Symbolic Execution with Mixed Concrete-Symbolic Solving

Corina S. Păsăreanu
Carnegie Mellon/NASA Ames
M/S 269-2
Moffett Field, CA 94035, USA
corina.s.pasareanu@nasa.gov

Neha Rungta
SGT Inc./NASA Ames
Moffett Field, CA 94035, USA
neha.s.rungta@nasa.gov

Willem Visser
University of Stellenbosch
Private Bag X1
7602 Matieland, South Africa
willem@gmail.com

ABSTRACT

Symbolic execution is a powerful static program analysis technique that has been used for the automated generation of test inputs. Directed Automated Random Testing (DART) is a dynamic variant of symbolic execution that initially uses random values to execute a program and collects symbolic path conditions during the execution. These conditions are then used to produce new inputs to execute the program along different paths. It has been argued that DART can handle situations where *classical* static symbolic execution fails due to incompleteness in decision procedures and its inability to handle external library calls.

We propose here a technique that mitigates these previous limitations of classical symbolic execution. The proposed technique splits the generated path conditions into (a) constraints that can be solved by a decision procedure and (b) complex non-linear constraints with uninterpreted functions to represent external library calls. The solutions generated from the decision procedure are used to simplify the complex constraints and the resulting path conditions are checked again for satisfiability. We also present heuristics that can further improve our technique. We show how our technique can enable classical symbolic execution to cover paths that other dynamic symbolic execution approaches cannot cover. Our method has been implemented within the Symbolic PathFinder tool and has been applied to several examples, including two from the NASA domain.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*

General Terms

Test Case Generation, Verification

Keywords

Symbolic Execution, DART, Constraint Solving

ISSTA'11, July 17–21, 2011, Toronto, ON, Canada

1. INTRODUCTION

Modern software is becoming increasingly complex and needs to be highly reliable. Testing is commonly used for ensuring the reliability of software but it is still mostly a manual, and therefore expensive process. Automating the testing process can significantly reduce the cost of producing software and it can increase software reliability by enabling more thorough testing.

Symbolic execution techniques [15, 5, 6, 14, 29, 7] have shown great promise in automatically generating test cases that achieve high code coverage. Symbolic execution is a program analysis technique that executes programs with symbolic rather than concrete inputs and maintains a *path condition* (PC). The PC is updated whenever a branch instruction is executed, to encode the constraints on the inputs that reach that instruction. Test generation is performed by solving the collected constraints, using an off-the-shelf decision procedure or constraint solver.

In previous work, we have developed a symbolic execution framework [14, 19, 18] that uses a model checker to generate and explore different symbolic execution paths. Whenever a path condition is updated, it is checked for satisfiability using an off-the-shelf solver; if the path condition becomes unsatisfiable, it means that the corresponding branch condition is unreachable, and the model checker is instructed to backtrack. The backtracking mechanism has the advantage that it avoids the potentially expensive re-execution of large parts of the code and it enables incremental decision procedure and constraint solving. Our approach further leverages the model checker's built-in search capabilities, such as different search heuristics, generation and exploration of different thread interleavings, partial order reductions, etc.

An alternative popular symbolic execution approach is implemented in DART (Directed Automated Random Testing) [12], CUTE [23], PEX [26], SAGE [11] and many other related tools. DART[1] and all these other tools perform a concrete execution on random inputs and at the same time they collect the path constraints along the executed paths. The collected constraints are systematically negated and solved to obtain new inputs that drive the program along alternative concrete paths. The process is repeated until the desired coverage criteria (or a timeout) have been reached. The approach does not use backtracking and is instead based on *re-executing* the program with different inputs.

[1]Although these tools are all subtly different they all use the general techniques from the original DART paper and as such we will refer only to DART in the rest of the paper

It has been argued [12, 9] that the main advantage of DART is its ability to fall back on concrete run-time values when "classical" symbolic execution would fail, i.e. when the decision procedure can not handle the complex mathematical constraints that are generated or when analyzing code that uses native or external libraries. Indeed, whenever symbolic execution does not know how to generate a constraint for a program statement depending on some inputs, DART uses the concrete run-time values of those inputs to simplify those constraints.

In this paper we propose a technique that enhances "classical" symbolic execution (as implemented in our framework) to solve cases were it failed before (incomplete decision procedures and handling of native libraries) while still maintaining its advantages. The technique performs *satisfiability checking with mixed concrete-symbolic solving*. Similar to DART, we use concrete values to simplify constraints that can not be handled by the decision procedure directly. However, unlike DART, we do not use the *run-time* values of program variables, but instead we use the concrete solutions of the *solvable constraints* in the current path condition. As a result we may use different concrete values (corresponding to different conditions) along the *same* symbolic path, which will correspond to multiple paths in DART. Our proposed technique does not rely on re-execution, and therefore can be used in conjunction with model checking and backtracking.

Our proposal has two ingredients: (1) use of uninterpreted functions to represent calls to unknown or complex functions during symbolic execution and (2) delayed (or lazy) concretization of these functions, based on the concrete values obtained by solving the "simple", solvable constraints in the path condition ($simplePC$). The result of this concretization is a set of simplified constraints that are solved again, using an off-the-shelf solver, hence the name "mixed concrete-symbolic solving" for our technique. If these new constraints are satisfiable, then the original PC is also satisfiable, and the symbolic execution continues on that path; otherwise it backtracks.

Note that, similar to all the other related approaches [12, 3, 23, 26, 11], our technique is inherently incomplete, i.e. there are cases where the collected path conditions are satisfiable but our proposed technique fails to show that. In such cases, symbolic execution will backtrack and will fail to generate test inputs for paths that are feasible. We do show however that our technique can be more powerful than DART, i.e. it obtains full path coverage where DART fails (see Section 2), but that in fact the two techniques are incomparable (see discussion in Section 3). We also show that our technique is more powerful (in terms of both statement and path coverage) than another related symbolic execution technique, EXE [3], that relies on a simple concretization.

To alleviate the incompleteness problem, we also propose two simple heuristics to enhance our technique. The intuition behind these heuristics is that they force the solver to generate "more interesting solutions" from $simplePC$ in the hope that these solutions will have a better chance at leading to satisfiability proofs when used to simplify the complex part of the PC. The first heuristic uses *incremental solving* to generate multiple solutions for $simplePC$. These solutions are then used to systematically concretize the external functions and perform iterative mixed concrete-symbolic solving (up to a user specified limit).

The second heuristic leverages user annotations that *partition* the domains of the uninterpreted functions into subsets that are deemed interesting by the user. Such partitions can be simple abstractions (e.g. partition the inputs between positive and negative values) or they can come from some form of black-box analysis of the complex functions (e.g. margin analysis [17]). The extra constraints are systematically added to the current PC and mixed concrete-symbolic solving is called systematically for each newly obtained PC.

