Software Model Checking via Summary-Guided Search

RUIJIE FANG, University of Texas at Austin, USA
ZACHARY KINCAID, Princeton University, USA
THOMAS REPS, University of Wisconsin—Madison, USA

In this work, we describe a new software model-checking algorithm called GPS. GPS treats the task of model checking a program as a directed search of the program states, guided by a compositional, summary-based static analysis. The summaries produced by static analysis are used both to prune away infeasible paths and to drive test generation to reach new, unexplored program states. GPS can find both proofs of safety and counter-examples to safety (i.e., inputs that trigger bugs), and features a novel two-layered search strategy that renders it particularly efficient at finding bugs in programs featuring long, input-dependent error paths. To make GPS refutationally complete (in the sense that it will find an error if one exists, if it is allotted enough time), we introduce an instrumentation technique and show that it helps GPS achieve refutation-completeness without sacrificing overall performance. We benchmarked GPS on a diverse suite of benchmarks including programs from the Software Verification Competition (SV-COMP), from prior literature, as well as synthetic programs based on examples in this paper. We found that our implementation of GPS outperforms state-of-the-art software model checkers (including the top performers in SV-COMP ReachSafety-Loops category), both in terms of the number of benchmarks solved and in terms of running time.

CCS Concepts: • **Theory of computation** → **Verification by model checking**; *Program analysis*; *Program verification*; • **Software and its engineering** → **Model checking**; *Software verification*; *Automated static analysis*; *Formal software verification*; *Dynamic analysis.*

Additional Key Words and Phrases: Model Checking, Static Analysis, Algebraic Program Analysis

ACM Reference Format:
Ruijie Fang, Zachary Kincaid, and Thomas Reps. 2025. Software Model Checking via Summary-Guided Search. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 364 (October 2025), 28 pages. https://doi.org/10.1145/3763142

1 Introduction

Three prominent approaches to reasoning about safety properties of programs are static analysis, automated testing, and software model checking. Static analysis is typically concerned with computing conservative over-approximations of program behavior—static analyzers can prove safety, but not refute it. Automated testing derives inputs on which to run programs, aiming to maximize the amount of code exercised by the tests—testing can be used to refute safety (that is, find bugs), but not verify it. Finally, software model-checking techniques take both a program and property of interest, and attempt to determine whether it holds. Software model checkers are capable of both verification and refutation, but are typically slow, and sometimes fail to terminate.

One may understand the failure modes of software model checkers by viewing model checking as a search problem: *given a control-flow graph that represents the program and a designated error location, find a path from entry to error that is feasible (in the sense that it corresponds to a program execution)*. Assuming that the error location is *unreachable*, a model checker may fail to discover a

Authors' Contact Information: Ruijie Fang, University of Texas at Austin, Austin, TX, USA, ruijief@cs.utexas.edu; Zachary Kincaid, Princeton University, Princeton, NJ, USA, zkincaid@cs.princeton.edu; Thomas Reps, University of Wisconsin—Madison, Madison, WI, USA, reps@cs.wisc.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/10-ART364
https://doi.org/10.1145/3763142

proof because it proves the safety of individual paths one path at a time, failing to find a correctness argument that generalizes to all paths. Assuming that the error location is *reachable*, a model checker may fail to find the error path because it exhausts its computational resources searching paths that are either infeasible or do not reach the error location.

This paper proposes a software model-checking algorithm, GPS (Guided by Path Summaries), that uses static analysis and testing techniques to mitigate both of these failure modes. In particular, GPS makes use of a recent line of work on program summarization via algebraic program analysis [31, 33], a static-analysis technique that uses SMT solving to derive a *path summary*, which is a transition formula that over-approximates the input/output relation of a given set of paths. Summarization is immediately applicable to the *unreachable* failure mode: rather than prove safety of paths one-by-one, we may compute a summary F and prove that F is unsatisfiable using an SMT solver. Our key insight is if F is instead *satisfiable*, then a model of F is an input/output pair of states and the *input* state corresponds to a test that may reach the error location. We can thus address the *reachable* failure mode by using this kind of "directed testing" to guide exploration of paths through the program, ensuring that the paths searched by GPS are feasible and at least *might* lead to the error location.

Of course, safety checking is an undecidable problem, and GPS has its own failure modes. The summary F for a set of paths is over-approximate, and so a test input generated from a model of F may fail to reach the error location either because (1) the error location is unreachable under any input (but the summary F was not precise enough to prove it), or (2) the error location *is reachable via some program path*, but cannot be reached under the given test input. In an attempt to determine which, GPS explores different paths through the program (similarly to symbolic execution), and for each path π, GPS tries to generate a new test that satisfies two criteria: (a) the new test *extends* π (ensuring a novel test because it traverses a previously-unexplored path) and (b) the new test *may* reach the error (where "may" is judged according to a path summary for all continuations of π to the error location). Successful test generation results in more paths being explored, making progress towards refuting safety. Failure to find a test case identifies a "dead end"—a program path that cannot be extended to reach the error location. Dead-end detection can help find bugs by leading the search toward more productive areas of the search space, and can prove safety by establishing that all paths lead to dead ends. Furthermore, by using Craig interpolation from failed test generation, GPS can synthesize candidate invariants for a proof of safety. GPS may still fail to terminate on some programs, but it is *refutationally complete* in the sense that GPS will find an error if one exists, if provided enough computational resources (similar to bounded model checking).

Contributions. We show how to combine synergistically the strengths of a summary-producing static-analysis method, a test generator, and a software model checker, leveraging the following mechanisms from prior work:
(1) Directed test generation à la DART, SYNERGY, DASH, and SMASH [4, 22, 24, 25];
(2) Path summaries à la algebraic program analysis [20, 33];
(3) Craig interpolation and abstract reachability trees for invariant synthesis from the lazy abstraction for interpolants (IMPACT) algorithm [37].
The main novelties of our work are as follows:
(1) GPS provides a generic method of using over-approximate summaries for *directed testing*, where the goal is to find a test that reaches an error location rather than to maximize code coverage. Aided by summaries, GPS is particularly effective at finding non-deterministic error paths: over-approximate summaries can guide GPS's test generator to effectively resolve non-determinism.
(2) GPS makes use of *failed* attempts at generating directed tests to generate candidate invariants that we call *dead-end interpolants*. Dead-end interpolants are high quality in the sense that they

are suitable for proving infeasibility of a possibly infinite (regular) set of paths—namely, all possible extensions of the dead-end path.

(3) GPS employs a novel, two-layered search strategy that interleaves a depth-first algorithm for executing tests and a breadth-first algorithm for generating new tests. To ensure *refutation-completeness* of this strategy (that is, if a bug exists, GPS will eventually find it), we require that the test executor does not get stuck executing an infinite-length test. We achieve this goal by instrumenting the program with a special variable called `gas`, along with additional assumptions that ensure (a) the `gas` variable is bounded from below by zero at all times, and (b) every infinite path through the control-flow contains infinitely many decrements to `gas`. Rather than searching for iteratively higher amounts of gas, GPS uses summaries to infer values of gas in each test it generates, which, when combined with GPS's search strategy, allows for a *non-uniform* search that is capable of finding long counter-examples. We further show that this instrumentation technique can help the algorithm converge on certain classes of benchmarks featuring highly input-dependent loops (Section 5.3).

(4) Almost every step of the GPS algorithm requires a *single-target* path-summary query to the static-analysis engine. We show that such queries can be efficiently precomputed offline by using Tarjan's algorithm for finding single-source/multi-target path expressions [46] in dual mode, so that it instead computes single-target/multi-source path expressions (Section 6.1).

We implemented GPS using a variant of compositional recurrence analysis [20] to compute summaries, and evaluated GPS on a rich suite of benchmarks, including intraprocedural programs from the Software Verification Competition; from prior literature; as well as synthetic programs based on examples in this paper. Our experiments show that *GPS outperforms several state-of-the-art software model checkers* (Table 3) while being especially effective at bug-finding for a class of programs featuring input-dependent loops, which we dub *"Lock & Key" problems* (defined in Section 2).

The remainder of the paper is structured as follows. Section 2 illustrates how GPS can prove and refute safety using a series of examples. Section 3 reviews necessary background. To facilitate the presentation of our technique, we develop it in two steps: GPSLite and GPS. GPSLite (Section 4) is a simplified variant of GPS that illustrates how program summaries can be used for both verification and refutation. GPS (Section 5) augments GPSLite with the facility to generate invariants from dead-end interpolants. Section 6 discusses how algebraic program analysis can be used to compute path summaries, and gives an efficient algorithm for computing the single-target/multi-source path summaries that are required for GPS. Experimental results appear in Section 7, and a discussion of related work is given in Section 8. Additional statistics about our experiments, as well as proofs about theorems in this paper may be found in the extended version of this paper [19].

```
int main() {
  int N = havoc();
  int i = 0, min = 0;
  while (i < N) {
    min++; i++;
  }
  if (min < 1000) return 0
  else assert(0)
}
```

Fig. 1. Example EX-1 in Section 2.

2 Overview

In this section, we highlight key insights behind the GPS algorithm via three motivating examples. The first example illustrates how over-approximate path summaries produced by static analysis can aid in finding an assertion violation. The second and third examples illustrate how GPS uses over-approximate (but not necessarily precise) summaries in verifying safety.

2.1 Summary-Directed Test Generation

We consider a bug-hunting task adapted from a real-world web-server bug, discussed in [41] and presented in Figure 1. EX-1 is a buggy program with a long error path—the loop must be executed N times—and moreover, for a tool to exercise the bug, it must find one of the values for N that trigger the bug—in this case, at least 1000. This combination of a long error path and input-dependent control flow is challenging for existing tools because they must find an appropriate value for a nondeterministic expression (in this case, $N = \text{havoc}()$) *and* efficiently traverse a long path to trigger the error. One could imagine another kind of buggy program that takes in an arbitrary integer value as input, but an assertion violation could only be triggered when one guesses the specific input value of 37. Collectively, we refer to these kinds of programs as *"Lock and Key" (L&K) problems*: *unlocking* the assertion violation is contingent on a good *key* (i.e., correct values for non-determinstic expressions). Algorithms that search for errors along paths of increasing length (e.g., bounded model checking or IC3 [11]) may exhaust their resources before reaching the necessary length bound, while testing techniques may fail to find the correct value of N to reach the error (e.g., Saxena et al. [41] identified EX-1 as a challenging problem for concolic-execution engines).

The GPS algorithm begins by computing a summary F that over-approximates all paths from entry to the assertion location (using an off-the-shelf static analyzer):

$$F \triangleq \exists k.k \geq 0 \land (k \geq 1 \Rightarrow 0 < N \land i' \leq N') \land min' = i' = k \land N = N' \land min' \geq 1000$$

In the formula F, program variables correspond to their values at the beginning of the program, while primed variables correspond to their values at the assertion location. GPS then uses an SMT solver to determine the satisfiability of F. If this formula is *unsatisfiable*, then the assertion is safe; however in this instance it is satisfiable—for example, we have the model

$$M \triangleq \{N \mapsto 1000, i \mapsto 0, min \mapsto 0, N' \mapsto 1000, i' \mapsto 1000, min \mapsto 1000\}$$

Because summaries are over-approximate, satisfiability of the summary F does not imply that the assertion can fail. However, in this case, M corresponds to an actual counter-example input that witnesses the assertion violation. GPS is thus able to confirm that the assertion can be violated by simulating executing from the initial state $\{N \mapsto 1000, i \mapsto 0, min \mapsto 0\}$ (corresponding to the values of the *unprimed* variables in the model M). Here, the key insight is that *if a summary-based static analysis fails to prove safety of an assertion, the failure provides an opportunity to refute the assertion by finding a test case using the summary.*

Insight #1: If a given path summary is satisfiable, we can use a model recovered from a satisfiable query as a new test.

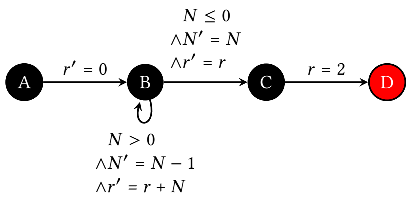
2.2 Verification through Dead-End Detection

EX-1 demonstrates how path summaries can be used to guide test generation toward an error. However, path summaries are over-approximate in nature, and so a test generated using a path summary may not actually reach the error location. In such cases (where the generated test is spurious), we must manage the search space for errors to allow GPS an opportunity to select another potential error path, or to exhaust the space and prove that the error is unreachable. This section illustrates how summaries can be used for *dead-end* detection to prove safety in the context of the simplified algorithm GPSLite (which omits the invariant-synthesis capability of GPS, and will be illustrated in Section 2.3).

