CORAL: Solving Complex Constraints for Symbolic PathFinder

Matheus Souza[1], Mateus Borges[1], Marcelo d'Amorim[1], and Corina S. Păsăreanu[2]

[1] Federal University of Pernambuco, Recife, PE, Brazil
{mbas,mab,damorim}@cin.ufpe.br
[2] CMU SV/NASA Ames Research Center, Moffett Field, CA, USA
corina.s.pasareanu@nasa.gov

Abstract. Symbolic execution is a powerful automated technique for generating test cases. Its goal is to achieve high coverage of software. One major obstacle in adopting the technique in practice is its inability to handle complex mathematical constraints. To address the problem, we have integrated CORAL's heuristic solvers into NASA Ames' Symbolic PathFinder symbolic execution tool. CORAL's solvers have been designed to deal with mathematical constraints and their heuristics have been improved based on examples from the aerospace domain. This integration significantly broadens the application of Symbolic PathFinder at NASA and in industry.

1 Introduction

Systematic testing is widely accepted in academia and industry as a major approach to improve quality of general-purpose software. Perhaps less popularized is the role of testing as an economic viable technique to improve reliability of critical systems. In the aerospace domain, for instance, systematic testing has been used to reduce cost of bug finding, i.e., to increase application reliability. NASA, in particular, maintains open-source tools to assist systematic testing.

Symbolic execution [15] is an automated technique to generate test input data. The input to symbolic execution is a parameterized method m of the application under test and the output is a test suite that maximizes path coverage for m. Internally, a symbolic execution tool is organized in two components: the constraint generator and the constraint solver. The constraint generator builds constraints on the input parameters of m for achieving path coverage while the solver attempts to solve these constraints, i.e., to generate concrete assignments to input parameters. A major obstacle for techniques that build on constraint solvers, such as symbolic execution, is the inability to deal with complex constraints. In particular, constraints that build on undecidable theories, constraints that build on decidable theories but are very expensive to deterministically solve, and constraints that the solver cannot handle.

The goal of this work is to improve the solving of constraints that use floating-point variables and complex mathematical functions. Such constraints often occur in the analysis of software from the aerospace domain; for example, consider software such as TSAFE [4, 5] that helps air-traffic controllers in detecting and resolving short-term conflicts between aircrafts. This software estimates the location of an aircraft based on several factors including speed and direction and makes extensive use of floating-point variables and trigonometric functions. Good handling of complex constraints is fundamental for testing software of this kind using a symbolic execution tool such as NASA Ames' Symbolic Pathfinder [20].

Symbolic PathFinder (SPF) is a symbolic execution tool used at NASA and Fujitsu for testing complex applications. This paper reports the results of using the constraint solver `CORAL` to solve the complex mathematical constraints generated with SPF. `CORAL` uses meta-heuristic search, such as genetic algorithms [12] and particle-swarm optimization [14], to look for solutions to constraints that the SPF tool generates. The hypothesis is that *search* can be effective in solving such constraints not managed by traditional decision procedures. The principle of meta-heuristic search is to iteratively refine a set of solution candidates, initially chosen at random, for a fixed number of times. Informed fitness functions evaluate the quality of a candidate to solve a constraint in each generation round. A new generation of candidates is obtained with modifications to the best fit candidates. The search terminates after a determined number of iterations. In our case, it succeeds only when the best fit candidate is also a solution to the input constraint.

To deal with numeric constraints `CORAL` uses a specialized fitness function that conceptually measures the distance of a candidate solution to satisfying a particular constraint. To reduce the search space, it additionally tries to infer the units and ranges of variables from the functions where these variables are used. The design of `CORAL` has been influenced in part by the constraints that SPF generated from the analysis of several NASA applications. In particular, some rewriting rules have been added and the fitness function has been adjusted based on examples from the NASA domain.

This paper makes the following contributions:

– **New constraint solver:** We present `CORAL` a meta-heuristic constraint solver specialized to handle complex mathematical constraints;
– **Integration:** We report the integration of SPF and `CORAL`. This integration moves forward the limits of symbolic execution to manage a wider range of programs;
– **Evaluation:** We evaluate this integration on several examples from NASA and also compare the use of `CORAL` with other constraint solvers (with some support for real arithmetic) that have been previously integrated in SPF.

The rest of the paper is organized as follows. Section 2 briefly illustrates how symbolic execution works. Section 3 describes the Symbolic PathFinder tool. Section 4 describes `CORAL`. Section 5 evaluates the integration of `CORAL` in Symbolic PathFinder. Finally, Section 6 discusses related work and Section 7 gives our conclusions.

2 Symbolic execution

Symbolic execution is a program analysis technique that executes a program with symbolic inputs as opposed to concrete inputs. It computes the effect of program execution on a symbolic state, which maps variables to symbolic expressions. When execution evaluates a branching instruction, the technique 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 the case of a symbolic execution the evaluation of the boolean expression is a symbolic value, so both branches can be taken resulting in different paths 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. Note, however, that programs with loops and recursion can have an infinite number of paths. In those cases, symbolic execution needs to bound the number of paths it explores.

