Scalable Path-Sensitive Program Analysis via Dynamic Programming

Joxan Jaffar
National University of Singapore
joxan@comp.nus.edu.sg

Jorge A. Navas
National University of Singapore
navas@comp.nus.edu.sg

Andrew E. Santosa
National University of Singapore
andrews@comp.nus.edu.sg

Abstract

Path-sensitivity improves program analysis by excluding infeasible paths and avoiding the merging of paths into a single abstraction if they exhibit different behavior. The main challenge however is that path-sensitive analysis is not scalable. In this paper, we present a symbolic execution-based framework which uses *dynamic programming* (DP) in order to reuse analyses arising from symbolic execution already performed. A straightforward implementation of DP, however, will allow little reuse. This is because symbolic execution repeatedly considers similar subtrees with different contexts. The first of our contributions to use a method of *interpolation* in order to generalize the result of symbolic execution so that it can be reused in another context. This has the effect of pruning the symbolic execution space, and is the basis of making our analysis scalable. However, interpolation introduces inaccuracy because some of the paths pruned may in fact be infeasible. We therefore introduce a technique to ensure precision by testing that a notion of *witness paths* that concretely demonstrate a particular analysis, and we require that pruned paths satisfy the witness criterion. In the end, we show that, in a sense defined by traditional abstract interpretation, our framework loses no accuracy beyond the abstraction that is required to close loops via their invariants.

We finally demonstrate practicality of our framework by instantiating it with a driving application: static backward slicing analysis. We show that our approach can produce slices significantly smaller than a path-insensitive version scaling up to tens of thousand of lines of C code.

1. Introduction

Static analysis using abstract interpretation [7] and dataflow analysis [22] is an efficient way of extracting information about all possible executions of a program. It has been successfully used in the area of compiler optimization and program verification. Given the program's *control-flow graph* (*CFG*), it performs an enumeration of the paths in the CFG while propagating abstract information reflecting the effect of the execution of program statements along each path. In the presence of loops or recursive calls, a fixpoint computation or *widening* is needed. Abstract interpretation and dataflow analysis achieve efficiency by eliminating details that might be irrelevant to the objective of the analysis, however, this comes with a loss of accuracy, consisting of:

1. extraction of information from paths that are infeasible, and
2. merging of different abstractions into a single abstraction that is coarser than the sum of its parts.

Such inaccuracies result in lost opportunities for compiler optimizations or false positives in program verification. A systematic way to avoid these inaccuracies is to perform *path-sensitive* analysis, but doing so has not been scalable so far.

This paper presents such an analysis method. It is based on *dynamic programming* (DP), a widely used technique to solve combinatorial optimization problems exhibiting certain characteristics. One major characteristic of DP is the existence of overlapping subproblems. This allows for the *reuse* of a solution of a subproblem to solve another subproblem. Each subproblem is executed in a *context*, which is an abstraction of the computation history so far. Whenever the same subproblem is encountered in a *similar* context, the previous solution can be reused. Now, in classical uses of DP, such as the *shortest-path problem* in graphs, reuse is always possible and the problem is solvable efficiently. In our more general setting, this is not the case[1].

The main technical contribution of this paper can now be described as defining conditions of reuse in a such a way that reuse is frequent in practice, thus bringing path-sensitive analysis practical for realistically sized programs.

In our framework, we first define analysis as the extraction of information, in the form of abstract formulas, from symbolic execution. We associate each path in the symbolic execution tree with a collection of *constraints* from which this information is extracted. It will be seen later that though this setup is completely general, our framework is essentially geared for analysis toward a *target program point*. This is mainly because the framework prescribes a bottom-up processing of the symbolic execution tree.

Our DP formulation will suggest a depth-first traversal where completed subtrees will give rise to a subsolution, expressed as a *summarization*. Certain pairs of subtrees are "similar" in the sense that they are addressing the same symbolic execution space[2], but they may have different *contexts*. In further traversal, the key then is to determine if a subtree "similar" to one already summarized satisfy the conditions that make the summarization reusable.

There are two conditions. The first concerns infeasible paths. More precisely, when a subtree is analysed and then summarized in a certain context, the symbolic execution performed would encounter, because of path-sensitivity, some infeasible paths in general. Therefore, when encountering a similar subtree with a differ-

[1] In fact the analysis problem, once formalized, is NP-hard.

[2] This is later formalized as representing the same program point.

ent context, reuse of the summarization is, in fact, *unsound*. This is because the symbolic execution tree of the latter subtree would, in general, contain more paths than the original tree.

We address this by discovering a *generalization* of the context of an analyzed subtree in such a way that the computed analysis continues to hold. One way is to ensure that the symbolic execution, if re-performed on the subtree but with the generalized context, would have all of the infeasible paths as with the original context. We call this generalization process *interpolation* in concert with its now well-known use in program verification. Once interpolation is performed, further similar subtrees are far more likely. Our specific method for interpolation will involve a simple and efficient process of considering path constraints one at a time.

We now describe a second condition for reuse. This time its purpose is not to preserve soundness, but instead to preserve *accuracy*.

Recall that in reuse, a summarization is applied to a subtree which has *fewer* feasible paths than the subtree originating the summarization. This means that the resulting analysis due to the reusing subtree is, in general, not accurate. In fact, it can be arbitrarily inaccurate. We introduce the concept of *witnesses* in order to ensure that the candidate summarization is indeed accurate. More precisely, a summarization, though formally representing the set of all answers for a subtree, is often represented by one *optimal* answer. For example, if we were seeking the upper bound of a variable, then a summarization would need be just one number. In general, such an optimal solution is realized in the subtree just some, and not all, of its (feasible) paths. The idea here is to choose such a subset of paths as witnesses to the (optimal answer of the) summarization. We thus have our second condition for reuse: the witness paths must be feasible in the candidate subtree.

Typical dynamic programming problems are acyclic, however, programs contains loops. To handle loops, our algorithm performs loop invariant generation, which we outline informally as follows. Given a tree which contains a subtree that is similar to itself, we compute a generalization of the parent node which then subsumes the context of the subtree. This is a straightforward way to enforce termination of symbolic execution in the presence of loops. What is important is that this invariant discovery is executed in a *lightweight* manner where the invariants for each path are constructed by deleting the constraints in the original context that are not invariant through the "cyclic" paths.

The final phase of our framework is to employ, in a standard way, a fixpoint computation on the "closed" symbolic execution tree to compute the abstract answers of the particular analysis.

Organization. The rest of this paper is organized as follows. Sec. 3 provides an informal overview of our approach illustrating several examples. We choose static backward slicing analysis [26] as our driving analysis due to its broad spectrum of applicability. Sec. 4 introduces relevant concepts and definitions required for the rest of the paper. Sec. 5 presents our framework rigorously but still informally. In this section, we show that, in a sense defined by traditional abstract interpretation, our framework loses no accuracy beyond the abstraction that is required to close loops via invariants.

Sec. 6 presents our algorithm in detail. It is here that we detail the central and most complex part which comprises an integrated computation of two fixpoints: one which concerns symbolic execution, and for which loop invariant discovery is used, and second, a more traditional computation for the abstract answers, as in abstract interpretation.

Finally, Sec. 7 shows our experimental evaluation of our framework, instantiated with our driving example of slicing, on a set of C benchmarks. Our experiments demonstrate that our path-sensitive framework produces slices significantly smaller than a path-insensitive slicer scaling up to tens of thousand of lines.

2. Related Work

Our closest related work has been recently presented in [20]. Here, dynamic programming (DP) is used to solve not an analysis problem, but rather, a *combinatorial optimization* problem: the Resource-Constrained Shortest Path (RCSP) problem. Although [20] was an inspiration for this paper, there are key advances here. First and most importantly, [20] is totally defined in a finite setting. Thus for program analysis, this would mean considering only loop-free programs. In contrast, we introduce a loop-invariant discovery method for making symbolic execution finite and integrate this with a fixpoint method of abstract interpretation. Second, [20] considered only the extraction of bounds of variables. Here, we present a general framework for program analysis. Third, [20] provided only for the use of one witness (though they considered storing redundant extra copies). Finally, this paper presents a full implementation and demonstrates its use on large programs.

We next outline previous approaches to path-sensitive analysis. We distinguish between program analysis for verification which attempts to prove absence or existence of errors and program analysis techniques for discovery of certain relationships, like ours. Note that in a sense verification techniques are not comparable to ours due to different objectives, however, their techniques to prune the search space are closely related to ours.

Software Model Checking. Current state-of-the-art software model checking is based on *counterexample abstraction refinement* (*CEGAR*) using *predicate abstraction* [3, 16]. The program is modeled via a coarse abstraction and then, the abstraction is refined until the property is refuted or proved. Abstraction refinement methods may not terminate since the sequence of refinements is not guaranteed to terminate. A crucial improvement for scalability has been lazy abstraction with interpolants [15, 24] in order to eliminate the consideration of irrelevant predicates in the abstraction. Our approach posses a commonality with CEGAR in the fact that both use interpolants to eliminate irrelevant facts to prune the search space.