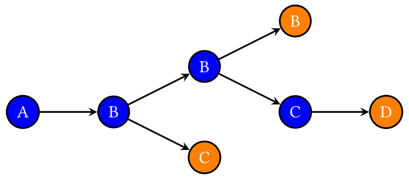
Consider the example EX-2 shown in Figure 2. EX-2 contains a safe assertion; we wish to verify it. GPSLite begins by computing a summary that over-approximates the set of paths from the entry

```
void main(int N) {
  int r = 0;
  while (N > 0) {
    r = r + N--;
  }
  assert(r != 2);
}
```

(a) Pseudo-code for program EX-2.



(b) A control-flow graph for (a).



(c) A *path tree* produced by the test $\{r \mapsto 0, N \mapsto 1\}$

Fig. 2. EX-2: An illustration of the use of dead-end detection to verify safety.

vertex A to the error vertex D. This summary can be understood as the sequential composition of the formula $F_0 \triangleq r' = 0 \land N' = N$, which labels the edge from A to B, with the formula

$$F \triangleq \exists k.((k = 0 \land r' = r) \lor (k \geq 1 \land 0 < N \land N' \geq 0)) \land N' = N - k \land r + k \leq r' \land r' = 2,$$

where F summarizes *all* paths from B to D (including any number of iterations of the B loop). The formula F is over-approximate: it represents the fact that r must increase by *at least* k after k iterations of the B loop, but does not represent its exact closed form.

The over-approximation in turn makes the formula $F_0 \circ F$ *satisfiable*;[1] thus, GPSLite generates a test input $M \triangleq \{N \mapsto 1, r \mapsto 0\}$ from a model of $F_0 \circ F$. In contrast to EX-1, simulating execution from M does *not* lead to an assertion violation. Instead, the path traversed by this test is used to unwind the control-flow graph of EX-2, producing the *path tree* shown in Figure 2c. In the path tree, blue nodes represent the path traversed by the test, while the leaves of the path tree are endpoints of unexplored paths, shown in orange.[2]

GPSLite continues its search by selecting a path to one of the orange vertices, say A $\rightarrow$ B $\rightarrow$ C, and checks whether the composition of the transition formulas along this new path ($N \leq 0 \land N' = N \land r' = 0$) and the summary of all paths from C to D ($r = 2$) is satisfiable. The formula obtained in this way is not satisfiable—meaning that *no feasible path* with prefix A $\rightarrow$ B $\rightarrow$ C can arrive at the error vertex D—so GPSLite declares the path A $\rightarrow$ B $\rightarrow$ C to be a dead end.

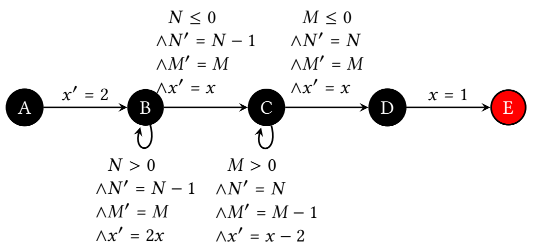
GPSLite proceeds to consider a path represented by the next leaf in the path tree, π = A $\rightarrow$ B $\rightarrow$ B $\rightarrow$ B, and again checks whether the composition of the transition formulas along π ($r' = 2N - 1 \land N' = N - 2 \land N > 1$) and the summary of paths from B to D (formula F above) is satisfiable. Again, the answer is no: r must be at least 3 after executing π, whereas the path-to-error summary F indicates that r is increasing and bounded above by 2; thus, π is a dead end as well.

Finally, GPSLite considers the last unexplored path A $\rightarrow$ B $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D, finds that this path too is a dead end, and concludes that the program is safe. The behavior of the full algorithm, GPS, follows the same recipe outlined above. This example illustrates how GPS(Lite) can use path

[1]As defined in Section 3, here $\circ$ denotes relational composition of two transition formulas.
[2]Concretely, the path tree includes all prefix paths of A $\rightarrow$ B $\rightarrow$ B $\rightarrow$ C.

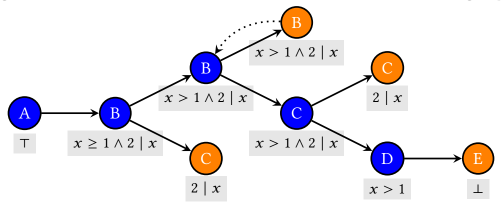
```
void main(int N, int M) {
  int x = 2;
  while (N-- > 0)
    x = 2 * x;
  while (M-- > 0)
    x = x - 2;
  assert(x != 1);
}
```

(a) An example program



$x' = 2$
$N \le 0 \wedge N' = N - 1 \wedge M' = M \wedge x' = x$
$M \le 0 \wedge N' = N \wedge M' = M \wedge x' = x$
$x = 1$
$N > 0 \wedge N' = N - 1 \wedge M' = M \wedge x' = 2x$
$M > 0 \wedge N' = N \wedge M' = M - 1 \wedge x' = x - 2$
A
B
C
D
E

(b) A control-flow graph for (a).



(c) An *abstract reachability tree* (ART)—a path tree with labels and covering edges (shown in dashes)—proving safety of EX-3.

Fig. 3. EX-3: An illustration of the use of dead-end interpolation to verify safety.

summaries to prove safety, even in cases where the path summary for the whole program is not sufficiently precise to do so.

Insight #2: Over-approximate summaries can help establish that a control-state is a dead end, thereby eliminating a whole class of paths from consideration.

2.3 Invariant Synthesis with Dead-End Interpolation

As previously described, GPSLite is essentially a variant of concolic testing [36, 42], but augmented with the use of static analysis-generated summaries to guide test generation and detect dead ends. In some cases (such as EX-2), dead-end detection is sufficient to prove safety; in other cases—especially programs with complex loops—it is necessary to synthesize an invariant that separates the reachable states of the program from the error. Here, we give an example of how the full GPS algorithm can synthesize such invariants by combining dead-end detection—the idea illustrated in Section 2.2—with Craig interpolation and abstract reachability trees from the Impact family of software-model-checking algorithms [3, 8, 37].

Example EX-3. Consider another program with a safe assertion, shown in Figure 3. Suppose that one were to run GPSLite on EX-3. We begin by computing a summary that over-approximates all paths from the entry vertex Ⓐ to the error vertex Ⓔ. Conceptually, we can understand this summary as the sequential composition of the two summaries

$$F_1 \triangleq \exists k_1. N' = N - k_1 \wedge M' = M \wedge x' \ge 2 + k_1 \wedge (k_1 \ge 1 \Rightarrow N' \ge 0) \wedge N' \le 0$$

$$F_2 \triangleq \exists k_2. N' = N \wedge M' = M - k_2 \wedge x' = x - 2k_2 \wedge (k_2 \ge 1 \Rightarrow M' \ge 0) \wedge M' \le 0 \wedge x' = 1$$

where F_1 summarizes paths from Ⓐ to Ⓒ (excluding looping paths at Ⓒ), and F_2 summarizes paths from Ⓒ to Ⓔ. Note that F_2 is precise, while F_1 *is not* (in particular, it does not represent the fact that $x' = 2^{k_1+1}$, and hence is over-approximate).

GPSLite determines that the composition of F_1 and F_2 is satisfiable, and generates a test input $\{N \mapsto 1, x \mapsto 2, M \mapsto 0\}$ from a model. Similar to the previous example, simulating execution with this input does not lead to an assertion violation, and instead produces a path tree, depicted in Fig. 3c (for now, ignore the gray boxes and dashed edges). GPSLite is able to determine that Ⓐ → Ⓑ → Ⓒ is a dead end, but (because F_1 over-approximates the behaviors along the paths from Ⓐ to Ⓒ) cannot establish that Ⓐ(→ Ⓑ)^i is a dead end for all $i \in \mathbb{N}$; as a result, GPSLite diverges for EX-3, producing infinitely many tests, none of which reach the error location (because the error location is unreachable).

Our full algorithm, GPS, is able to prove safety of EX-3 by exhibiting a *complete and well-labeled abstract reachability tree (ART)*, shown in Figure 3c. Intuitively, we can think of an ART as a Floyd/Hoare proof that every path in EX-3 is a dead end. More concretely, an ART is a path tree in which (a) each path is labeled with a state formula (depicted in gray)—which describes an *over-approximation* of the states reachable along the path, and functions as an invariant candidate—and (b) there is an additional set of *covering edges* (dashed lines) between identical control locations on a path. The presence of a covering edge means that an inductive invariant has been found for a loop. Observe that each leaf in the pictured ART (with the exception of the Ⓑ leaf, which has an outgoing covering edge) is labeled with a condition that is inconsistent with the path-to-error summary for that node, thus indicating that it is a "dead end". For instance, each Ⓒ leaf is labeled with the condition "$2 \mid x$" (i.e., x is even), which is inconsistent with F_2, and the Ⓔ leaf is labeled with "$\bot$" ($\bot$ denotes *false*, which is inconsistent with the identity transition formula). The label of the Ⓑ leaf is not a dead-end condition, however it is an inductive invariant of the first loop in EX-3—and indeed, this is indicated by the dashed covering edge between Ⓑ and the second Ⓑ node.

To derive the ART shown in Figure 3c—and thus arrive at a safety proof—GPS generates an initial test, attempts to explore the path Ⓐ → Ⓑ → Ⓒ, and discovers that it is a dead end. GPS uses the unsatisfiability proof from dead-end detection to generate a sequence of three state formulas $\top$, $2 \mid x$, $2 \mid x$, which constitute a Floyd/Hoare proof that the path Ⓐ → Ⓑ → Ⓒ is a dead end: we call these formulas *dead-end interpolants* for the path Ⓐ → Ⓑ → Ⓒ. Similarly, GPS finds dead-end interpolants for the paths Ⓐ → Ⓑ → Ⓑ → Ⓒ ($\top$, $2 \mid x$, $2 \mid x$, $2 \mid x$, $2 \mid x$) and Ⓐ → Ⓑ → Ⓑ → Ⓒ → Ⓓ → Ⓔ ($\top$, $x \geq 1$, $x > 1$, $x > 1$, $x > 1$, $\bot$), and conjoins the dead-end interpolants across these three paths to arrive at the labels pictured in Figure 3c. Finally, GPS checks that the label of the path Ⓐ → Ⓑ → Ⓑ can be extended to Ⓐ → Ⓑ → Ⓑ → Ⓑ (i.e., $x > 1 \land 2 \mid x$ is an inductive invariant of the first loop), thus completing the proof.

Intuitively, dead-end interpolants are good candidates for inductive invariants, because they prove that *all* extensions of a given path—of which there are typically infinitely many—are unsatisfiable.

Insight #3: Interpolants generated from summary-based dead-end detection can be used to derive high-quality candidate invariants.

There is mutually beneficial synergy between Insight #2 and Insight #3: each step of the GPS algorithm typically considers a non-error leaf of the ART, and thus the path-to-error summary for the leaf describes a (possibly infinite) set of paths. If a dead end is detected in that step, interpolation is forced to generalize with respect to that set of paths. When the set of paths includes a loop, such generalization steps produce good candidates for inductive invariants.

3 Background

Fix a set of variable symbols X. We use $\mathbf{SF}(X)$ denote the set of *state formulas* over X (whose free variables are drawn from X). A **transition formula** over X is a formula whose free variables range over $X \cup X'$, where X' denotes the set of "primed" variants of the variables in X. We use $\mathbf{TF}(X)$ to denote the set of all transition formulas over X. For transition formulas F and G in $\mathbf{TF}(X)$, we use

$F \circ G \triangleq \exists X''.F[X' \mapsto X''] \wedge G[X \mapsto X'']$ to denote the relational composition of F and G. For states M and M' and a transition formula F, we use $M \xrightarrow{F} M'$ to denote that M can transition to M' along F. For a transition formula F and a state formula ψ, we define $post(\psi, F) = (\exists X.\psi \wedge F)[X' \mapsto X]$ to be the strongest post-condition of ψ along F.