We illustrate symbolic execution using a simple example. Consider the fragment of code from Figure 1 (left) taken from a flight abort executive:

```
if(pressure < 640.0 ||
   pressure > 960.0) {
  abort();
} else { continue(); }
```

1.	$SYM < 640.0$
2.	$SYM >= 640.0 \wedge SYM > 960.0$
3.	$SYM >= 640.0 \wedge SYM <= 960.0$

Fig. 1: Abort example and corresponding path conditions.

If the value of the *input* variable `pressure` is outside nominal values 640.0 and 960.0, then the mission is aborted, otherwise the mission is continued. Traditional testing of this code involves assigning some concrete values to the inputs and executing the code; for example, if the value of variable `pressure` is `460.0`, testing will exercise only one path through the code, corresponding to the condition `pressure < 640.0` being true, resulting in an abort. In contrast, symbolic execution assigns a symbolic value to the input variable `pressure` and analyzes all the three possible paths through the code, corresponding to the three path conditions in Figure 1 (right). The path conditions correspond respectively to the cases where the first term of the disjunction ("`||`") is satisfied, the second term is satisfied, and none is satisfied. Note that due to the short-circuit operator, it is only possible to satisfy the second term of the condition negating the first. Solving these path conditions with a constraint solver gives the test inputs that achieve complete path coverage through the code.

3 Symbolic PathFinder

Symbolic PathFinder (SPF) is a symbolic execution tool for Java bytecode. SPF is used primarily for automated test case generation of code and also of Simulink/Stateflow and UML models, via a translation into bytecode [19]. SPF has been used at NASA (JSC Onboard Abort Executive, fault tolerant protocols, PadAbort-1 models, T-SAFE Java code), in industry (most notably at Fujitsu – 60K LOC), and in various research projects from academia. SPF is part of the Java PathFinder verification tool-set [8], a freely available open-source project. We describe here SPF's main features and how it builds complex mathematical constraints, which are then used with `CORAL`'s heuristic solvers.

Features. The Java Pathfinder tool-set includes the JPF-core project, an explicit-state model checker for Java programs, and several extension projects, one of them being SPF (`jpf-symbc` Java project). The JPF-core implements an extensible custom Java Virtual Machine (VM), equipped with state storage and backtracking capabilities, different search strategies, as well as listeners for monitoring and influencing the search. By default, JPF-core executes the program based on the standard semantics of Java. SPF replaces this concrete execution semantics with a non-standard symbolic interpretation of bytecodes. It uses a custom bytecode instruction factory for that. More precisely, SPF uses the instruction factory class `SymbolicInstructionFactory` to build bytecode instructions that manipulate symbolic values and expressions. For example, the result of the symbolic interpretation of the bytecode `IADD` is to pop from the stack two symbolic integers sym_1 and sym_2 and to push the symbolic expression $sym_1 + sym_2$ back to the stack. SPF stores these symbolic values that symbolic execution computes in special "attributes" associated with the program data, i.e. variables, fields and stack operands.

The symbolic execution of conditional instructions (such as `if` statements) leads to the exploration of distinct program paths, corresponding to the boolean expression of the conditional evaluating to *true* or to *false*. SPF relies on the JPF-core framework to systematically explore the different choices of symbolic execution paths as well as thread interleavings. These choices are explored exhaustively (up to some bounds) using a mechanism of the JPF-core known as choice generators. The SPF implementation uses a specialized choice generator, the `PCChoiceGenerator`, for the construction of path conditions. Each generated choice is associated with a path condition encoding the condition or its negation, respectively. The path conditions are checked for satisfiability using off-the-shelf decision procedures or constraint solvers. If the path condition is satisfiable, the search continues; otherwise, the search backtracks (meaning that the path is unreachable).

Decision Procedures and Constraint Solvers. To check the feasibility of path conditions, SPF uses multiple decision procedures and constraint solvers through a generic interface. Currently, SPF supports the following solvers: `CHOCO` for integer/real constraints, `CVC3` for linear constraints, and the interval arithmetic solver `IASolver`, as well as the SMT decision procedures `CVC3` and `YICES`. Both `CHOCO` and `IASolver` have support for handling constraints on reals and complex mathematical functions, however they both perform poorly in practice (in terms of correctness, speed and tool support). This paper reports the integration of a new constraint solver to SPF for handling complex mathematical constraints, namely `CORAL`.

Handling Math Functions. SPF uses JPF-core's native peers mechanism to model native libraries and any other program parts that cannot be analyzed directly with symbolic execution. Most notably, SPF incorporates native peers models for the methods in the `java.lang.Math` library; these models create symbolic expressions encoding the mathematical functions, that are left uninterpreted. Such use of native peers lifts the interpretation of Math functions from the concrete level to the abstract "model" level: whenever the symbolic execution reaches a call to a complex Math function, that call is intercepted by SPF and it is used as a symbolic operator to build a new symbolic expression. The path conditions containing such expressions are dispatched to an appropriate constraint solver that can handle complex Math constraints, such as `CORAL`.

