interval-solving support. First note that the use of different seeds can make some difference; in particular, in rp+pso where the variance in number of constraints solved across the use of different seeds appeared higher. Note that the AVM solver is used here as reference for comparison with rp+avm. The plot shows that results remain consistent when considering several different random seeds. For all cases the combined solver was able to solve many more constraints compared to the best cases of both PSO and AVM search.

G. Time efficiency

Figure 9 shows, in boxplot notation, the distributions of time for the cases the solver can find a solution (9(a)) and the cases it cannot (9(b)). The additional cost due to the introduction of

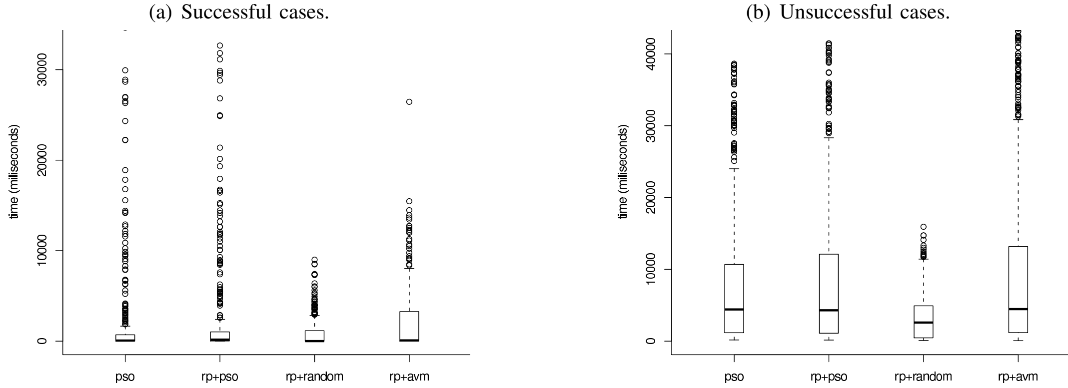

Fig. 9. Distributions of time for each solver, considering all subjects. Figures 9(a) and 9(b) show, respectively, cost distributions for the cases where solutions are found and where solutions are not found.

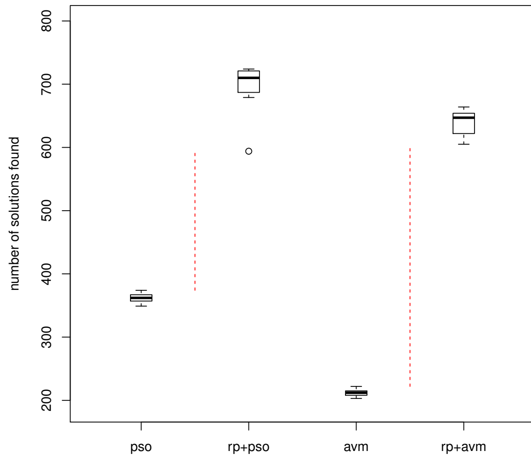

Fig. 8. Distributions of number of solutions found with pso and rp+pso considering different seeds.

RealPaver can be observed comparing the distribution for pso and the distribution for rp+pso. It indicates that RealPaver in general returns quickly for the precision and time bounds we informed. Note from the sizes of the boxplots from Figure 9(a) that, in most cases, each solver reports solutions very fast. The outliers (circles above the boxplots) of the solvers that use pso appear higher in the plot. Conceptually, this indicates that it is worthwhile to use pso for finding the solution. In the most extreme case the solving can take about 40s to finish. It is important to recall that the path conditions generated with the symbolic execution of Apollo and CDx can be very large. Finally, the larger sizes of the boxes from Figure 9(b) relative to the boxes from Figure 9(a) indicates that overall solving time is dominated by the cases the solver cannot find an answer.

