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*
[‡]*TRACLabs 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*—A challenging problem in symbolic execution is to solve complex mathematical constraints such as constraints that include floating-point variables and transcendental functions. The inability to solve such constraints limit the application scope of symbolic execution. In this paper, we present a new method to solve such complex math constraints. Our method combines two existing: meta-heuristic search and interval solving. Conceptually, the combination explores the synergy of the individual methods to improve constraint solving. We implemented the new method in the CORAL constraint-solving infrastructure, and evaluated its effectiveness on a set of publicly-available software from the aerospace domain. Results indicate that the new method can solve significantly more complex mathematical constraints than previous techniques.**

***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. 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 serves to determine (in)feasibility of paths and to generate actual test-inputs. Thus, effectiveness of symbolic execution to generate 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 for constraint solving [2], [3], [5], 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 *may* contain solutions to the constraint problem. Our approach consists of two steps: (1) an interval solver is used to generate a box for a given path condition and (2) a meta-heuristic search is used to solve the path condition. Unlike our prior work, where CORAL generates the initial population randomly, our new approach uses the intervals from the box reported at step 1 to seed the initial population. Conceptually, a box serves to reduce CORAL's search space.

Our combined approach leverages the power of interval solvers and meta-heuristic search to overcome their individual limitations. Interval solvers can efficiently compute parts of the problem's search space that are likely to contain solutions. However, they typically cannot 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 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 [6] into the CORAL infrastructure. CORAL supports two meta-heuristic search techniques: (1) Particle Swarm Optimization (PSO) [7], which is a global search method, and (2) Alternating Variable Method (AVM) [8], which is

a local search method. The AVM technique was added to CORAL as part of the work reported here. We evaluated the approach on programs from the aerospace domain, that use complex mathematical functions, non-linear operations, and floating-point input variables. We use Symbolic Pathfinder [9] 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 variables. 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 that uses RealPaver as the interval solver, and supports two meta-heuristic search techniques (i.e., PSO and AVM). The implementation is publicly available (see http://pan.cin.ufpe.br/coral), and has been integrated with the Symbolic PathFinder [9] symbolic execution system.
- Experimental results of comparison between a new version of CORAL that incorporates the proposed approach and a previous version of CORAL (that does not use interval-based constraint solving) on a set of subjects from the aerospace domain. Results indicate that use of both PSO and AVM in combination with interval-based constraint solving lead to a significant increase in the number of constraints that could be solved.

II. Background

This section provides background 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 a boolean expression yields 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 assign a bound on the number of paths that can be explored with symbolic execution.

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

1. $log(in\_SYM) < CONST\_4.0$
2. $log(in\_SYM) == CONST\_4.0$
3. $log(in\_SYM) > CONST\_4.0$

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

2) Constraint Solving: Symbolic execution uses a constraint solver in two cases: (i) to check path feasibility and (ii) to generate test inputs. In the first case, symbolic execution checks if the current path is feasible by checking if the path condition of that path is satisfiable. Exploration of a path is interrupted if its path condition becomes unsatisfiable. In the second case, symbolic execution uses a constraint solver to solve path conditions of 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 (http://babelfish.arc.nasa.gov/trac/jpf), a freely available open-source project. It 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`. It directly interprets the (simpler) remaining methods from the Math library.

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:

$$(1.5 - (x * (1.0 - y))) = 0.0 \quad (1)$$

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. This box contains the following solution to the input constraint: `x = 99.99999999999991, y = 0.985`. However, it is important to remark that one reported box does *not* necessarily contain solutions (see note below).

Inner and Outer Boxes. 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 [6] uses advanced constraint-satisfaction techniques [11]. Interval solvers of this kind report *inner* and *outer* boxes on output [6, § 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 if the solver reports no box [6, § 3.2]. In that case, the input constraint is unsatisfiable. The approach does not guarantee which kinds of boxes are reported first. RealPaver classified the box above as an outer box, but, as already observed, the box does indeed contains a solution.

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 their search. One of our goals is to evaluate how far distant from the boxes reported by the interval solver the search could find solutions. We used PSO [7] for global search and AVM [8] for local search. The PSO search is similar to Genetic Algorithms [4]: the population evolves during the search according to some user-defined evolutionary 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. AVM is essentially an adaptation of Hill Climbing [4] to numeric problems. 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.