Program Analysis for Program Verification. SAT-based methods have been used successfully for bounded model checking (e.g., F-Soft [18]). These techniques rely on conflict analysis and clause learning for pruning the search space. SAT-based approaches suffer significantly in presence of loops. The solution is often incomplete by considering paths up to a bounded length. Recently, another SAT-based method has been presented in [14]. The method enumerates feasible paths, and for each one, different techniques (e.g., predicate abstraction, abstract interpretation, symbolic execution, etc) can be used to find the proof or a violation. This flexibility allows, for instance, the use of abstract interpretation to handle unbounded paths. Another incomplete SAT-based system is Saturn [1, 11] which can only detect infeasibility by reasoning about finite domains. Other approaches to path-sensitive analysis for verification based on heuristics are ESP [9] and *Trace Partioning* [4, 23]. ESP achieves scalability by keeping track of some branch correlations under the assumption that different branches that produce different results should be treated differently. Trace partioning use different heuristics to decide whether or not merge the abstract states at the join points in the CFG. This technique can be also used for discovery properties.

Program Analysis for Discovery Properties. Here the detection of infeasible paths is partial (e.g., branch correlation and conflict sets) and the merging operation at the join nodes in the CFG is performed based on heuristics. Profiling techniques [2] have been used to identify those paths more frequently visited during the execution of multiple tests ("hot paths"). Then, join points in the CFG that belong to hot paths are split if the dataflow analysis may incur in a loss of precision. ESP has been extended for dataflow analysis

in [13] but the method is still based on heuristics. An interprocedural path infeasibility analysis is described in [5] to restructure the original CFG by eliminating some conditional branches. The scalability is achieved at the expense of a partial detection of infeasible paths based on simple branch correlations. An application for dataflow analysis of this work is presented in [6] for computing a more precise set of def-use pairs. The technique is demand-driven and computes the pairs and checks their feasibility using the branch correlation analysis from [5]. Recently, another method to detect infeasible paths based on conflict sets is described in [25]. The heuristics-based technique also splits nodes in the CFG if the dataflow analysis may incur in loss of imprecision.

3. Basic Idea

In this section, we describe informally our method through several examples using program slicing. Static slicing [26] defines the *backward slice* of a program wrt a program point p and a variable x (called the *slicing criterion* $(p,\{x\})$) as all statements of the program that might affect the value of x at p. We follow here Weiser's dataflow approach. Assuming that a CFG is defined simplistically as pair (N,E) where $E \subseteq N \times N$ and N is the set of nodes, we define $\mathcal{D}_n$ as the set of variables at node $n \in N$ that may affect the slicing criterion[3]. *Data dependencies* can be formulated using this set by defining two kinds of dataflow information. Given an edge $e \equiv (i,j) \in E$ we denote $def(e)$ and $use(e)$ as the variables altered and used, respectively, at e. Then,

$$\mathcal{D}_i = (\mathcal{D}_j \setminus def(e)) \cup use(e). \tag{1}$$

If j reaches the slicing criteria (i.e., $j = p$) then $\mathcal{D}_j = \{x\}$. *Control dependencies* can also affect the criterion. Informally, a branch $e \equiv (i,j) \in E$ is included if some of the statements under its scope are included in the slice, and

$$\mathcal{D}_i = \mathcal{D}_j \cup use(e) \tag{2}$$

In presence of loops the set $\mathcal{D}_n$ must be inferred via a fixpoint computation which terminates due to the finiteness of the domain. Finally, an edge $e \equiv (i,j) \in \mathcal{E}$ is included in the slice if

$$\mathcal{D}_i \cap def(e) \neq \emptyset \tag{3}$$

Interpolation. Consider Fig. 1(a) and we slice on the criterion $\langle\langle 7\rangle,\{x\}\rangle$, Path-insensitive static slicing would not be able to eliminate any statement. However, the assignment `x=z` at $\langle 6\rangle$ is not executable since the execution path to it is infeasible. By exposing infeasible paths, a static slicer would be able to produce a more precise slice. Here, our algorithm produces the most precise slice, which is the empty program. However, detection of infeasible paths comes at a cost of traversing the execution tree of the program, whose size is potentially exponential to the size of the program.

Fig. 2(a) shows the execution tree traversed by our algorithm. The nodes are labeled with P:C (P is a program point and C is a context identifier to distinguish nodes with the same program point) and edges between two locations labeled by the instruction that executes when control moves from the source to the destination. Feasible transitions are denoted by (black) solid edges, and (red) infeasible transitions by zigzag edges. We denote subsumed nodes by (green) dotted edges and the label "(s)".

Our method first reaches the node 6:1 with the path formula $\Psi_{6:1} \equiv c = 0 \wedge a > 0 \wedge y = 0 \wedge b > 0 \wedge z = y \wedge c > 0$, which is infeasible. We now generate an *interpolant*. Given two formulas Ψ and Φ where $\Psi \Rightarrow \Phi$, an interpolant is a formula $\overline{\Psi}$ whose variables belong to both Ψ and Φ, and both $\Psi \Rightarrow \overline{\Psi}$ and $\overline{\Psi} \Rightarrow \Phi$. Here we want to find a formula $\overline{\Psi}_{6:1}$ such that $\Psi_{6:1} \Rightarrow \overline{\Psi}_{6:1}$ and $\overline{\Psi}_{6:1} \Rightarrow \mathit{false}$.

While traversing the tree, we simultaneously produce *answers* for our analysis. In the context of program slicing, at every node n, we store the set $\mathcal{D}_n$. At 6:1 the set $\mathcal{D}_{6:1} = \perp$ as the path is infeasible.

After generating the interpolant and the answer of 6:1, the algorithm backtracks to 5:1 and visits 7:1. Again, the algorithm needs to generate an interpolant that should be as general as possible. Since the formula path is satisfiable, it generates $\overline{\Psi}_{7:1} \equiv \mathit{true}$, which is the most general possible interpolant. Furthermore, the algorithm updates $\mathcal{D}_{7:1} = \{x\}$, provided directly by the slicing criteria.

We next use the pairs of interpolants and answers at 6:1 and 7:1 to produce a new pair at 5:1.

First, we explain how to produce the interpolant. Given $\overline{\Psi}_{6:1}$ as a postcondition, we compute the first candidate interpolant at 5:1, denoted $\overline{\Psi}^1_{5:1}$ as the formula satisfying $\Psi_{5:1} \Rightarrow \overline{\Psi}^1_{5:1}$ and $\overline{\Psi}^1_{5:1} \Rightarrow (c > 0 \Rightarrow \overline{\Psi}_{6:1})$. Here, $\Psi_{5:1}$ is a conjunction of the constraints along the path that ends in 5:1 (i.e., $c = 0 \wedge a > 0 \wedge y = 0 \wedge b > 0 \wedge z = y$). The constraint $c > 0$ is obtained from the transition relation between $\langle 5\rangle$ and $\langle 6\rangle$, and $\overline{\Psi}_{6:1}$ is *false*. The most general $\overline{\Psi}^1_{5:1}$ is $c \leq 0$, which corresponds to the *weakest liberal precondition* (*wlp*) [10] of the postcondition $\overline{\Psi}_{6:1} = \mathit{false}$ wrt the transition relation $c > 0$[4]. The second candidate interpolant $\overline{\Psi}^2_{5:1}$ is computed from $\overline{\Psi}_{7:1}$ (*true*), such that: $\Psi_{5:1} \Rightarrow \overline{\Psi}^2_{5:1}$ and $\overline{\Psi}^2_{5:1} \Rightarrow (c \leq 0 \Rightarrow \overline{\Psi}_{7:1})$. The most general $\overline{\Psi}^2_{5:1}$ is *true*. Now the final interpolant for 5:1, denoted $\overline{\Psi}_{5:1}$ is the conjunction $\overline{\Psi}^1_{5:1} \wedge \overline{\Psi}^2_{5:1}$ which is $c \leq 0$.

For the answers, the algorithm following Eqs. 1 and 2, and considering $\mathcal{D}_{6:1} = \perp$ and $\mathcal{D}_{7:1} = \{x\}$, computes $\mathcal{D}_{5:1} = \{x\}$ by performing set union for merging those answers. Note that due to the infeasibility of the branch from 5:1 to 6:1, the Eq. 3 is not applicable, and hence, no statement is added into the slice so far. Indeed, no statement will be included in the slice since the variable z was never propagated, and hence, $\mathcal{D}_k = \{x\}$, for all node k.