SPF uses native peers for the following functions from the Java Math library: ACOS, ASIN, ATAN, ATAN2, COS, EXP, LOG, POW, ROUND, SIN, SQRT, TAN. For the rest of the Math functions, which are much simpler, we provide simple implementations that are interpreted directly by SPF.

```
if (Math.pow(in,2.0)>16.0) {
  do1();
} else { do2(); }
```

1. $pow(in\_SYM, CONST\_2.0) < CONST\_16.0$
2. $CONST\_16.0 == pow(in\_SYM, CONST\_2.0)$
3. $pow(in\_SYM, CONST\_2.0) > CONST\_16.0$

Fig. 2: Example with Math function and corresponding path conditions.

Figure 2 shows one example that uses the `pow` math function. Variable `in` stores the symbolic input in_SYM. The symbolic execution of this code produces the three path conditions to the right side of this figure. As mentioned before SPF does not directly interpret the call to the standard Java library function `Math.pow`. Instead, it constructs a symbolic expression `pow(in_SYM,CONST_16.0)` 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[3]. 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.

4 CORAL heuristic solvers

This section describes design and implementation of the `CORAL` heuristic constraint solvers. We first elaborate on the representation of the search space and the search strategies used by `CORAL`. Then we illustrate the fitness function used, and finally, the optimizations.

4.1 Search Algorithms

Representation of space and search. Our characterization of a *candidate solution* is a map from symbolic variables to concrete values. A *population* corresponds to the set of candidates that are active in a given moment in the search. Our work follows an evolutionary approach to search. In this setting, the

[3] The 3-way split reflects the three possible outcomes of the Java bytecode that compares two doubles, according to the Java semantics.

population evolves during the search according to some user-defined principle. Conceptually, each evolution step approximates the candidates to a solution. The search starts with a population obtained from the random assignment of values to variables and terminates after a fixed number of iterations or when it finds candidates with optimal fitness.

The `CORAL` infrastructure provides two different search strategies: random and Particle-Swarm Optimization (PSO). We discuss here PSO, the strategy that performed best in our experiments. Random search is described elsewhere [21, 22]. PSO is a search algorithm, similar to the popular genetic algorithm search (GA), used in combinatorial optimization problems. Both PSO and GA use special operators to mutate candidates during the evolution process. While GA mimics biological evolution (e.g., with mutation and reproduction) PSO mimics movements of a group of animals in swarms. Although GA and PSO operate similarly with successive refinements of the population, they have different computational costs. At each iteration, GA needs to eliminate less fitted individuals, add new ones with crossover, and modify existing ones with mutation. The PSO algorithm updates the search state more efficiently: it uses efficient matrix arithmetic to update a fixed-size population. In PSO terminology candidate solutions are called *particles*. The particles collaborate to compute a solution (this is a central difference between GA and PSO). Each particle has a *position* in the search space and a contributing factor to the population, typically called *velocity*, which PSO uses to update the next position of each particle. The next position of a particle depends on its current position and velocity. The next velocity of a particle depends on the best position the swarm has seen from the start of the search (global) and the best position of that particle from the start (local). Details on design and implementation of these algorithms can be found elsewhere [12, 14].

Fitness functions. The role of a fitness function (a.k.a. objective function) is to drive the search towards (fitter) solutions. This function gives a score denoting the quality of an input candidate to solve the problem. Our solvers use a variation of the Stepwise Adaptive Weighting (SAW) fitness function that dynamically adjusts the importance of different sub-problems for solving the whole problem [9]. For constraint solving, the problem is to solve the entire path condition $pc(\overrightarrow{x}) = \bigwedge b_i$ and the sub-problem is to solve a clause b_i of the input path condition. The definition of SAW is as follows:

$$f(\overrightarrow{x}) \ = \ \sum_{i} \ w_i * g_i(\overrightarrow{x})$$

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. This score is given in the continuous interval $[0.0, 1.0]$ with higher values (respectively, low) indicating better (respectively, worse) fitness. The search goal is to maximize 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. We note that a final solution is only relevant if it satisfies all clauses b_i.

SAW was originally created to solve SAT problems, i.e., propositional formula with boolean variables. We adjusted the definition to handle numeric variables. Recall that $g_i(\overrightarrow{x})$ denotes the score of $\overrightarrow{x}$ on b_i. Function g_i is defined as follows, where each clause b_i is a disjunction of terms $b_{i1} \vee \ldots \vee b_{im}$:

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

Note that the codomain of functions g_i and d are the same; the interval $[0.0, 1.0]$. Function d conceptually measures "how far" the candidate $\overrightarrow{x}$ is from a solution that satisfies the term b_{ij}. We want to maximize g_i and for that we need to minimize d, the distance to solution. For example, for the case where b_{ij} is an equality expression of the form $eq(e_1, e_2)$ we define the distance d as $norm(|e_2(\overrightarrow{x}) - e_1(\overrightarrow{x})|)$. The modulo of the difference denotes the distance between the evaluations of the expressions e_1 and e_2 on input $\overrightarrow{x}$. The function *norm* normalizes the distance in the expected range. This function considers any input above some defined threshold t to return the upper bound 1 for the distance, otherwise it divides the input by t to obtain a value in the expected range. The evaluation of function d on a satisfying solution produces value 0. Definitions of the distance function d to other relational operators are similar.