Other similar heuristics can be added easily, as we maintain a clear separation between the path exploration and the mixed constraint solving in our algorithms. Furthermore, similar to DART, we have added an option to use *random values* for all the inputs that are unconstrained in the path condition.

We have implemented all the proposed techniques in the Symbolic PathFinder tool [18] and have applied them to test generation for Java code. The implementation and the examples are available at [13]. We note however that our results are not particular to Java or to Symbolic PathFinder, as they can apply to any "classical" symbolic execution approach and associated tools.

The rest of the paper is organized as follows. Section 2 illustrates our techniques on an example and compares them with the related approaches DART and EXE. Section 3 describes in detail satisfiability checking with mixed solving, the two heuristics and the random solving option, that is the default for our techniques. That section also provides a more in-depth comparison with DART. Section 4 describes the implementation of the proposed techniques in the Symbolic PathFinder tool and Section 5 describes the experience of applying the implementation to several examples, including two from the NASA domain. Finally we provide more related work in Section 6 and conclusions and future work in Section 7.

2. EXAMPLE

We use the example in Figure 1 to illustrate our proposed techniques and to compare it to the related approaches DART [12] and EXE [3].

Figure 1(a) shows a simple program using Java-like syntax. We analyze method `test` that invokes another method `hash`. Assume that `hash` is a complex mathematical function that our off-the-shelf constraint solver can not handle or a function whose code is simply unavailable for analysis (e.g. *native* method in Java). For illustrative purposes, let us assume that $hash(x) = 10 * x$, for $0 \leq x \leq 10$ and 0 otherwise. "S0", "S1", "S3" and "S4" denote statements that we wish to cover with our automated testing techniques (their exact content does not matter for our purposes here).

Figures 1 (b), (c), and (d) show the program executions that are generated as a result of analyzing method `test` with our method, EXE, and DART, respectively, using the same depth-first search order of exploration[2]. DART and our technique cover all the program statements, while EXE does not; the example also shows that DART and EXE do not cover all the paths through the code, while our technique does.

Let us look at this example in more detail. The path condition PC to reach the execution of "S0" at line `#4` has

[2]Assume for simplicity that the constraint solver always returns the smallest possible solution.

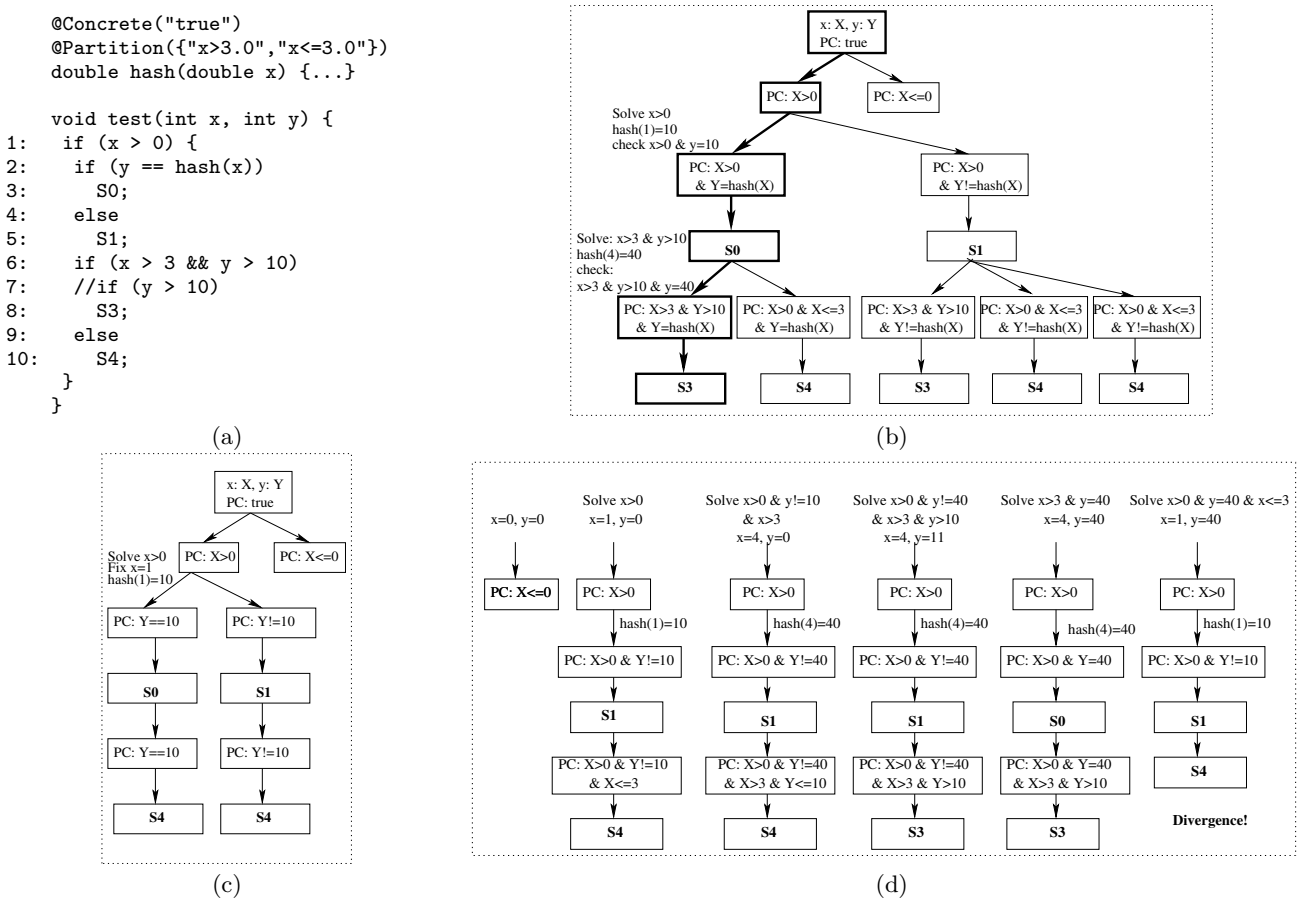

Figure 1: (a) An illustrative example program. (b) Paths explored using our technique; all the statements and paths are covered. (c) Paths explored using EXE; statement "S3" is not covered. (d) Paths explored using DART; all statements are covered but path "S0, S4" is not covered.

the value `X>0 & Y=hash(X)`, where `X` and `Y` denote the two input symbolic values for method `test`. Note that we use upper case letters to denote the symbolic representation of the equivalent variables defined in lower cases letters. "Classical" symbolic execution would get stuck given our assumption that the constraint solver cannot handle `hash` directly so it cannot generate two values for the inputs x and y that drive the execution of `test` through the `then` branch of the conditional at line `#3`. On the other hand, DART starts execution by generating random values for inputs `x` and `y`. If the concrete value of `x`, `v`, satisfies `X>0`; then DART can easily generate a value for `y` that is equal to `hash(v)`. The value of `hash(v)` is known at run-time. If `v` does not satisfy `X>0`, then DART performs an extra iteration where it first solves `X>0` and sets the value of `x` to the solution. DART then re-executes the program and finds a value for `y` that is equal to the run-time value of `hash(x)`. By first picking randomly and then fixing the value of `x`, DART can drive the execution of `test` through different program paths.