This dynamic programming process continues recursively in a post-order manner until the entire tree has been explored. A key feature is that the algorithm uses the interpolants generated at each program point, instead of the stronger original context, to weaken the criteria for reusing answers corresponding to the point. For instance, consider the node 3:2 in the tree. The path formula at 3:2 is $\Psi_{3:2} \equiv c = 0 \wedge a \leq 0$. Before exploring the subtree, our method tests if $\Psi_{3:2}$ is *subsumed* by $\overline{\Psi}_{3:1}$, that is, if $\Psi_{3:2} \Rightarrow \overline{\Psi}_{3:1}$, which is the case since $\overline{\Psi}_{3:1}$ is $c \leq 0$. Therefore, without exploring the subtree emanating from 3:2, the algorithm simply reuses $\mathcal{D}_{3:1}$ as $\mathcal{D}_{3:2}$. Note that the subsumption holds due to the use of the interpolant $\overline{\Psi}_{3:1}$. The original context at $\Psi_{3:1}$ ($a > 0 \wedge c = 0 \wedge y = 0$) would not have subsumed $\Psi_{3:2}$. At 3:2 we want an interpolant $\overline{\Psi}_{3:2}$ that satisfies both $\Psi_{3:2} \Rightarrow \overline{\Psi}_{3:2}$ and $\overline{\Psi}_{3:2} \Rightarrow \overline{\Psi}_{3:1}$. Here, the right interpolant is $\overline{\Psi}_{3:1}$ of the subsuming node 3:1.

Witnesses. While subsuming a node may save search space and yet preserve the correctness of the analysis, it does not preserve necessarily the accuracy of the analysis. Consider the program fragment in Fig. 1(b). The slice wrt x at $\langle 5\rangle$ is obtained by our algorithm so far, as the original program. However, a more detailed analysis of the program discovers that the statement `y=z` at program point $\langle 2\rangle$ does not affect the computation of the criterion variable x. The reason is that y can only affect x if statement at $\langle 4\rangle$ is executed, but $\langle 4\rangle$ is unreachable because it is located at an infeasible path whenever the program reaches $\langle 2\rangle$.

[3] For simplicity and w.l.o.g., we assume a single slicing criterion. Therefore, we can omit it from the expression $\mathcal{D}_n$.

[4] In our implementation, we adopt the approaches of *constraint deletion* or *slackening* described in [21] which are efficient, yet usually do not give us the weakest interpolant.

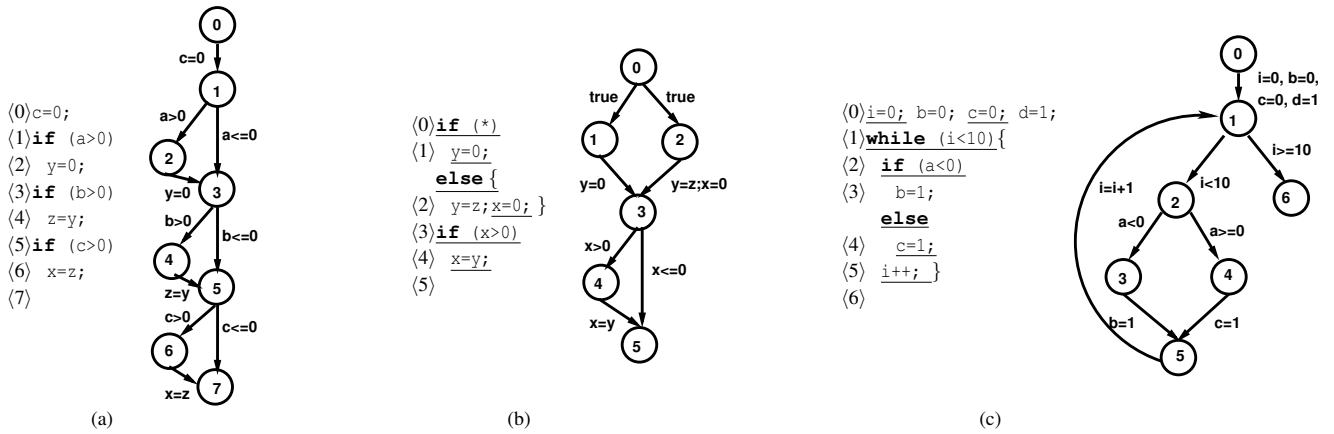

Figure 1. Three Programs, Their Control Flow Graphs, and Their Slices (Underlined Statements) on $\langle\langle 7\rangle,\{x\}\rangle$, $\langle\langle 5\rangle,\{x\}\rangle$, and $\langle\langle 6\rangle,\{c\}\rangle$, respectively

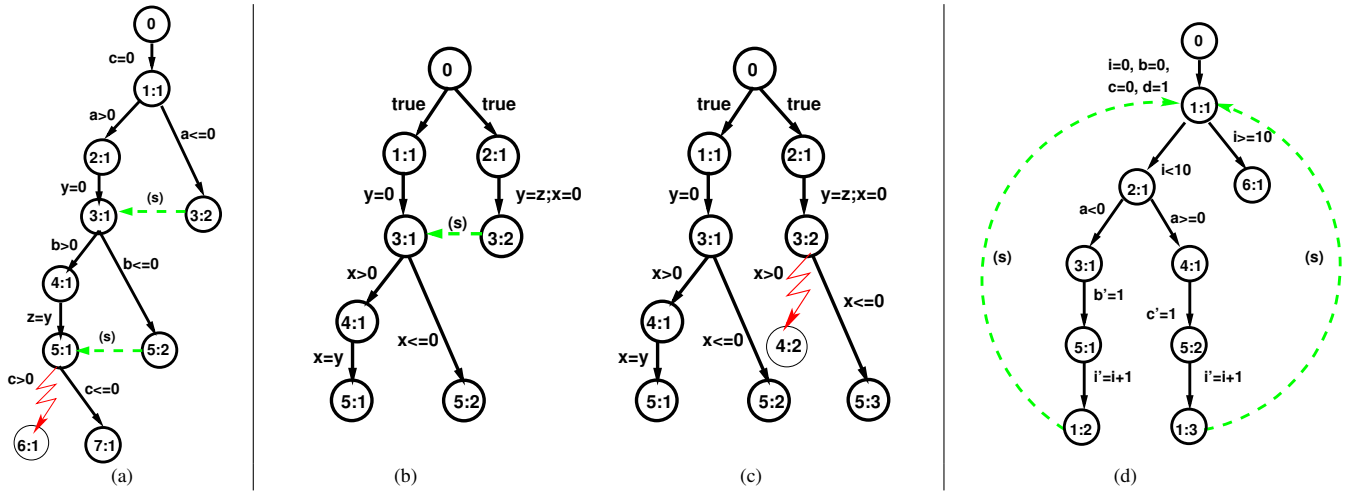

Figure 2. Their Interpolated-based Symbolic Execution Trees

The symbolic execution tree computed by our algorithm, explained so far, is shown in Fig. 2(b). At node 3:1 the algorithm has joined the solutions from its children $\mathcal{D}_{4:1} = \{y\}$ and $\mathcal{D}_{5:2} = \{x\}$ obtaining $\mathcal{D}_{3:1} = \{x, y\}$. The interpolant at 3:1 is *true*, due to the nonexistence of infeasible paths in the subtree rooted at 3:1. This causes subsumption to hold at 3:2, at which its answer is $\mathcal{D}_{3:2} = \mathcal{D}_{3:1} = \{x, y\}$. Due to this answer, we include all the statements along the path from 1:1 to 3:2 in the slice, the end result being all statements included in the slice.

For the sake of discussion, let us consider that the node 3:2 is not subsumed by 3:1. The symbolic execution tree is now shown in Fig. 2(c). The key observation is that the new subtree rooted at 3:2 contains an infeasible path (the zigzag edge from 3:2 to 4:2) since $\Psi_{3:2} \Rightarrow x = 0$, and therefore the more accurate result for $\Psi_{3:2}$ should be $\mathcal{D}_{3:2} = \{x\}$. Due to this, the assignment `y=z` at $\langle 2\rangle$ would not be included in the slice.

The program in Fig. 1(b) has illustrated once again that the existence of infeasible paths is essential in order to obtain the most precise slice. Moreover, it has also discovered that we need to strengthen the condition that decides whether a node can be subsumed or not in order to be precise. Hence, given the set of dependencies $\mathcal{D}_n$ our algorithm will also store, for each variable $x \in \mathcal{D}_n$, the path formula ω_x that "gives rise" to the answer. We call this formula ω_x, the *witness* to the dependency on x.

Informally, a node n with context Ψ_n is subsumed by n' with interpolant $\overline{\Psi}_{n'}$ and reuse dependencies $\mathcal{D}_{n'}$ if $\Psi_n \Rightarrow \overline{\Psi}_{n'}$ (as before), and for all x in $\mathcal{D}_{n'}$ each representative path formula ω_x is possible with the new context Ψ_n. That is, $\omega_x \wedge \Psi_n$ is satisfiable.