We represent a program as a weighted graph. A **weighted graph** is a tuple $G = (V, E, w, s, t)$ where V is a finite set of vertices, $E \subseteq V \times V$ is a set of directed edges, $w : E \longrightarrow \mathbf{TF}(X)$ is a weight function associating each edge with a transition formula, and $s, t \in V$ are designated *source* and *terminal* vertices, such that s is the unique vertex of in-degree 0 in G and t is the unique vertex of out-degree 0 in G. A **path** in G is a sequence of edges $e_1e_2 \ldots e_n$ such that the destination of each e_i is the source of e_{i+1}. For vertices $u, v \in V$, we use $Paths_G(u, v)$ to denote the set of paths in G that begin at u and end at v.

Let $\pi = e_1e_2 \cdots e_n$ be a path. We use $src(\pi)$ to denote the source vertex of e_1, $dst(\pi)$ to denote the destination vertex of e_n; if π is the empty path, define $src(\pi) = dst(\pi) = s$. We extend the weight function w to paths by defining $w(\pi) = w(e_1) \circ w(e_2) \circ ... \circ w(e_n)$ to be the composition of edge weights along π. For paths π and π', we use $\pi \sqsubseteq \pi'$ to denote that π is a prefix of π'; we call this the *prefix order.* For two paths π and σ with $dst(\pi) = src(\sigma)$, we write $\pi\sigma$ to denote the concatenation of π with σ. For any prefix $\tau = e_1 \ldots e_i$ of π, let $\tau^{-1}\pi$ denote the suffix $e_{i+1} \ldots e_n$; note that in this notation, $\tau(\tau^{-1}\pi) = \tau e_{i+1}, ..., e_n = \pi$.

Let $G = (V, E, w, s, t)$ be a weighted graph. We say that G is **unsafe** if there exists a path $\pi \in Paths_G(s, t)$ from s to t such that $w(\pi)$ is satisfiable; if so, we say that such a path π is a **feasible** *s*-*t* path. If no such feasible *s*-*t* path exists, we say that G is safe. The GPS algorithm solves the following problem:

Problem 1 (Control-state Reachability). *Given a weighted graph G, decide whether G is safe.*

For presenting the GPS algorithm, we assume access to a *path summary oracle* $\mathsf{Sum}(G, u, v)$, which given a weighted graph G and two vertices u and v, returns a transition formula that over-approximates all paths from u to v in G (i.e., $w(\pi) \models \mathsf{Sum}(G, u, v)$ for all $\pi \in Paths_G(u, v)$). In Section 6, we review how to use algebraic program analysis [33] to produce such an oracle, and give an algorithms that is particularly well-suited to GPS. However, the GPS algorithm is agnostic as to how path summaries are obtained.

4 GPSLite: Summaries for Test Generation and Dead-End Detection

In this section, we describe GPSLite, a simplified variant of the GPS algorithm that we use to stage the exposition of the GPS algorithm—and highlight the role of path summaries in generating tests and detecting dead ends. In Section 5, we describe how GPSLite can be combined with Craig interpolation and abstract reachability trees à la lazy abstraction with interpolants (Impact) [37] to perform invariant synthesis, thus yielding the GPS algorithm.

GPSLite can be understood as a combination of path summaries with testing and symbolic execution, similarly to concolic testing (as in Dart and Cute). In contrast to pure concolic testing, GPSLite uses summaries to generate test inputs that are “likely” to reach an error location (rather than generating inputs randomly, or to maximize code coverage).

GPSLite uses a **path tree** to represent its search space. A path tree for a weighted graph $G = (V, E, w, s, t)$ is a finite collection of paths of G emanating from the source vertex $s \in G$. Intuitively, the non-maximal paths of a path tree T represent paths that have already been explored (and proven to be feasible), while the maximal paths of T (the *leaves* of the path tree) represent paths that are either dead ends (cannot be extended to an error trace) or remain to be explored.

Definition 1 (Path Tree). *A **path tree** for a weighted graph $G = (V, E, w, s, t)$ is a set of paths $T \subseteq \bigcup_{u \in V} Paths_G(s, u)$ satisfying the following conditions:*

- *T **is prefix-closed:** For any path $\pi \in T$ and any prefix τ of π, we have $\tau \in T$.*
- *T **is full:** For any path $\pi \in T$, if there is some edge $e \in E$ such that $\pi e \in T$, then $\pi e' \in T$ for any edges $e' \in E$ with $\mathrm{dst}(\pi) = \mathrm{src}(e')$.*

Note that these two conditions ensure that for any path π beginning at s, there is a leaf of T that is a prefix of π—in this sense, the leaves of a path tree represent all possible opportunities for expanding the tree.

The GPSLite algorithm is shown in Figure 4. GPSLite takes as input a weighted graph $G = (V, E, w, s, t)$, and determines whether G is safe. It maintains two data structures: a path tree T, and a queue frontierQueue, with the invariant that each maximal path $\pi \in T$ is either a dead end or belongs to frontierQueue. It terminates when it either finds a feasible s-t path witnessing that G is unsafe with respect to ψ, or it exhausts frontierQueue (proving that G is safe with respect to ψ by virtue of establishing each leaf of T as a dead end).

In each iteration of the main loop, GPSLite selects a path π from frontierQueue and attempts to generate a *directed* test that extends π—that is, a state M that (1) *is reachable* along the path π and (2) *may reach* the target location t along the path summary $\mathrm{Sum}(G, \mathrm{dst}(\pi), t)$. This step is accomplished using the following primitive (which can be implemented via a query to an SMT solver):

Primitive $\mathrm{Check}^{+}(F, G)$:
- **Input:** A pair of transition formulas F and G
- **Output:** Returns either:
  1. (SAT, M) if $F \circ G$ is satisfiable, where M is a state such that $L \xrightarrow{F} M \xrightarrow{G} N$ for some L and N (i.e., M is a witness for the midpoint of the composition of F and G).
  2. Or UNSAT, if $F \circ G$ is unsatisfiable.

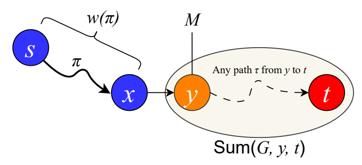


Illustration of $\mathrm{Check}^{+}(w(\pi), \mathrm{Sum}(G, \mathrm{dst}(\pi), t))$ where $y = \mathrm{dst}(\pi)$ and x is predecessor of y.

If $\mathrm{Check}^{+}(w(\pi), \mathrm{Sum}(G, \mathrm{dst}(\pi), t))$ fails to find a model M, then π is a dead end: for any path τ from $\mathrm{dst}(\pi)$ to t, we must have that $w(\tau)$ entails $\mathrm{Sum}(G, \mathrm{dst}(\pi), t)$, so if $w(\pi) \circ \mathrm{Sum}(G, \mathrm{dst}(\pi), t)$ is unsatisfiable, so is $w(\pi) \circ w(\tau) = w(\pi\tau)$. On the other hand, if $\mathrm{Check}^{+}(w(\pi), \mathrm{Sum}(G, \mathrm{dst}(\pi), t))$ finds a model M, then we have established that π is feasible, and executing from M *might* to lead to the error because M lies in the domain of $\mathrm{Sum}(G, \mathrm{dst}(\pi), t)$; thus, we can simulate execution from M to find yet more feasible paths (extending π), with the goal of reaching t.

GPSLite uses the Explore sub-routine (Figure 4) to simulate executing a test. Given a weighted graph G, a path tree T, a frontier queue frontierQueue, a (feasible) path π, and model M (reachable along π), Explore simulates execution of G beginning at location $dst(\pi)$ with state M. Should $\mathrm{dst}(\pi) = t$, then π is a feasible s-t path, and so we may halt execution (and the GPSLite algorithm). Otherwise, we enter the loop at line 15 of Explore(−) and simulate execution along each edge e emanating from $dst(\pi)$. For each edge e along which execution may progress—we have $M \xrightarrow{w(e)} M'$ for some state M'—we may recursively explore the (feasible) path πe starting at state M'. For each edge e for which no such M' exists, πe is added to frontierQueue, to be revisited in the main loop of GPSLite.

Should an edge e correspond to a deterministic program command, we may find the state M' such that $M \xrightarrow{w(e)} M'$ (should it exist) simply by executing it. However, our program model permits non-deterministic commands: in general there may be infinitely many models M' such

```
1:  procedure GPSLite($G, \psi$)
    input: Weighted graph $G = (V, E, w, s, t)$
    output: Either Safe, if there exists no feasible $s$-$t$ path in $G$, or (Unsafe, $\pi$) where $\pi$ is a feasible $s$-$t$ path.
2:    $T \longleftarrow \{\epsilon\}$;
3:    frontierQueue $\longleftarrow [\epsilon]$
4:    while frontierQueue is not empty do
5:      $\pi \longleftarrow$ frontierQueue.dequeue()
6:      match Check$^+(w(\pi), \mathrm{Sum}(G, \mathrm{dst}(\pi), t))$ do
7:        case Sat(M) do
8:          match Explore($G, T$, frontierQueue, $\pi, M$) do
9:            case Safe do continue
10:           case (Unsafe, $\tau$) do return (Unsafe, $\tau$)
11:       case Unsat do continue
12:   return Safe
13: procedure Explore($G, T$, frontierQueue, $\pi, M$)
    input: Weighted graph $G = (V, E, w, s, t)$, path tree $T$, path $\pi \in T$, model $M$
    output: Safe when $t$ is not reached; otherwise (Unsafe, $\tau$), where $\tau$ is a feasible $s$-$t$ path
14:   if $dst(\pi) = t$ then return (Unsafe, $\pi$);                ▷ Terminal location reached; return
15:   for $e \in E$ with $dst(\pi) = src(e)$ do
16:     $T$.add($\pi e$);
17:     $Next \longleftarrow w(e)[X \mapsto M(X)]$                 ▷ Set of states $M'$ such that $M \xrightarrow{w(e)} M'$
18:     match Check$^+(Next, \mathrm{Sum}(G, dst(e), t))$ do
19:       case SAT($M'$) do                                        ▷ $M$ has successor $M'$ along $e$.
20:         match Explore($G, T$, frontierQueue, $\pi e, M'$) do
21:           case (Unsafe, $\tau$)) do return (Unsafe, $\tau$)
22:           case Safe do continue
23:       case UNSAT do                                            ▷ $M$ has no successor along $e$.
24:         frontierQueue.enqueue($\pi e$);                        ▷ Halt simulation at $\pi e$ and add to frontier.
25:   return Safe;
```

Fig. 4. The GPSLite algorithm.

that $M \xrightarrow{w(e)} M'$. One may determine whether such an M' exists by checking satisfiability of the formula $Next \triangleq w(e)[X \mapsto M(X)]$ (replacing each pre-state variable with its corresponding value in M, thus producing formula whose models correspond exactly to the set $\{M' : M \xrightarrow{w(e)} M'\}$). The Explore(-) procedure refines this approach by further constraining the successor state M' to be one that may reach the target location (according to the path summary Sum(G, $dst(e)$, t)). By doing so, we reap two benefits: (1) this choice may do a better job of directing execution toward the target location, and (2) we can terminate execution early if Explore(-) cannot reach the error location using M'.

Discussion. The main insight behind GPSLite is two-fold: first, (over-approximate) summaries can effectively guide test generation; second, the summary-guided tests can be used to guide exploration of program states in a two-layered search: a breadth-first exploration algorithm can be used to generate targeted tests along frontiers of the path tree, and a depth-first search can be used to execute each generated test. Intuitively, this search strategy can be particularly performant when the quality of generated tests are good (which, as we observe in Section 7, is often true) but additional care must be taken to ensure that the test generator does not get stuck executing an infinite-length test; we discuss in detail how to address this issue in Section 5.3.

5 GPS: Combining GPSLite with Abstraction Refinement

GPSLite has no facility for invariant generation—it is capable of proving safety of a program only when path summaries are sufficiently strong to show that all paths are dead-ends. This section presents GPS, a software-model-checking algorithm that integrates GPSLite with Craig interpolation and abstract reachability trees from the lazy-abstraction-with-interpolants algorithm [37]. In particular, we develop *dead-end interpolation*, which uses summaries to generate high-quality candidate invariant assertions that prove safety of a regular set of paths simultaneously.

5.1 From Path Trees to Abstract Reachability Trees (ARTs)

The main ingredient that bridges GPSLite (described in Section 4) and GPS is an *abstract reachability tree* (ART). Informally speaking, an ART is an extension of a path tree (Definition 1), where every path inside the path tree is also paired with a *label*—a state formula that over-approximates the post-state of the path—and some pairs of paths are connected by *covering edges*, which describe entailment relationships between paths and are used to check for inductiveness of loop labels. An example of an ART for the program EX-3 in Section 2 appears in Figure 8.