H. Discussion and Lessons Learned

We explored alternative design options during our study. For example, we evaluated the option of adding new clauses to the input constraint so to enforce solutions to be within boxes. Such experiment was ineffective, meaning not only that we were unable to find solutions to constraints outside the box (as expected) but also that we did not improve solver's capability of finding constraints inside the box. We understood that the fitness functions we used performed well in guiding the search towards the most effective regions of the search space, may it be inside or outside the box. We also evaluated the option of inverting the order of the combination, i.e., to make the PSO search compute boxes and to use those boxes to specify the domains of variables of RealPaver. However, we realized that PSO is not a very appropriate kind of search for this considering it evolves the individuals of the population as birds in a flocking movement. By the end of the search individuals would be close together; hence if a solution is not found towards the end of the search (when individuals starts to settle) and the overall fitness value is high, it is more likely that the search found a local maxima.

Considering the expected complementary nature of heuristic solving (see Figures 5 and 6) and the potential high cost associated with the cases where solve queries do not yield solutions (see Figure 9(b)), it is worth considering running all solvers in parallel. This should increase the chances of finding a solution. Furthermore, when the symbolic execution is run in "concrete-symbolic" mode [19], [20], it is perhaps beneficial to run the solvers asynchronously together with the process that explores path conditions. This would reduce the time to solve the generated constraints and would enable increasing the bounds used to control resource usage.

In summary, we found that the interval constraint solving combined with the meta-heuristic search is highly effective in solving complex constraints and therefore it increases the applicability of symbolic execution.

VI. Related Work

Various techniques have been proposed recently to deal with undecidable fragments of constraints generated from symbolic execution [2], [19]–[21]. These techniques fall back on concrete values and use randomization to help simplify complex constraints that can not be solved directly. While successful in practice, none of these techniques can effectively solve constraints such as $sin(x) = cos(y)$ (in this case all the previous approaches reduce to random solving). Our approach

is orthogonal to these previous approaches and uses interval solving to seed the meta-heuristic search in CORAL to solve such complex constraints. It remains to evaluate the benefit of other constraint solvers in seeding CORAL; for example, solvers that handle integer arithmetic used in the works above.

The FloPSy [22] constraint solver has been recently developed with similar purpose and approach as CORAL. CORAL and FloPSy use a similar notion of distance in their fitness functions. Different from CORAL, FloPSy does not adjust the weights of constraint clauses in its fitness function as the search advances. As for the search, FloPSy uses genetic algorithms for global search and a variation of the AVM method [7] for local search. Another difference is that CORAL performs some optimizations which are orthogonal to the search (e.g., inference of domains and particular simplifications). FloPSy is used under the concolic (concrete-symbolic) execution of PEX [23], developed at Microsoft Research. CORAL has been customized specially for SPF; this could not be done readily with FloPSy. The work discussed in this paper is orthogonal to the method of search. Our experiments with using AVM for the meta-heuristic search indicate that FloPSy could benefit from the boxes reported by an interval solver the same way as CORAL did.

Decision procedures [24]–[26] determine satisfiability of constraints involving the decidable theories that the procedures support. If decidable, the procedures can additionally provide solutions to the constraints. In prior work [27], we evaluated some of these decision procedures in the context of symbolic execution. We observed that the constraints generated with symbolic execution often involve undecidable theories that are not supported by the procedures. Solvers such as ABsolver [28], CalCS [29], and iSAT [30] provide some limited support for handling non-linear constraints. However, to the best of our knowledge, none of them can handle constraints over floating-point variables, and support all types of mathematical functions (e.g., sin, log, pow) that CORAL supports. Meta-heuristic search based constraint solvers, such as CORAL and FloPSy, treat constraints as black-boxes and therefore can support arbitrary functions.

VII. Conclusions

One important challenge in applying symbolic execution is dealing with complex mathematical constraints. This paper shows that meta-heuristic solving driven by the intervals provided by an interval solver can significantly improve the effectiveness of symbolic execution. We have incorporated the combined solving technique in the CORAL constraint solving infrastructure available at the following link: pan.cin.ufpe.br/coral.