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

Figure 2. Fitness function of CORAL.

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 a candidate solution. 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 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 on each clause b_i as that clause remains unsolved for longer than some given number of times. The weights help the search to positively differentiate candidates that satisfy "difficult" clauses from those that satisfy many "easy" clauses. Note that a candidate solution is relevant only 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.

1	**Constraint:** $(sin(x1) - cos(x2)) = 0.0 \wedge x1 >= -1 \wedge x2 <= 1$ **Intervals:** x1=[0.9640..., 0.9646...], x2=[0.6061..., 0.6067...] **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=[99.999...,100.0], x2=[99.999...,100.0], x3=[89.999...,90.000...], x4=[99.999...,100.0], x5=[99.999...,100.0], x6=[99.999...,100.0] **Solution:** x1=100.0, x2=98.4818097287292, x3=3.0825564323322396E-11, x4=83.0313044811115, x5=73.32021467962014, x6=41.927226898838214

Figure 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. Variables $c*$ denote constants; `CORAL` substitutes them prior to search.

A. Improving Interval Solving

Consider the constraint 1 from Figure 3. 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"). Despite that, a floating-point solution does not appear to exist there. To confirm this we increased the precision of RealPaver to 16 digits and obtained the intervals: `x1=[0.9642330272329055, 0.9642330272329056]` and `x2=[0.6065632995619922, 0.6065632995619923]`. Note that the fitness values are very close to 100% for any of the 4 pairs of concrete assignments that these intervals admit, but still not a hit. Increasing the precision bound does not change results. We evaluated this constraint in Java with each of the 4 possible assignments from the box above and could not find a floating-point solution. In contrast, `CORAL` (with or without seeding) finds one solution (see row "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. `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 fitness function derived from the constraint did not give much guidance. 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 that RealPaver reports for variable $x1$ admits only one value: 0.0 and that the substitution $x1 \leftarrow 0$ in the constraint makes the search trivial as the assignment to all other variables becomes unconstrained.

C. Improving 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 scenarios where the interaction between the two search methods 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 have higher chances of finding solutions if the search space associated with the reported intervals were small, but this is 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 combining 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 one of the following reasons: (a) the reported box (on the reals) does not admit a floating-point solution (see Section III-A) or (b) the reported box does not contain a real solution (see Section II-B). The (in)accuracy of the box that the interval solver reports is determined by the following factors:

- User-specified precision;
- Removal of conjuncts from the input constraints containing unsupported operators. Reported boxes therefore over-approximate the solution space of the original constraint;
- The effect of the locality problem [6, § 2.5, Figure 3], which can appear on specific inputs.

Despite 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

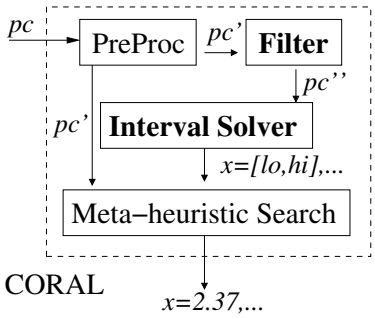


Figure 4. Organization of CORAL with interval solver.

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 with its four processing steps. Boxes with bold fonts highlight new components introduced with this integration.

- In the first step, CORAL applies semantic-preserving simplifications to the input constraints [3]. E.g., it eliminates variables whose values fully depend other variables. The symbol pc' denotes the simplified constraint.
- In the second step, CORAL filters conjuncts that contain operations not handled by the underlying interval solver. For example, RealPaver does not support type casting (e.g., asint(x), where x is a double variable), disjunction, and negation. We note that SPF can avoid the addition of negated constraints to path conditions, by introducing extra non-determinism in the path exploration. As a result of this filtering, note that a box for pc'' could include a concrete solution to pc'' that is not a solution to pc'.
- In the third step, the filter passes pc'' to the interval solver for processing. The interval solver forwards to CORAL the first solution box found within the user-specified timeout. If it cannot find one, CORAL behaves as usual.
- Finally, the fourth 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 (PSO and AVM) are not confined to the box the interval solver reports. Our hypothesis, substantiated with experimental results, is that CORAL often finds concrete solutions to the input constraint which are not necessarily within the box but are close to it; the solution box can therefore improve CORAL's effectiveness by reducing the scope of the search.