Definition 2 (Abstract Reachability Trees). *An abstract reachability tree (ART) for a weighted graph $G = (V, E, w, s, t)$ is a 3-tuple $\Lambda = (T, L, \mathrm{Cov})$, with each element of the 3-tuple defined as follows:*
*(1) **Path tree**. T is a path tree of G (Cf. Definition 1),*
*(2) **Label function**. $L : T \longrightarrow \boldsymbol{SF}(X)$ maps each path $\pi \in T$ to a state formula in $\boldsymbol{SF}(X)$.*
*(3) **Covering relation**. $\mathrm{Cov} \subset T \times T$ is called the covering relation, such that for every $(\pi, \pi') \in \mathrm{Cov}$, π' is a prefix of π and $\mathrm{dst}(\pi) = \mathrm{dst}(\pi')$*

For a path $\pi = e_1 e_2 \cdots e_n$, if π is non-empty, we use $\mathit{parent}(\pi)$ to denote its immediate prefix $e_1 e_2 \cdots e_{n-1}$ and say that π is a **child** of $\mathit{parent}(\pi)$ (corresp. $\mathit{parent}(\pi)$ is a **parent** of π). Furthermore, recall from Section 3 that $\sqsubseteq$ denotes the prefix order over paths. Consider an ART $\Lambda = (T, L, \mathrm{Cov})$. For a path $\pi \in T$, we refer to π as a **leaf path** if it maximal w.r.t. $\sqsubseteq$. We say that a path $\pi \in T$ is **pruned** if $L(\pi) \wedge \mathrm{Sum}(G, \mathrm{dst}(\pi), t)$ is unsatisfiable (where t is the terminal vertex of the weighted graph associated with ART Λ). One may think of a pruned path π as a path for which the ART represents a proof that π is a dead-end—every pruned path is a dead end, but not every dead end is necessarily pruned. For a path $\pi \in T$, we say that π is **covered** if there is a pair $(\pi, \pi') \in \mathrm{Cov}$. We say a leaf path π is a **frontier** if it is neither pruned nor covered. Intuitively, π is a frontier path if it *might* be the prefix of a previously unexplored, feasible s-t path in G. We define Λ to be **complete** when there are no frontier paths in T (i.e., *every* maximal path $\pi \in T$ is either pruned or covered).

For any $\pi \in T$, we can associate π with a set of paths $\mathit{Paths}_\Lambda(\pi)$ in G by treating Λ as a finite-state automaton with start state ϵ, final state π, and with an e-labeled transition from π to πe for each $\pi e \in T$, and an ϵ-transition from π to π' for each $(\pi, \pi') \in \mathrm{Cov}$. For instance, the set of paths associated with Ⓐ $\longrightarrow$ Ⓑ $\longrightarrow$ Ⓑ in Figure 3c is Ⓐ $\longrightarrow$ Ⓑ$(\longrightarrow$ Ⓑ$)^+$. Formally, Paths_Λ is the least function such that $\mathit{Paths}_\Lambda(\epsilon)$ contains ϵ, $\mathit{Paths}_\Lambda(\pi e)$ contains $\{\tau e : \tau \in \mathit{Paths}_\Lambda(\pi)\}$, and $\mathit{Paths}_\Lambda(\pi')$ contains $\mathit{Paths}_\Lambda(\pi)$ for each $(\pi, \pi') \in \mathrm{Cov}$.

Let Λ be an ART for a weighted graph G. The following *well-labeledness* condition ensures that for any $\pi \in T$ and any $\tau \in \mathit{Paths}_\Lambda(\pi)$, we have $\mathit{post}(\top, w(\tau)) \models L(\pi)$.

Definition 3 (Well-labeledness). *Given an ART Λ of a weighted graph G, we call Λ well-labeled (w.r.t. G) if the following conditions are satisfied:*
*(1) **Initiation**. The formula labeling the root ($L(\epsilon)$) is $\top$.*
*(2) **Consecution**. For any path $\pi e \in T$, $\mathit{post}(L(\pi), e) \models L(\pi e)$*
*(3) **Well-covered**. For any $(\pi, \pi') \in \mathrm{Cov}$, we have $L(\pi) \models L(\pi')$*

```
1:  procedure GPS($G$)
    input: Weighted graph $G = (V, E, w, s, t)$
    output: Safe if $G$ is safe; otherwise (Unsafe, $\pi$), where $\pi$ is a feasible $s$-$t$ path in $G$.
2:   $\Lambda \longleftarrow$ CreateART($G$);            ▷ Initialize ART containing only the empty path $\epsilon$
3:   frontierQueue.enqueue($\epsilon$);
4:   while frontierQueue $\neq$ [] do
5:     $\pi \longleftarrow$ dequeue(frontierQueue);
6:     if TryCover($\Lambda, \pi$) then                 ▷ Attempt to cover $\pi$ by an ancestor
7:       continue;                                      ▷ $\pi$ is covered in $\Lambda$, no need to explore further
8:     else
9:       $Fs \longleftarrow$ map($w, \pi$) :: Sum($G, dst(\pi), t$);   ▷ Sequence of $|\pi|+1$ transition formulas
10:      match Check($Fs$) do
11:        case UNSAT($\phi_0, \phi_1, ..., \phi_n$) do Refine($\Lambda, [\phi_0, ..., \phi_{n-1}], \pi$);
12:        case SAT($M_0, ..., M_n, M_{n+1}$) do
13:          match Explore($G, \Lambda, M_n, \pi$) do
14:            case Safe do continue;
15:            case (Unsafe, $\tau$) do return (Unsafe, $\tau$);
16:  return Safe;                                       ▷ $\Lambda$ is complete and the procedure is proven safe.
```

Fig. 5. Main procedure for the full GPS algorithm.

```
1:  procedure TryCover($\Lambda = (T, L, \mathrm{Cov}), \pi$)
2:   for each $\tau \sqsubset \pi$ with $dst(\tau) = dst(\pi)$ do
3:     $I \longleftarrow L(\tau)$
4:     match Check($I$, map($w, \tau^{-1}\pi$) :: $\neg I$) do
5:       case UNSAT($T_0, ..., T_n$) do
6:         Refine($\Lambda, \pi, [T_0, ..., T_n]$);
7:         Add $(\pi, \tau)$ to Cov;
8:         return $\top$;
9:       case SAT($M_0, ..., M_n$) do return $\bot$;
```

```
1:  procedure Refine($\Lambda, \pi e_1 \dots e_n, \phi_1 \dots \phi_n$)
2:   $(T, L, \mathrm{Cov}) \longleftarrow \Lambda$;
3:   for $i = 1$ to $n$ do
4:     $\pi' \longleftarrow \pi e_1 \dots e_i$
5:     for each $\tau$ with $(\tau, \pi') \in \mathrm{Cov}$ do
6:       if $L(\tau) \not\models \phi_i$ then
7:         Remove $(\tau, \pi')$ from Cov;
8:         frontierQueue.enqueue($\tau$);
9:     $L(\pi') \longleftarrow L(\pi') \wedge \phi_i$;
```

Fig. 6. Auxilliary procedures for the GPS algorithm: Refine(-) and TryCover(-). We use $a :: t$ to denote concatenating a head element a with a list t, and use map(f, L) to denote the result of applying a function f to every element of list L (as is standard in functional programming languages).

The following lemma states that the safety of a weighted graph G can be witnessed by a complete and well-labeled ART Λ:

Lemma 1. *Let $G = (V, E, w, s, t)$ be a weighted graph, and let Λ be a well-labeled ART for G. If Λ is complete and well-labeled, then G is safe.*

5.2 From GPSLite to GPS

At a glance. Recall that as discussed in Section 4, the basic idea of the GPSLite algorithm is as follows: in each iteration of the main loop of Figure 4, we select a path π from frontierQueue—which is a queue storing paths that are potentially frontiers. Then, GPSLite tries to generate a longer path with π as a prefix, in an effort to make progress in a search for a feasible source-sink path in a given weighted graph G. Such an attempt is not always guaranteed to be successful; notably, the query to $\mathsf{Check}^{+}(w(\pi), \mathsf{Sum}(G, \mathsf{dst}(\pi), t))$ can return UNSAT (See line 22 in Figure 4), showing that π leads to a dead-end. The key idea here—borrowed from lazy abstraction with interpolants [37]— is that in the case of UNSAT, one can extract a sequence of intermediate state formulas that can form candidate invariants constituting a proof of infeasibility from the unsatisfiable input formula—and such intermediate state formulas can be maintained as *labels*, or invariant candidates, in an ART. This in turn enables us to synthesize invariants from UNSAT proofs by maintaining coverings between ART paths.

Similar to GPSLite, the GPS algorithm maintains a queue frontierQueue of leaf paths in Λ, maintaining the invariants that (1) all *frontier paths of* Λ are in frontierQueue, and (2) every path in frontierQueue is either a frontier path or a pruned path. We call the paths in frontierQueue *possible frontiers.* Recall that in GPSLite, possible frontier paths are selected in breadth-first order from the queue frontierQueue; for each selected path π GPSLite makes π a *non*-frontier by either generating a test that uses π as a proper prefix, or proves that π is a dead-end. The full GPS algorithm differs in two main ways: (1) GPS extracts information from dead-end detection — in form of interpolants — and uses them to refine candidate invariants; (using the Refine procedure in Figure 6) (2) GPS uses the TryCover procedure shown in Figure 6 to check whether the candidate invariants are in fact inductive. The Refine and TryCover procedures in GPS build upon the procedures of the same name in the Impact algorithm [37]. The main difference is that our Refine procedure incorporates information from path summaries (discussed below), and our TryCover procedure only introduces covering edges between descendants and ancestors.[3]

In what follows, we will first explain how the covering step and the summary-based dead-end refinement works, and then reinforce our explanation through a concrete example.

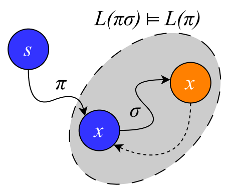
Synthesis of candidate invariants via dead-end interpolation. We assume access to a primitive $\mathsf{Check}(-)$, extending the $\mathsf{Check}^{+}(-)$ primitive from the previous section. It takes as input a sequence of transition formulas $F_1, \dots, F_n$, determines whether the path $F_1, \dots, F_n$ is feasible, and produces either a witness sequence of states demonstrating feasibility, or a sequence of intermediate state formulas constituting a proof of infeasibility. The intermediate assertions can be computed, for instance, using Craig interpolation [39].

Primitive $\mathsf{Check}(F_1, ..., F_n)$:
- **Input:** A sequence of transition formulas $F_1, \dots, F_n$
- **Output:** Returns either:
  1. $(\mathsf{SAT}, [M_0, \dots, M_n, M_{n+1}]$ if $(F_1 \circ \dots \circ F_n)$ is satisfiable, where $M_0, \dots, M_{n+1}$ is a sequence of states such that $M_i \xrightarrow{F_i} M_{i+1}$ for all $i \in \{0, \dots, n\}$
  2. $(\mathsf{UNSAT}, [\phi_0, \dots, \phi_n, \phi_{n+1}])$ if $(F_1 \circ \dots \circ F_n)$ is unsatisfiable, where $\phi_0, \dots, \phi_{n+1}$ is a sequence of state formulas such that $\phi_0 = \top$, $post(\phi_i, F_i) \models \phi_{i+1}$ for all $i \in \{0, \dots, n\}$, and ϕ_{n+1} is $\bot$.

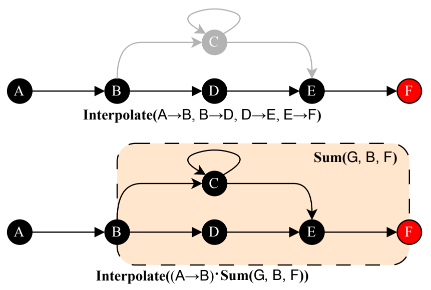
Thus, the $\mathsf{Check}(-)$ primitive generalizes $\mathsf{Check}^{+}(-)$ in two ways: (1) it checks satisfiability of the composition of a sequence of transition formulas rather than two, and (2) when the result is UNSAT, $\mathsf{Check}(-)$ produces a proof of unsatisfiability.