Back to the tree in Fig. 2(c), the context at node 3:2 is $\Psi_{3:2} \equiv a \leq 0 \wedge x = 0 \wedge y = z$. The interpolant at 3:1 is $\overline{\Psi}_{3:1} \equiv \mathit{true}$. It is straightforward to see that $\Psi_{3:2} \Rightarrow \overline{\Psi}_{3:1}$. In addition, we test that, for each variable in the dependency set, their witnesses are satisfiable. That is, both $\Psi_{3:2} \wedge \omega_x$ and $\Psi_{3:2} \wedge \omega_y$ are satisfiable, where ω_x is $x \leq 0$ and ω_y is $x > 0 \wedge x = y$, each being constraints along the paths that give rise to the corresponding dependencies. The second formula is unsatisfiable. Therefore, the algorithm must explore node 3:2, which, as shown, results in a more precise slice including all underlined statements in the program in Fig. 1(b).

Loops. We explain now how our dynamic programming method handles loops. This can be seen as a two-phase process.

In the first *loop invariant generation* phase, it abstracts the state at the looping point with a loop invariant that is computed *on-the-fly* in a *lightweight* manner. Consider the looping program in Fig. 1(c) and its symbolic execution tree in Fig. 2(d). When node

1:2 is reached, we attempt to compute a set of abstractions of the state at node 1:1 that are invariant for the cyclic path $\langle 0\rangle$-$\langle 1\rangle$-$\langle 2\rangle$-$\langle 3\rangle$-$\langle 5\rangle$-$\langle 1\rangle$. We note that the original formula at 1:1 is $\Psi_{1:1} \equiv i = 0 \wedge b = 0 \wedge c = 0 \wedge d = 1$. Among the atomic constraints, our algorithm attempts to find a subset of constraints that are *individually invariant* through the cycle $\langle 1\rangle$-$\langle 2\rangle$-$\langle 3\rangle$-$\langle 5\rangle$-$\langle 1\rangle$. For example, $i = 0$ is not individually invariant, as the value of the variable i at 1:2 is 1, while $d = 1$ is individually invariant, as its value is preserved at 1:2. The sought after set is therefore $\{c = 0, d = 1\}$, named $S_{1:1}$. When interpreted as conjunction, any subset of this set is a path-based invariant through the cycle, the strongest being $c = 0 \ \wedge \ d = 1$. We then stop the traversal at 1:2 and compute interpolants given $c = 0 \wedge d = 1$ at 1:2. Here, instead of interpolating on infeasibility, we interpolate on a more general postcondition, which is the path-based invariant. For example, the interpolant $\overline{\Psi}_{5:1}$ is a condition that satisfies $\Psi_{5:1} \Rightarrow \overline{\Psi}_{5:1}$ and $\overline{\Psi}_{5:1} \Rightarrow (i' = i+1 \Rightarrow c = 0 \wedge d = 1)$, where $\Psi_{5:1}$ is $i = 0 \wedge b = 0 \wedge c = 0 \wedge d = 1 \wedge i < 10 \wedge a < 0 \wedge b' = 1$. A suitable $\overline{\Psi}_{5:1}$ would be $c = 0 \wedge d = 1$.

We next traverse the else branch of the if conditional after replacing $\Psi_{1:1}$ with the more general path-based invariant $c = 0 \ \wedge \ d = 1$. The else branch is satisfiable under this abstraction, and we reach 1:3 with the formula $\Psi_{1:3} \equiv c = 0 \wedge \ d = 1 \wedge \ i < 10 \wedge \ a \geq 0 \wedge \ c' = 1 \wedge i' = i+1$ (here we denote different variable versions using primes). As the value of c is 1 at 1:3, among the elements of $S_{1:1}$, only $d = 1$ is invariant through this path. Therefore the loop invariant computed for this path is $d = 1$. This step clarifies the use of individually invariant constraints: the set $S_{1:1}$ includes only constraints that even if any removed, the remaining constraints are still invariant through the already-traversed cycle. In this way, at 1:3 we are certain that $d = 1$ is invariant through both cycles in the loop, and since there are no more cycles, $d = 1$ is the loop invariant. We then backtrack and exit the loop reaching 6:1 with the formula $\Psi_{6:1} \equiv d = 1 \wedge i \geq 10$.

Next, during the *answer fixpoint generation* phase, the algorithm backward-propagates the answers through all execution paths, including the cyclic ones, until the answers converge in a fixpoint. For illustration, assume that the loop exit is taken first since the variable i is not invariant, and $\mathcal{D}_{6:1} = \{c\}$. This dependency is propagated to 1:1 such that $\mathcal{D}_{1:1} = \{c\}$. Then, our methods starts computing a standard fixpoint for the loop. In the first iteration, the method traverses forward the loop body reaching 1:2. At 1:2, our methods returns the answer $\mathcal{D}_{1:2} = \mathcal{D}_{1:1} = \{c\}$, as both 1:2 and 1:1 correspond to the same program point. Similarly, at 1:3 it produces $\mathcal{D}_{1:3} = \{c\}$. Then, dependencies are propagated back following Eqs 1 and 2 and slice computed using Eq 3. Due to control dependencies it adds the variables a and i into $\mathcal{D}_{1:1}$, such that it becomes $\{i, a, c\}$. Since $\{i, a, c\} \not\subseteq \{c\}$ another fixpoint iteration is required. After the second iteration, our method converges into a fixpoint since no more dependencies can be added. The slice is described by all underlined statements in program in Fig. 1(c).

4. Preliminaries

We outline the framework of *Constraint Logic Programming* (*CLP*) [19] which formalizes a program as CLP rules. The purpose is to use the operational semantics of CLP, which constructs derivation trees, to symbolic execution of the program.

The *universe of discourse* is a set of terms, integers, and arrays of integers. A *constraint* is written using a language of functions and relations. An *atom* is of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and the $\tilde{t}$ a sequence of terms. A *rule* is of the form $A :\!\!- \tilde{B} \wedge \phi$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint ϕ constitute the *body* of the rule. We use rules to represent state transitions and program end points, and restrict their syntax to $p(k,\tilde{x}) \ :\!\!- \ p(k',\tilde{x}'), \rho(\tilde{x},\tilde{x}')$. In this syntax, k and k' are program points, $\tilde{x}$ and $\tilde{x}'$ represent state variables before and after the execution of the program, and $\rho(\tilde{x},\tilde{x}')$ represent the transition relation given by the statement between points k and k'. A set of rules is called a program, denoted $\mathbf{P}$.

For example, given a program fragment with two variables `x` and `y`, the assignment $\langle 5\rangle$ `x = y+1` $\langle 6\rangle$ is represented as the rule $p(5,x,y) \ :\!\!- \ p(6,x',y') \wedge y' = y \wedge x' = y+1$. For a conditional $\langle 6\rangle$ `if (x>0)` $\langle 7\rangle$, we represent the transition between $\langle 6\rangle$ and $\langle 7\rangle$ by the rule $p(6,x,y) \ :\!\!- \ p(7,x',y') \wedge y' = y \wedge x' = x \wedge x > 0$.

A *substitution* simultaneously replaces each variable in a term or constraint into some expression, notationally $[\tilde{e}/\tilde{x}]$, where $\tilde{x}$ is a sequence $x_1,\ldots,x_n$ of variables and $\tilde{e}$ a sequence $e_1,\ldots,e_n$ of expressions, such that x_i is replaced by e_i for all $1 \leq i \leq n$.

A *state* is of the form $p(k,\tilde{x}) \wedge \phi(\tilde{x})$, where k is an integer constant, $\tilde{x}$ is a sequence of variables called *primary* variables, which represent the variables that appear in the original program, and ϕ a constraint on them. Given a state $\mathcal{G}$, we denote the constraint part as $cons(\mathcal{G})$. Sometimes we write $(c,\mathcal{G})$ to denote the state $\mathcal{G}$ augmented with additional constraints c. This notation is to capture a symbolic state which instantiated with a context c.

We say that a state $\overline{\mathcal{G}} : p(k,\tilde{x}) \wedge \overline{\Psi}(\tilde{x})$ *subsumes* another state $\mathcal{G} : p(k',\tilde{x}') \wedge \Psi(\tilde{x}')$ if $k = k'$ and if both states are renamed apart, $\Psi(\tilde{x}') \Rightarrow \exists \overline{\Psi}[\tilde{x}'/\tilde{x}]$[5]. We also say that $\overline{\mathcal{G}}$ *generalizes* $\mathcal{G}$.