Example. This example illustrates how the meta-heuristic search operates to find a solution to the constraint $sin(a) = -sin(b) \wedge sin(a) > 0$ using the fitness function we defined. Table 3 illustrates the evolution of a fixed-size population of only two candidates. Each row details one candidate in a given iteration. Columns "it.", "(a, b)", "distance(weight)", and "fitness" show respectively the iteration number, the input assignment (candidate), the distance to satisfy a clause of the constraint with the current weight of the clause in parenthesis, and the fitness value of the candidate. The constraint is satisfied when the fitness equals the sum of the weights. Iteration 0 denotes the initial population. `CORAL` performs 4 iterations to find a solution. Note the increase in weight of the first clause (equality) relative to the second (inequality) as the search progresses.

it.	(a, b)	distance (weight)		fitness
		$sin(a){=}{-sin(b)}$	$sin(a) > 0$	
0	0.0000,0.0000	0.0 (1)	0.01 (1)	1.9900
	0.3927,5.7596	0.0011 (1)	0.0 (1)	1.9988
1	0.3927,6.2832	0.0038 (1)	0.0 (2)	2.9962
	0.3927,5.4978	0.0032 (1)	0.0 (2)	2.9968
2	0.5236,6.2832	0.0049 (2)	0.0 (2)	3.9900
	0.3927,5.7596	0.0011 (2)	0.0 (2)	3.9977
3	0.5236,5.2360	0.0036 (3)	0.0 (2)	4.9890
	0.3927,5.7596	0.0011 (3)	0.0 (2)	4.9965
4	0.0000,6.2832	0.0 (4)	0.01 (2)	5.9800
	0.5236,5.7596	0.0 (4)	0.0 (2)	6.0000

Fig. 3: Fitness-guided constraint solving.

Implementation. We have integrated `CORAL` in SPF by specializing SPF's generic decision procedure interface for `CORAL`; this involves encoding SPF's symbolic expressions into a format that is suitable for solving with `CORAL` and reading the solutions from `CORAL` back into SPF. `CORAL` currently uses the opt4j Java library [3] for implementing the search. The library essentially requires the user to define a fitness function and the representation of candidate solution, which, in our case, is a vector of integers and reals.

1.	$Math.Pow(E, E1) == Math.Pow(E, E2) \quad \Rightarrow \quad E1 = E2$
2.	$Math.Pow(E1, E) == Math.Pow(E2, E) \quad \Rightarrow \quad E1 = E2$
3.	$Math.Log(E) == c \quad \Rightarrow \quad E = POW(2, c)$
4.	$Math.Log_{10}(E) == c \quad \Rightarrow \quad E = POW(10, c)$
5.	$x_1\ [+, -, *, /]\ x_2 == E \quad \Rightarrow \quad x_1 = E\ [-, +, /, *]\ x_2$
6.	$x_1 + c * x_1 = E \quad \Rightarrow \quad x_1 = E/(1 + c)$

Table 1: Some rewriting rules of CORAL.

4.2 Optimizations

This section describes optimizations that CORAL uses.

Inference of variable domains. The quality of initial states is an important factor to determine overall search quality: a solution is obtained with a sequence of modifications on candidate inputs, starting from their initial assignments. CORAL tries to improve the quality of initial random assignments by inferring specific domains associated to each symbolic variable. The principle is that the search becomes more exhaustive when confined to a smaller space. For example, it infers the unit radian for variables that appear free within the context of sine and cosine expressions. For variables of this kind, CORAL starts the search assigning random values from a selection of values in the range $0 - 2\pi$. It also infers ranges which are explicit on the input constraint. For example, it will update the range $[lo_0, hi_0]$ associated to variable v to $[c, hi_0]$ if the constraint $v >= c$ is observed in the path condition and $c > lo_0$ holds, where c is a constant.

Elimination of variables. Before passing a constraint to the search procedure, CORAL attempts to simplify the input formula. The approach it uses for that is to identify variables whose values can be fully determined by others. This is similar to a decision procedure for equality that partitions expressions in equivalence classes [17]. CORAL uses rewriting rules in attempt to isolate variables. Table 1 shows some of the rewriting rules it uses. Note that rule 2 is lossy, e.g., $E1{=}{-2}$ and $E2{=}2$. Rule 5 inverts the side of the arithmetic operation to isolate the variable. Rule 6 factors variable x_1 and inverts the side of the multiplication factor. Note that, considering fixed-precision arithmetic, the rules could lead to incorrect results. However, the search only terminates successfully if the optimal input satisfies the original constraint.