GPS uses the $\mathsf{Check}(-)$ primitive analogously to how GPSLite uses $\mathsf{Check}^{+}(-)$—to generate a test from a path π drawn from the frontier queue or to detect that it is a dead end. In addition, when GPS detects a dead end, it uses the resulting sequence of state formulas (*dead-end interpolants*) to strengthen the labels of π and its prefixes with the help of the auxiliary Refine routine (Figure 6)

[3]We chose to only introduce coverings between descendants and ancestors to simplify presentation. In practice, one could still implement an Impact-like covering strategy of allowing coverings between paths that aren't ancestors and descendants.



(a) The covering condition.


A
B
C
D
E
F
Interpolate(A→B, B→D, D→E, E→F)
Sum(G, B, F)
A
B
C
D
E
F
Interpolate((A→B)·Sum(G, B, F))

(b) Two styles of interpolation (G denotes the weighted graph).

Fig. 7. (a): Illustration of the covering condition in an ART: If taking path σ from π (with $x = \text{dst}(\pi)$) ends in the same vertex x, and $L(\pi\sigma) \models L(\pi)$, then $\pi\sigma$ and π essentially end in the same states, and $L(\pi\sigma)$ is an inductive invariant of the loop at σ. (b): Two styles of interpolation explained via a weighted graph with A, ..., F being vertices: (top) "One path-to-error at a time" interpolation as in the Impact algorithm; versus (bottom) GPS-style dead-end interpolation, which interpolates a concrete prefix path (from A to B) and an (over-approximate) summary-to-error (shown in orange; representing all paths that start at B and end at F).

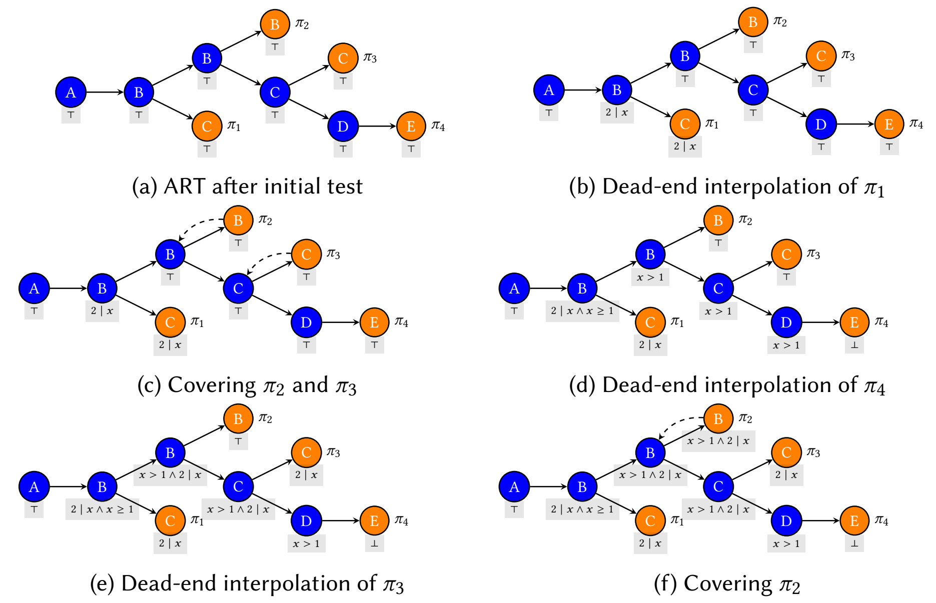
so that π becomes a pruned path in the resulting ART. Given an ART $\Lambda = (T, L, \text{Cov})$, a path $\pi e_1 \dots e_n$, and a sequence of state formulas $\phi_1 \dots \phi_n$, Refine (due to [37]) adds ϕ_i as a conjunct to the state-formula label of $\pi e_1 \dots e_i$ in Λ. Strengthening the label at a path τ may violate the *well-covered* condition (Definition 3) for covering edges to τ; thus, Refine checks if the condition remains satisfied (line 5) and removes covering edges that now violate the condition.

The novelty of GPS is not in the Check(−) primitive, but the way in which it is used. GPS derives interpolants from a sequence of $n + 1$ transition formulas, where the first n correspond to a path π and the last to a summary $\text{Sum}(G, \text{dst}(\pi), t)$ of a set of paths.

Figure 7b illustrates the main difference between refinement in GPS and refinement in the Impact algorithm [37]. In this figure, vertices Ⓐ through Ⓕ denote weighted-graph vertices, with Ⓐ being the source and Ⓕ being the sink (which represents the error vertex). In Impact, interpolation-based refinement is performed only for each source-to-error path, one path at a time (shown in the top half of the figure). In contrast, in GPS interpolation-based refinement is performed for a *regular set of paths* $\{\pi\tau : \tau \in \mathit{Paths}_G(\text{dst}(\pi), \mathit{error})\}$: π is a particular path-prefix that GPS chooses (of the form s-to-$\text{dst}(\pi)$), and $\mathit{Paths}_G(\text{dst}(\pi), \mathit{error})$ is the set of all suffixes to error. This approach increases the generality of the resulting interpolants (shown in the bottom half of the figure).

Using coverings to test for inductive invariants. Consider the situation illustrated in Figure 7a. Suppose that we have a well-labeled ART $\Lambda = (T, L, \text{Cov})$ for a weighted graph $G = (V, E, w, s, t)$, and that a path $\pi \in T$ is a prefix of another path $\pi' \in T$ such that $\text{dst}(\pi) = \text{dst}(\pi')$. Let σ be a suffix path such that $\pi' = \pi\sigma$. Then σ is necessarily a *cycle* in G—as σ starts at $\text{dst}(\pi)$ and ends at the same vertex $\text{dst}(\pi') = \text{dst}(\pi)$. By the well-labeledness condition of Λ, we have that $\text{post}(L(\pi), w(\sigma)) \models L(\pi')$. Now, note that if $L(\pi') \models L(\pi)$, then we may conclude that $L(\pi')$ is a loop invariant for the cycle σ. In other words, one can view the labels of the ART as *candidate* loop invariants, and checking whether a given candidate is an invariant is simply an implication check.

Invariant-checking is implemented by the TryCover(Λ, v) subroutine (Figure 6) of GPS (again essentially due to [37]). This routine seeks to test whether the label of a path $\pi \in T$ can be strengthened to permit π to be covered by one of its ancestors τ w.r.t. the prefix order (establishing


(a) ART after initial test (b) Dead-end interpolation of π_1

(c) Covering π_2 and π_3 (d) Dead-end interpolation of π_4

(e) Dead-end interpolation of π_3 (f) Covering π_2

Fig. 8. Illustration of GPS on the example EX-3

the label at τ to be an inductive invariant of the cycle $\tau^{-1}\pi$). For each candidate ancestor τ (with $dst(\tau) = dst(\pi)$), the TryCover routine forms an inductiveness query that checks whether there is an execution of $\tau^{-1}\pi$ that begins in a state satisfying $L(\tau)$ and ends in a state satisfying $\neg L(\tau)$ (line 4). If such an execution exists, the candidate τ is discarded; if it does not, then TryCover uses sequence interpolants derived from the inductiveness check to refine the labels of prefix paths on the path from τ to π (ensuring that $L(\pi) \models L(\tau)$ and Λ remains well-labeled), and adds (π, τ) to Cov.

Putting everything together: how GPS works on example EX-3 from Section 2. We illustrate these steps by revisiting example EX-3 from Section 2, with the CFG $G = (V, E, w, s, t)$ and ART $\Lambda = (T, L, \text{Cov})$ given in Figures 3, and the full execution trace of GPS on EX-3 (shown as a sequence of ART operations) shown in Figure 8.

As discussed in Section 2, GPS first executes a test $\{N \mapsto 1, x \mapsto 2, M \mapsto 0\}$ to obtain the (incomplete) ART shown in Figure 8a. After this step, no refinements have been performed, so the label for each path in the ART is simply $\top$. Next, GPS then detects that π_1 is a dead end via interpolating $w(\pi_1)$ composed with $\text{Sum}(\text{dst}(G, \pi_1), t)$, producing Figure 8b where the labels of the vertices on π_1 have been refined with dead-end interpolants.

Since π_2 and π_3 both have the same label as their parent ($\top$), both can be covered during a TryCover(-) call to produce Figure 8c. These covering operations indicate that $\top$ is an inductive invariant of the Ⓑ and Ⓒ (though not necssarily an invariant that separates the initial states from the error states–GPS will determine this later and uncover these paths). GPS then detects that π_4 is a dead end, producing Figure 8d—note that π_2 and π_3 become *uncovered*, because the label of their parent has changed and so they are no longer considered redundant. GPS detects that π_3 is a dead end to produce Figure 8e, and finally the covering edges from π_2 to its parent is reinstated to arrive at the complete and well-labeled ART in Figure 8f, and GPS reports that the assertion is safe.

Theorem 1 (Soundness of GPS). *Let G be a weighted graph. If GPS(G) returns* Safe, *then G is safe; if it returns* (Unsafe, π)*, then π is a feasible path from the source of G to its target.*

```
void main() {
  int s0, s1, s2, c;
  int state = 0;
  while (1) {
    c = *; s0 = *; s1 = *; s2 = *;
    /* state machine */
    if (c == 68 ∧ state == 0) state = 1;
    if (c == 79 ∧ state == 1) state = 2;
    if (c == 71 ∧ state == 2) state = 3;
    if (state == 3 ∧ s0 == 67 ∧ s1 == 65 ∧ s2 == 84)
        error();
  }
}
```

(a) An example program implementing a state-machine that loops indefinitely until the user enters a correct input of ASCII-encoded characters `C, A, T` at once (via $s0 - s2$) and `D, O, G` in succession (via c).

```
void main() {
  int s0, s1, s2, c;
  int state = 0;
  int gas = *; // instrumented
  while (1) {
    if (gas-- < 0) break; // instrumented
    c = *; s0 = *; s1 = *; s2 = *;
    /* state machine */
    if (c == 68 ∧ state == 0) state = 1;
    if (c == 79 ∧ state == 1) state = 2;
    if (c == 71 ∧ state == 2) state = 3;
    if (state == 3 ∧ s0 == 67 ∧ s1 == 65 ∧ s2 == 84)
        error();
  }
}
```

(b) Instrumented version of the program on the left.

Fig. 9. An example illustrating a class of loop patterns on which the refutation-completeness transformation for GPS also speeds up convergence.

5.3 Refutation Completeness

The GPS procedure described in Section 5.2 is not complete for refutation: if the program has a non-terminating execution, it is possible for the Explore procedure in Figure 6 to diverge. In fact, the *only* obstacle to refutation completeness is when the Explore(−) procedure fails to terminate by executing a test input that leads to a trace of infinite length. To address this issue, this section describes a transformation that makes GPS complete by bounding the depth of test executions. We also formally state and prove a refutation-completeness result. The detailed proof may be found in the extended version of this paper [19].

Instrumentation with the global variable `gas`. GPS instruments the entire program using a global variable named gas. The purpose of the instrumentation is to ensure the termination of Explore by using gas to represent a model-checking bound. Using the havoc(−) primitive, we treat gas as being initialized to a non-deterministic, positive integer value. At each loop-header vertex, we add an assumption that gas ≥ 0 and a statement to decrement gas by 1. In a nutshell, the gas-bound instrumentation ensures termination of (calls to) the Explore(−) procedure (Figure 4). Without gas-instrumentation, GPS might generate a test that is unbounded in length (e.g., by entering a non-terminating loop), in which case the Explore(−) procedure would get stuck executing this infinite test, therefore breaking refutation completeness. Intuitively, to guarantee refutation completeness we need to make sure that the algorithm fairly explores all traces of the program-under-test. This property is in turn used in the proof of refutation completeness. (See the proof of Lemma 9 in the extended version of this paper [19])

Theorem 2 (Refutation completeness). *Let G be a weighted graph, instrumented with gas as above. If P is unsafe, then GPS(P) terminates with the result "Unsafe."*

Discussion. The purpose of gas-instrumentation is to obtain refutation-completeness as a theoretical guarantee of GPS. Its effect on *practical* performance is another matter: on the one hand, gas may be helpful by terminating exploration of paths that do not lead to the error location, but on the other it may be harmful by terminating exploration of paths that would (if provided with enough gas). We will return to this question in our experimental evaluation (Section 7). For the moment, we remark that the potential negative effects of the transformation are mitigated by the use of summaries in GPS: by performing gas-instrumentation before computing summaries, we obtain summaries that (can) provide useful information about how long an error path must be. In the remainder of this section, we discuss such an example.