Let $\mathcal{G}_i$ be $p(k,\tilde{x}_i) \wedge \phi(\tilde{x}_i)$ and $\mathbf{P}$ denote a state and a program respectively. Let rule $R \in \mathbf{P}$ be $p(k_i,\tilde{x}) \ :\!\!- \ p(k_{i+1},\tilde{x}') \wedge \rho(\tilde{x},\tilde{x}')$, written so that none of its variables appear in $\mathcal{G}_i$. A *reduct* of $\mathcal{G}_i$ using R, denoted $reduct(\mathcal{G}_i, R)$ is a state $\mathcal{G}_{i+1}$ of the form $p(k_{i+1},\tilde{x}_{i+1}) \wedge \phi(\tilde{x}_i) \wedge \rho(\tilde{x}_i,\tilde{x}_{i+1})$. Reduction is a means to symbolically compute strongest postcondition, where $\mathcal{G}_i$ represents a symbolic pre-state, R the transition, and $\mathcal{G}_{i+1}$ the symbolic post-state.

A *derivation sequence* is a possibly infinite sequence of states $\mathcal{G}_0, \mathcal{G}_1, \cdots$ where $\mathcal{G}_i$, for $i > 0$, is a reduct of $\mathcal{G}_{i-1}$. Given τ any sequence $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n$, we say that τ (or its end state $\mathcal{G}_n$) is *feasible* if $cons(\mathcal{G}_n)$ is satisfiable, and *infeasible* otherwise. A state $\mathcal{G}_i : p(k,\tilde{x}) \ \wedge \ \Psi(\tilde{x})$ is *looping* if either it is derived from another state with the same k (its *looping parent*) through one or more reduction steps, or there is a state derived from it with the same k. A sequence that ends in a state $\mathcal{G}_n$ is *terminal* when it is feasible and cannot be reduced further.

A *derivation tree* for a state, $\mathcal{G}_0$, has as branches all derivation sequences emanating from $\mathcal{G}_0$. It is *closed* if all its leaf states are either terminal, infeasible, or subsumed by some other state. The last state $\mathcal{G}_n$ in a derivation sequence has the syntax

$$p(k_n,\tilde{x}_n), \phi(\tilde{x}_0) \wedge \textstyle\bigwedge_{i=1}^{n} \rho_i(\tilde{x}_{i-1},\tilde{x}_i),$$

where $\phi(\tilde{x}_0)$ is $cons(\mathcal{G}_0)$ from which the derivation sequence starts, and each $\rho_i(\tilde{x}_{i-1},\tilde{x}_i)$ is added in the i-th derivation step.

Here we call $cons(\mathcal{G}_n)$ a *solution*, and a *suffix constraint* is a constraint $\psi : \bigwedge_{i=m}^{n} \rho_i(\tilde{x}_{i-1},\tilde{x}_i)$. For a suffix constraint ψ and some renamed-apart state $\mathcal{G} : p(k,\tilde{x}) \wedge \phi$ in a derivation tree, the constraint $\phi \wedge \psi[\tilde{x}/\tilde{x}_{m-1}]$ is a solution belonging to some successful state. Here we say that ψ extends $\mathcal{G}$ into a solution.

In what follows, we shall use the terms symbolic execution path and tree as synonymous with derivation sequence and tree.

5. The Framework

In this section, we present informally the core concepts that underpin the main algorithm schema defined later. While these concepts were exemplified above in Sec. 3, here we provide a more general setting. A precise realization of the framework in the next Sec. 6.

[5] Here we use a more restricted version of subsumption than $[\![\overline{\mathcal{G}}]\!] \supseteq [\![\mathcal{G}]\!]$, a set inclusion wrt the declarative semantics of the CLP program. Here $[\![\mathcal{G}]\!]$ is the set of the groundings of the variables in $\mathcal{G}$ such that the ground atoms belong to the declarative semantics.

The key subsection here is 5.2 which explains how a notion of *reuse* is realized. This is the key to scalability.

5.1 Abstract Domain for Answers

We begin by defining an abstract domain in the traditional way, by having a (possibly infinite) set of abstract formulas that we call *answers*. An abstract domain is a lattice $\langle L, \sqsubseteq \rangle$ (ordered by $\sqsubseteq$) that represent a set of abstract states (or answers). Let Σ denote the universe of concrete program states. The abstraction function $\alpha : 2^{\Sigma} \to L$ and concretization function $\gamma : L \to 2^{\Sigma}$ relates elements of the abstract lattice to concrete set of states forming a *Galois Connection* [7]. An answer is applied to a *suffix path*, which is a path from a state reachable from the initial state, to a terminal state. An answer associated to this state intuitively represents an abstraction of the constraints of the path, and is written in terms of the primary variables of the state. We shall not require the abstract transfer operation here, but will define it below.

Note that this definition of answer tacitly states that we are interested in properties that hold at the *end* of the program. Indeed, our bottom-up formulation is geared toward bottom-up analysis, where information about a state is obtained from it successor, and not predecessor, states. In general, our framework can be used for analysis for *any* target program point. (Technically, we can inject a statement **if** `(*)` **goto** `END` before the target point so that we can continue to consider the final point `END` as the the target.) However, for the sake of simplicity, we shall hereafter assume that the target point is the final point.

An analysis of a program can now be defined as the set of all answers that apply to all of its terminating suffix paths. As usual, we usually seek not the entire set, but a representative subset, as we exemplify later.

5.2 Dynamic Programming

A *summarization* σ is simply a formula on program variables that represents the answers to a path Π in such a way that the answers to an extended path c, Π can easily be computed. Where σ is a summarization for a path Π, we write $c \oplus \sigma$ to denote the summarization for the extended path c, Π. This in effect represents our abstract transfer operation. A summarization is extended to apply to subtrees in the obvious way, i.e., it is a formula which produces the answers to a subtree given any context. First, define that for two paths emanating from the same state having summarizations σ_1 and σ_2 respectively, that $\sigma_1 \otimes \sigma_2$ denotes a summarization that represents a (not necessarily strict) superset of the answers in σ_1 and σ_2. We shall call this the *join-summarization* operation, and when it is precisely the union of the answers in σ_1 and σ_2, we shall say that it is *lossless*.

Now if $\mathcal{G}$ is a state with up to (say) two descendants $c_1, \mathcal{G}_1$ and $c_2, \mathcal{G}_2$, then we define our dynamic programming framework in levels $i > 0$, as follows:

$$summ^{(i)}(\mathcal{G}) = \begin{cases} \sigma_{\mathcal{G}} & \text{if } \mathcal{G} \text{ is terminal} \\ \bot & \text{if } \mathcal{G} \text{ is infeasible} \\ summ^{(i-1)}(\mathcal{G}) & \text{if } \mathcal{G} \text{ is looping} \\ (c_1 \oplus summ^{(i)}(\mathcal{G}_1)) \otimes & (c_2 \oplus summ^{(i)}(\mathcal{G}_2)) \quad \text{otherwise} \end{cases}$$

where $\sigma_{\mathcal{G}}$ denotes the base case summarization for $\mathcal{G}$, $\bot$ denotes the *undefined* summarization which provides no information, and $summ^{(0)}(\mathcal{G})$ always returns $\bot$.

This definition implies that more than just a simple recursive descent algorithm will be required in order to compute summarizations. Instead, such an algorithm will require some kind of fixpoint computation mechanism. This fixpoint computation is standard and so we will relegate further discussion to the next section.

In general, the specific definition of a summarizations is application dependent and hence manual.

Example 1. Consider first the variable dependency example from Sec. 3. Given a path Π, an answer is simply a subset S of the initial variables of the path upon which may affect the selected final variables. A summarization σ can therefore be defined to be just S. Formally, this summarization represents the set of all supersets of S. Now we can compute a summarization for $c \oplus \sigma$ as the subset of variables in c upon which the variables in σ depend. Here we use directly the sets *def* and *use* from Eqs 1 and 2, Sec. 3. Given two summarizations for two paths σ_1 and σ_2 emanating from the same state, $\sigma_1 \otimes \sigma_2$ can be defined as the summarization obtained by the set union of the variables in σ_1 and σ_2.

Example 2. In this second example, consider upper bounds analysis on loop-free programs which is set up as follows. Let there be a distinguished variable T which appears in the program only in the form $T = T + k$ for some positive constant k. Its initial value is 0. An answer for a path is a formula $T_f - T \leq k$ where where T_f is another distinguished variable which is used to capture the value of T at the end of the program, and k is constant. Therefore an answer captures an upper bound for the increment of T along one path, and an answer for all paths will contain the upper bound of T for the program. We can now define a summarization of a path Π to be a single number, k, representing the (biggest) difference in value the of the *current* value of T to the final. Note that the current value of T is, formally, the initial value of T for the path Π. Now we can compute a summarization for $c \oplus \sigma$ as simply k in case c does not increment T, or $k + k'$, in case c increments T by k'. Given two summarizations for two paths σ_1 and σ_2 emanating from the same state, $\sigma_1 \otimes \sigma_2$ can be defined as the *maximum* of the two summarization σ_1 and σ_2.