B. Implementation

We evaluated two interval solvers for this integration: RealPaver [6] and the ICOS solver (https://sites.google.com/site/ylebbah/icos). To support our decision of which one to select, we considered the robustness of the tools, the supported operators that are relevant to our benchmarks, and the quality of results, including time efficiency and precision of the boxes reported. 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. For each call, the combined solver translates the input format of CORAL to that of RealPaver, AMPL (A Mathematical Programming Language) [13]. We used the default configurations of RealPaver except that we set the 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 to 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 outline the research questions that we want to answer with the experiments. In Section V-B we describe the subject programs that we analyzed, 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 very tight? If the answer is positive, any search method that focuses on these intervals would perform equally well.

RQ-2. How effective are the proposed solvers for dealing with constraints to which neither baseline solver could find solutions? Our baseline solvers are original CORAL and RealPaver (with random search over the intervals).

RQ-3. Is it possible that the result of a combined solver outperforms the baseline solvers due to coincidence? Is it possible that a positive result for a combined solver is the result of a(n) (un)fortunate selection of random seeds?

B. Subject Programs

We describe below the subjects we used.

Apollo. The Apollo Lunar Autopilot is a Simulink model that was automatically translated to Java using the Vanderbilt tool-set [14]. 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 (http://www.mathworks.com/products/simulink/demos.html). 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 (http://sss.cs.purdue.edu/projects/cdx/). 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, 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. This system, called NextGen, has several components, including one called TSAFE (Tactical Separation Assisted Flight Environment). TSAFE is designed to prevent near misses and collisions that are predicted to happen in the near future (within thirty seconds to three minutes). It is still under development, 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. This 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. Path conditions for this subject contain the `atan2` function.

Table I shows the subject sizes and complexity metrics for the constraints that symbolic execution generates. Column "subject" shows the name of the subject and column "size" shows the size computed with JavaNCSS (http://www.kclee.de/clemens/java/javancss/). The following columns show the average number of conjunctions per path condition and the average number of math functions per path condition.

subject	LOC	avg. # clauses	avg. # functions
Apollo	2.6K	39	3
CDx	16.4K	63	6
Conflict Probe	45	7	5
Turn Logic	42	3	20

Table I
SIZE OF SUBJECTS AND COMPLEXITY OF CONSTRAINTS.

C. Experimental Setup

Constraints considered. The set of constraints we analyzed originate from symbolic executions of 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 count the difference in the number of cases one solver can solve a constraint and the other cannot. We use Venn-diagrams to highlight the distinct sets of constraints each solver can solve (in response to RQ-2).

Solvers considered. The list below describes the baseline solvers we used in our evaluation.

- **pso** is the solver that uses `CORAL` with Particle Swarm Optimization search (PSO) [7]. 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 with random search. At each iteration, random search chooses one random input assignment from the intervals (associated with the reported box) and computes the fitness values associated with it. Random search will have higher chances of finding solutions if the intervals are tight (in response to RQ-1). *This is the baseline solver for RealPaver.*

We describe below the combined solvers that we propose:

- **rp+pso** is the combination that uses interval solving with the PSO search, as provided by `CORAL` (see **pso** above). It uses the intervals from the first box that RealPaver reports to seed the initial population of the PSO search. The search is not constrained within the box used.
- **rp+avm** is the combination that uses interval solving with AVM search [8]. 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. For AVM, we use the intervals to seed the search and also to reset the search at random restarts, which serve to workaround local maxima. We translated to Java the publicly-available C# implementation of AVM from `FloPSy` (http://pexarithmeticsolver.codeplex.com) and used in this combination.