Consider the program in Figure 9. This program implements a small state machine accepting user inputs via the variables c, $s1$, $s2$, and $s3$. To reach the error location, GPS's test generator has to *both* guess the value for c correctly across three successions of the loop—as well as guessing correct values for variables $s0$, $s1$, and $s2$ when the state machine is in its accepting state.

Note that this example also falls within the category of "Lock & Key" programs we discussed in Section 2—here, the key to unlock a feasible path-to-error is precisely the sequence of inputs required to arrive at the accepting state of the state machine. Without performing gas-instrumentation, GPS times out when running its first test. The essential problem is that the Explore algorithm uses the path-to-error summary to maintain the invariant that the state *may* reach the error location, but there is no inherent bias for an SMT solver to choose values for the $c, s1, s2$, and $s3$ variables that make *progress* towards the error. Because every execution can be extended to reach the error location, Explore may simply choose the value 0 for all non-deterministic expressions and enter an infinite loop. In contrast, once the instrumentation is performed (the result is shown in Figure 9(b)), a summary-generating static analysis—such as compositional recurrence analysis [20]—is able to generate a loop summary of the following form:

$$\exists k. k \geq 0 \land gas' = gas - k \land state \leq state' \leq state + k,$$

which, in essence, forces the test generator to select paths in which variable *state* changes between loop iterations, because the *possible* change in *state* is bounded by the decrease in *gas* (e.g., from a state with $\{gas \mapsto 2; state \mapsto 0\}$, GPS *must* select $c = 68$, because if variable *state* fails to advance then the error state cannot be reached while consuming ≤ 2 units of *gas*). As such, the gas-instrumentation technique is potentially beneficial in *helping GPS generate higher-quality tests* in loops where non-deterministically-valued expressions can lead to non-terminating executions.

6 Summary Computation

The GPS algorithm heavily relies on the use of path summaries: the test generator uses summaries to generate tests; the test executor uses summaries to help guide path selection; summaries are also instrumental in helping prove that some paths are dead ends. In this section, we present background information on how *path summaries* may be computed via static analysis—in particular, algebraic program analysis [33].

Algebraic program analysis is a program-analysis framework in which one analyzes a program, represented by a control-flow graph (CFG), by (1) computing a regular expression recognizing a set of paths of interest, and then (2) interpreting that regular expression over an algebra corresponding to the analysis task at hand. Typically, step (1) is accomplished with an algorithm for the *single-source path-expression problem*, which computes, for some fixed graph G and vertex u, a path expression $PathExp_G(u, v)$ recognizing the set of paths $Paths_G(u, v)$ for each vertex $v \in V$. In this paper, we are interested in using algebraic program analysis to compute transition formulas that summarize sets of paths in a CFG, represented by a weighted graph $G = (V, E, w, s, t)$. There are several program analyses that can be used for this task, which operate in the same fashion [14, 20, 32, 45]. The idea is to instantiate the interpretation step (step (2)) of algebraic program analysis to evaluate regular expressions to a transition formula by interpreting each edge $e \in E$ as its weight $w(e)$, 0 as $\perp$, 1 as $\bigwedge_{x \in X} x' = x$, + as $\lor$, $\cdot$ as $\circ$, and $(-)^*$ as some over-approximate reflexive-transitive-closure operator. The analyses referred to above differ essentially in their choice of the operator used to interpret

$(-)^*$. In our implementation of GPS, we use a variant of *Compositional Recurrence Analysis* (CRA) [20], which, given a formula F representing a loop body, computes F^* by extracting recurrence relations from F and computing their closed forms.

6.1 Efficient Offline Computation of Single-Target Summaries

As previously discussed, in line 9 of the Explore(-) procedure (Figure 5), GPS relies on path-summary queries from an arbitrary CFG vertex u to the terminal vertex, and such queries are made on almost every step of executing a test. Thus, relying on a path-summary query via Tarjan's algorithm [46] would incur a somewhat expensive single-source path-expression query with a different vertex at almost every step of the GPS algorithm. This section describes a key optimization that pre-computes *all* static queries required by GPS offline, using a single call to Tarjan's algorithm, thus lowering pre-computation costs by a linear factor and saving the work of re-analyzing loops.

In particular, we can *dualize* Tarjan's algorithm to solve the *single-target path-expression* problem: i.e., given graph G and vertex v, for each vertex $u \in V$ compute (in near-linear time) a path expression $PathExp_G(u, v)$ whose language is the set of paths $Paths_G(u, v)$. This step is accomplished via the following three-step process:

- **Step 1: Reverse weighted graph G.** Given $G = (V, E, w, s, t)$, its reversal $G^R = (V, E^R, w^R, t, s)$ reverses the direction of all edges and swaps the s and t vertices. Formally, $E^R \triangleq \{(v, u) : (u, v) \in E\}$ and $w^R(u, v) \triangleq w(v, u)$.
- **Step 2: Define a new sequential-composition operator $\circ^R$.** Define the *reversed* sequential-composition operator $F \circ^R G \triangleq G \circ F$, which reverses the direction of composition of two transition formulas $F, G \in \mathbf{TF}(X)$ to compensate for the fact that all edge directions in G^R have been reversed. Note that $F_n \circ^R \ldots \circ^R F_1 = F_1 \circ \ldots \circ F_n$.
- **Step 3: Run the usual analysis on G^R, using $\circ^R$ to interpret $(\cdot)$.** Given the results of the two steps above, one can run algebraic program analysis as usual, using Tarjan's algorithm to compute path expressions over the reversed weighted graph G^R with $\circ^R$ as the new sequential-composition operator, producing a transition formula $F(v)$ for each vertex v. We have that $F(v)$ over-approximates all paths in $Paths(G, v, t)$.

Via dualization, one can efficiently compute all single-target path expressions off-line before invoking the GPS algorithm. As such, the queries discharged by the test-execution procedure Explore(−) can be answered efficiently.

7 Implementation and Evaluation

We prototyped a software model checker for C implementing the GPS algorithm, using a variant of compositional recurrence analysis (CRA) [20] for path summarization, and a variant of Newton refinement [17] for interpolation. The implementation of Newton refinement, and thus the GPS algorithm, is limited to programs that can be expressed in linear integer arithmetic (LIA; in particular, our implementation cannot handle programs with pointers or arrays). We disabled non-linear invariant generation in CRA to ensure that summaries are expressed in LIA. These limitations are features of our implementation and not intrinsic to GPS *per se.* As described here, GPS is an intra-procedural algorithm; our implementation also makes a best-effort attempt to in-line procedure calls, and then analyzes the resulting call-free program.

Research Questions. GPS combines components from testing, static analysis, and software model checking. We start with two research questions aimed towards understanding how each of these components contribute to the overall performance of GPS:

- **RQ #1 (Effectiveness of different GPS components):** Which features of GPS are crucial in its overall performance?

Table 1. Details about the L&K programs benchmark suite. 10 benchmarks of different parametrized values in range [10, 10000] are created for each parametrizable benchmark.

Benchmark	Type	Description	Source
`ccbse-refute1.c`	Parametrizable	Example from Fig. 4 of Ma et. al	[35]
`ccbse-refute2.c`	Parametrizable	Example from Fig. 6 of Ma et. al	[35]
`godefroid-issta-1a.c`	Parametrizable	Example from Fig. 1a of Godefroid et. al	[23]
`godefroid-issta-1b.c`	Parametrizable	Example from Fig. 1b of Godefroid et. al	[23]
`godefroid-issta-2.c`	Non-parametrizable	Example from Fig. 2 of Godefroid et. al	[23]
`lese.c`	Parametrizable	Example from Saxena et. al	[41]
`majumdar-icse07.c`	State machine	10-state, 7-input string-processing state machine	[36]
`godefroid-ndss09.c`	State machine	5-state, 4-input string-processing state machine	[40]
`refcomplete-ex.c`	State machine	4-state, 4-input state machine example from Section 5.3	Synthetic
`cadabra.c`	State machine	8-state state machine accepting 'cadabra'	Synthetic
`abracadabra.c`	State machine	12-state state machine accepting 'abracadabra'	Synthetic
`abracadabraabra.c`	State machine	16-state state machine accepting 'abracadabraabra'	Synthetic
`abracadabraabracadabra.c`	State machine	23-state state machine accepting 'abracadabraabracadabra'	Synthetic

- **RQ #2 (GPS vs. baseline algorithms):** GPS builds upon both CRA (its underlying summarization algorithm) and Impact [3, 6, 37] (the basis of its invariant-synthesis mechanism). How does GPS compare with each?

Our last research question compares GPS with the state of the art:

- **RQ #3 (Overall effectiveness of the GPS algorithm):** How does GPS compare against state-of-the-art tools, especially the top-performers in the ReachSafety-Loops category of SV-COMP?

Benchmark Selection. We performed the comparisons on three suites of benchmarks, with 297 programs in total.

- **Suite #1: 230 intraprocedural benchmarks from SV-COMP.** We sub-setted 230 benchmarks from the ReachSafety-Loops category of the Software Verification Competition (SV-COMP) 2024 benchmark suite [5], filtering out *only* programs containing non-LIA syntax.
- **Suite #2: 9 Safety-verification examples from this paper.** We created a benchmark suite (of safe programs) consisting of examples **EX-2** and **EX-3** in Section 2, as well as 7 variants of these examples that exhibit similar loop patterns; this suite includes a total of 9 programs.
- **Suite #3: 58 Lock & Key-style unsafe benchmarks.** We created a suite of "L&K"-style benchmarks similar to **EX-1** discussed in Section 2, as well as the state-machine example discussed in the section on `gas`-instrumentation (Section 5.3). This suite of benchmarks consists of the following benchmarks (for details, see Table 1):
  1. 6 programs with "L&K"-style, input-dependent loops from prior literature [23, 35, 36, 41]—each program in this collection features paths to errors that can only be discovered by successfully guessing the "key"—i.e., there are non-deterministically-valued expressions inside the program.
  2. We further observed that 5 of the 6 L&K examples are *parametrizable*—that is, the length of the shortest feasible path-to-error can be prolonged by enlarging a constant inside the program, i.e., changing the "lock." To test the robustness of GPS in handling these examples, we benchmarked the tools in our comparison on 10 different values of the parameter (in the range [10, 10000]) for each of the parametrizable programs.
  3. The remaining 7 benchmarks are loops implementing string-processing state machines where an assertion violation is triggered when the state machine arrives at an accepting state. As shown in Table 1, this suite includes two examples from prior work [36, 40], the state-machine example from Section 5.3, and four synthetic state-machine examples.

  In total, this benchmark suite consists of 58 unsafe programs (i.e., 1 unparametrizable benchmark, 50 parametrized benchmarks, and 7 state machine benchmarks).

Table 2. Overall benchmark statistics for RQ #1 and #2. The Failures row contains triples $T/M/U$ indicating the number of benchmarks for which the tool exceeded the Time bound, exceeded the Memory bound, or reported Unknown, respectively. "GPS" refers to unablated version of GPS, "GPSLite" refers to the version of GPS without abstraction refinement presented in Section 4, "GPS-nogas" refers to ablated GPS without gas-instrumentation, "GPS-nogas-nosum" refers to ablated GPS without CRA-generated summaries, "CRA" refers to only using CRA to prove safety, and Impact refers to the lazy abstraction with interpolants implementation in CPAChecker.

		GPS		GPSLite		GPS-nogas		GPS-nogas-nosum		Impact		CRA	
	#tasks	#correct	time	#correct	time	#correct	time	#correct	time	#correct	time	#correct	time
Suite #1 (SV-COMP)	230	**214**	7389.1	200	10302.9	210	7845.7	168	37969.2	119	68616.9	155	752.3
Safe	191	**180**	4852.7	166	8301.5	176	5482.0	136	33948.5	95	59559.3	155	746.4
Unsafe	39	**34**	2536.4	**34**	2001.4	**34**	2363.7	32	4020.7	24	9057.5	0	5.9
Suite #2 (Safe Exs)	9	**9**	6.0	6	837.5	**9**	6.4	1	2780.7	2	4209.4	1	0.7
Suite #3 (L&K)	58	**57**	623.3	**57**	620.9	54	1749.5	2	27718.5	15	27808.9	0	22.6
Unparametrized	1	**1**	0.2	**1**	0.2	**1**	0.2	**1**	0.2	**1**	2.3	0	0.2
Parametrized	50	**50**	10.8	**50**	8.5	**50**	8.9	1	23511.6	11	25370.3	0	7.9
State Machines	7	**6**	612.3	**6**	612.2	3	1740.4	0	4206.7	3	2436.3	0	14.5
Total	297	**280**	8018.4	263	11761.3	273	9601.6	171	68468.4	136	100635.2	156	775.6
Failures		8/9/0		13/21/0		9/15/0		110/16/0		160/0/1		2/0/139	

Table 3. Overall benchmark statistics for RQ #3. Times are in seconds. The Failures row contains triples $T/M/U$ indicating the number of benchmarks for which the tool exceeded the Time bound, exceeded the Memory bound, or reported Unknown, respectively.