Example 3. Next consider points-to analysis, where a summarization is a pair from $V \times 2^V$ where V are the program variables. Thus a summarization σ indicates, for each variable v, which set of (other) variables v may point to. As above, we can compute a summarization $c \oplus \sigma$ for an extension c of Π as the merging of some sets of σ in the obvious way, e.g. if c indicates that v_1 now may point to v_2, we then adjoin the current points-to set of v_1 with that of v_2. The joining of two summarizations σ_1 and σ_2 is equally straightforward: each variable v now points to the union of its points-to set in σ_1 and σ_2.

Example 4. Finally consider interval analysis, and assume for simplicity that the domain of discourse is a finite set of positive integers. A summarization can simply be a pair of integers. Extending a summarization σ for a path Π with a constraint c is a straightforward matter of computing, for each new variable assigned in c, its interval based on the intervals of the other variables in σ. The joining of two summarizations σ_1 and σ_2 is simply to assign to each variable an interval which best covers its two intervals indicated by σ_1 and σ_2.

Note that in these four examples, the first three have join-summarizations $\otimes$ that are lossless.

5.3 Reuse of Summarizations

The DP formulation above holds promise for efficient execution whenever reuse is possible. Here we discuss two aspects which *prevent* reuse. The first is mandatory, for its use affects *soundness*. The second affects *accuracy*.

Interpolant

Suppose σ is a summarization for a symbolic execution tree Π for a state $\mathcal{G}$, and this summarization was computed using a context c, that is, σ applies to the state $(c, \mathcal{G})$. Suppose we wish to reuse σ

for a variant state $(c', \mathcal{G})$, whose tree is Π'. Now σ was obtained by considering only feasible paths in Π. Therefore there could be paths in Π' that are nonexistent in Π. This in turn means that any use of σ is no longer sound.

However, if it can be shown that every feasible path in Π' is also feasible in Π, then reuse is clearly sound. This is the reason why, when we compute a summarization, we also compute an *interpolant* c" which is a context equal to or more general than c, such that the tree of $(c", \mathcal{G})$ contains only feasible paths in Π. The idea, of course, is that c" is as general as possible, to maximize the applicability of reuse.

We now have our first of two conditions for the reuse of a summarization: the candidate state must entail the interpolant.

Witnesses

We have chosen a dynamic programming formulation to obtain solutions from subsolutions. Thus in the quest for an "optimal" solution[6], we require optimal subsolutions of subproblems. For the resource bound example, an optimal solution is one which indicates the lowest bound. In the variable dependency example, an optimal solution mentions the smallest set of variables. Similarly for the points-to example. Finally, in the interval analysis example, an optimal solution mentions the narrowest interval.

We thus introduce, for each tree Π, a notion of *witnesses*. These shall essentially be a subset of the terminating or looping paths in Π which by themselves *exhibit* the optimal solution that ultimately is represented in the form of a summarization for Π. Clearly we can define that all the paths in Π are witnesses. But the point clearly, is that we often require just a small number and in some cases, e.g. our resource bounds example, just one.

We can now introduce our second condition for the reuse of summarizations: that the witnesses associated with the summarization are feasible in the candidate state.

5.4 A Characterization of the Framework

Here we provide a particular view of traditional analysis. While it is not technically essential to the presentation of the framework, its purpose is to explain the framework in a traditional setting.

Abstract Interpretation [7] uses some abstract domain, a lattice $\langle L, \sqsubseteq \rangle$ equipped with abstraction α and concretization γ functions. Representing a program by a monotone abstract transfer function, program analysis is then simply a fixpoint computation over abstract formulas, and termination is guaranteed by ensuring that progressing through the lattice structure is bounded. Failing this, *widening* and *narrowing* operators can be used to guarantee convergence. This computation is essentially driven by a traversal of the *control flow graph* (CFG) of the program. One particular aspect of this traversal is that when a node is visited more than once, the abstract formulas computed so far are *merged*. A CFG is defined, as usual, as a tuple $\langle N, E, \rho \rangle$ where N is a set of nodes, $E \subseteq N \times N$ is a set of edges, and each edge $e \in E$ is labeled by a *transition relation* $\rho(V, V')$ where V is the set of variables before and after the transition. The point of interest here is to consider *different implementations* of a CFG. The typical one has its nodes corresponding exactly to the program points in the program. However, in order to provide more accuracy, a common practice is to *split* nodes in a CFG in order to avoid abstraction by the abovementioned merge operation. Consider for example the CFG in Figure 1(a) and note that node 3 has two incoming edges. Figure 2(a), when read as a CFG (instead of an execution tree), depicts such a split. It is easy to see that performing traditional analysis on the latter CFG will, in general, be more accurate. Note that whenever a node is split into two, each of these continues to *represent* the same program point.

We now consider a classification of CFG's. Before proceeding, we shall restrict to CFG's in which there is a *feasible path* from the source node to every other node[7]. We shall also allow CFG's to be infinite.

DEFINITION 1. *(Path Sensitivity) The analysis is*

- fully path-sensitive *if the (in general, infinite) CFG is a tree.*
- path-sensitive modulo loops *if the only loops in the CFG are those from a node to a parent which represents the same program point.*
- partially path-sensitive *if in the CFG there are two nodes which represent the same program point.*
- path insensitive *if every node in the CFG represents a distinct program point.*

We can now characterize our framework, whenever its join is lossless, as producing analyses that are *path-sensitive modulo loops*.

6. Algorithm

We start with a few definitions. We denote by $ext(W, \mathcal{G})$ the set of suffix constraints of $\mathcal{G}$ witnessed by the elements of W. An *answer tuple* is a tuple $\langle \mathcal{G}; W; \sigma; \overline{\mathcal{C}}_{\mathcal{T}} \rangle$, where $\mathcal{G}$ is a goal, W a set of witnesses, and σ a summarization, and $\overline{\mathcal{C}}_{\mathcal{T}}$ a sequence of generalized ancestor states of $\mathcal{G}$, which is used by the loop invariant discovery mechanism. We define a partial order $\sqsubseteq$ on summarizations (with the strict version $\sqsubset$) and define *val* to be a mapping from suffix constraints to summarizations.

Our algorithm in Fig. 3 performs depth-first post-order traversal of the derivation tree of a state, returning an answer tuple after traversing a closed subtree. It maintains a global *memo table* $\mathcal{M}_{\mathcal{T}}$ that is a set of computed triples containing the first three components of an answer tuple. The table is initially empty. The procedure takes as input a state, say $\mathcal{G}$, and a sequence of pairs called $\mathcal{C}_{\mathcal{T}}$, also initially empty. The purpose of $\mathcal{C}_{\mathcal{T}}$ is to record all potential looping parents for a given program point. At each recursive call, the algorithm terminates when encountering a terminal, infeasible or subsumed node. A node can be subsumed by either a state of a triple in $\mathcal{M}_{\mathcal{T}}$ or an ancestor state in $\mathcal{C}_{\mathcal{T}}$, called the "looping" case. If none of the above cases for termination is satisfied, the algorithm traverses the derivation tree further by generating the reducts of the current state corresponding to a strongest postcondition step and performs recursive calls on those reducts. The result of each case is an answer tuple, whose first component is a generalization $\overline{\mathcal{G}}$ of $\mathcal{G}$.

The first INFEASIBLE case (Lines 2-3) is triggered when the state $\mathcal{G}$ is infeasible. As mentioned, our algorithm always attempts to generalize $\mathcal{G}$ by weakening its constraints while preserving infeasibilities. Here, $\mathcal{G}$'s constraint is equivalent to $false$, and since we want to preserve infeasibility, we cannot weaken this constraint further. Hence, here we return an answer tuple with a state with *false* as constraint, an empty set of witnesses, and undefined set of summarizations. The last component is only meaningful when there is a looping point in the derivation tree of $\mathcal{G}$. Since $\mathcal{G}$ is non looping, we return the no-generalization symbol ε.

In the second, TERMINAL (Lines 4-5), the execution reaches the end of the path. Since the path is satisfiable, the input state $\mathcal{G}$ can be fully generalized. Here we return a generalization of $\mathcal{G}$ with its constraint *true*. The second component of the answer tuple is again an empty set of witnesses. The third component is $\sigma_{\mathcal{G}}$, which is the base-case summarization defined by the analysis problem at hand (e.g., the variable in slicing criterion). This is not a looping

[6] Recall that while an analysis returns sets of answers, often we just require one of them.

[7] Thus we exclude only "dead" nodes.