Evaluation of boolean expressions in postfix notation. In our context, evaluation refers to the operation that checks whether a candidate solution satisfies the input formula. Random(ized) search is very sensitive to evaluation time in general [12]. In principle, random search performs increasingly better as evaluation time decreases: more distinct inputs will be selected from an uniform distribution in the same allotted time. It is in our interest to improve evaluation time for a fair comparison with random solving and for more efficient solving. To that end the solver uses a postfix notation to evaluate path conditions on a given input. A postfix expression is scanned from left to right, therefore the operators can be applied efficiently to the operands located at the top of an operand stack. We use a fixed-size array of reals to implement such stack.

4.3 Sample constraints

Table 2 gives a set of representative constraints that CORAL is able to solve. The first column shows the constraint and the second shows the source of the

constraint	source
$(1.5 - x1 * (1 - x2)) == 0$	Beale
$(-13 + x1 + ((5 - x2) * x2 - 2) * x2) + (-29 + x1 + ((x2 + 1) * x2 - 14) * x2) == 0$	Freudenstein and Roth
$pow((1 - x1), 2) + 100 * (pow((x2 - x1 * x1), 2)) == 0$	Rosenbrock
$((pow(((x * (sin((((y * 0.017) - (z * 0.017)) + (((((((((pow(w, 2.0))/((sin((t* 0.017)))/(cos((t*0.017)))))/68443.0)*0.0)/w)*-1.0)*x)/(((pow(x, 2.0))/((sin((t* 0.017)))/(cos((t * 0.017)))))/68443.0)))))) - (w * 0.0)), 2.0)) + (pow(((x* (cos((((y * 0.017) - (z * 0.017)) + (((((((((pow(w, 2.0))/((sin((t * 0.017)))/(cos((t* 0.017)))))/68443.0)*0.0)/w)*-1.0)*x)/(((pow(x, 2.0))/((sin((t*0.017)))/(cos((t* 0.017)))))/68443.0)))))) - (w * 1.0)), 2.0))) == 0.0$	TSAFE
$((exp(x) - exp((x * -1.0)))/(exp(x) + exp((x * -1.0)))) > (((exp(x) + exp((x * -1.0))) * 0.5)/((exp(x) - exp((x * -1.0))) * 0.5))$	PISCES
$x^{tan(y)} + z < x * atan(z) \wedge sin(y) + cos(y) + tan(y) >= x - z \wedge atan(x) + atan(y) > y$	manual

Table 2: Sample of constraints that CORAL handles.

constraint. Some of these constraints are taken from the literature while others were generated by CORAL users and also by SPF from the analysis of NASA applications. Capitalized names indicate subjects from NASA. Note that most of the constraints are non-linear and use mathematical functions. The first 3 constraints are used elsewhere to evaluate the FloPSy constraint solver [18] (see also Section 6). The PISCES subject is discussed in Section 5.4. The manual constraints have been written by 3 users of CORAL. Note that solving equality constraints such as the first 4 in this table is challenging with random and heuristic search as they significantly reduce the solution space.

5 Evaluation

This section presents our evaluation of CORAL. Section 5.1 shows the setup of the various constraint solvers we used in our comparison. Section 5.2 compares the use of CORAL in SPF with other public solvers already integrated to SPF and also compares variations of CORAL. Section 5.3 discusses the impact of the number of search iterations set in CORAL on effectiveness and runtime. Finally, sections 5.4 and 5.5 discuss the analysis of the NASA PISCES library and the Java translation of the Apollo Lunar Autopilot Simulink model.

5.1 Setup

Solvers. The user can control the duration of a solving task in CORAL either by time or number of iterations. In our experiments we use number of iterations to obtain deterministic results. When not mentioned otherwise CORAL uses in each query request PSO as search strategy and 600 iterations (See Section 5.3). We consider the following solvers in our comparison: CORAL, CHOCO [1], CVC3 [2], and YICES [7]. All these solvers have been already integrated to SPF. We note that these solvers have different goals. For example, CVC3 and YICES are decision procedures for Satisfiability Modulo Theories (SMT). In particular, YICES can decide over linear real arithmetic and CVC3 can decide over rational linear arithmetic. But neither CVC3 nor YICES can handle complex mathematical functions directly. CHOCO, on the other hand, is a constraint-programming solver for the theories of integers and reals with support to mathematical functions.

The Wrapper solver. In order to compare the different solvers, we developed a "wrapper solver" to encapsulate all the solvers considered in our evaluation.

Similar to the basic solvers, such solver needs to implement a SPF-defined Java interface with operations for building the objects denoting the terms of a constraint and for calling the solver. We implemented the general solver for two reasons. First, it is possible that one of the solvers fails to solve a constraint that appears in a shallow exploration depth even though it could solve more elaborated constraints. With the wrapper solver, exploration will continue if at least one solver answers positively to a satisfiability check query. All solvers have the chance to answer each query generated with the symbolic execution. Second, the wrapper solver was useful to detect discrepancies between results that would often point to a bug in the SPF-solver integration or the solver itself.

5.2 Comparison with other solvers

Results for decidable constraints. We evaluated `CORAL` with all the other solvers for the symbolic execution of two set data-structures popularly used in testing: binary search tree and tree map. For these subjects, we used the implementation and test drivers available on the SPF codebase. The test drivers explore all sequences of method calls and inputs up to informed bounds. The symbolic execution of these data-structures generates constraints that only involve decidable theories. A decision procedure with support for linear integer arithmetic should be able to find solutions to all satisfiable path constraints. We observed that `CORAL` could solve as many constraints as any other solver in this experiment. The test driver was set to generate sequences up to bound 5. Solving decidable fragments is also important in this context as it is often the case that the input constraint mix decidable and undecidable parts.

Results for constraints with Math functions. We evaluated `CORAL` with 78 manually-written test cases including mathematical function expressions. In this setup, three users of `CORAL` first developed the constraints with the help of the Wolfram Alpha visualization tool [6] and then translated to Java. Although each constraint is satisfiable the translation to Java creates unsatisfiable paths. To note that Java models short-circuit boolean expression with control flow. In this setup we compared only `CORAL` and `CHOCO` since they provide support to math functions. Out of 678 queries `CORAL` solved 595 (87.7% of total). Of these, `CHOCO` did not solve 526. In addition, for no query `CHOCO` could solve and `CORAL` could not. `CHOCO` solved a total of 68 constraints (10.1% of total).

Results for different configurations of `CORAL`. In this experiment, we used all manually-written test cases. This includes complex constraints with and without math functions. The table from Figure 4 compares four instances of `CORAL` in this setup. We use a matrix to show how many constraints one solver could solve that another could not. More specifically, each cell $A[i, j]$ of the square matrix A stores the number of constraints that solver i could solve and solver j

	pso-opt	pso	ran-opt	ran	total
pso-opt	-	116	38	209	722
pso	36	-	50	118	642
ran-opt	10	102	-	179	694
ran	12	1	10	-	525
Total: # Queries=838, SOLVED=763					

Fig. 4: Different configurations of `CORAL`.

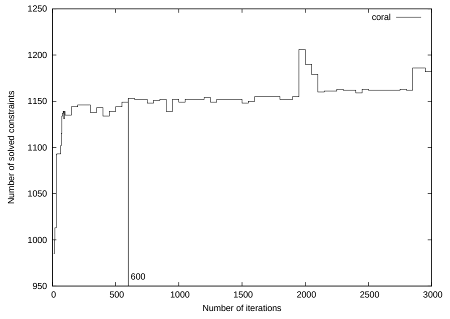
could not. Last column and row show summaries. Last column shows the total of constraints the solver in that line could solve. Last row shows the total number of queries submitted to the solvers and the total number of queries solved. The label "pso" refers to `CORAL` using particle swarm optimization, while "ran" refers to use of random search. The label "-opt" indicates that the solver *enabled* optimization with the inference of variable domains and attempted to isolate variables that no other variables depend as discussed in Section 4. The random solvers use a bound of 360,000 iterations while the PSO solvers a bound of 600 corresponding to approximately the same time of search. (See Section 5.3.) We make the following observations:

- `CORAL` performed well even for cases where it was not designed for. It solved well the linear integer constraints generated from the symbolic execution of binary search and treemap. This result is important considering that symbolic execution of scientific applications builds constraints with both decidable and complex parts.
- `CORAL` performed significantly better than `CHOCO` for the queries including Math functions derived from constraints manually-written by `CORAL` developers. In particular, we found cases when `CHOCO` would report incorrect solutions (e.g. the constraint `Math.sin(x)+Math.cos(y)==1`). We note that we did not tweak any parameter of `CHOCO`. We used the configuration set in SPF.
- Figure 4 shows that the versions of `CORAL` with optimizations found more solutions on average. In addition, "pso-opt" found more solutions than "ran-opt". In some cases the optimized solver missed the solution of some constraints that its non-optimized version finds. As discussed in Section 4.2 the optimizations can reduce not only the search space but also the solution space. Note also that the difference in total number of constraints solved between "pso-opt" and "ran-opt" is not huge. We observed that one affecting factor for this result is the relative high number of inequality constraints (e.g., >=) compared to that of equality constraints for which random search would conceptually have more difficulty to find solutions.

5.3 Impact of number of iterations on precision and runtime

This section discusses the impact of the number of iterations (using the PSO search) in runtime and precision (as measured by the number of solutions found) and present the method used to select a default value for the maximum number of iterations per query to the solve.

We considered manually-written and NASA's benchmarks in this experiment. We varied the number of iterations from 10 to 3000 and measured how many solutions the solver can find for each selection. The leftmost plot from Figure 5 relates number of iterations with numbers of solutions that `CORAL` finds for each assignment. The plot indicates that the ratio of increase varies in different rates. For the lower end of the range (say, less than 500 iterations) the increase is sharp; for larger values the increase is smoother and often unpredictable. For example, `CORAL` finds 1153 solutions when using 600 iterations and only 51 more when using 3000 iterations (which is 5x increase in number of iterations). The



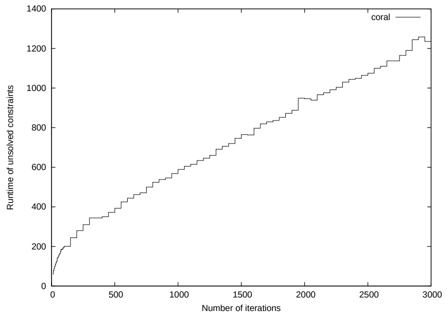


Fig. 5: Left plot relates number of iterations with number of solutions. Right plot relates number of iterations with runtime.

vertical line in the figure shows this point of "stabilization", which we use as default selection for the maximum number of iterations. It is perhaps worth mentioning that the plot is not increasing monotonically with the number of iterations. This occurs because the search algorithm in the opt4j library uses the number of iterations itself as a factor to regulate the perturbation of candidate solutions. That does not imply, however, that the search is non-deterministic for given seed and maximum number of iterations.