		GPS		VeriAbs		Symbiotic		CPAChecker (Portfolio)	
	#tasks	#correct	time	#correct	time	#correct	time	#correct	time
Suite #1 (SV-COMP)	230	**214**	7389.1	201	12362.6	158	62130.2	146	64674.9
Safe	191	**180**	4852.7	169	9223.0	126	56825.7	120	56785.3
Unsafe	39	**34**	2536.4	32	3139.6	32	5304.5	26	7889.6
Suite #2 (Safe Exs)	9	**9**	6.0	7	752.7	2	4667.6	2	4617.7
Suite #3 (L&K)	58	**57**	623.3	29	14853.2	23	20700.5	40	14751.1
Unparametrizable	1	**1**	0.2	0	392.7	**1**	0.7	**1**	3.3
Parametrizable	50	**50**	10.8	22	14393.0	17	20425.6	32	14054.4
State Machines	7	6	612.3	**7**	67.5	5	274.2	**7**	693.4
Total	297	**280**	8018.4	237	27968.5	183	87498.3	188	84043.7
Failures		8/9/0		9/9/42		96/18/0		108/0/1	

Evaluation Setup. We ran all experiments on a Dell XPS 15 laptop running Ubuntu 24.04 LTS with an Intel i7-13700H CPU and 16 GiB of RAM. We used the benchexec script standard for evaluating tools in SV-COMP [9]. The CPU that we used has 6 high-performance cores and 8 efficient cores. To reduce variance in benchmark timings, we configured benchexec to only use the 6 high-performance cores. For all benchmarks, we used benchexec to enforce a timeout threshold of 600 seconds and a memory limit of 15GB.

We now address each research question in turn.

RQ #1: Which features of GPS are crucial in its overall performance?

There are several key ingredients to GPS: (1) the path summaries generated by a static-analysis tool (in our case, CRA); (2) the test-generation and refinement procedures inside the GPS model-checking algorithm; (3) the use of the instrumentation variable gas to make GPS refutation-complete. To understand the impact of the various components of GPS, we performed an ablation study with

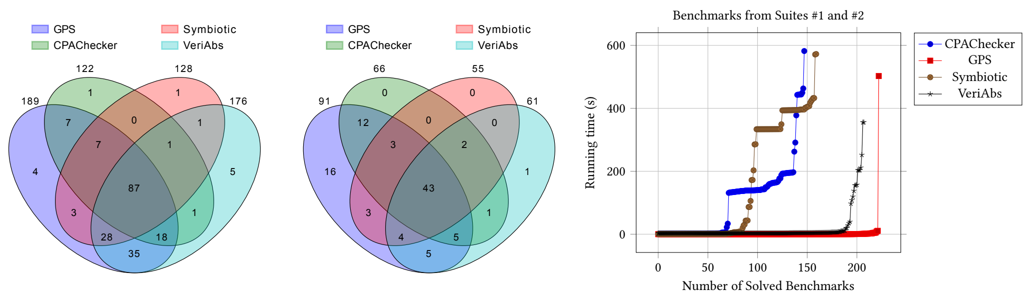


Fig. 10. Benchmark results for RQ #3. From left to right: (a): Venn diagram of safe benchmark results, across all three benchmark suites, for RQ #3 (200 benchmarks total); (b): Venn diagram of unsafe benchmark results, across all three benchmark suites, for RQ #3 (96 benchmarks total); (c) Cactus plot of results for RQs #3.

several variation of GPS, the results of which are reported in Table 2. Below, we describe each variation and the result of the comparison.

- **(GPS-nogas-nosum).** Instead of using CRA as the summary oracle, we use a trivial oracle ($\text{Sum}(G, u, v) = \top$ for all vertices u and v), and additionally we do not perform gas-instrumentation because some of the other tools that we wanted to compare with also did not perform gas-instrumentation. This variation essentially omits the main novel features of GPS: with trivial summaries, summary-directed test generation coincides with Dart-style directed test generation, dead-end detection always fails, and dead-end interpolation coincides with Impact-style infeasible-path interpolation. The resulting algorithm essentially follows the tradition of Synergy, Dash, and McVeto in that it combines concolic testing with abstraction refinement.
  We find that GPS-nogas-nosum performs significantly worse than GPS on both safe and unsafe benchmarks. The impact of using summaries to generate high-quality tests is apparent when one compares GPS with GPS-nogas-nosum on the unsafe SV-COMP suite (on which GPS-nogas-nosum does well, because that suite is dominated by deterministic benchmarks where summary-guidance is not needed) with the L&K suite (on which it does poorly due to absence of this guidance).
- **(GPSLite)** We measure the impact of invariant synthesis through dead-end interpolation (Section 5) by comparing GPS with GPSLite (Section 4, instantiated with gas-instrumentation), which omits this feature. We find that GPSLite is somewhat faster than GPS on unsafe benchmarks (because it avoids the overhead of computing interpolants), but GPS dominates GPSlite on safe benchmarks.
- **(GPS-nogas).** We study the repercussions of the refutation-completeness transformation described in Section 5.3 by comparing GPS with GPS-nogas, a variant of GPS without the transformation. We find that the theoretical guarantee of refutation-completeness translates to positive effect on the practical performance of GPS.

RQ #2: How does GPS compare with baseline techniques it builds upon?

To address RQ #2, we benchmarked GPS against a re-implementation of the Impact algorithm inside CPAChecker [6, 7]. We configured the latest release[4] of CPAChecker to use the option "`-predicateAnalysis-ImpactRefiner-SBE.properties`", which tells CPAChecker to use the Impact algorithm under the hood. The results for Impact are given in Table 2.

[4]Version 4.0 at the time of writing.

Findings. One way to answer RQ #2 is by thinking of the starting point as Impact (see Table 2, column 6), which uses Craig interpolation and abstract reachability trees, to which we have added (i) a different search strategy combining breadth-first search and concolic testing, and (ii) the novel features of summary-directed testing, dead-end detection, and dead-end interpolation. Here, to control for Impact not using gas instrumentation, we use GPS-nogas as the right end-point:

$$\textsc{Impact}\ \left(\frac{136}{297}\right) \xrightarrow[\text{search strategy}]{+35} \text{GPS-nogas-nosum}\ \left(\frac{171}{297}\right) \xrightarrow[\text{novel features}]{+102} \text{GPS-nogas}\ \left(\frac{273}{297}\right)$$

While the different search strategy gives a substantial improvement in the number of benchmarks solved, the novel features of GPS provide a much larger one. The overall improvement is almost a factor of 2x over Impact. (The full GPS algorithm did even better: 280/297.)

Another way to answer RQ #2 is by thinking of the starting point as CRA (see Table 2, column 7), with its abilities to create summaries, and to look at the improvement in verification capability—the number of safe benchmarks solved—obtained as we add dead-end detection and then dead-end interpolation to the CRA baseline. We see the following improvement, with roughly equal contributions by each:

$$\text{CRA}\ \left(\frac{156}{200}\right) \xrightarrow[\text{dead-end detection}]{+16} \text{GPSLite}\ \left(\frac{172}{200}\right) \xrightarrow[\text{dead-end interpolation}]{+17} \text{GPS}\ \left(\frac{189}{200}\right)$$

While the overall improvement in the number of benchmarks solved, compared to CRA, is a respectable 21.2%, GPS also adds the ability to find counter-examples (and is refutation-complete).

RQ #3: How does GPS compare against state-of-the-art tools?

To address RQ #3, we compared our prototype implementation of GPS against three portfolio-based model checkers that implement a diverse range of techniques.

Baseline Comparisons for RQ #3. We compared GPS against three other model checkers:

- **VeriAbs** [1], a portfolio of model checkers, which (along with its variant VeriAbsL) ranked first (and second place, for VeriAbsL) in the SV-COMP 2024 ReachSafety-Loops category.
- **Symbiotic** [29], a portfolio-based symbolic-execution tool, which was the third-best performer of the SV-COMP 2024 ReachSafety-Loops category.
- **CPAChecker** [7], a portfolio-based model checker implementing a suite of different strategies; although CPAChecker was not a top scorer in the ReachSafety-Loops category of SV-COMP 2024, the motivation for including it was to compare GPS with CPAChecker's portfolio mode, because RQ #2 compared GPS against the Impact implementation inside CPAChecker.

For each tool involved in this comparison, we used the version and configuration provided for its SV-COMP 2024 contest entry. In particular, in this experiment, both Symbiotic and CPAChecker were used in portfolio modes as specified in their competition entries [15, 29].

Findings. Key statistics about this comparison are shown in Table 3. We observe that GPS has the best performance (both in terms of the number of successful benchmarks and running time) for all benchmark suites. A cactus plot depicting a more fine-grained view of success rate vs. running time appears in the right of Figure 10.

Our result confirm that GPS has excellent performance on L&K examples, owing to the fact that it uses summaries to generate a high-quality test case and then efficiently simulates execution on that test case. Other tools are more sensitive to the length of the shortest counter-example, which we illustrate in the extended version of this paper [19]. GPS solves all except one L&K

benchmarks, with the one unsolved benchmark being a 23-state state machine benchmark on which the underlying static analysis (CRA) times out when generating the initial path summary.

Venn diagrams for the sets of benchmarks on which each tool is successful can be found in Figure 10, from which we conclude that GPS can be profitably incorporated into portfolio solvers.

8 Related Work

GPS builds upon three different lines of work: directed testing, static analysis, and software model checking—we discuss each of the three below.

Directed test generation and execution. *Directed testing* sometimes refers to a technique that aims to maximize code coverage (e.g., DART [22]); in contrast, GPS directs testing toward a particular error location. More in line with the goals of GPS is *directed greybox fuzzing* [10, 12, 28, 43], which aims to generate test inputs that reach a designated error location. Perhaps most similar to our technique is Beacon [28], which uses a backwards static analysis (propagating information from the error location back to the entry of the program) to over-approximate the set of states that *may* reach the error. GPS uses a dualized variant of Tarjan's algorithm to compute path-to-error summaries for each location to achieve a similar result. *Directed symbolic execution* [35] uses heuristics for guiding a symbolic-execution algorithm toward a particular error location, instead of maximizing coverage. The main difference between this work and GPS is that GPS employs summarization and program instrumentation to guide the search toward particular paths to error, whereas directed symbolic execution leverages a distance metric over the structure of an interprocedural control-flow graph. The salient difference between GPS and the aforementioned techniques is that GPS uses a synergistic interaction of static-analysis and software-model-checking techniques that make it capable of *verifying* safety properties while achieving a completeness result for property refutation, even for programs with an unbounded state space.

Saxena et al. [41] identified programs like EX-1 (Figure 1) as being challenging for concolic-execution engines. EX-1 is an L&K problem in which there is no chain of flow dependences between variable `N` and the branch-condition "`min < 1000`," but only *implicit flow* [16] between them. In compiler terminology, `N` influences `min` only via the *control dependence* [21] from `i < N` to `min++`.

Invariant checking and synthesis. The invariant-finding features of GPS consist of two components: (1) invariant synthesis from static-analysis-generated summaries and Craig interpolation, and (2) an Impact-based [37] invariant-checking procedure to check whether the synthesized invariants are inductive. We discuss prior work related to each component below.

- *Static analysis and invariant generation.* A recent line of work on algebraic program analysis [33] demonstrates that it is possible to design summary-based static analyses that are competitive with state-of-the-art software model checkers in terms of verification ability. The design of GPS was inspired by the question of whether this precision could be harnessed for property *refutation* as well. In principle, GPS can be instantiated with any backward static-analysis algorithm, and the dualization of Tarjan's method presented in Section 6.1 can be seen as a way of implementing a backward analysis using algebraic program analysis. From this perspective, the crucial feature for GPS's success is the fact that algebraic program analyses can be very precise, whereas historically backward analyses are imprecise (relative to forward analyses).
- *Invariant checking.* GPS uses the covering procedure in Impact to test whether ART labels constitute an inductive invariant. The induction principle underlying the covering procedure is as follows: In the ART, we add a covering edge from path $\pi\tau$ to its ancestor π when the label at π is an inductive invariant for τ, and τ is an m-step unfolding of a loop (for some m). Thus, the covering procedure (in GPS and Impact) tests whether an ART label is an invariant that is inductive with respect to an m-step unfolding of some loop.