Our goal is to achieve something similar to DART in the context of "classical", static symbolic execution. We want to fall back on concrete values when the symbolic execution encounters functions of the form `hash`, while still allowing backtracking.

One simple solution is to first solve the simple, "solvable", part of the PC, i.e. constraint `X>0`, and to use the obtained solution (e.g. 1 for `X`) to "concretize" the value of `x`, i.e. to fix the value of the input `x` to be 1 for the rest of the execution. From that point on, execution involving concrete values follows standard semantics, hence `hash(1)` will be executed concretely and its result, `hash(1)=10`, will be used for updating PC to `Y=10`[3], corresponding to the execution of the `then` branch of the conditional at line `#3`. The negation of the constraint, corresponding to the `else` branch, can also be analyzed easily, so one can generate test cases to cover the execution of both "S0" and "S1".

While very simple, this approach can work well in practice and it was used for example in the EXE tool [3]. However, it has the drawback that by fixing some symbolic inputs to concrete values, it may be overly restrictive so it may miss covering some large parts of the code under analysis. For example, by fixing `x` to be 1, the condition at line `#7` can not be satisfied, so EXE will fail to generate test inputs that cover statement "S3" (see Figure 1 (c) that shows the statements and paths covered by EXE).

[3]For simplicity we ignore here extra constraints that are due to casting from `int` to `double`

DART solves this problem by re-execution: assume DART starts by first executing `test` on inputs `x=0` and `y=0`, it will then collect `X<=0` in the PC, negate the constraint, solve it, so that next time it will execute, it will use the solution `X=1` to re-run the program. In this way, by repeated re-execution, all the statements in `test` get covered (see Figure 1(d)). Note however that some paths (e.g. "S0, S4") remain un-covered, due to divergence, meaning that the "predicted" path ("S0, S4") is different from the one that is actually taken ("S1, S4") [12]. Various tools that implement the DART algorithm handle divergence differently, e.g. the original DART implementation would re-start with new random values, that is not of too much help here.

In contrast, we propose here a technique that does not rely on re-execution, and therefore can be used in conjunction with backtracking. We propose to use *uninterpreted functions* to represent calls to unknown or complex functions during symbolic execution. For our example, this means that when symbolic execution reaches the invocation of `hash`, it will not actually execute it (concretely or symbolically) but rather collect it and add it to the path condition. A special annotation @Concrete("true") is used to mark the methods that are to be kept uninterpreted. Whenever symbolic execution needs to decide feasibility of alternate paths, we first solve the simple part of PC, we use those values to concretize `hash`, and then we solve again the path condition that was thus simplified.

For example, in Figure 1(b), in order to execute the path corresponding to "S0, S3" (marked with bold lines in the figure), we need to decide if PC: `X>0 & Y=hash(X)` is satisfiable. We first split PC into $simplePC$: `X>0` and $complexPC$: `Y=hash(X)` and solve $simplePC$. The obtained solution `X=1` is used to compute `hash(1)=10` and to simplify $complexPC$ into `Y=10`. This constraint is conjoined back with $simplePC$ and the result is $newPC$: `X>0 & Y=10` which is satisfiable.

Similarly, the second PC along the path "S0, S3" is PC: `X>0 & X>3 & Y>10 & Y=hash(X)`, which is equivalent to PC: `X>3 & Y>10 & Y=hash(X)`. Our mixed concrete-symbolic solving technique will first solve $simplePC$: `X>3 & Y>10`, it will use solution `X=4` to compute `hash(4)=40` and will then solve again the simplified constraints $newPC$: `X>3 & Y>10 & Y=40`. Note that, to ensure soundness, we need some extra equality constraints that for simplicity we omitted here but we will discuss in detail in Section 3. In this way, our technique can cover all the paths through the code. Note that unlike DART, we do not use the run-time values of program variables but instead we use the solutions of the collected constraints. As a result we may use different concrete values along the same symbolic path; this would correspond to multiple concrete paths in DART; e.g., we considered two different concrete values for `X` along the path "S0, S3".

Assume now that in the code of Figure 1(a), we replace line `#6` with line `#7`. Then it becomes harder to cover the paths through `test`. For example, DART would not be able to generate a test input that covers "S0, S3" (which it covered before). The reason is that it can only progress based on (the negation of) constraints that it has seen so far, and now that we have removed the condition `x>3`, DART would not be able to generate an input for `x` that satisfies `X>0 & Y=hash(X) & Y>10`, which is the PC corresponding to "S0, S3". For a similar reason, our technique will not be able to cover that path, since it does not have enough information in $simplePC$ to decide the satisfiability of the overall PC. EXE will behave as before.

To overcome the problem, we developed some simple heuristics that allow us to cover "S0, S3" as well. The first heuristic uses iterative solving to generate multiple solutions for the $simplePC$: `X>0 & Y>10` and then uses these solutions to repeatedly concretize `hash` and to perform mixed concrete-symbolic solving. Thus, after running for two iterations, solution `X=2` is found and this is good enough for making `X>0 & Y>10 & Y=hash(X)` true, since `Y=hash(2)=20` and `X>0 & Y>10 & Y=20` is satisfiable.

The second heuristic uses extra constraints provided by the user via a @Partition annotation to help finding solutions (see Figure 1(a) that defines two partitions `x>3` and `x<=3`). For the example, applying the heuristic entails adding the constraints describing the partitions to PC. As a result, we obtain two new path conditions: PC `& x>3` and PC `& x<=3` and we apply mixed concrete-symbolic solving for each resulting path condition. For PC `& X>3` equal to `X>0 & Y=hash(X) & Y>10 & X>3`, we are able to find a solution so we stop.

3. SATISFIABILITY CHECKING WITH CONCRETE-SYMBOLIC SOLVING

In this section we describe our proposed method that uses mixed concrete-symbolic solving to check the satisfiability of the path conditions that are generated using symbolic execution. We also discuss the potential for unsoundness in our method and how we remedy it using extra equality constraints. As mentioned, our method is incomplete, which may lead to missing the execution of feasible paths. We describe two simple heuristics that help alleviate this incompleteness problem. Furthermore, we describe how we use random values for the unconstrained inputs. Finally, we provide a discussion that compares our mixed concrete-symbolic solving technique with DART.

The method shown in Figure 2 checks the satisfiability of a path condition using a combination of concrete and symbolic solving. The input to the method `mixedIsSatisfiable` is the path condition that is being solved ($workingPC$), while the outputs are: a result of the satisfiability check and a set of extra constraints ($extraPC$) that are described in more detail below. The method assumes that there are certain constraints in the path condition that can be solved by a constraint solver and it simplifies the rest using concrete values. The combinations of the symbolic and concrete solving determines the satisfiability of the path condition.

3.1 Main Method

The constraints whose satisfiability can be checked by the constraint solvers (the constraints that can be symbolically solved) are added to the $simplePC$, while all the other constraints are added to the $complexPC$, e.g. non-linear integer constraints, complex mathematical constraints and constraints containing calls to external libraries. The $simplePC$ and $complexPC$ are initialized to null at the beginning of the method. Next, the $workingPC$ is split into $simplePC$ and $complexPC$. If $simplePC$ is un-satisfiable, then it means that the original $workingPC$ is also un-satisfiable, so the method returns false on line `#5`, together with null for the $extraPC$. If $simplePC$ is satisfiable and $complexPC$ is null,

procedure mixedIsSatisfiable(*workingPC*)
1: *simplePC* := null;
2: *complexPC* := null;
3: split *workingPC* into *simplePC* and *complexPC*;
4: **if** solve(*simplePC*) == false **then**
5: **return** ⟨*false*, *null*⟩;
6: **if** *complexPC* == null **then**
7: **return** ⟨*true*, *null*⟩;
8: use solutions of *simplePC* to simplify *complexPC*;
9: *extraPC* := extra *EQ* constraints from solutions;
10: *newPC* := *simplePC* ∧ *simplified_complexPC* ∧ *extraPC*;
11: **return** ⟨*isSatisfiable*(*newPC*), *extraPC*⟩;

Figure 2: Mixed Concrete-Symbolic Solving

```
1:  void test(int x, int y) {
2:    if(x >= 0 && x > y && y == x*x)
3:      S0;
4:    else
5:      S1;
6:  }
```

Figure 3: Example illustrating potential for unsoundness

the method returns true (and null). If, however, *complexPC* is not null, the solution generated for the *simplePC* is used to simplify the *complexPC*, meaning that the concrete solutions are used to execute all the uninterpreted functions in *complexPC* and the concrete returned values are then used instead for these functions. The result of simplification is assigned to *simplified_complexPC*. The values of the solution are used to create extra equality constraints on line `#9` of Figure 2. These additional equality constraints need to be added to ensure soundness for our technique (see next sub-section). Furthermore, these extra equality constraints are used in our incremental solving heuristic, described later in this section. A *newPC* that is a conjunction of the *simplePC*, *simplified_complexPC*, and *extraPC* is generated. This *newPC* can then be solved using the constraint solver at hand, and the result is returned, together with the equality constraints from *extraPC*.

3.2 Potential for Unsoundness

The example in Figure 3 illustrates the reason for additional equality constraints that are needed to ensure the soundness of our method. The path condition corresponding to the execution of "S0" is `X>=0 & X>Y & Y=X*X`. Note however that "S0" is unreachable, i.e. the path condition is not satisfiable. If we split this path condition into the part we can solve, namely *simplePC*: `X>=0 & X>Y`, and the part we cannot solve[4], namely *complexPC*: `Y=X*X`, then we can obtain a result for the first part that suggests we should use `X=0` in the non-linear part. This will simplify the non-linear side to `Y=0`. In turn this will lead to a simplified combined constraint of `X>=0 & X>Y & Y=0` which is satisfiable with `X=1` and `Y=0`; which could lead us to believe that `S0` is reachable. The problem here is that we introduced unsoundness when we ignored the solutions obtained from the solvable side when using the simplified result from the non-linear side.

We therefore always add extra equality constraints on the solutions that we use to simplify *complexPC*. These extra constraints are stored in *extraPC*. For this example, *extraPC* is `X=0` and the final constraint becomes `X>=0 & X>Y & Y=0 & X=0` which is not satisfiable. The constraint on the solution for `X`, namely `X=0`, that we used to simplify `Y=X*X` is thus added back into the final constraint.

In our setting omitting these solution constraints can cause unsoundness, but an analogous situation can occur in DART. However, the unsoundness in the DART approach will not lead to analyzing unreachable code as it could in our case, but will instead lead to divergence, i.e. DART will produce inputs to cover code that it will then not cover. To the best of our knowledge the DART approach doesn't address this problem by recording constraints on the concrete values, as we are doing, but instead just checks if the paths it is trying to cover are indeed covered by the inputs. If a divergence is detected, DART is restarted anew with random input values.

3.3 Incremental Solving Heuristic

The incremental solving heuristic tries multiple solutions for *simplePC* in an attempt to find a solution that leads to the satisfiability of the overall path condition. The algorithm for the incremental solving is shown in Figure 4. When the technique is run with the incremental heuristic `isSat_Incremental`, the *workingPC* is provided as input to the method. In a loop bounded by *MAX_TRIES* (user defined) the method `mixedIsSatisfiable` (shown in Figure 2) is invoked with the current *workingPC*. The method returns true if the mixed path condition is satisfiable at lines `3--5` in Figure 4.

The heuristic comes into play when the mixed path condition is not satisfiable. As mentioned, the additional path condition, *extraPC*, returned by `mixedIsSatisfiable` contains the equality constraints that represent solutions to the *simplePC* and were used for simplifying the *complexPC*. We negate these constraints and add them to *workingPC* to force the solver to generate new solutions, in the hope of exercising the different behaviors from the complex functions encoded in *complexPC*.

3.4 Partitioning Heuristic

The partitioning heuristic uses additional constraints defined in code annotations to partition the domain of the external (un-interpreted) functions when searching for solutions in *simplePC*. The goal is to generate inputs to exercise the external functions on some *interesting* values. The method for the partitioning heuristic is shown in Figure 5. From lines `#1` to `#5`, a call to `mixedIsSatisfiable` is made with the *workingPC*, if it is satisfiable we return true, otherwise we check the *partitionPCs*. The *partitionPCs* is a set of path constraints on the parameters of the complex functions that appear in the *PC*; they are collected during the first run of `mixedIsSatisfiable`. Each *partitionPC* in *partitionPCs* is then added to the *workingPC* one at a time. Whenever the *new_workingPC* is satisfiable the heuristic returns true.

For example, assume that the current working *PC* is `Z+1>0 & Y=hash(Z+1) & Y>10`. And assume that the @Partition annotation for method `hash(double x)` defines two parti-

[4]Assuming here our solver cannot deal with non-linear integer arithmetic

```
procedure isSat_Incremental(workingPC)
 1: tries := 0;
 2: while tries < MAX_TRIES do
 3:    ⟨result, extraPC⟩:=mixedIsSatisfiable(workingPC);
 4:    if result == true then
 5:       return true;
 6:    if extraPC == null then
 7:       return false;
 8:    workingPC := workingPC ∧ ¬extraPC;
 9:    tries := tries +1;
10: return false;
```

Figure 4: Incremental Solving Heuristic

```
procedure isSat_Partitioning(workingPC)
 1: ⟨result, extraPC⟩ := mixedIsSatisfiable(workingPC);
 2: if result == true then
 3:    return true;
 4: if partitionPCs == null then
 5:    return false;
 6: for each partitionPC in partitionPCs do
 7:    new_workingPC := workingPC ∧ partitionPC;
 8:    result := mixedIsSatisfiable(new_workingPC);
 9:    if result == true then
10:       return true;
11: return false;
```

Figure 5: Partitioning Heuristic

tions: `x>3` and `x<=3`. During the first call to `mixedIsSatisfiable`, the partition constraints are instantiated with the symbolic parameters of `hash` and collected into *partitionPCs*; thus for our example, *partitionPCs* will contain `Z+1>3` and `Z+1<=3`. Since the result of `mixedIsSatisfiable` is false, we try to add each one of these partition constraints back to *workingPC* and solve again. The result of `mixedIsSatisfiable` on `Z+1>0 & Y=hash(Z+1) & Y>10 & Z+1>3` is true so we stop (the result being true).

3.5 Random Solving

We also discuss here an option that we have added to our approach. Whenever the *simplePC* contains no constraints on some input variables, we have the freedom to choose any solution for those inputs. We have added the option of using random values for such un-constrained inputs. Similar to the incremental solving heuristics, these random solutions can be generated iteratively in a loop bounded by a user specified limit. These solutions are used to simplify *complexPC* as before. This option, together with SPF's ability to generate and explore paths in random order, give us a random execution capability similar to DART. We had made this option the default.

```
public static void test(int x, int y) {
  if (x*x*x > 0) {
    if (x>0 && y==10)
      abort();
  } else {
    if (x>0 && y==20)
      abort();
  }
}

public static void abort() {
  assert(false);
}
```

Figure 6: Example Illustrating Random Solving.

Consider the example shown in Figure 6. The example is taken from the DART paper [12] where it was used to show a weakness of "classical" symbolic execution. Note that only the first abort is reachable. The *workingPC* corresponding to the `then` branch of the first `if` statement is `X*X*X>0` that is split into the *simplePC* as null and the *complexPC* as `X*X*X>0` since it is a non-linear constraint. The random solving option assigns a random value to `X` and obtains the cubed value of `X` greater than zero with a probability of 0.5. The path condition corresponding to the execution of the first abort is `X>0 & Y=10 & X*X*X>0` which can then be solved easily with our mixed concrete-symbolic solving

Similarly, the path condition for the else branch tries to find a value less than equal to zero. In summary random values are used to solve the *complexPC* when *simplePC* under-constraints the inputs.

A more sophisticated heuristic would be to sample random values from the space of solutions of *simplePC*. This could be done in the cases when the constraint solver provides ranges for the solutions (such as the Choco solver [4]); we plan to explore this option in future work.

3.6 Discussion

In this section we provide a more in-depth comparison of our technique with DART. As for our technique, we consider here the main method (`mixedIsSatisfiable`) with the random solving option.

Incompleteness. As mentioned, both techniques are incomplete. Our technique can miss a path when the solution to *simplePC* cannot be used to simplify *complexPC* and generate a satisfying assignment for *newPC* (even when one exists). Recall that *simplePC* encodes the constraints that can be solved by a decision procedure, while *complexPC* has all the other constraints.

DART on the other hand may miss covering a path due to unsound concretization and divergence. The additional equality constraints used in our approach can be similarly used in DART to get rid of the unsound concretization. However, this would result in severely under-constraining the analysis. This restriction comes into play because any additional constraints added in DART would be kept not just for the current path, but also for the future paths (until the constraint is negated).

We have seen in Section 2 that "classical" symbolic execution with mixed concrete-symbolic solving can be "more powerful" than DART in some cases, meaning it covers more feasible paths than DART under those cases. We show here that in fact the two techniques are *incomparable in power*.

```
public void test (boolean b, int x, int y) {
 if (b) {
   if(y <= 0)
     { ... }
   else
     { ... }
 }
 else {
   // identityF returns the input
   if(x <= 0 && identityF(y) == 1)
     { ... }
   else
     { ... }
 }
}
```

Figure 7: Example illustrating "residual solving"

Our technique vs. DART. Let us revisit the example in Section 2. The reason that DART fails to cover path "S0, S4" is that the constraints for the predicted path `x>0 & y=40 & x<=3` give a solution that leads to a different path ("S1, S4"). The culprit is constraint `y=40` which is used for predicting the execution of "S0"; it comes from negating a constraint that was previously created during a concretization of `hash` (when the concrete value of `x` was 4 and `hash(x)` was 40). In reality the constraint `y=10` would have been useful, since `x=1` is used for the current path and `hash(1)=10`. The problem here is that DART tries to satisfy a constraint that was specific to a previous run. Both constraints `y=40` and `y=10` correspond to the same condition in the code, namely `y=hash(x)`. On the other hand, our technique does not have that problem, since it maintains explicitly the condition `y=hash(x)` in the PC. Furthermore, our technique may use different concrete values to solve different PCs along the same symbolic path and it does not need to carry along the additional constraints that result from concretization. Therefore solving a PC is not affected by the additional constraints that arise from solving previous PCs.

Let us consider now the cases where DART can potentially cover a path p that our approach cannot. Remember that our approach misses a path p when the solutions to $simplePC$ can not be used to render $newPC$ satisfiable. There are two cases to consider: (1) Some additional information is available that can in essence further constrain $simplePC$ and (2) p is generated using the initial random input values. We discuss both cases.

(1) Additional information is available to DART. It may well happen that the additional "artificial" constraints discussed above and created in previous runs may have the opposite, beneficial effect that they can actually help in further constraining the $simplePC$. An alternative is that for the variables that are unconstrained on the current path, solutions created from previous runs are used. We refer to this phenomenon as "residual solving".

An example in Figure 7 is used to illustrate how residual solving can aid DART. For this example, DART generates the following program paths:
(1) `b=true,y=0,x=0` with these values it will generate the constraint `b & y <=0` where the last term is negated to have (2) `b=true, y=1, x=0` and generate the constraint `b & y > 0`; there are no more paths for `y` so it now negates `b` (3) `b=false, y=1, x=0` This will follow the true branch for `(x<=0 & identityF(y) == 1)`. Here it just happened that the last value DART solved for `y` on a *different path* is equal to the one that is needed to make `identityF(y)==1`. Using our technique, `y` is not constrained in the corresponding $simplePC$, so we can pick any value for it, but that will not necessarily satisfy `identityF(y)==1`. Assuming DART followed the same search order it will execute the true branch of `(x <=0 & identityFunction(y) == 1)` every time.

We believe, however, that the advantage to DART illustrated by the example above is rather a low level implementation detail that could be easily added to our technique. We could provide a setting in our method to pick the last stored solution in the case of unconstrained variables, and a random value if no solution was previously stored.

(2) Random execution vs. random solving Randomization is an important and powerful technique in DART and it can lead to the discovery of path p. On the other hand, random solving is used within our method when there are no constraints within $simplePC$ on variables that need to be concretized. Hence random solving enables our technique to harness some of the power of randomization. Combined with a randomized search order in SPF, our technique could potentially reap similar benefits from randomization as DART. Such a claim is, however, hard to prove theoretically. A careful experimental comparison would need to be done between the two approaches. Such an empirical study is beyond the scope of this paper and we leave it for future work.

Finally, we remark that our proposed technique is applicable only in the cases of "pure", side-effect free functions, or functions whose side effects are deemed unimportant for the analysis. On the other hand DART is more widely applicable, although it is likely to diverge in those cases.

4. IMPLEMENTATION

In this section we describe the implementation of our proposed methods in the Symbolic PathFinder (SPF) tool [18], in the context of symbolic execution of Java programs. Note however that our proposed methods are not particular to the Java language or the SPF tool and they are applicable to any technique or tool that performs symbolic execution.

4.1 Symbolic PathFinder (SPF)

SPF is a symbolic execution tool for Java bytecode. SPF is part of the Java PathFinder verification tool-set [13] which includes JPF-core, an explicit-state software model checker, and several extension projects, one of them being SPF. JPF-core implements an extensible custom Java Virtual Machine (VM), 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 concretely based on the standard semantics of the Java and uses a large collection of *models* for native libraries.

SPF replaces the concrete execution semantics of JPF-core with a non-standard symbolic interpretation of bytecodes. For example, when adding two symbolic integers $sym1$ and $sym2$ (by executing the `IADD bytecode`) the result is a symbolic expression representing $sym1 + sym2$. The symbolic values and expressions computed by symbolic

execution are stored in data "attributes" associated with the program variables, fields and stack operands. SPF uses JPF-core to explore the different symbolic execution paths, as well as different thread interleavings.

The symbolic execution of conditional instructions (such as `if` statements) involves exploration of paths corresponding to the branch condition evaluating to both true and false. These choices are generated non-deterministically by a "PC choice generator". 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 branch is unreachable.

4.2 Mixed Concrete-Symbolic Solving and Uninterpreted Functions

We have implemented the procedure described in Figure 2 and the three heuristics described in Section 3 as additional procedures for checking satisfiability of path conditions within SFP.

By default, all the non-linear integer constraints and the constraints involving `java.lang.Math` methods are left uninterpreted (i.e. are added to *complexPC*). In addition, all the Java methods (native or not) that are annotated by the user (see below) can now be handled with the newly implemented technique.

To represent uninterpreted functions, we created a new class `FunctionExpression` whose instances record the signatures of the external methods. The main elements of a `FunctionExpression` are: (a) the fully qualified method name, (b) a set of symbolic arguments and their respective types to the method, (c) list of conditions—if they are specified by the @Partitions(..) annotation, and (d) the return type of the method. We use Java reflection to perform the actual invocation of the methods during the concretization and simplification of the generated *PC*s. Reflection in Java enables dynamic retrieval of classes and methods defined within the classes; it also provides the mechanism for invoking the retrieved methods.

Using reflection we are able to concretely execute the method within the `FunctionExpression` on the host VM. This allows us to concretely execute native methods without providing a model class for the native class as is required within the JPF tool-kit to handle native code. The value returned after the execution of the method is used to check the satisfiability of the *newPC*. Note that our approach works only for the external methods that are "pure", side-effect free, or for methods whose side effects are deemed uninteresting for the analysis (e.g. printing statements). All the other methods still need models. Alternatively, one can use a simpler concretization approach, similar to the one discussed in Section 2 and implemented in the EXE tool.

4.3 User Annotations and Listener

The only input required from the user is in the form of annotations, that specify which methods to leave uninterpreted. We use a Java annotation (@Concrete("true")) to indicate methods that we are unable to execute symbolically, for e.g, a native method or a method that generates non-linear constraints that an underlying constraint solver is unable to solve. Another annotation (@Partition(..)) provides a list of conditions that is used in the partitioning heuristic (as discussed in Section 3).

```
class Bessel
{
  @Concrete("true")
  public static native double bessely0(double x);

  public static void main(String[] args)
    {
      run_bessel(0.0);
    }

  public static void run_bessel(double x) {
       System.out.println("Calls of  bessely0");
       double y = bessely0(x);

        if(x>=1.25 && y > 0.2)
          System.out.println("Br1");
        else
          System.out.println("Br2");
  }
}
```

Figure 8: Example containing native method

We provide a listener, `ConcreteExecutionListener`, that monitors the symbolic execution of the program and detects when a method invoked during execution contains the annotation @Concrete("true"). In that case, the listener generates an uninterpreted function (instance of `FunctionExpression`) for that method.

After generating the `FunctionExpression`, the listener sets a concrete random value as the return value (this value is never used) and its corresponding symbolic attribute as the generated `FunctionExpression`. The listener finally sets the instruction following the invoke instruction as the next instruction to be executed, thus omitting to execute the method either concretely or symbolically.

5. EXPERIENCE

All the examples presented in this paper have been analyzed using the implementation of our technique; the examples and the implementation are available from the JPF repository[5]. In this section we discuss the application of our technique to (a) an example that uses native method calls and (b) two larger examples from the NASA domain: TSAFE and the Apollo lunar autopilot.

5.1 Native Method Example

An example that invokes a native method that is annotated with @Concrete is shown in Figure 8. The example is taken from the NAG C Library from the Numerical Algorithms Group (www.nag.com). This is a mathematical and statistical library containing routines for linear algebra, optimization, quadrature, differential equations, regression analysis, and time-series analysis. The example calls a C library routine directly from Java using the Java Native Interface (JNI). There is a two fold advantage of reusing the native libraries; one it avoids the arduous task of rewriting the library in Java and two the CPU intensive operations run much faster as machine dependent native code

[5] `http://babelfish.arc.nasa.gov/trac/jpf`, see jpf-symbc/src/examples/concolic

compared to interpreted Java bytecode. Our method now provides an easy way to invoke these native methods using the @Concrete annotation. Classical symbolic execution in SPF requires a model implementation of the native method and cannot analyze the example in Figure 8 without it.

The native method, `bessely0`, shown in Figure 8 has a single argument and a single return value. The `bessely0` is part of the Standard C Math Library. The various Bessel functions, of which this is one, are named for 18th-century German astronomer Friedrich Wilhelm Bessel.

The `main` method invokes the `run_bessel` method with an input parameter that is treated as a symbolic variable. The symbolic variable `x` is used to invoke a native function `bessely0`. Since the method has the @Concrete annotation it returns an uninterpreted function: `bessely0(D` x_{sym}`)D`. The method signature indicates that it takes x_{sym} of type double as input and returns a double. As the symbolic execution continues after skipping the native method, it reaches the conditional branch. The *workingPC* for the true branch of the path is:

$x_{sym} >= 1.25 \wedge$ `bessely0(D` x_{sym}`)D` > 0.2

The *workingPC* is split into $x_{sym} >= 1.25$, the *simplePC*, and `bessely0(D` x_{sym}`)D` >0.2, the `complexPC`. The concrete value generated from the solution of the `simplePC` is used to invoke `bessely0` using reflection. Thus, with the help of a simple annotation, SPF is able to handle native methods without modeling them. There are 11 states generated in order to explore both the branch statements in the `run_bessel` method.

5.2 TSAFE

Tactical Separation Assisted Flight Environment (TSAFE) seeks to predict and resolve loss of separation between two or more air crafts in the time horizon between 30 seconds to 3 minutes. The separation assurance prototype contains inputs that include real values for position, velocity, and direction of the multiple air crafts. The inputs also encompass the conditions whether the air crafts are eligible for change in course as directed by air traffic controllers. Confidence in the system depends on how it reacts to unexpected scenarios. We present the analysis results for the conflict probe class in TSAFE. The conflict probe computes whether a loss of separation can occur. TSAFE has been analyzed extensively in recent work [8], where SPF was tried but could not be used because the TSAFE component contains complex mathematical functions that could not be handled by the constraint solvers.

The component that we analyzed contains approximately 400 SLOC, makes use of various complex math functions such as `sqrt`, `pow`, `sin` etc. The component contains a bounded loop that generates a large number of constraints in the system during execution. Recall that random solving is used when there are solutions from *simplePC* that can be used. To capture the variations due to randomization we execute ten trials of the method on the TSAFE example and record the average number of states generated, average time taken in seconds, and maximum branch coverage obtained across the different trials. Our method explores 12,683 states in 15 seconds on average. It manages to obtain a 100% branch coverage where five out of the five branches are covered, and it generates 6 test cases. One in about every three trials obtained a 100% branch coverage while the others obtained 80% branch coverage. The path conditions generated during the process have at most 22 constraints. Note that in order to obtain similar branch coverage using black box testing thousands of tests were generated in [8].

5.3 The Apollo Lunar Autopilot

We have applied our techniques to the analysis of the Apollo Lunar Autopilot, a Simulink model that was automatically translated to Java using the Vanderbilt toolset [20]. The translated Java code has 2.6 KLOC in 54 classes. The Simulink model was created by an engineer working on the Apollo Lunar Module digital autopilot design team. The goal was to study how the model could have been designed in Simulink, if it had been available in 1961. The model is available from MathWorks[6]. It contains both Simulink blocks and State flow diagrams and makes use of complex Math functions (e.g. `Math.sqrt`). The model could not be analyzed before using Choco [4] or the SMT solvers incorporated in SPF, due to errors and omissions in capabilities of the constraint solver.

In this experiment, the goal was to generate test sequences that lead to certain values at the beginning of a Simulink block. This process mimics what the developers often do: try to generate test cases that stress certain blocks inside the model under limit values. We wrote an Observer Automaton that runs synchronously with the Autopilot Simulink model; the automaton transitions to the error state when a certain value in the model (`tjcalc`) is above a threshold. Since this is a reactive model that is driven by sequences of time steps, we ran the model for different time-step sequences and for different thresholds. We used the main method `mixedIsSatisfiable` and the random solving option. We used Choco as the off-the-shelf solver. As an example, for threshold value 10 and sequence size 5, we obtained a five step test sequence that drives the model to the error (running for 6 m 1 s and using 15 MB of memory). In contrast, for values below the threshold, the technique generates easily a two step sequence (running for 1 s and using 15 MB of memory).

5.4 Discussion

For all the examples presented in this section, we used the main method, with the random solving option to generate both test inputs and test sequences to drive the execution of the example programs. All these examples show the feasibility of our main method and demonstrate that "classical" symbolic execution as implemented in SPF has been enhanced to handle cases that could not be handled before. The heuristics did not play any significant role in analyzing these examples; the randomization in our technique provided enough variation in the solution space. Further experimentation is needed to assess the proposed heuristics.

We note that random values are only used for variables unconstrained in *simplePC*. In the Apollo and TSAFE examples, *simplePC* is not empty and the solutions of *simplePC* are successfully and extensively used to simplify *complexPC*. An example *PC* in Apollo contains 37 constraints in *simplePC* and 6 in *complexPC*; a *PC* in TSAFE contains 14 constraints in *simplePC* and 8 in *complexPC*.

[6] `http://www.mathworks.com/products/simulink/\\demos.html?file=/products/demos/shipping/simulink/\\aero\_dap3dof.html`

6. RELATED WORK

Our work is related to the large body of work on white-box test-case generation and static bug-finding, but we focus here on the more closely related works. "Classical" symbolic execution has been introduced in the 70s [15, 5] and it has since been explored in a large body of work [6, 24, 21, 7, 29, 14, 28, 19] in the context of test case generation. The work presented here is enhancing all these classic symbolic execution approaches to handle incompleteness in decision procedures and native library calls, by using concrete solutions to simplify constraints that encode uninterpreted functions. In addition a number of static bug-finding tools also use classic symbolic execution and can thus also benefit from these techniques [27, 1].

In the preceding sections we have already highlighted the close relationship between our work and two other algorithms, EXE and DART, the latter being implemented in many other tools, such as CUTE [23, 22], PEX [26], and SAGE [11]. We will thus discuss here more on extensions of these works.

Firstly, the KLEE [2] tool improves on EXE in an orthogonal direction to what we do, namely, they rather built models of external (unknown or unanalyzable) functions. This approach allows them to therefore stay completely symbolic, rather than concretize inputs for functions that cannot be analyzed, as we do here. Note that such a modeling approach is standard practice in software model checking. The approach however requires considerable manual effort.

Maybe the most closely related work to ours is that of Godefroid in [10], which was independently and concurrently developed. The goal of the work in [10] is to improve the current DART approach to address some of its limitations that we have also discovered here. In contrast, our goal is to improve "classical" symbolic execution to address its limitations. Godefroid in [10] proposes to use a combination of validity checking and uninterpreted functions, and to use the validity proofs to generate test cases. It is shown that theoretically this approach is more general than DART, but to make it work in practice one needs to capture input-output pairs from observing execution of the functions. The paper also addresses the unsoundness of the original DART approach, and suggests a solution similar to that of EXE where the concretization constraints are added to the PC for the rest of the path. As we have shown, adding these constraints permanently will over-constrain the paths. In [10] this point is also made, in the context, that sometimes taking the unsound route has its advantages in that it might by accident discover useful inputs; in our approach, we can pick different solutions on the same path, which cannot be done in DART. Furthermore, we do not rely on validity checking which poses an implementation challenge. Instead we rely on standard constraint solving, a widespread technology.

Orthogonal constraint solving techniques [25, 16] use artificial intelligence methods, such as genetic algorithms and particle swarm optimizations, to help solving complex mathematical constraints in the context of symbolic execution. Those works can not deal with native or external libraries as we do here.

7. CONCLUSIONS

Symbolic execution is a powerful technique for the automated generation of test cases that achieve high code coverage. DART is a variant of symbolic execution that performs iterative run-time symbolic execution along concrete paths, starting with a random run. It has been argued that the main advantage of DART is its ability to handle situations where classical symbolic execution techniques fail, due to incompleteness in decision procedures and handling of external library calls.

We have proposed a technique that helps classical symbolic execution gain in power in the cases where it failed before. Our technique uses satisfiability checking with mixed concrete symbolic solving. Similar to DART, we use concrete values to simplify constraints that can not be handled by the decision procedure directly. However, unlike DART, we do not use the *run-time* values of program variables, but instead we use the concrete solutions of the *solvable constraints* in the current path condition. As a result we may use different concrete values (corresponding to different conditions) along the *same* symbolic path, which corresponds to multiple paths in DART. Our proposed technique does not rely on re-execution, and therefore can be used in conjunction with model checking and backtracking.

We have shown that our technique can be more powerful than DART and a related approach, EXE, that performs a simple concretization. We have further proposed two heuristics to enhance our technique. We provide an implementation in the SPF symbolic execution tool. We have the proposed techniques to a series of examples, including two realistic ones from the NASA domains.

In the future, we plan to robustify our implementation and to perform more experimentation to fully assess the merits of the proposed technique. We also plan to fully evaluate the two heuristics that we have proposed; fine-tuning them is needed as they may become expensive in practice. We also plan to devise more powerful heuristics and also explore combinations of existing heuristics.

Our results show how mixed concrete-symbolic solving can help "classical" symbolic execution in the cases it failed before, due to incomplete decision procedures, handling native libraries, etc. The results presented here are generally applicable; they not particular to the Java language or the SPF tool. In order to apply the presented techniques to more realistic Java programs, we will need to look into handling the data structures that are executed outside symbolic execution. One can use JPF's serialization mechanism for that.

Acknowledgments

We thank Patrice Godefroid and Koushik Sen for useful discussions on the DART algorithm, for confirming the paths taken by DART on the example presented in Section 2 and for explaining to us the divergence. We also thank Cristi Cadar for his feedback on EXE. Finally, we thank Heinz Erzberger at NASA Ames Research Center for allowing access to the TSAFE example.

8. REFERENCES

[1] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[2] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.

[3] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. *TISSEC*, 12(2):1–38, 2008.

[4] Choco Solver. http://www.emn.fr/z-info/choco-solver/.

[5] L. A. Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491, 1976.

[6] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé. Using symbolic execution for verifying safety-critical systems. In *ESEC/FSE*, page 151. ACM, 2001.

[7] X. Deng, Robby, and J. Hatcliff. Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In *TAICPART-MUTATION*, pages 3–12, 2007.

[8] D. Giannakopoulou, D. Bushnell, J. Schumann, H. Erzberger, and K. Heere. Formal testing for separation assurance. In *To Appear, Annals of Mathematics and Artificial Intelligence*. Springer, 2011.

[9] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54. ACM, 2007.

[10] P. Godefroid. Higher-Order Test Generation. *Proc. PLDI*, 2011.

[11] P. Godefroid, P. de Halleux, A. Nori, S. Rajamani, W. Schulte, N. Tillmann, and M. Levin. Automating software testing using program analysis. *Software, IEEE*, 25(5):30–37, 2008.

[12] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[13] Java PathFinder Tool-set. http://babelfish.arc.nasa.gov/trac/jpf.

[14] S. Khurshid, C. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Proc. TACAS*, pages 553–568, 2003.

[15] J. C. King. Symbolic execution and program testing. *Comm. ACM*, 19(7):385–394, 1976.

[16] K. Lakhotia, N. Tillmann, M. Harman, and J. De Halleux. Flopsy: search-based floating point constraint solving for symbolic execution. In *ICTSS*, pages 142–157, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] T. Menzies and Y. Hu. Just enough learning (of association rules): the tar2 "treatment" learner. *Artif. Intell. Rev.*, 25(3):211–229, 2006.

[18] C. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE*, pages 179–180. ACM, 2010.

[19] 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 *Proc. ISSTA*, 2008.

[20] C. S. Păsăreanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine, and S. Neema. Model based analysis and test generation for flight software. In *Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology*, pages 83–90, Washington, DC, USA, 2009. IEEE Computer Society.

[21] R. Santelices and M. J. Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *ISSTA*, pages 195–206, 2010.

[22] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Proc. HVC*, volume 4383 of *LNCS*, pages 166–182. Springer, 2007.

[23] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE-13*, pages 263–272, New York, NY, USA, 2005. ACM.

[24] S. Siegel, A. Mironova, G. Avrunin, and L. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *ISSTA*, pages 157–168. ACM, 2006.

[25] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu. CORAL: solving complex constraints for Symbolic Pathfinder. *Proc. NFM*, 2011.

[26] N. Tillmann and J. De Halleux. Pex: white box test generation for. NET. In *TAP*, pages 134–153. Springer-Verlag, 2008.

[27] A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *Proc. ISSTA*, pages 97–107, New York, NY, USA, 2007. ACM Press.

[28] W. Visser, C. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *ISSTA*, pages 37–48. ACM New York, NY, USA, 2006.

[29] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. *TACAS*, pages 365–381, 2005.