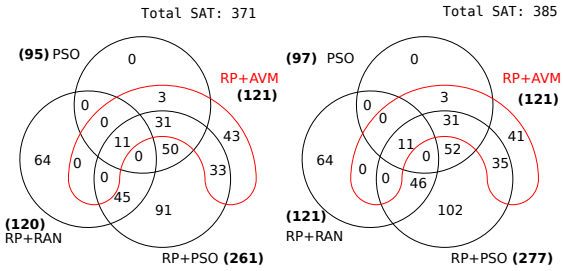
PSO, as a global search, looks for solutions across the entire state space. If the use of one box does not result in good fitness the search can try longer jumps using the many candidate solutions in the search population. AVM, as a local search, looks for solutions close to the boxes. It performs incremental and systematic changes to the single candidate it uses across the search. At random restarts the AVM search re-initiates the search from another point within the 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: while the local search often returns to the box, the global search is not forced to do so. We want to evaluate how they compare. Both approaches have been incorporated to the infrastructure of `CORAL`.

D. Differences between solvers

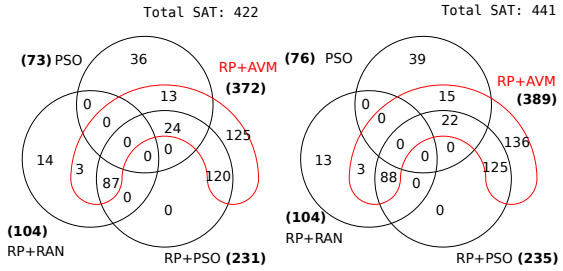
This section discusses the differences between the sets of constraints that each solver is able to solve. Each set in the Venn diagrams from Figure 5 corresponds to one solver and includes the constraints which that solver can solve. Each partition of a set shows a number indicating how many constraints are solved in that partition; the number close to the solver name indicates the total number of constraints that solver solves. For each subject, we show two diagrams: one with averages and the other with median results (numbers rounded) across the 10 seeds used in our experiment. We call a solution *distinct* if only one solver can find it.

1) Apollo: Figure 5(a) shows the results for the Apollo subject. We make the following observations:

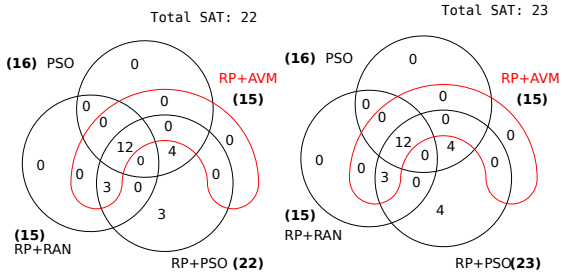
- The rp+pso solver outperforms the pso (baseline for `CORAL`) solver: rp+pso missed on average only 2 constraints that pso finds. On the other hand, rp+pso missed, on average, a total of 64 constraints that rp+ran (baseline for RealPaver) found and 46 constraints that rp+avm found.
- The sets of constraints that pso and rp+ran solve are almost disjoint. This happens because most of the solutions pso finds are not within the intervals that RealPaver reports.
- The sets of constraints that rp+ran and rp+avm solve are also almost 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, the proposed solvers are complementary: rp+pso found many more solutions than any other solver (on average 261, of which 91 are distinct). However, both rp+ran and rp+avm found a significant number of distinct solutions (on average 64 and 43, respectively).



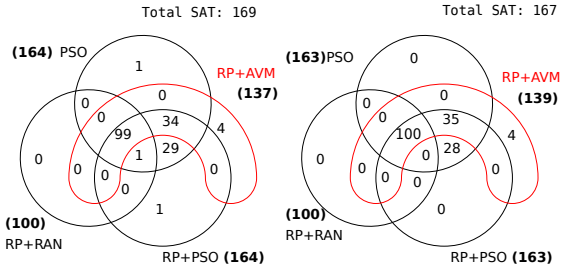
(a) Apollo: average values (left), median values (right)



(b) CDx: average values (left), median values (right)



(c) TSAFE-Confl. Probe: avg. values (left), median values (right)



(d) TSAFE-Turn Logic: avg. values (left), median values (right)

Figure 5. Differences of constraints solved; 10 different random seeds.

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 (88.2%=372/422). Also, the rp+avm solver found *all* solutions that rp+pso found and, on average, 125 distinct ones. We observed that the search landspace and the good results of RealPaver, as observed in the number of constraints solved with rp+ran, made a positive difference for rp+avm, which is optimistic about the intervals RealPaver reports.
- In contrast to the case of Apollo the pso solver found on average a total of 36 distinct solutions. 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: The symbolic execution of the Conflict Probe subject is expected to produce a small number of feasible paths; this module of TSAFE consists of one small function. Overall, we noticed that the rp+pso combination solved all constraint that other solvers could solve and 3 distinct constraints. For this case, rp+pso effectively subsumed all other solvers.