The rightmost plot shows average runtime in milliseconds for each assignment of number of iterations. For that, we used a machine with an Intel Core i7-920 processor (8M Cache, 2.66GHz), 8GB RAM, and running Ubuntu 10.04 32bits. In contrast to the previous experiment, we only considered unsolved constraints as they dominate runtime. In principle, the cost of a search iteration varies with the size of the constraint. In this setup, however, the size of the constraints does not vary significantly and the plot reveals an apparent linear relationship between number of iterations and average runtime.

Considering only the constraints that the solver could solve in the experiments from Section 5.2, `CORAL` took on average 60ms, `CHOCO` 3ms, `CVC3` 9ms, and `YICES` <1ms. As mentioned, this runtime difference can increase favorably to non-`CORAL` solvers when considering the constraints that `CORAL` cannot find solutions. Section 7 points to our plans to improve `CORAL`'s runtime.

5.4 Analysis of the PISCES library

We have applied SPF with the new `CORAL` solvers to the analysis of the PISCES (Platform Independent Software Components for the Exploration of Space) mathematical library. PISCES implements a collection of mathematical utility functions and it is used at NASA's Johnson Space Center for Web-based, collaborative development of computer programs for planning trajectories and trajectory-related aspects of spacecraft-mission design.

We have analyzed 20 methods in the library (version 2006), that perform complex mathematical computations such as hyperbolic (arc) sine, cosine, tangent, floating point reminder, factorial, as well as converting time and degrees into radians and back, etc. We were able to analyze all the methods with `CORAL`, and we discovered some problems, that were due to illegal arguments not properly

caught in the code. Furthermore, we tested the implementations by performing checks of known mathematical properties of the PISCES functions.
For example, we checked the following:

```
public static void testHyperbolicTangent(double x) {
  double sinH = MathFunctions.sinh(x); /* hyperbolic sine */
  double cosH = MathFunctions.cosh(x); /* hyperbolic cosine */
  double tanH = MathFunctions.tanh(x); /* hyperbolic tangent */
  assert (tanH == sinH/cosH);
}
```

SPF with `CORAL` generates 6 path conditions, and it correctly determines that only 2 are feasible and that the assertion is not violated. If the assertion is changed to `assert (tanH != sinH/cosH)`, SPF correctly finds two cases when the assertion is violated.

5.5 Analysis of the Apollo Lunar Autopilot

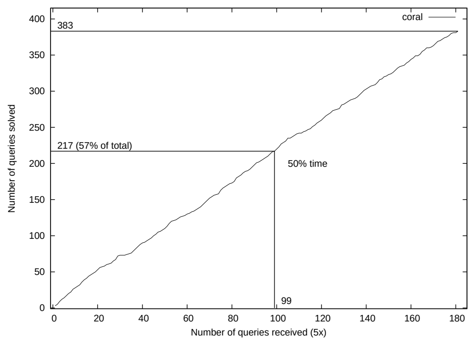


Fig. 6: Number of queries received vs. solved.

We have also applied SPF with `CORAL` to the analysis of the Apollo Lunar Autopilot, a Simulink model that was automatically translated to Java using the Vanderbilt tool-set [19]. This 2.6KLOC subject is deployed in a single package with 54 classes. (Numbers computed with the JavaNCSS tool [13].) 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[4]. It contains both Simulink blocks and Stateflow diagrams and makes use of complex Math functions (e.g. `Math.sqrt`). The model could not be analyzed using `CHOCO` (or other constraint solvers that were previously in SPF), since these solvers could not handle the `sqrt` operation. In this experiment we set the bound on the length of a path condition to 50 and the bound on time to 2h. The bound on length makes the search to backtrack when it makes more than 50 consecutive branching choices. `CORAL` could solve 383 out of 905 queries generated (i.e., 42% of total) during the state-space exploration. Figure 6 summarizes the search. In one axis it shows the number of queries the constraint solver received (note the 5x scale) and the other shows the number of solutions found. The figure highlights the 1h data point. Note from the plot a small increase in saturation as time advances: in 50% of the time 57% (217 out of 383) of the total number of solutions are found. One reason for this is the increase of the path condition size (and cost of solving) with the increase of exploration depth. `CORAL` is sensitive to the path condition size in two ways. On the one hand as the path

[4] `http://www.mathworks.com/products/simulink/demos.html?file=/products/demos/shipping/simulink/aero_dap3dof.html`

condition grows bigger the evaluation time also increases. On the other hand, a fitness function conceptually makes better judgments when more conjuncts appear in the path condition.

6 Related Work

Random-symbolic solving [11, 21, 22] has been recently proposed as an approach to solve constraints with undecidable fragments. The approach is to selectively randomize variables from the input constraint before passing a simplified version of it to a decision procedure. Empirical results show that such collaboration is very promising. We plan to investigate novel ways to promote collaboration between `CORAL` and decision procedures. For example, to first pass the input constraint to a decision procedure (with mathematical functions uninterpreted) and use solutions to seed the initial state of `CORAL`.

The constraint solver `FloPSy` [18] 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 a variation of the AVM method [16] and genetic algorithms. Another difference is that `CORAL` performs some optimizations (e.g., inference of domains and rewriting to eliminate variables) which are orthogonal to the search. (See Section 5.) `FloPSy` is used under the concolic execution of PEX [23], developed at Microsoft Research. `CORAL` has been customized specially for SPF; this could not be done readily with `FloPSy`.

Heuristic search has been previously proposed to improve random (concrete) testing [24, 10] as opposed to symbolic testing. In the context of a concrete execution the fitness function operates directly over program elements. It measures how close execution is to discover a new program path using structural path coverage. One central distinction between the concrete and symbolic approaches is that, to evaluate fitness with concrete testing, one needs to execute the program to collect path coverage data while in the context of symbolic execution one needs to evaluate path conditions, which is an abstraction of the path.

7 Conclusions

This paper proposes the meta-heuristic solver `CORAL` for dealing with constraints involving mathematical functions and floating-point variables that symbolic execution can generate. The integration of `CORAL` with the NASA's Symbolic PathFinder tool (SPF) indicates that the approach is promising. The use of `CORAL` broadens the application of SPF at NASA and industry. `CORAL` is publicly available for use at the following address.

`http://pan.cin.ufpe.br/coral`

In future work, we plan to add incremental solving capability to `CORAL` (within the context of symbolic execution) and to investigate novel ways to collaborate with decision procedures. Finally, we plan to thoroughly evaluate `CORAL` in the context of constraints generated from the analysis of other NASA applications.

Acknowledgments. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[5]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08. Matheus Souza is supported by the CNPQ fellowship 118428/2010-1.

References

1. CHOCO web page. http://www.emn.fr/z-info/choco-solver/.
2. CVC3 web page. `http://www.cs.nyu.edu/acsys/cvc3/`.
3. Opt4J web page. `http://opt4j.sourceforge.net/`.
4. TSAFE maryland. `http://www.cs.umd.edu/~mvz/cmsc435-s09/`.
5. TSAFE mit. `http://sdg.csail.mit.edu/TSAFE/downloads/`.
6. Wolfram Alpha web page. `http://www.wolframalpha.com/`.
7. YICES web page. `http://yices.csl.sri.com/`.
8. JPF project, 2010. `http://babelfish.arc.nasa.gov/trac/jpf`.
9. T. Bäck, A. E. Eiben, and M. E. Vink. A superior evolutionary algorithm for 3-SAT. In *Evolutionary Programming (EP)*, pages 125–136, UK, 1998.
10. L. Baresi, P. L. Lanzi, and M. Miraz. Testful: An evolutionary test approach for java. In *ICST*, pages 185–194, 2010.
11. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223, 2005.
12. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
13. JavaNCSS website. *JavaNCSS - A Source Measurement Suite for Java*. `http://www.kclee.de/clemens/java/javancss/`.
14. J. Kennedy and R. Eberhart. Particle swarm optimization. In *IEEE Neural Networks*, pages 1942–1948, 1995.
15. J. C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7):385–394, 1976.
16. B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.
17. D. Kroening and O. Strichman. *Decision Procedures – an Algorithmic Point of View*. EATCS. Springer, 2008.
18. K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. FloPSy - search-based floating point constraint solving for symbolic execution. Springer Verlag, November 2010. (To Appear)*In Intl. Conference on Testing Software and Systems (ICTSS)*.
19. 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.
20. 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*, pages 15–26, 2008.
21. M. Takaki, D. Cavalcanti, R. Gheyi, J. Iyoda, M. d'Amorim, and R. Prudencio. A comparative study of randomized constraint solvers for random-symbolic testing. In *NFM*, pages 56–65. NASA, May 2009.
22. M. Takaki, D. Cavalcanti, R. Gheyi, J. Iyoda, M. d'Amorim, and R. Prudencio. Randomized constraint solvers: a comparative study. *Innovations in Systems and Software Engineering (ISSE)*, 6(3):243–253, September 2010.
23. N. Tillmann and J. de Halleux. Pex: White box test generation for .NET. In *Tests and Proofs*, volume 4966 of *LNCS*, pages 134–153. 2008.
24. P. Tonella. Evolutionary testing of classes. In *ISSTA*, pages 119–128, New York, NY, USA, 2004. ACM.

[5] www.ines.org.br