In the future we plan to use CORAL for finding discontinuous frontiers in robustness symbolic analysis [31] and for finding overflow and round-off error bounds in functions that manipulate floating-point values.

Acknowledgments. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES: http://www.ines.org.br).

References

[1] J. C. King, "Symbolic execution and program testing," *Communications of ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[2] M. Takaki, D. Cavalcanti, R. Gheyi, J. Iyoda, M. d'Amorim, and R. B. C. Prudêncio, "Randomized constraint solvers: a comparative study," *ISSE*, vol. 6, no. 3, pp. 243–253, 2010.
[3] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu, "CORAL: Solving Complex Constraints for Symbolic PathFinder," in *NASA Formal Methods*, 2011, pp. 359–374.
[4] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.
[5] L. Granvilliers and F. Benhamou, "Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques," *ACM Transactions on Mathematical Software*, vol. 32, pp. 138–156, March 2006.
[6] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *IEEE Neural Networks*, 1995, pp. 1942–1948.
[7] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.
[8] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *ISSTA*, 2008, pp. 15–26.
[9] "JPF project," 2010, http://babelfish.arc.nasa.gov/trac/jpf.
[10] P. Van Hentenryck, D. McAllester, and D. Kapur, "Solving polynomial systems using a branch and prune approach," *SIAM Journal of Numerical Analysis*, vol. 34, pp. 797–827, April 1997.
[11] R. Dechter, *Constraint processing*. Elsevier Morgan Kaufmann, 2003.
[12] T. Bäck, A. E. Eiben, and M. E. Vink, "A superior evolutionary algorithm for 3-SAT," in *Evolutionary Programming (EP)*, UK, 1998, pp. 125–136.
[13] "ICOS web page," https://sites.google.com/site/ylebbah/icos.
[14] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2002.
[15] C. S. Pasareanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karasai, H. Nine, and S. Neema, "Model based analysis and test generation for flight software," in *Proceedings of SMC-IT*, 2009.
[16] "JavaNCSS," http://www.kclee.de/clemens/java/javancss/.
[17] "Cdx collision detector," http://sss.cs.purdue.edu/projects/cdx/.
[18] "FloPSy web page," http://pexarithmeticsolver.codeplex.com.
[19] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI*, 2005, pp. 213–223.
[20] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *CCS*, 2006, pp. 322–335.
[21] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *ISSTA'11*, 2011, pp. 34–44.
[22] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux, "Flopsy - search-based floating point constraint solving for symbolic execution," in *ICTSS*, 2010, pp. 142–157.
[23] N. Tillmann and J. de Halleux, "Pex: White box test generation for .NET," in *Tests and Proofs*, ser. LNCS, 2008, vol. 4966, pp. 134–153.
[24] C. Barrett and S. Berezin, "CVC Lite: A new implementation of the cooperating validity checker," in *CAV*, 2004, pp. 515–518.
[25] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV*, 2007, pp. 519–531.
[26] B. Dutertre and L. M. de Moura, "A fast linear-arithmetic solver for dpll(t)," in *CAV*, 2006, pp. 81–94.
[27] S. Anand, C. S. Pasareanu, and W. Visser, "JPF-SE: A symbolic execution extension to Java Pathfinder," in *TACAS*, 2007, pp. 134–138.
[28] A. Bauer, M. Leucker, C. Schallhart, and M. Tautschnig, "Don't care in smt: building flexible yet efficient abstraction/refinement solvers," *STTT*, vol. 12, no. 1, pp. 23–37, 2010.
[29] P. Nuzzo, A. Puggelli, S. A. Seshia, and A. L. Sangiovanni-Vincentelli, "CalCS: SMT solving for non-linear convex constraints," in *FMCAD*, October 2010, pp. 71–79.
[30] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert, "Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure," *JSAT*, vol. 1, no. 3-4, pp. 209–236, 2007.
[31] R. Majumdar and I. Saha, "Symbolic robustness analysis," in *RTSS*, ser. RTSS '09, 2009, pp. 355–363.