4) TSAFE: Turn Logic: The use of interval solvers was not as helpful for this case but there was no loss compared to pso: RealPaver reported empty boxes on most (73%) of the cases. The reason for this behavior is that RealPaver doesn't support the only math function involved in this experiment: $atan2(y, x)$. So we translated it to $atan(y/x)$, which does not cover all corner cases in the original $atan2$ definition. For example, when the first argument is positive, and the second is zero, the result is the closest value to $\pi/2$.

In summary, we observed that the collaboration of metaheuristic search and interval solving was highly effective considering the subjects we used. As expected, there was no clear winner among the search methods used in the proposed combinations. For the Apollo subject the rp+pso combination solved more constraints than any other and more distinct ones. For the CDx subject the rp+avm combination solved more constraints than any other and significantly more distinct ones. For the two smaller subjects we considered, the contribution of interval solving was smaller.

E. Distance to the box

Figure 6 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 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.

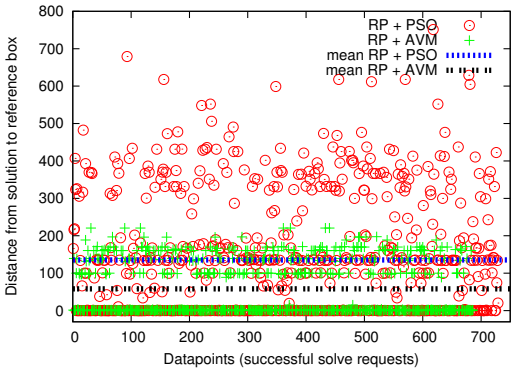


Figure 6. Distance from solutions to the frontier of the reference box. Only considering successful cases of rp+pso and rp+avm.

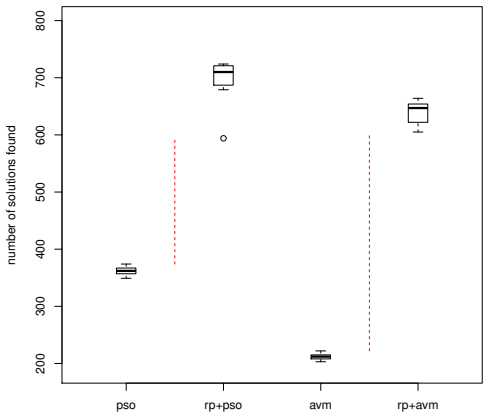
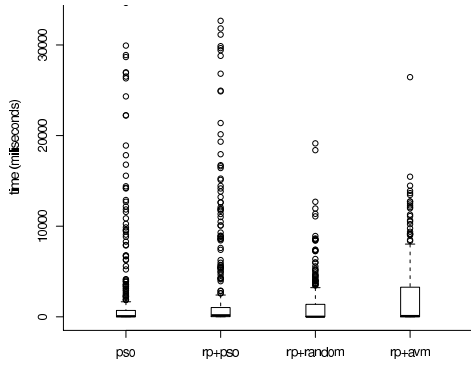


Figure 7. Distributions of number of solutions found per solver, with and without interval solver (`rp`), considering different seeds.

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 7 shows, in boxplot notation, the distributions of number of constraints solved per solver for 10 different random seeds considering all experiments.

(a) Successful cases.



(b) Unsuccessful cases.

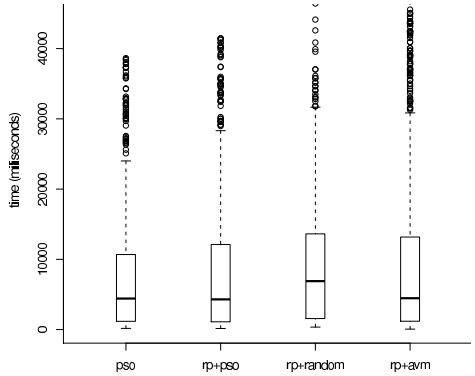


Figure 8. Distributions of time for each solver, considering all subjects. Figures 8(a) and 8(b) show, respectively, cost distributions for the cases where solutions are found and where solutions are not found.

We considered pso, rp+pso, avm, and rp+avm solvers in this experiment. In this experiment, the PSO solver is reference for comparison with rp+pso and the AVM solver is reference for rp+avm. The dotted vertical lines highlight the difference between the maximum point in the distribution of the baseline solvers (i.e., those 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 for the interval-based solver, but results remain consistent when considering the 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 8 shows, in boxplot notation, the distributions of time for the cases the solver can and cannot find solution. The impact due to the introduction of RealPaver can be observed comparing the distributions of pso and rp+pso. One can observe that RealPaver in general returns quickly for the precision and time bounds we set. From the squeezed median lines in the boxplots from Figure 8(a), note also that solvers typically reports solutions very fast. From the time difference across cases; in the most extreme case solving can take ∼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. That is the reason why we wanted to bound execution by number of iterations instead of time; giving chance to the solvers to find solutions to constraints of this kind. The larger sizes of the boxes from Figure 8(b) relative to the boxes from Figure 8(a) indicates that overall solving time is dominated by the cases the solver cannot find an answer. As expected random is slower than other solvers on average (runs 10^6 iterations as opposed to 600 of PSO), but distributions are similar.

H. Discussion and Lessons Learned

We explored alternative design options during our study. For example, we evaluated the option incorporating the reported intervals on the input constraint. Such experiment was ineffective: not only we were unable to find more solutions to constraints outside the box (as expected) but also could not improve solver's capability for finding solutions inside the box. We also evaluated the option of inverting the order of use of interval and meta-heuristic solvers. For example, to make the search compute boxes and then use those boxes to specify the domains of variables for the interval solver. However, we realized that population-based search, such as PSO, is not very appropriate to this. It evolves population individuals as birds in a flocking movement. By the end of the search individuals start to settle and get close together; if a solution is not found towards the end of the search and the overall fitness value is still high, it is more likely that the search found a local maxima.

Considering the expected complementary nature of heuristic solving (see Figure 5) and the potential high cost associated with the cases where solve queries do not yield solutions (see Figure 8(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 [15], [16], it is perhaps beneficial to run the solvers and the process that explores path conditions *asynchronously*. 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 interval solving combined with meta-heuristic search is highly effective in solving complex math constraints. Given the complementary results obtained we recommend running proposed solvers in parallel.

VI. Related Work

Various techniques have been proposed recently to deal with undecidable fragments of constraints generated from symbolic execution [2], [15]–[17]. 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 [5] 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 [8] for local search. Another difference is that CORAL performs some optimizations which are orthogonal to the search (e.g., inference of domains and algebraic simplifications). FloPSy is used under the concolic (concrete-symbolic) execution of PEX [18], 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 determine satisfiability of constraints involving the decidable theories they support. The procedures can additionally provide solutions to satisfiable constraints. In prior work [19], we evaluated some of these decision procedures in the context of symbolic execution. As expected, we observed that constraints generated with symbolic execution often involve undecidable theories. A number of existing decision-procedure solvers do provide some support for handling such constraints. For example, iSAT (http://isat.gforge.avacs.org/), in particular, combines interval-solving with sat solving. However, to the best of our knowledge, none of them supports all types of math functions that meta-heuristic search based constraint solvers, such as CORAL and FloPSy, can deal with. We remain to evaluate how these different approaches compare.

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 pan.cin.ufpe.br/coral. In the future we plan to use CORAL for finding discontinuous frontiers in robustness symbolic analysis [20] 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] 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.

[6] 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.

[7] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *IEEE Neural Networks*, 1995, pp. 1942–1948.

[8] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[9] 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.

[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] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2002.

[14] 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 *SMC-IT*, 2009, pp. 83–90.

[15] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[16] 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.

[17] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *ISSTA'11*, 2011, pp. 34–44.

[18] 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.

[19] S. Anand, C. S. Pasareanu, and W. Visser, "JPF-SE: A symbolic execution extension to Java Pathfinder," in *TACAS*, 2007, pp. 134–138.

[20] R. Majumdar and I. Saha, "Symbolic robustness analysis," in *RTSS*, 2009, pp. 355–363.