Table 4. A comparison between GPS and related algorithms. The "Invariant Synthesis" column refers to whether the algorithm considers a single path or multiple paths when proposing candidate invariants.

	Only explores feasible paths	Invariant checking/synthesis	Refutation complete
GPS (Our work)	✓	Set of paths	✓
Dash [4]	✗	Set of paths	✗
Impact [37]	✗	Single path	✗
Dart [22]	✓	N/A	✗

A different, but related, approach is to test whether invariants are k-inductive using the principle of k-induction [44]; however, the k-induction principle is incomparable to the induction principle behind Impact and GPS. See the extended version of our paper [19] for a more thorough discussion of this topic. There is also work that leverages the k-induction principle as the underlying invariant checker in an invariant-synthesis engine [26, 30, 48]; incorporating the k-induction principle into the GPS framework is an interesting avenue for future work.

Software model checking. There has been much work on software model checking; we focus on the most relevant works below, and give a comparison between GPS and related algorithms in Table 4.

GPS builds on *lazy abstraction with interpolants* (Impact) [37], and similarly uses an abstract reachability tree to store explored states, and interpolation to perform refinement. There are two key innovations that GPS provides. First, Impact expands the ART by a depth-first search, which is in fact *not* refutation-complete—as the algorithm can be forced to always extend the ART in an unbalanced manner (e.g., by always extending the *leftmost* leaf). In contrast, GPS builds the tree by simulating execution from summary-directed tests and formally guarantees refutation-completeness, as the frontier paths in GPS are explored in a breadth-first manner. Second, while Impact proves safety one source-to-target path at a time, GPS uses path-to-target summaries to prove safety of regular sets of such paths.

Perhaps most related to GPS's idea of combining test execution with interpolation-based refinement is the line of work on Synergy [25] and its successors Dash [4], Smash [24], and McVeTo [47]. Synergy's test-generation strategy might be thought of as an intermediate point between generating tests to maximize coverage and to reach an error location. It finds a path to error through a finite-state abstraction of the program, and then uses the frontier edge of that path (which crosses from the part of the state space that has been explored to the part that has not) to generate a new test. Using path summaries, GPS is able take the entire error path into account for test generation. For instance, in EX-1, GPS discovers the bug using a single test, whereas the Synergy family of algorithms requires 1000 tests, each test unrolling the loop one more step.

Lazy Annotation [38] features an instrumentation step similar to GPS's instrumentation-by-gas procedure to aid the algorithm's convergence in the face of unbounded loops; however, lazy annotation does not explicitly guarantee refutation-completeness, nor does it investigate the impact of such an instrumentation procedure on algorithm performance.

Refutation-completeness. GPS benefits from the gas-instrumentation procedure (Section 5.3) for being refutation-complete; however, such a procedure only works due to how GPS performs refinement in *breadth-first order* but eagerly explores generated tests in *depth-first order.* This unique, two-layered search strategy allows GPS to efficiently explore long paths-to-errors, while still retaining refutation-completeness due to the breadth-first refinement order. In contrast, Impact-like algorithms [2, 3, 8, 13, 37]—many of which are described as algorithms implementing depth-first refinement—suffer from an inherent tension between achieving refutation-completeness and the issue of efficiently handling deep counterexamples. In particular, these algorithms can either be modified to explore program states (and refine) in breadth-first order, and thus achieve refutation-completeness at the expense of performance, or perform both exploration and refinement in depth-first order, thus sacrificing refutation-completeness. When restricted to intraprocedural programs, the Whale algorithm of Albarghouthi et al. [2] implements Impact with breadth-first refinement and is thus refutation-complete, but (due to the tension described above) does not address the issue of handling deep counterexamples.

Reasoning about sets of paths. This paper introduces *dead-end interpolation* as a means for generating candidate invariants that are high-quality in the sense that they are sufficient for proving that a (possibly infinite) set of paths cannot reach the error location. Large-block encoding [6] shares a similar goal of enabling software model checkers to reason about sets of paths; in contrast to our work these sets are always finite. In a similar spirit, Whale [2] and Ufo [3] extend Impact to operate over *abstract reachability graphs* (ARGs), and perform refinement along subgraphs of the ARG. This approach also allows these algorithms to refine multiple paths at a time, but again each subgraph only represents a finite number of paths. Another related technique is acceleration [27], which can be used to reason about infinitely many paths, and which is similar in spirit to summarization. They differ in that acceleration is *precise*, and applies only to loops of a particular form, whereas summarization is over-approximate and broadly applicable.

9 Conclusion

We presented a novel algorithm, GPS, for checking control-state reachability of intraprocedural programs. Our experiments confirmed that GPS is a particularly performant model-checking algorithm that can handle both challenging verification tasks and challenging refutation tasks. GPS opens up several exciting avenues for future work. First, the way GPS employs static analysis-generated summaries to help both test generation and refinement means that *any improvement* in the precision of the (summary-generating) static analysis leads to an improvement in both the quality of the a directed test and an interpolant sequence generated by GPS. Thus, any future work on more precise path summary-generation methods would immediately benefit GPS. It is also interesting to consider using GPS to answer intra-procedural reachability queries in an inter-procedural model checking algorithm such as Spacer [34]. Finally, it would be interesting to consider whether GPS can be extended to handle richer classes of programs, such as those manipulating arrays, pointers, or heaps.

Data-Availability Statement

The artifact related to this paper (including benchmarks, an implementation of GPS, CRA, as well as other tools we compared against in the evaluations) is packaged as a Docker container on Docker Hub: https://hub.docker.com/r/ruijiefang/gps-oopsla25-ae/. Instructions for using the Docker container, as well as all benchmarks used, are available at the following Zenodo Link: https://doi.org/10.5281/zenodo.16914159 [18].

Acknowledgments

This work was supported in part by the NSF under grant number 1942537. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] Mohammad Afzal, A. Asia, Avriti Chauhan, Bharti Chimdyalwar, Priyanka Darke, Advaita Datar, Shrawan Kumar, and R. Venkatesh. 2019. VeriAbs : Verification by Abstraction and Test Generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1138–1141. https://doi.org/10.1109/ASE.2019.00121

[2] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. 2012. Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In *Verification, Model Checking, and Abstract Interpretation*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 39–55. https://doi.org/10.1007/978-3-642-27940-9_4

[3] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. 2012. Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 672–678. https://doi.org/10.1007/978-3-642-31424-7_48

[4] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. 2008. Proofs from Tests. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1390630.1390634

[5] Dirk Beyer. 2024. State of the Art in Software Verification and Witness Validation: SV-COMP 2024. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 299–329. https://doi.org/10.1007/978-3-031-57256-2_15

[6] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, Simon Fraser University, and Roberto Sebastiani. 2009. Software model checking via large-block encoding. In *2009 Formal Methods in Computer-Aided Design.* 25–32. https://doi.org/10.1109/FMCAD.2009.5351147

[7] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[8] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. 2010. Predicate abstraction with adjustable-block encoding. In *Formal Methods in Computer Aided Design.* 189–197.

[9] Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2019. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer* 21, 1 (2019), 1–29. https://doi.org/10.1007/s10009-017-0469-y

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[11] Aaron R. Bradley. 2012. Understanding IC3. In *Theory and Applications of Satisfiability Testing – SAT 2012*, Alessandro Cimatti and Roberto Sebastiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–14. https://doi.org/10.1007/978-3-642-31612-8_1

[12] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2095–2108. https://doi.org/10.1145/3243734.3243849

[13] Alessandro Cimatti and Alberto Griggio. 2012. Software Model Checking via IC3. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 277–293.

[14] John Cyphert and Zachary Kincaid. 2024. Solvable Polynomial Ideals: The Ideal Reflection for Program Analysis. *Proc. ACM Program. Lang.* 8, POPL, Article 25 (Jan. 2024), 29 pages. https://doi.org/10.1145/3632867

[15] Priyanka Darke, Sakshi Agrawal, and R. Venkatesh. 2021. VeriAbs: A Tool for Scalable Verification by Abstraction (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 458–462. https://doi.org/10.1007/978-3-030-72013-1_32

[16] D.E. Denning and P.J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (1977), 504–513. https://doi.org/10.1145/359636.359712

[17] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. 2017. Craig vs. Newton in software model checking. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 487–497. https://doi.org/10.1145/3106237.3106307

[18] Ruijie Fang, Zachary Kincaid, and Thomas Reps. 2025. Artifact for "Software Model Checking via Summary-Guided Search". https://doi.org/10.5281/zenodo.16914159

[19] Ruijie Fang, Zachary Kincaid, and Thomas Reps. 2025. Software Model Checking via Summary-Guided Search (Extended Version). arXiv:2508.15137 [cs.PL] https://arxiv.org/pdf/2508.15137

[20] Azadeh Farzan and Zachary Kincaid. 2015. Compositional recurrence analysis. In *2015 Formal Methods in Computer-Aided Design (FMCAD).* 57–64. https://doi.org/10.1109/FMCAD.2015.7542253

[21] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. *SIGPLAN Not.* 40, 6 (jun 2005), 213–223. https://doi.org/10.1145/1064978.1065036

[23] Patrice Godefroid and Daniel Luchaup. 2011. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 23–33. https://doi.org/10.1145/2001420.2001424

[24] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. 2010. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 43–56. https://doi.org/10.1145/1706299.1706307

[25] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. 2006. SYNERGY: A New Algorithm for Property Checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Portland, Oregon, USA) *(SIGSOFT '06/FSE-14)*. Association for Computing Machinery, New York, NY, USA, 117–127. https://doi.org/10.1145/1181775.1181790

[26] Arie Gurfinkel and Alexander Ivrii. 2017. K-Induction Without Unrolling. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 1–9. https://doi.org/10.23919/FMCAD.2017.8102243

[27] Hossein Hojjat, Radu Iosif, Filip Konečný, Viktor Kuncak, and Philipp Rümmer. 2012. Accelerating Interpolants. In *Automated Technology for Verification and Analysis*, Supratik Chakraborty and Madhavan Mukund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 187–202. https://doi.org/10.1007/978-3-642-33386-6_16

[28] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*. 36–50. https://doi.org/10.1109/SP46214.2022.9833751

[29] Martin Jonáš, Kristián Kumor, Jakub Novák, Jindřich Sedláček, Marek Trtík, Lukáš Zaoral, Paulína Ayaziová, and Jan Strejček. 2024. Symbiotic 10: Lazy Memory Initialization and Compact Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 406–411. https://doi.org/10.1007/978-3-031-57256-2_29

[30] Dejan Jovanovic and Bruno Dutertre. 2016. Property-Directed k-Induction. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 85–92. https://doi.org/10.1109/FMCAD.2016.7886671

[31] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. 2017. Compositional recurrence analysis revisited. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 248–262. https://doi.org/10.1145/3062341.3062373

[32] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. 2017. Non-linear reasoning for invariant synthesis. *Proc. ACM Program. Lang.* 2, POPL, Article 54 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158142

[33] Zachary Kincaid, Thomas Reps, and John Cyphert. 2021. Algebraic Program Analysis. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 46–83. https://doi.org/10.1007/978-3-030-81685-8_3

[34] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 17–34. https://doi.org/10.1007/978-3-319-08867-9_2

[35] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Static Analysis*, Eran Yahav (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 95–111. https://doi.org/10.1007/978-3-642-23702-7_11

[36] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *29th International Conference on Software Engineering (ICSE'07)*. 416–426. https://doi.org/10.1109/ICSE.2007.41

[37] Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 123–136. https://doi.org/10.1007/11817963_14

[38] Kenneth L. McMillan. 2010. Lazy Annotation for Program Testing and Verification. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 104–118. https://doi.org/10.1007/978-3-642-14295-6_10

[39] Kenneth L. McMillan. 2018. Interpolation and Model Checking. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer International Publishing, Cham, 421–446. https://doi.org/10.1007/978-3-319-10575-8_14

[40] David Molnar, P Godefroid, and M Levin. 2008. Automated whitebox fuzz testing. In *Network and distributed system security symposium, NDSS*. 416–426.

[41] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended symbolic execution on binary programs. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago,