Boosting Concolic Testing via Interpolation

Joxan Jaffar, Vijayaraghavan Murali
National University of Singapore
{joxan, m.vijay}@comp.nus.edu.sg

Jorge A. Navas
The University of Melbourne
jorge.navas@unimelb.edu.au

Abstract. Concolic testing has been very successful in automatically generating test inputs for programs. However one of its major limitations is path-explosion that limits the generation of high coverage inputs. Since its inception several ideas have been proposed to attack this problem from various angles: defining search heuristics that increase coverage, caching of function summaries, pruning of paths using static/dynamic information etc. We propose a new and complementary method based on *interpolation*, that greatly mitigates path-explosion by subsuming paths that can be guaranteed to not hit a bug. We discuss new challenges in using interpolation that arise specifically in the context of concolic testing. We experimentally evaluate our method with different search heuristics using Crest, a publicly available concolic tester.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution, testing tools*

General Terms
Algorithms, Reliability

Keywords
Concolic testing, interpolation, symbolic execution

1. INTRODUCTION
Testing is the most commonly used method to ensure software quality. It executes a given program with some inputs and the objective is then to find bugs or validate the program with respect to the given inputs. Traditionally, testing was carried out using manually generated inputs which became cumbersome and ineffective. Random testing alleviates this problem by generating random test inputs, but suffers from poor code coverage. Recently, more intelligent methods [15, 9, 5, 3] based on *concolic testing*, a variant of symbolic execution, have emerged, that generate inputs by systematically exploring program paths attempting to increase coverage.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491425

The main idea of concolic testing is to execute the program simultaneously with concrete values and symbolic values. When the program is executed, symbolic constraints along the executed path are collected in a formula called *path condition*. Then, a branch is picked and negated from the path condition resulting in a new formula which is then fed to a constraint solver to check for satisfiability. If it is satisfiable, concrete test inputs are generated to follow the new feasible path. If it is unsatisfiable, the new path is infeasible and another branch has to be picked to be negated. This way concolic testing attempts to improve the poor code coverage of random testing. A key characteristic of concolic testing is that path conditions can be simplified using concrete values whenever the decidability of their symbolic constraints goes beyond the capabilities of the underlying constraint solver.

One major problem with concolic testing is that there are in general an exponential number of paths in the program to explore, resulting in the so-called *path-explosion* problem. Recently, several methods have been proposed to attack this problem from various angles: using heuristics focused on branch coverage [3], function summaries [8], using static/dynamic program analysis [2] and so on. We propose a new method based on *interpolation*, largely complementary to existing approaches, that significantly mitigates path-explosion by pruning a potentially exponential number of paths that can be guaranteed to not encounter a bug.

Our method, inspired by [12], aims at assisting concolic testing by making use of the concept of *interpolation* [7] interleaved with the concolic execution process. The use of interpolation for pruning paths in the context of symbolic execution is well-known (see e.g., [12, 14]). The idea is as follows: first, assume that the program is annotated with certain bug conditions of the form "if C then **bug**", where if the condition C evaluates to true along a path, the path is buggy. Then, whenever an unsatisfiable path condition is fed to the solver, an interpolant is generated at each program point along the path. The interpolant at a given program point can be seen as a formula that *succinctly* captures the reason of infeasibility of paths at the program point. In other words it succinctly captures the reason why paths through the program point are not buggy. As a result, if the program point is encountered again through a different path such that the interpolant is implied, the new path can be *subsumed*, because it can be guaranteed to not be buggy. The exponential savings are due to the fact that not only is the new path subsumed, but also the paths that this new path would spawn by negating its branches.

Unfortunately, methods such as [12, 14] cannot be used directly for concolic testing due to several challenges. First, the soundness of these methods relies on the assumption that an interpolant at a node has been computed after exploring the entire "tree" of paths that arise from the node. In concolic testing, this assumption is invalid as the tester can impose an arbitrary search order. For example, concolic testers such as Crest [3] and KLEE [4] use often many heuristics that may follow a random walk through the search space, thus making this method unsound. To address this problem, we need to keep track of nodes whose trees have been explored fully (in which case we say the node is annotated with a *full-interpolant*) or partially (similarly, a *half-interpolant*).

Under this new setting, only nodes with full-interpolants are capable of subsumption in a sound manner. As a result, the amount of subsumption depends on how often nodes get annotated with full-interpolants from the paths explored by the concolic tester. Unfortunately our benchmarks in Section 6 showed that the above method by itself results in very few nodes with full-interpolants, thereby providing poor benefit to the concolic tester, because the tester rarely explores the entire tree of paths arising from a node. Hence, an important challenge now is to "accelerate" the formation of full-interpolants in order to increase subsumption. For this, we introduce a novel technique called *greedy confirmation* that performs limited path exploration (i.e., execution of a few extra paths) by itself, guided by subsumption, with an aim to produce a full-interpolant at nodes currently annotated with a half-interpolant. It is worth mentioning that this execution of few paths is done without interfering with the search order of the concolic tester. This technique ultimately resulted in a significant increase in subsumption for our benchmarks, and is vital for the effectiveness of our method.

We implemented our method and compared it with a publicly available concolic tester, Crest [3]. We found that for the price of a reasonable overhead to compute interpolants, a large percentage of paths executed by those heuristics can be subsumed thereby increasing their coverage substantially.

2. RELATED WORK

The main innovation introduced by concolic testing (originally presented in DART [9] and Cute [15]) was the fact that concrete inputs can be generated based on some intelligent decision by symbolically negating one of the executed branches. Since the seminal papers of [9, 15] many works have been published improving concolic testing in different ways. We limit our discussion to related works that attempt at mitigating the scalability issues in concolic testing due to the exponential numbers of paths.

Recent extensions (e.g., EXE [5], Crest [3], Sage [10] and KLEE [4]) have tried to address this challenge by using novel heuristics to guide the exploration of paths improving the naive depth-first search strategy originally used in DART. They target *branch coverage* (i.e., number of branches evaluated to true and false) rather than *path coverage.* Although branch coverage does not suffer from path-explosion it is understood that the ultimate goal of the concolic tester is path coverage. Branch coverage is just an inexpensive measure of the quality of a test suite, and a good branch coverage is more of a necessary requirement for quality than sufficient. The main difference with the above methods is that we focus on path coverage rather than branch coverage while respecting those search strategies.

Another interesting line of research has addressed the caching of *function summaries* as a way of reducing the exponential number of paths (e.g., SMART [8] and [1]). Our method performs function inlining (not intelligent interprocedurally) while these methods execute the DART algorithm to generate reusable summaries (not intelligent intraprocedurally). This suggests that both approaches are orthogonal and could work together to benefit from each other.

The closest related tester to ours in the line of pruning paths is [2]. This method eliminates redundant paths by analysing the values read and written in memory by the program. The main idea is to prune paths that only differ in program variables that are not subsequently read (i.e., *dead variables*). We share with them the high-level idea of removing irrelevant information in order to increase the chances of subsumption, but the similarity ends there. The most important difference is that [2] defines "irrelevant information" using live (dead) variables so the subsumption test is simply a subset operation. We use interpolation to prune paths so our subsumption test involves logical entailment checks, which are more expensive. However, interpolants are much more powerful for subsumption, which can result often in a greater amount of pruned paths. We exemplify these differences with [2] through an example in Section 3.

Finally, as mentioned before, we have been clearly inspired by [12] in the idea of interpolating infeasible paths which has been also applied in [14]. However, [12, 14] assume full control of the search space by performing a DFS-traversal to compute their interpolants, which ensures an interpolant at a node always represents the entire tree of paths below a node. This is the key to making sound subsumption. This assumption is no longer valid in concolic testing since the tester controls the search space using some heuristic which may not be DFS. Interestingly, Crest [3] compares different heuristics for concolic testing and concluded that DFS is actually the worst in terms of branch coverage. Thus, [12, 14] are not readily suitable for concolic testing. More importantly, as we will see in Section 6, even if somehow the DFS-restriction is lifted in these methods (e.g., by augmenting them with half and full-interpolants), it is not enough to provide a reasonable benefit to concolic testing, as they scale poorly without greedy confirmation.

3. RUNNING EXAMPLE

Consider the program in Fig. 1, where a `read()` call signals the concolic tester to generate a concrete input. In this case, the inputs are used to decide the program's control flow. Assume initially the concolic tester generates a positive value for both inputs. This drives the tester down the path $\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_5 \cdot \ell_6 \cdot \ell_8$, which is shown by the left-most path in the program's symbolic execution tree. A symbolic execution tree represents the set of paths traversed by the program where each node corresponds to a program location and an edge between two nodes corresponds to the program

```
ℓ1 int s=0;
ℓ2 if (read())
ℓ3   s=s+1;
ℓ4 if (read())
ℓ5   s=s+2;
ℓ6 if (s>3)
ℓ7   bug;
ℓ8
```

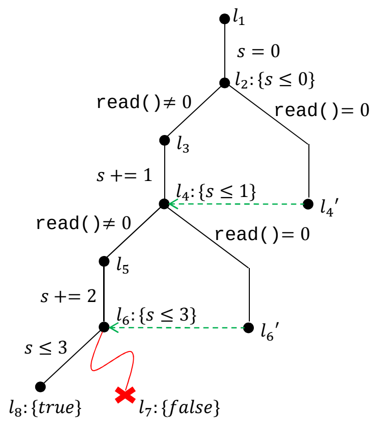


Figure 1: A program and its symbolic execution tree

transition.

Now, the concolic tester provides us this path, on which we perform symbolic execution and annotate it with interpolants in a backward manner. At ℓ_8 the path is not buggy and there is no infeasibility, therefore the interpolant *true* is stored there (denoted by $\ell_8 : \{true\}$). Propagating this to ℓ_6 we note that there is another branch (to ℓ_7) that has not been explored, so we annotate the interpolant *true* at ℓ_6 as a *half-interpolant* to denote this fact. Once a node has been annotated with a half-interpolant, we stop and give control back to the concolic tester.

Assume now the concolic tester attempts to "turn-around" at ℓ_6 into the path $\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_5 \cdot \ell_6 \cdot \ell_7$. This path is infeasible as its path condition $s = 0 \wedge s' = s + 1 \wedge s'' = s' + 2 \wedge s'' > 3$ is unsatisfiable (for simplicity we omitted the `read` constraints). Now we generate interpolants for this path by first annotating $\ell_7 : \{false\}$ because it is an unreachable node. Propagating this back to ℓ_6 through the label $s > 3$, we obtain the interpolant $s \le 3$[1]. We now note all paths from ℓ_6 have been explored, so we conjoin all interpolants at ℓ_6 ($true \wedge s \le 3$) annotating it with the *full-interpolant* $s \le 3$, denoting that the entire tree of paths under ℓ_6 has been explored.

Now, when the concolic tester generates a zero for the second `read()` and executes the path $\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_6'$, we check if the path condition at ℓ_6', $s = 0 \wedge s' = s + 1$ implies the full-interpolant at ℓ_6, $s' \le 3$ (after proper renaming). It does, so we can guarantee the entire tree of paths below ℓ_6' to not be buggy and *subsume* it. The exponential savings is because we saved the concolic tester from executing a potentially exponential number of paths in the tree under ℓ_6' (in this case, two paths, but in general exponential). Importantly, note that only full-interpolants are capable of subsumption and the amount of savings provided by our method is directly proportional to the number of full-interpolants formed from exploring entire trees of paths.

Unfortunately, the method discussed so far has a catch. We conveniently assumed the concolic tester would execute the paths in that specific order, which resembles a depth-first search (DFS). If after executing the first path the tester did not attempt to turn-around at ℓ_6 (thus leaving ℓ_6 with a half-interpolant), ℓ_6 would not have been able to subsume ℓ_6'. Even worse is the fact that because of the half-interpolant at ℓ_6 all its ancestors along the path also become incapable of subsumption, thereby losing a great amount of savings.

Thus, we need to "accelerate" the formation of full-interpolants instead of simply relying on the concolic tester to explore the paths. However a challenge in concolic testing is that whatever technique is employed for this purpose, it must always stay *proportional* to the executed path, that is, not become intractable. For this, we introduce a technique called *greedy confirmation* that is both tractable (proportional to the path length in the worst-case) and accelerates the formation of full-interpolants resulting in a substantial increase in subsumption.

The basic idea is while backtracking along a path, once we encounter a node with another branch that has not been explored by the concolic tester, instead of simply annotating the node with a half-interpolant and halting the process, we explore the other branch ourselves (while the concolic tester is waiting to generate the next path) and attempt to confirm whether a full-interpolant can be generated from it, so that we can annotate the node as full and continue the backward propagation. However, the sub-tree under the other branch could be exponentially large, in which case we must avoid exploring it to remain tractable. Hence we introduce a restriction: while exploring the sub-tree, each program point is allowed to be explored at most once. If a program point is visited along more than one path, we demand that it be subsumed. If not, we declare that the greedy confirmation process failed at the original branch node, which will remain annotated with a half-interpolant.

The reasoning behind the restriction is as follows. During greedy confirmation, we want to give each program point at least one chance to become subsumed, so we must allow at least one visit to each program point. However, we do not want to resort to a full search within the sub-tree, which could become intractable. Thus, this restriction ensures that the "search" is linearly bounded by the longest path in the program[2]. The motivation behind this technique is that the difference between the two branching sub-trees might not affect the bug condition and so one tree can quickly subsume the other.

Let us see how greedy confirmation works on the example. When backtracking along the first path, $\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_5 \cdot \ell_6 \cdot \ell_8$, once we reach ℓ_6 with the interpolant *true*, we trigger greedy confirmation and attempt to explore the other tree under ℓ_6. We immediately notice the other branch to ℓ_7 is infeasible, and annotate it with $false$. Propagating this back to ℓ_6 we get the *full*-interpolant $s \le 3$, which can now be propagated back further. Hence at ℓ_5 we get the full-interpolant $s \le 1$. Propagating this to ℓ_4 makes it a half-interpolant since there is another branch from ℓ_4 to ℓ_6'. Triggering greedy

[1]Note that for this example we use interpolants based on weakest preconditions, but our method is not limited to them and any interpolation method can be used.

[2]We tried restricting to two, three and other constant instances of each program point, but we found no difference in our benchmarks.

confirmation, this time at ℓ_4, we notice that ℓ'_6 is subsumed by ℓ_6 with the full-interpolant $s \leq 3$. Propagating this to ℓ_4 and conjoining it with the existing half-interpolant, we get $s \leq 3 \wedge s \leq 1$, or simply $s \leq 1$ at ℓ_4, which is now a *full*-interpolant that can be propagated back. Similar reasoning at ℓ_2 subsumes a large tree of paths under ℓ'_4 resulting in the full-interpolant $s \leq 0$ at ℓ_2.

Now, in this contrived example, if the concolic tester executes any path in the program, we are able to subsume it immediately at ℓ_2. Essentially, greedy confirmation has pushed subsumption from happening at lower levels in the symbolic execution tree to upper levels. A simple but effective optimisation is if greedy confirmation failed at a node, thereby leaving it with a half-interpolant, we can simply halt the annotation process because there is no use propagating a half-interpolant to the node's parents.

Finally, note importantly that this example does not contain any dead variables, so techniques like [2] will not be able to prune any path, whereas with interpolation we are able to.

Remark. The correctness of our method relies on the fact that whenever we subsume a path (i.e., we skip its execution) we can ensure that the path will not be buggy (see Theorem 5.1 in Sec. 5). This guarantee is only feasible if the program is annotated with assertions. Without them we have no basis to make such a guarantee and subsume the path. A key observation is that without assertions each path is unique because at least it will differ in one branch from the rest of paths, and hence, there is no hope for boosting the concolic execution. However, with an assertion many paths can be considered equivalent and this allows our method reporting savings. In conclusion, our method is only effective if we consider programs annotated with assertions. We believe that this step can be done automatically and it is not a big hassle in practice.

4. PRELIMINARIES

Syntax. Most of the formalism used here has been borrowed from [11, 13]. We restrict our presentation to a simple imperative programming language where all basic operations are either assignments or assume operations, and the domain of all variables are integers. The set of all program variables is denoted by *Vars*. An *assignment* $\mathsf{x} := \mathsf{e}$ corresponds to assign the evaluation of the expression e to the variable x. In the *assume* operator, $\mathsf{assume}(\mathsf{c})$, if the Boolean expression c evaluates to *true*, then the program continues, otherwise it halts. The set of operations is denoted by *Ops*. We then model a program by a *transition system*. A transition system is a quadruple $\langle \Sigma, I, \longrightarrow, O\rangle$ where Σ is the set of states and $I \subseteq \Sigma$ is the set of initial states. $\longrightarrow \subseteq \Sigma \times \Sigma \times Ops$ is the transition relation that relates a state to its (possible) successors executing operations. This transition relation models the operations that are executed when control flows from one program location to another. We shall use $\ell \xrightarrow{\mathsf{op}} \ell'$ to denote a transition relation from $\ell \in \Sigma$ to $\ell' \in \Sigma$ executing the operation $\mathsf{op} \in Ops$. Finally, $O \subseteq \Sigma$ is the set of final states.

Symbolic Execution. A *symbolic state* σ is a triple $\langle \ell, s, \Pi\rangle$. The symbol $\ell \in \Sigma$ corresponds to the current program location. For clarity of presentation in our algorithm, we will use special symbols for initial location, $\ell_{\mathsf{start}} \in I$, final location, $\ell_{\mathsf{end}} \in O$, and bug location $\ell_{\mathsf{bug}} \in O$ (if any). W.l.o.g we assume that there is only one initial, final, and bug location in the transition system.

The symbolic store s is a function from program variables to terms over input symbolic variables. Each program variable is initialised to a fresh input symbolic variable. This is done by the procedure *init_store()*. The *evaluation* $[\![c]\!](s)$ of a constraint expression c in a store s is defined recursively as usual: $[\![v]\!](s) = s(v)$ (if $c \equiv v$ is a variable), $[\![n]\!](s) = n$ (if $c \equiv n$ is an integer), $[\![e \; \mathsf{op}_r \; e']\!](s) = [\![e]\!](s) \; \mathsf{op}_r \; [\![e']\!](s)$ (if $c \equiv e \; \mathsf{op}_r \; e'$ where e, e' are expressions and op_r is a relational operator $<, >, =, !=, >=, <=$), and $[\![e \; \mathsf{op}_a \; e']\!](s) = [\![e]\!](s) \; \mathsf{op}_a \; [\![e']\!](s)$ (if $c \equiv e \; \mathsf{op}_a \; e'$ where e, e' are expressions and op_a is an arithmetic operator $+, -, \times, \ldots$).

Finally, Π is called *path condition*, a first-order formula over the symbolic inputs that accumulates constraints which the inputs must satisfy in order for an execution to follow the particular corresponding path. The set of first-order formulas and symbolic states are denoted by *FO* and *SymStates*, respectively. Given a transition system $\langle \Sigma, I, \longrightarrow, O\rangle$ and a state $\sigma \equiv \langle \ell, s, \Pi\rangle \in SymStates$, the symbolic execution of $\ell \xrightarrow{\mathsf{op}} \ell'$ returns another symbolic state σ' defined as:

$$\sigma' \triangleq \begin{cases} \langle \ell', s, \Pi \wedge [\![c]\!](s)\rangle & \text{if } \mathsf{op} \equiv \mathsf{assume}(\mathsf{c}) \text{ and} \\ & \Pi \wedge [\![c]\!](s) \text{ is satisfiable} \\ \langle \ell', s[x \mapsto [\![e]\!](s)], \Pi\rangle & \text{if } \mathsf{op} \equiv \mathsf{x} := \mathsf{e} \end{cases} \quad (1)$$

Note that Equation (1) queries a constraint solver for satisfiability checking on the path condition. We assume the solver is sound but not necessarily complete. That is, the solver must say a formula is unsatisfiable only if it is indeed so.

Abusing notation, given a symbolic state $\sigma \equiv \langle \ell, s, \Pi\rangle$ we define $[\![\sigma]\!] : SymStates \to FO$ as the formula $(\bigwedge_{v \;\in\; Vars} [\![v]\!](s)) \wedge \Pi$ where *Vars* is the set of program variables.

A *symbolic path* $\pi \equiv \sigma_0 \cdot \sigma_1 \cdot \ldots \cdot \sigma_n$ is a sequence of symbolic states such that $\forall i \bullet 1 \leq i \leq n$ the state σ_i is a *successor* of σ_{i-1}, denoted as $\mathsf{SUCC}(\sigma_{i-1}, \; \sigma_i)$[3]. A path $\pi \equiv \sigma_0 \cdot \sigma_1 \cdot \ldots \cdot \sigma_n$ is *feasible* if $\sigma_n \equiv \langle \ell, s, \Pi\rangle$ such that $[\![\Pi]\!](s)$ is satisfiable. If $\ell \in O$ and σ_n is feasible then σ_n is called *terminal* state. Otherwise, if $[\![\Pi]\!](s)$ is unsatisfiable the path is called *infeasible* and σ_n is called an *infeasible* state. If there exists a feasible path $\pi \equiv \sigma_0 \cdot \sigma_1 \cdot \ldots \cdot \sigma_n$ then we say σ_k $(0 \leq k \leq n)$ is *reachable* from σ_0 in *k steps*. We say σ'' is reachable from σ if it is reachable from σ in some number of steps. A *symbolic execution tree* contains all the execution paths explored during the symbolic execution of a transition system by triggering Equation (1). The nodes represent symbolic states and the arcs represent transitions between states.

Concolic Testing. We first define a *concolic path* p as the path executed by the concolic tester represented as a sequence of program locations $\ell_1 \cdot \ldots \cdot \ell_n$ and define the

[3]W.l.o.g, we assume each state has at most two successors.

```
GenericConcolic(program $P$, path $p$)
1: while termination conditions are not met do
2:     $b_i$ := pick a branch from $p$
3:     $p'$ := construct a path passing through
               the branches $b_0, \ldots, b_{i-1}, \overline{b_i}$
4:     if $\exists$ satisfying assignment $I$ forcing $P$ through $p'$ then
5:         $q \leftarrow$ ConcretePath($P$,$I$)
6:         process $q$ by either $p \leftarrow q$ or
           GenericConcolic($P$,$q$)
7:     endif
8: endwhile
```

Figure 2: A Generic Concolic Tester

transition relation $\ell \xrightarrow{\mathsf{op}} \ell'$ as before. In order to manipulate concolic paths we also define *prefix*(p,ℓ) of a path p wrt a location ℓ as the prefix up to ℓ without including ℓ. We also define *constraints*(p) that maps a concolic path into a formula representing the conjunction of the symbolic constraints along the path. Of course, this formula is properly renamed (i.e., SSA form) to take into account variables that are redefined more than once.

We now show in Fig. 2 a generic algorithm that performs concolic testing as described in [3]. The algorithm is generic in the sense that it can be parameterised with different search strategies by simply choosing different heuristics to pick a branch at line 2. The algorithm is instantiated with the program P and an initial concolic path p, usually chosen by running the program with random inputs. In line 2, a branch b_i is chosen from the path p based on the heuristic used. In line 3, a new path p' is built by keeping the prefix up to b_i and negating the constraints at the branch b_i. In line 4 the symbolic constraints associated with p' are then fed to a constraint solver to check for satisfiability. If they are unsatisfiable, the algorithm returns to line 2 and picks another branch to negate. Otherwise, an assignment I (concrete inputs) is extracted from the solver which is used to guide the next concrete path along the negated branch. This is done using the call to ConcretePath(P,I) in line 5. Once the new path, say q, is executed, depending on the heuristic, it is processed by either replacing the old path p with q or by making a recursive call to GenericConcolic with q in line 6. The entire process continues until some termination condition is met (usually a fixed number of iterations or recursive call depth) shown in line 1.

5. CONCOLIC TESTING WITH INTERPOLATION

We now present our symbolic execution based algorithm that runs synchronised with GenericConcolic and helps the concolic testing strategy mitigate the path-explosion problem. First, we give few definitions critical to our algorithm:

Definition 1 (Craig Interpolant). *Given two formulas A and B such that $A \wedge B$ is unsatisfiable, a Craig* interpolant *[7] (INTP(A,B)) is another formula Ψ such that (a) $A \models \Psi$, (b) $\Psi \wedge B$ is unsatisfiable, and (c) all variables in Ψ are common to A and B.*

An interpolant allows us to remove irrelevant information in A that is not needed to maintain the unsatisfiability of $A \wedge B$. That is, the interpolant captures the essence of the reason of unsatisfiability of the two formulas. Efficient interpolation algorithms exist for quantifier-free fragments of theories such as linear real/integer arithmetic, uninterpreted functions, pointers and arrays, and bitvectors (e.g., see [6] for details) where interpolants can be extracted from the refutation proof in linear time on the size of the proof.

Definition 2 (Full and half interpolants). *An interpolant annotated at a symbolic state σ is a* full-interpolant *if either:*

(a) σ is a leaf node (terminal, infeasible or subsumed) in the symbolic tree, or

(b) $\forall \sigma'$ such that SUCC(σ,σ'), σ' is annotated with a full-interpolant.

An interpolant that is not full is called a half-interpolant*.*

Definition 3 (Subsumption check). *Given a current symbolic state $\sigma \equiv \langle \ell, s, \cdot \rangle$ and an already explored symbolic state $\sigma' \equiv \langle \ell, \cdot, \cdot \rangle$ annotated with the interpolant Ψ, we say σ is* subsumed *by σ' (SUBSUME$(\sigma, \langle \sigma', \Psi \rangle)$) if Ψ is a full-interpolant and $[\![\sigma]\!](s) \models \Psi$.*

To understand the intuition behind the subsumption check, it helps to know what a full-interpolant at a node actually represents. A full-interpolant Ψ at a node σ' succinctly captures the reason of infeasibility of all infeasible paths in the symbolic tree rooted at σ' (let us call this tree T_1). Then, if another state σ at ℓ implies Ψ, it means the tree rooted at σ (say, T_2) has exactly the same, or more, infeasible paths compared to T_1. In other words, T_2 has exactly the same, or *less feasible paths* compared to T_1. Since T_1 did not contain any feasible path that was buggy, we can guarantee the same for T_2 as well, thus subsuming it. Note that if Ψ was a half-interpolant, we cannot guarantee this because T_1 has not been fully explored.

5.1 Symbolic execution of paths with interpolants

We first present the algorithm SymExec (Figure 3) that takes a path chosen by the concolic tester and executes it symbolically in order to annotate each program point in it with interpolants. A collection of such annotated paths implicitly represents the symbolic execution tree. Then, we will present GenericConcolicWithPruning, a modified version of GenericConcolic that can prune paths using the annotated symbolic execution tree.

SymExec takes as arguments a symbolic state σ and a path p. Typically, the state σ refers initially to the first location in p. It is also assumed that all procedures have access to the program's original transition system P, which can considered a global variable to the algorithm. The actual object of interest is the annotation done by the procedure which is persistent across multiple calls to it. For the sake of simplicity, ignore the gray box which we will come to later.

```
SymExec($\sigma \equiv \langle \ell, \cdot, \cdot \rangle$, $p$)
1:  if TERMINAL($\sigma$) then $\langle \Psi, \mathsf{f} \rangle$ := $\langle true, \mathsf{full} \rangle$
2:  else if INFEASIBLE($\sigma$) then $\langle \Psi, \mathsf{f} \rangle$ := $\langle false, \mathsf{full} \rangle$
3:  else if $\exists\ \sigma' \equiv \langle \ell, \cdot, \cdot \rangle$ annotated with $\langle \Psi', \mathsf{full} \rangle$ such that SUBSUME($\sigma, \langle \sigma', \Psi' \rangle$) then
4:     $\langle \Psi, \mathsf{f} \rangle$ := $\langle \Psi', \mathsf{full} \rangle$
5:  else $\langle \Psi, \mathsf{f} \rangle$ := UnwindPath($\sigma, p$)
6:  endif
7:  annotate $\sigma$ with $\langle \Psi, \mathsf{f} \rangle$
8:  if f $\neq$ full then halt

UnwindPath($\sigma \equiv \langle \ell, s, \Pi \rangle$, $p$)
1:  $\Psi$ := existing interpolant at $\sigma$
2:  $\sigma' \triangleq \begin{cases} \langle \ell', s, \Pi \wedge [\![c]\!](s) \rangle & \text{if } \ell \xrightarrow{\mathsf{op}} \ell' \in p \wedge \mathsf{op} \equiv \mathsf{assume(c)} \\ \langle \ell', s[x \mapsto S_x], \Pi \rangle & \text{if } \ell \xrightarrow{\mathsf{op}} \ell' \in p \wedge \mathsf{op} \equiv \mathsf{x := e} \text{ and } S_x \text{ fresh variable} \end{cases}$
3:  SymExec($\sigma', p$) and let the resulting annotation at $\sigma'$ be $\langle \Psi', \cdot \rangle$
4:  $\Psi$ := $\Psi \wedge$ INTP($constraints(prefix(p, \ell)), constraints(\ell \cdot \ell') \wedge \neg\ \Psi'$)
5:  if $\exists \sigma'' \equiv \langle \ell'', \cdot, \cdot \rangle$ such that $\sigma''$ is not annotated $\langle \cdot, \mathsf{full} \rangle$ and SUCC($\sigma, \sigma''$) then
6:     GreedyConfirmation($\sigma''$) and let the resulting annotation at $\sigma''$ be $\langle \Psi'', \mathsf{f}'' \rangle$
7:     if f'' = full then
8:        $\Psi$ := $\Psi \wedge$ INTP($constraints(prefix(p, \ell) \cdot \ell), constraints(\ell \cdot \ell'') \wedge \neg\ \Psi''$)
9:     endif
10: endif
11: $\mathsf{f} \equiv \begin{cases} \mathsf{full} & \text{if } \forall \sigma''' \text{ s.t. } \mathsf{SUCC}(\sigma, \sigma'''), \sigma''' \text{ is annotated with } \langle \cdot, \mathsf{full} \rangle \\ \mathsf{half} & \text{otherwise} \end{cases}$
12: return $\langle \Psi, \mathsf{f} \rangle$
```

Figure 3: Symbolic execution with interpolation along a path

First, lines 1-4 handle the three possible base cases for the symbolic execution of a path: terminal, infeasible and subsumed. In line 1, the function TERMINAL checks if $\ell = \ell_{\mathsf{end}}$. If yes, the path can be fully generalised returning the full-interpolant *true*, since it is feasible. In line 2, the function INFEASIBLE checks if the path condition Π of σ is unsatisfiable. If yes, again the path can be fully generalised, but this time to $false$ since it is infeasible. Finally, line 3 checks if there is another state σ' that can subsume σ, using the function SUBSUME which implements Definition 3. If yes, we can simply annotate σ with the (full) interpolant of σ' (line 4).

If the three base cases described above are not applicable, the algorithm executes symbolically the next location of the path by calling the procedure UnwindPath. This procedure, at line 1, obtains any interpolant that may be annotated at σ (it can be assumed that initially all symbolic states are annotated with the default interpolant $\langle true, \mathsf{half} \rangle$). In line 2, it executes one symbolic step along the path and calls the main procedure SymExec with the new successor state σ' (line 3). This mutually recursive call will in the end annotate σ' with an interpolant, say Ψ'. In line 4, it computes the interpolant for the current state σ, using Ψ' and the constraints along the transition $\ell \xrightarrow{\mathsf{op}} \ell'$ and then conjoining the result with any existing interpolant at σ. Finally, in line 11, it computes whether Ψ is a full or half-interpolant by implementing Definition 2. For example, consider the case where $constraints(prefix(p, \ell)) \equiv x_0 = 1 \wedge x_1 = x_0 + 3$, $constraints(\ell \cdot \ell') \equiv x_2 = x_1 + 2$, and $\Psi' = x_2 \leq 10$ (after proper renaming). The call at line 11 will generate an interpolant whose variables must be common to $constraints(prefix(p, \ell))$ and $constraints(\ell \cdot \ell')$ together with Ψ'. Thus, it can only include x_1. An interpolant generated by MathSAT [6] would be $x_1 \leq 4$. A weaker interpolant is $x_1 \leq 8$ which can be computed by weakest preconditions.

Returning back the tuple $\langle \Psi, \mathsf{f} \rangle$ to the caller SymExec, at line 7, it performs the actual annotation of σ. Finally, in line 8, SymExec checks if it just annotated the state with a full interpolant. If not, it halts in a normal manner and returns control to the concolic tester (this can be seen as a system-wide halt and is also notified to UnwindPath)[4]. The reason for halting is that there is no use propagating the half interpolants to parent nodes, because they will still remain as half interpolants and be of no use for subsumption. In other words, the algorithm only propagates full interpolants to parent nodes. This is a simple but very effective optimisation. Note however, that the annotation of states done so far is still persistent. More importantly, a half interpolant at a node now could get converted to a full interpolant later when all successors of the node get full interpolants.

We now present a modified concolic tester GenericConcolicWithPruning, shown in Fig. 4, with the gray boxes highlighting the changes made in order to prune paths using our method. First, after picking a branch b_i to negate

[4]This halting may make the algorithm seem "impure" but we believe it makes it much easier to understand.

```
GenericConcolicWithPruning(program $P$, path $p$)
1:  while termination conditions are not met do
2:      $b_i :=$ pick a branch from $p$
3:      $p' :=$ construct a path passing through
                the branches $b_0, \ldots, b_{i-1}, \overline{b_i}$
4:      if IsSubsumedPath($p'$) then continue
5:      endif
6:      if $\exists$ satisfying assignment $I$ forcing $P$ through $p'$ then
7:          $q \leftarrow$ ConcretePath($P$,$I$)
8:          SymExec($\sigma_0 \equiv \langle \ell_{\mathsf{start}}, \mathit{init\_store()}, \mathit{true} \rangle, q$)
9:          process $q$ by either $p \leftarrow q$ or
            GenericConcolicWithPruning($P$,$q$)
10:     endif
11: endwhile
```

Figure 4: A Generic Concolic Tester with Pruning

(line 2) and constructing the corresponding path p' that goes through it (line 3), the concolic tester makes a call to IsSubsumedPath with p' at line 4. This call queries the persistent annotated symbolic execution tree to check whether the path p' has been subsumed already. Note that this is just a look-up and hence does not involve symbolic execution. If yes, the tester can simply skip p' from being executed and continue, thus pruning a potentially exponential number of recursive calls at line 9 and all those paths it would have generated. The second change is that once a path has been *concolically* executed in line 7, the tester has to invoke our method so that we can annotate it with interpolants. Thus, line 8 makes a call to SymExec with the initial state of the path σ_0 and the path q that was executed.

5.2 Greedy Confirmation to accelerate formation of full-interpolants

The method discussed so far has laid out the framework to bring interpolation to concolic testing with the help of half and full-interpolants. The benefit provided by our technique relies heavily on the formation of full-interpolants from paths explored by the concolic tester (i.e., the amount of times f is assigned full in UnwindPath, line 11). Unfortunately the method explained so far does not perform well in practice (as we will see in Section 6) because relying only on the tester to explore paths results in poor formation of full-interpolants. This is because the tester's arbitrary search strategy seldom explores the entire tree of paths arising from a node, a requirement to annotate the node with a full-interpolant and enable it to subsume other nodes.

Hence, an important challenge to deal with in concolic testing is to "accelerate" the formation of full-interpolants. For this, we introduce a new concept called *greedy confirmation*. The basic idea was described in Section 3, here we explain it in technical terms (gray box in Fig. 3). If the call to SymExec (line 3 of UnwindPath) returned successfully, it means σ', the successor of σ, was annotated with a full-interpolant, say Ψ'. Now, we check (at line 5) if the other successor of σ (say, σ'') is also annotated with a full-interpolant. If not, then this half-interpolant, say Ψ'', is the one preventing Ψ from becoming a full-interpolant. Now, we greedily explore σ'' by ourselves in an attempt to confirm whether we can make Ψ'' a full-interpolant, so that we can immediately upgrade Ψ to full, thus enabling it to perform subsumption. The intuition behind this technique is that the difference between the trees below the two "siblings" σ' and σ'' might have no effect on the bug condition, and since the tree below σ' has been fully explored, thereby annotated with full-interpolants, it opens up opportunities to subsume nodes in the tree below σ''. Thus, we make the call to GreedyConfirmation in line 6.

GreedyConfirmation essentially performs symbolic execution of σ'' similar to SymExec. However, a key impediment is that the symbolic execution of σ'' must not become expensive and it should have a *proportional* cost to the length of path, otherwise we consider it intractable. However the tree under σ'' might be arbitrarily large and so we impose an important restriction to maintain tractability: each program point is allowed to be explored at most once during greedy confirmation of σ''. If a program point ℓ_k is visited in two states σ_{k_1} and σ_{k_2} then we demand that one of them be subsumed (not necessarily by the other). If it is not possible, then we declare that the invocation of greedy confirmation failed at σ'', which will not be annotated with a full-interpolant. This restriction ensures that in the worst case, greedy confirmation at any symbolic state is bounded linearly by the length of the longest path in the program.

Note that GreedyConfirmation does not need the help of the concolic tester to explore its paths. In fact, this is the whole point. Thus, it does not take as argument the path p, but instead explores paths on its own. Since GreedyConfirmation is almost identical to SymExec, for lack of space, we do not show any pseudo-code here. The only modification to SymExec is to keep track of counters for each visited program location that is not subsumed and stop if a counter becomes greater than one. Finally, to perform the symbolic step (line 3, UnwindPath), GreedyConfirmation must pick $\ell \xrightarrow{\mathsf{op}} \ell'$ from the program's transition system P instead of the path p.

Coming back to the description of UnwindPath, consider the call to GreedyConfirmation at line 6 produced the annotation $\langle \Psi'', \mathsf{f}'' \rangle$ at σ''. If suppose Ψ'' was a full-interpolant (i.e., greedy confirmation succeeded), then at line 8 we augment the interpolant Ψ at σ with this full-interpolant. More importantly, at line 11, f will be assigned full because both successors of σ (σ' and σ'') have been annotated with full-interpolants (Ψ' and Ψ'' respectively). This directly accelerates the formation of full-interpolants, resulting in more subsumption (line 3 of SymExec succeeding more often). In Section 6 we will see this greatly increases the savings provided for concolic testing, while still maintaining tractability.

Theorem 5.1. *When a symbolic state σ is subsumed by another state σ', the error location ℓ_{bug} is unreachable from σ.*

The proof follows trivially from the correctness of the algorithms described in [12, 14] and the fact we only subsume if the interpolant is full.

COROLLARY 5.2. *If* GENERICCONCOLIC *will execute a path p that leads to the bug location ℓ_{bug}, then* GENERICCONCOLICWITHPRUNING *will also execute p.*

Theorem 5.1 states that we never subsume a node from which the error location ℓ_{bug} would be reachable. Thus, it can be shown that we never prevent GENERICCONCOLICWITHPRUNING from executing a path that will reach ℓ_{bug}.

Finally, note that it is entirely possible that the concolic tester can detect infeasible paths after simplifying a complex constraint using concrete values that we cannot detect symbolically. Therefore if unsatisfiability cannot be determined symbolically, we cannot compute interpolants. As a result, the pruning of paths is limited only to those where unsatisfiability does not require reasoning about complex constraints. Moreover, it is possible for GreedyConfirmation to reach ℓ_{bug} during the exploration of a path in the presence of constraints whose unsatisfiability cannot be determined. In this case, we also declare that greedy confirmation failed and return the control to the concolic tester, since only the tester can decide the reachability of ℓ_{bug} in order to ensure zero tolerance for false alarms.

6. EXPERIMENTAL EVALUATION

We implemented our algorithm on the TRACER [13] framework for symbolic execution and modified the concolic tester Crest [3] to communicate with TRACER during its testing process. To achieve this, the algorithm in Fig. 3 was implemented in TRACER while Crest was slightly modified to implement the procedure in Fig. 4.

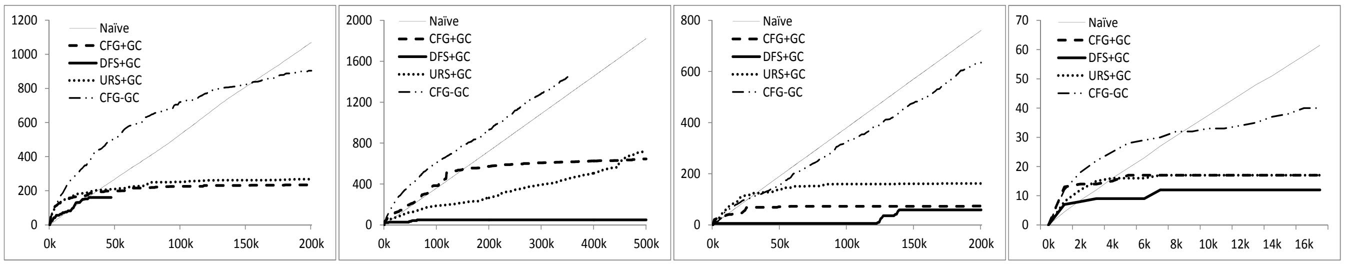
We conducted experiments on three strategies: Depth-First Search (DFS), Uniform Random Search (URS) and Control Flow Graph (CFG) directed search. DFS explores the paths in a depth-first order naturally forming full-interpolants in a bottom-up manner (even without the need for greedy confirmation) and maximising the benefit of subsumption, so it is the best-case strategy for our method. However according to [3], it is the worst strategy in terms of branch coverage, so we mainly provide it for completeness. CFG is a heuristic-based strategy that first computes the shortest distances between every basic block using the CFG of the program and decides on a "target" basic block to cover. Then, when a path is executed it chooses the next path by turning around at a branch along the path with the *least distance* to the target block. If it is unable to do so (due to infeasibility of constraints), it chooses the next closest branch to the target that is along the path, and so on. Once the target is reached, it randomly chooses a new target block to cover. It is worth noting that CFG-directed search was shown to be best strategy in terms of branch coverage by [3], so we consider it an illustrative concolic testing strategy. Finally, URS explores paths in a random order by choosing to turn-around at a random branch in a currently executed path. We included URS just to provide another example of a typical random concolic testing strategy. This random element in both CFG and URS hinders greatly with the formation of full-interpolants and is the main challenge for our algorithm to overcome.

As benchmarks, we used four programs from the ntdrivers-simplified category of SV-COMP'12: cdaudio, diskperf, floppy and kbfiltr. These programs range from 1000 to 2000 lines of code. In spite of their relative small sizes these programs have a large number of paths due to very complex control flows. Thus, we believe these programs can stress more the computation of interpolants and subsumption checks since symbolic paths contain a high number of constraints and it is often harder to reason about the infeasibility of these constraints. One of the consequences is that interpolants are stronger (in the logical sense) since they must consider multiple reasons of infeasibility and hence, they are less likely to be reusable. We ran three main experiments with them on an Intel 3.2Ghz with 2Gb memory.

A technical problem with the experiments is that if our algorithm prevented Crest from exploring subsumed trees, we would never know how many paths Crest *would have* executed in those trees (and subsequently the time taken for the same) had we not prevented it. Also, we do not want to meddle with the random number-based sequence in CFG and URS by forbidding Crest to execute certain paths. Hence for measurement purposes, we do not forbid Crest from exploring subsumed trees in our experiments, but instead take note when Crest is in subsumed trees and discard any measurements taken during that time.

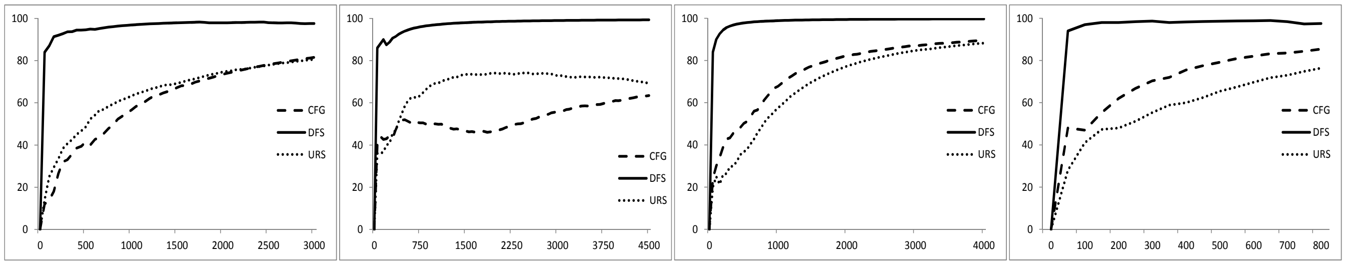
First experiment – Performance: At the outset we would like to measure the actual performance improvement provided by our method. For this we measure the time taken by each strategy of naive Crest (i.e., original version) and Crest aided by our method (i.e., algorithms in Figs. 3 and 4) to execute a target number of paths, chosen based on the size of the benchmark: 200k for cdaudio and floppy, 500k for diskperf and around 20k for kbfiltr (due to its smaller size). The results are shown in Fig. 5, where the number of executed paths is shown on the X-axis and the time taken (in seconds) to execute those paths is shown on the Y-axis. For naive Crest, we noticed that the time taken to execute a path is almost constant across all three strategies, so for simplicity we took the average time of the three strategies (shown as Naive in Fig. 5). When running with our method (and with greedy confirmation), we show the strategies separately labelled as CFG+GC, DFS+GC and URS+GC.

It can be seen that in each benchmark, naive Crest takes almost linear time to execute the target number of paths because the generation of each path involves simply one constraint solving, which takes almost constant time, and possibly a heuristic to choose the branch to negate, which is negligible. When aided by our method, Crest starts out a bit slower due to the overhead of interpolation, however after a certain time the benefits of interpolation (i.e., subsumption) start to payoff, ultimately reaching the target number of paths much faster. The magnitude of improvement depends on the strategy. As expected, our method works best with DFS, providing more than 10-20 times improvement, followed by CFG and URS with about 3-5 times improvement. For example, in diskperf, naive Crest takes on average 1700s to complete, whereas with our aid, CFG and URS take about 600s and DFS takes 50s. Note that every path subsumed in this experiment is a path executed by Crest that need not have been executed. More importantly, the *trend* of the graphs of Crest running with our algorithm indicates



(a) (b) (c) (d)

Figure 5: Timing for (a) cdaudio (b) diskperf (c) floppy (d) kbfiltr. X-axis: Paths, Y-axis: time in seconds



(a) (b) (c) (d)

Figure 6: Subsumption for (a) cdaudio (b) diskperf (c) floppy (d) kbfiltr. X-axis: Paths, Y-axis: % subsumption

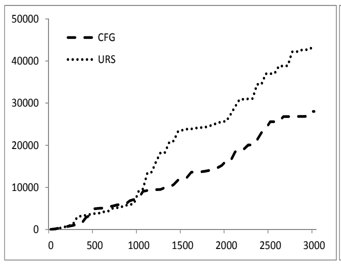
potentially exponential benefit in time over naive Crest.

To measure the effectiveness of greedy confirmation in our algorithm, we measured the timing of Crest with our algorithm but with greedy confirmation turned off (i.e., the gray box in Fig. 3 removed). Hence, the full-interpolants would be generated only by Crest exploring full sub-trees in the symbolic execution tree. We chose the canonical CFG strategy for this experiment, and its result is shown as CFG-GC (double-dotted line) in the graphs in Fig. 5. It can be seen immediately that the timing without greedy confirmation is much worse than otherwise. Especially for diskperf in Fig. 5(b), the timing is even slower than running naive Crest, because the overhead incurred due to interpolation does not payoff in subsuming other paths. In other benchmarks it pays off resulting in a benefit to Crest, but the amount of benefit is small compared to running CFG with greedy confirmation(CFG+GC). More importantly, the trend of CFG-GC does not appear to provide exponential benefit to Crest, if at all. This experiment shows that greedy confirmation is a indeed major contribution to the effectiveness of our algorithm and interpolation-based methods without it [14, 12] are not readily suitable for concolic testing.

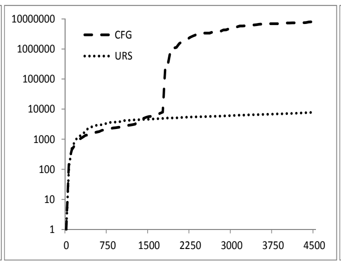
Second experiment – Subsumption: Now we would like to understand the basis on which the first experiment provided benefit. A good measure for this is the amount of subsumption that our method provides, i.e., the percentage of Crest paths that are subsumed. A thing to take note is that CFG and URS may repeat execution of some paths due to their random element which we cannot control. For this and the subsequent experiment, we do not include such paths in the calculations because a repeated path does not contribute to the percentage of subsumption. We included them in the previous experiment because they indeed contribute to execution time.

The results are shown in Fig. 6. In the graph of each benchmark, the X-axis represents the number of paths executed by Crest and the Y-axis represents the percentage of those paths subsumed. It can be clearly seen that DFS very quickly reaches almost 100% subsumption whereas the other two strategies CFG and URS show an increasing trend towards it, with DFS as their asymptote. This is in line with the first experiment where the magnitude of performance benefit was the greatest in DFS, followed by the other two. In diskperf we notice some noise, as CFG and URS fluctuate between 50-65% subsumption. The next experiment below suggests that this is because even though we subsume trees during this period, CFG and URS are not interested in executing paths in those trees. This is possibly why the magnitude of improvement was low (only around 2) for diskperf in Fig. 5(b). Theoretically, the rate of subsumption could decrease (even be zero) if Crest constantly avoids executing paths in subsumed trees, but we expect this to seldom happen in reality.

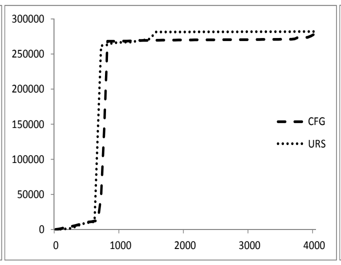
Third experiment – Extra path-coverage: Now we present a different view of the provided benefit. When we subsume a tree, we not only provide the direct benefit of saving the paths that Crest indeed executes in the tree (the first experiment), but also the indirect benefit of covering



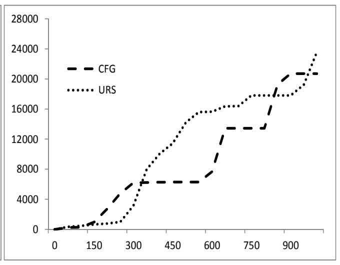
(a)



(b)



(c)



(d)

Figure 7: Extra coverage provided for (a) cdaudio (b) diskperf (c) floppy (d) kbfiltr by our method. X-axis: Crest path coverage, Y-axis: Additional path coverage from subsumption.

paths in the tree that Crest cannot cover within its budget. Although such paths are of low priority to the strategy's heuristics, they are provided "free of charge" by our method because the time we spend to subsume the whole tree is inclusive of these paths as well. This can be considered extra path coverage because Crest has no hope of executing them in its budget, but if its budget were longer it may execute them in future. Note that this experiment does not make sense for DFS, because when we subsume a tree, DFS would immediately proceed to execute *all* paths in the tree anyway.

In Fig. 7, we measure the number of such "free" paths per path executed by Crest. Note that we again do not include repeated paths in this experiment since they do not contribute to path coverage. In each graph, the X-axis shows the actual path coverage of Crest, and the Y-axis shows the extra path coverage obtained due to subsumption. The ratio of extra path coverage to Crest's actual path coverage varies greatly depending on the benchmark, from a magnitude of 10 in cdaudio to about 1000 in diskperf (note the logarithmic scale). Specifically, for the CFG strategy in diskperf, we subsume a huge number of trees around 1500 paths, but the previous experiment showed yet fluctuating percentage of subsumption around that time, indicating that CFG is not interested to execute paths in those subsumed trees.

In this experiment, by "taking credit" for entire subsumed trees, we measured the upper bound on the magnitude of benefit in path coverage that we can provide, a mean of 100. The lower bound, around 3 to 5, is dictated by the first experiment (although we did not explicitly measure it there, we can extract it from the timing), where we took credit only for the paths that Crest actually executed in the subsumed trees, within its budget. In general, one can expect the benefit we provide to lie somewhere in between, depending on the budget.

Fourth experiment – Terminating testing: Finally, we discuss a small but important experiment. We wanted to make a "pure" comparison of concolic testing with and without our method, notwithstanding the complications with measuring the number of subsumed paths, random number sequences and repeated paths we encountered in the previous experiments. In other words, we wanted to actively forbid Crest from exploring subsumed trees instead of letting it run and discarding measurements like before. The problem in doing this is that when terminating with a budget, the sequence of paths executed by naive Crest and our method would be different and hence, incomparable. However, if the testing process terminates having explored the entire search space (i.e., verified the program), the sequence of paths it took to do so does not matter.

We obtained a smaller non-buggy version of kbfiltr from the same benchmark suite and ran Crest's CFG strategy on it with and without our method, this time actively forbidding Crest from exploring subsumed trees. With our aid, Crest was able to completely verify the program in 20 seconds, whereas naive Crest was able to complete only after 256 seconds. This experiment shows concrete evidence that our method indeed accelerates a typical concolic testing strategy such as CFG towards more path coverage, in this case verifying the program much faster than otherwise. Unfortunately, we could not run this experiment on other benchmarks as they contain a prohibitive number of paths and concolic testing could not explore all of them in a reasonable amount of time making it not possible to make such an interesting comparison.

Remark. Although we focus on path coverage in this paper it is worthwhile to note that our method can be also used to improve *branch coverage*. In fact, in some of our preliminary experiments we targeted branch coverage and observed that we were achieving the same branch coverage but sometimes faster with our method than without. However, the problem of branch coverage is simpler than path coverage and hence, pruning is not always vital. In these cases, the overhead of interpolation and subsumption may not pay off.

7. CONCLUSION

We attacked the path-explosion problem of concolic testing by pruning redundant paths using interpolation. The challenge for interpolation in concolic testing is the lack of control of search order. To solve this, we presented the concept of half and full interpolants that makes the use of interpolants sound, and greedy confirmation that accelerates the formation of full-interpolants thereby increasing the likelihood of subsuming paths. We implemented our method and empirically presented its performance and path coverage gains.

8. REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *TACAS*, pages 367–381, 2008.

[2] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *TACAS*, pages 351–366, 2008.

[3] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. In *ASE*, pages 443–446, 2008.

[4] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, pages 209–224, 2008.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *CCS*, pages 322–335, 2006.

[6] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *TACAS'08*, pages 397–412, 2008.

[7] W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation*, 22, 1955.

[8] P. Godefroid. Compositional dynamic test generation. In M. Hofmann and M. Felleisen, editors, *34th POPL*, pages 47–54. ACM Press, 2007.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223, 2005.

[10] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.

[11] J. Jaffar, , J. Navas, and A. Santosa. Unbounded Symbolic Execution for Program Verification. In *RV 2011*, pages 396–411, 2011.

[12] J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *15th CP*, volume 5732 of *LNCS*. Springer, 2009.

[13] J. Jaffar, Vijayaraghavan Murali, J. Navas, and A. Santosa. TRACER: A Symbolic Execution Tool for Verification. In *CAV 2012*, pages 758–766, 2012.

[14] K. L. McMillan. Lazy annotation for program testing and verification. In T. Touili, B. Cook, and P. Jackson, editors, *22nd CAV*, volume 6174 of *LNCS*, pages 104–118. Springer, 2010.

[15] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13*, pages 263–272, 2005.