SymbolicExec($\mathcal{G}$,$\mathcal{C}_\mathcal{T}$)
1: **switch**($\mathcal{G} : p(k,\tilde{x}) \wedge \Psi(\tilde{x})$))
INFEASIBLE
2: **case** Ψ is unsatisfiable:
3: **return** $\langle p(k,\tilde{x}) \wedge \mathit{false}; \emptyset; \bot; \varepsilon\rangle$
TERMINAL
4: **case** $k = \mathsf{pc}_{\mathit{end}}$:
5: **return** $\langle p(k,\tilde{x}) \wedge \mathit{true}; \emptyset; \sigma_{\mathcal{G}}; \varepsilon\rangle$
SUBSUMED
6: **case** there exists $\langle p(k,\tilde{x}) \wedge \overline{\Psi}(\tilde{x}); W_\mathcal{T}; \sigma_\mathcal{T}\rangle \in \mathcal{M}_\mathcal{T}$ and
7: $\Psi(\tilde{x}) \Rightarrow \overline{\Psi}(\tilde{x})$ and $\forall \varphi \in W_\mathcal{T} : \Psi \wedge \varphi$ is satisfiable:
8: **return** $\langle p(k,\tilde{x}) \wedge \overline{\Psi}(\tilde{x}); W_\mathcal{T}; \sigma_\mathcal{T}; \varepsilon\rangle$
LOOPING
9: **case** $\mathcal{G}$ is looping
10: **if** there exists $\langle \mathcal{G}_P : p(k,\tilde{x}_P) \wedge \Psi_P(\tilde{x}_P); \sigma_P\rangle$ in $\mathcal{C}_\mathcal{T}$:
11: $\langle \Phi(\tilde{x}_P); \overline{\mathcal{C}}_\mathcal{T}\rangle \leftarrow$ INVARIANT($\mathcal{C}_\mathcal{T}$)
12: **return** $\langle p(k,\tilde{x}) \wedge \Phi(\tilde{x}); \emptyset; \sigma_P; \overline{\mathcal{C}}_\mathcal{T}\rangle$
13: **else**
14: $\mathcal{G}_c, \sigma_c \leftarrow \mathcal{G}, \bot$
15: $\langle\langle p(k,\tilde{x}) \wedge \overline{\Psi}(\tilde{x}); \_; \sigma; \overline{\mathcal{C}}_\mathcal{T}\rangle \leftarrow$ Unfold($\mathcal{G}_c, \langle \mathcal{G}_c; \sigma_c\rangle :: \mathcal{C}_\mathcal{T}$)
16: **if** ($\overline{\mathcal{C}}_\mathcal{T} = \langle p(k,\tilde{x}) \wedge \Phi(\tilde{x}); \_\rangle :: \overline{\mathcal{C}}'_\mathcal{T}$) **then**
17: $\overline{\mathcal{C}}_\mathcal{T} \leftarrow \overline{\mathcal{C}}'_\mathcal{T}$
18: **if** ($\Phi(\tilde{x}) \not\Rightarrow \overline{\Psi}(\tilde{x})$) **then**
19: $\mathcal{M}_\mathcal{T} \leftarrow \mathcal{M}_\mathcal{T} \setminus \{$tuples of recursive calls$\}$
20: $\mathcal{G}_c \leftarrow p(k,\tilde{x}) \wedge \Phi(\tilde{x})$
21: **goto** 15
22: **if** ($\sigma_c \sqsubset \sigma_c \otimes \sigma$) **then**
23: $\mathcal{M}_\mathcal{T} \leftarrow \mathcal{M}_\mathcal{T} \setminus \{$tuples of recursive calls$\}$
24: $\sigma_c \leftarrow \sigma_c \otimes \sigma$
25: **goto** 15
26: $\mathcal{M}_\mathcal{T} \leftarrow \mathcal{M}_\mathcal{T} \cup \{\langle p(k,\tilde{x}) \wedge \overline{\Psi}(\tilde{x}); \emptyset; \sigma_c\rangle\}$
27: **return** $\langle p(k,\tilde{x}) \wedge \overline{\Psi}(\tilde{x}); \emptyset; \sigma_c; \overline{\mathcal{C}}_\mathcal{T}\rangle$
RECURSIVE
28: **default**:
29: $\langle \overline{\mathcal{G}}; W; \sigma; \overline{\mathcal{C}}_\mathcal{T}\rangle \leftarrow$ Unfold($\mathcal{G}, \mathcal{C}_\mathcal{T}$)
30: $\mathcal{M}_\mathcal{T} \leftarrow \mathcal{M}_\mathcal{T} \cup \{\langle \overline{\mathcal{G}}; W; \sigma\rangle\}$
31: **return** $\langle \overline{\mathcal{G}}; W; \sigma; \overline{\mathcal{C}}_\mathcal{T}\rangle$

Figure 3. SymbolicExec

sequence, and similar to the infeasible case, we also return ε as the last component.

The third termination case, SUBSUMED (Lines 6-8), searches for an entry in $\mathcal{M}_\mathcal{T}$ such that the current state, $\Psi(\tilde{x})$ entails the interpolant associated to the entry, $\overline{\Psi}(\tilde{x})$. If the entailment holds, the state $\mathcal{G}$ has at most the same subsolutions as the stored $p(k,\tilde{x}) \wedge \overline{\Psi}(\tilde{x})$, and since we could not have found any new subsolution, we can stop the traversal at this point. However, the memoed summarization $\sigma_\mathcal{T}$ may be inaccurate if it was given by a suffix constraint that extends $\mathcal{G}$ to an infeasible state. For this, at Line 7, we also test that all witnesses in $W_\mathcal{T}$ are feasible in the current context. If both conditions are satisfied, we return the contents of the memoed entry itself with additional last component ε.

The LOOPING case (Lines 9-27) handles loops to make the symbolic execution finite. The test at Line 10 determines if $\mathcal{G}$ is a state at the end of a looping path, that is, if there is a looping ancestor $\mathcal{G}_P$ with summarization σ_P in $\mathcal{C}_\mathcal{T}$, in which case the algorithm executes Line 11. Otherwise, it executes Line 14.

At Lines 11-12, our algorithm builds *on-the-fly* a loop invariant by combining invariants that are correct for each looping path in the loop's body. We call such invariant a *path-based* invariant. Assume the current state $\mathcal{G} : p(k,\tilde{x}) \wedge \Psi(\tilde{x})$ is derived from $\mathcal{G}_P : p(k,\tilde{x}_P) \wedge \Psi_P(\tilde{x}_P)$ where $\langle \mathcal{G}_P; \_\rangle$ is in the sequence $\mathcal{C}_\mathcal{T}$, with the same

Unfold($\mathcal{G}$, $\mathcal{C}_\mathcal{T}$)
32: $\overline{\Psi}(\tilde{x}), \mathcal{G}_c : p(k,\tilde{x}) \wedge \Psi(\tilde{x}), W, \sigma \leftarrow \mathit{true}, \mathcal{G}, \emptyset, \bot$
33: **foreach** $\langle \mathcal{G}' : p(k',\tilde{x}') \wedge \Psi(\tilde{x}) \wedge \rho(\tilde{x},\tilde{x}'); R\rangle$
in $\{\langle \mathcal{G}'; R\rangle | \mathcal{G}' = \mathit{reduct}_{\mathbf{P}}(\mathcal{G}_c, R)\}$
34: $\langle p(k',\tilde{x}') \wedge \overline{\Psi}'(\tilde{x}'); W'; \sigma'; \overline{\mathcal{C}}_\mathcal{T}\rangle \leftarrow$ SymbolicExec($\mathcal{G}', \mathcal{C}_\mathcal{T}$)
35: $\overline{\Psi}(\tilde{x}) \leftarrow \overline{\Psi}(\tilde{x}) \wedge$ INTERP($\Psi(\tilde{x}), (\rho(\tilde{x},\tilde{x}') \Rightarrow \overline{\Psi}'(\tilde{x}'))$)
36: $W'', \sigma'' \leftarrow \rho \wedge W', (\mathit{val}(\rho) \oplus \sigma')$
37: $\sigma \leftarrow \sigma \otimes \sigma''$
38: $W \leftarrow$ SELECT($W \cup W'', \sigma$)
39: **if** ($\overline{\mathcal{C}}_\mathcal{T} \neq \varepsilon$) **then** $\mathcal{G}_c \leftarrow$ APPLYINVARIANT($\mathcal{G}_c, \overline{\mathcal{C}}_\mathcal{T}$)
40: **return** $\langle p(k,\tilde{x}) \wedge \overline{\Psi}(\tilde{x}); W; \sigma; \overline{\mathcal{C}}_\mathcal{T}\rangle$

Figure 4. Unfold

program point representing a cyclic control flow path. Note that formula $\Psi(\tilde{x})$ is necessarily of the form $\Psi_P(\tilde{x}_P) \wedge \Psi'(\tilde{x}_P,\tilde{x})$[8] where $\Psi'(\tilde{x}_P,\tilde{x})$ is the conjunction of all constraints along the looping path. To generate a path-based invariant, we generalize $\Psi_P(\tilde{x}_P)$ into $\Phi(\tilde{x}_P)$ such that a new sequence from $\overline{\mathcal{G}}_P : p(k,\tilde{x}_P) \wedge \Phi(\tilde{x}_P)$ by the same derivations results in $\overline{\mathcal{G}} : p(k,\tilde{x}) \wedge \Phi(\tilde{x}_P) \wedge \Psi'(\tilde{x}_P,\tilde{x})$, with $\overline{\mathcal{G}}_P$ subsuming $\overline{\mathcal{G}}$.

The procedure INVARIANT invoked in Line 11 derives, using a theorem prover, constraints that are *individually invariant* in the looping path. The constraints are obtained from the set of atomic constraints in $\exists \mathit{var}(\Psi_P) - \tilde{x}_P : \Psi_P$ (projection of Ψ_P onto the variables $\tilde{x}_P$)[9]. Name this conjunction (set) of constraints $S_{\mathcal{G}_P}$. We remove from it any constraint φ such that $\varphi \wedge \Psi'(\tilde{x}') \not\Rightarrow \varphi[\tilde{x}/\tilde{x}_P]$, that is, when the derivation of state $p(k,\tilde{x}_P) \wedge \varphi$ through the same cyclic path results in a state that is not subsumed by it. The remaining constraints are individually invariant through this path. The conjunction of the elements in this set is denoted by Φ. At Line 12, the algorithm returns an answer tuple containing a state with the computed invariant for the looping path, empty witnesses, σ_P as the summarization, the current recorded summarization for $\mathcal{G}_P$ (initially the empty set), which we will elaborate further, and weakening $\overline{\mathcal{C}}_\mathcal{T}$ of $\mathcal{C}_\mathcal{T}$. Generally, the looping derivation path from $\mathcal{G}_P$ to $\mathcal{G}$ also "passes through" nested looping points such that to maintain the correctness of the algorithm, the weakening of $\mathcal{G}_P$ into $\overline{\mathcal{G}}_P$ also necessitates constraint deletions in such looping states. These weakenings are reflected in $\overline{\mathcal{C}}_\mathcal{T}$.

Lines 14-27 handle the case when the current state is at the start of a loop. $\mathcal{G}_c$ is the generalization of $\mathcal{G}$ and it is a combination of all path-based invariants generated (in Line 11) so far. σ_c is the current summarization for the input state $\mathcal{G}$. Both are initialized to $\mathcal{G}$ and $\bot$, respectively in Line 14.

At Line 15, the algorithm calls the Unfold procedure (Fig. 4), which produces reducts and recursively calls SymbolicExec. We detail Unfold later, but for now, it is sufficient to know that it returns a state $p(k,\tilde{x}) \wedge \overline{\Psi}(\tilde{x})$ which is a generalization of $\mathcal{G}_c$, summarization σ, and a generalization $\mathcal{G}_P$ via path-based invariant generation of $\mathcal{G}_c$ or some ancestor state.

Lines 16-25 implement two different fixpoint computations which are dependent on the return values of Unfold:

1. The *loop invariant generation* phase is implemented by Lines 16-21. At Line 18 we test if $\mathcal{G}_P$ is a generalization of the current state, and if so, we also test if the generalization is stronger

[8] The existence of nested looping points between $\mathcal{G}_P$ and $\mathcal{G}$ may complicate the presentation, but here we prefer a simple formulation for readability.

[9] Note that this is still a correct symbolic description of the program state, as for all groundings θ_P of $\tilde{x}_P$, $(\exists \mathit{var}(\Psi_P) - \tilde{x}_P : \Psi_P)\theta_P$ is valid if and only if $\Psi_P\theta_P$ is satisfiable.

than the generalization $\overline{\Psi}(\tilde{x})$. If this is the case, then $\mathcal{G}_P$ reflects a strong enough loop invariant that ensures the infeasibilities in the derivation subtree. Otherwise, we perform a "restart" where, at Line 19, we first remove all memoed triples generated in the recursive calls (via Unfold). We then update $\mathcal{G}_c$ with the current loop invariant (Line 20). This update results in removing some infeasibilities maintained by $\overline{\Psi}(\tilde{x})$, and in the next iteration we will have a more general $\overline{\Psi}(\tilde{x})$. We then jump to Line 15 to continue a new iteration via Unfold.

2. Once the loop invariant generation ends, $\mathcal{G}_c$ is ensured to be invariant. However, the summarization σ (obtained by merging all summarizations from the recursive calls in Unfold) does not necessarily converge yet into a fixpoint. The *answer fixpoint generation* phase ensures that. At Line 22 we detect if there is a new summarization in σ that is not already in σ_c. If so, we have not reached a fixpoint, and we therefore perform another "restart," where at Line 23 we first clear the triples generated in the recursive calls of Unfold and merge the new summarizations to σ_c (Line 24).

   If a fixpoint has been reached, we return an answer tuple with empty witnesses and σ_c after memoing it in the memo table (Line 26). The reason for empty witnesses is that we have applied generalization to $\mathcal{G}$ (which is applied to all states derived from $\mathcal{G}$ at Line 39 in of Unfold) such that the suffix constraints $ext(W,\mathcal{G})$ may actually not extending the original, stronger state $\mathcal{G}$ into solutions.

Lines 29-31 handle the symbolic execution step. At Line 29 the algorithm calls the Unfold procedure, which returns generalizations, loop invariants, and a summarization. Here we memo the first three components of the returned answer tuple (Line 30) and return the tuple to the caller (Line 31).

We next discuss the Unfold procedure (Fig. 4), whose purpose is to execute recursive calls to SymbolicExec and combines the results. At any derivation tree node with state $\mathcal{G}$, the procedure performs reductions resulting in state $\mathcal{G}'$, and calling SymbolicExec in Line 34.

The recursive call to SymbolicExec (Line 34) returns an answer tuple containing a generalization of the reduct $\mathcal{G}' : p(k',\tilde{x}') \wedge \overline{\Psi}'(\tilde{x}')$. A key operation of the algorithm is the propagation of the answer tuples to the caller. Here we compute a formula $\overline{\Psi}(\tilde{x})$ that generalizes $\Psi(\tilde{x})$ and still preserves the path infeasibility in the derivation tree of the particular reduct $\mathcal{G}'$. That is, $\overline{\Psi}(\tilde{x})$ is an *interpolant* [8] satisfying the following conditions:

1. $\Psi(\tilde{x}) \Rightarrow \overline{\Psi}(\tilde{x})$ and
2. $\overline{\Psi}(\tilde{x}) \Rightarrow (\rho(\tilde{x},\tilde{x}') \Rightarrow \overline{\Psi}'(\tilde{x}'))$.

For example, when $\mathcal{G}'$ is itself infeasible instead of its further descendants, the call to SymbolicExec executes Line 3 where an answer tuple with *false* constraint is returned. Therefore, $\overline{\Psi}'$ obtained at Line 34 is *false*, and $\overline{\Psi}$ has to satisfy: $\overline{\Psi}(\tilde{x}) \Rightarrow (\rho(\tilde{x},\tilde{x}') \Rightarrow \mathit{false})$, or, $\overline{\Psi}(\tilde{x}) \Rightarrow \neg\rho(\tilde{x},\tilde{x}')$ to preserve the infeasibility of $\mathcal{G}'$. The function INTERP$(\Psi(\tilde{x}),\ (\rho(\tilde{x},\tilde{x}') \Rightarrow \overline{\Psi}'(\tilde{x}')))$ in Line 35 returns an interpolant. The final interpolant is the conjunction, also of all interpolants resulting from the calls.

Ideally, instead of an interpolant, we would like to compute the *weakest liberal precondition* (*wlp*) [10] which is the weakest interpolant possible, however, maintaining such formulas is problematic in practice [12]. Therefore, we derive $\overline{\Psi}(\tilde{x})$ efficiently (in polynomial time) as a less general interpolant provided by the greedy *constraint deletion* and *slackening* techniques described in [21].

Lines 36-38 deal with the witnesses and summarizations returned by the recursive call. The algorithm maintains two variables W and σ that are the set of witnesses and summarizations so far, each initialized to $\emptyset$ and $\bot$, respectively at Line 32. In essence, W'' is an extension of W' with the current transition relation of R. Ideally, we want $\sigma'' = val(ext(W'',\mathcal{G}_c))$, but we relax this into $\sigma'' \supseteq val(ext(W'',\mathcal{G}_c))$ to allow approximations. At Line 37, we merge σ'' with the current set σ of summarizations. The function SELECT at Line 38 selects among the witnesses collected so far the optimal set among them, that is, it returns a set of witnesses $W' \subseteq W \cup W''$ where $val(ext(W',\mathcal{G}_c)) = \sigma$.

The next step (Line 39) propagates the computed path-based invariants to further paths in order to make sure all paths were considered under the global loop invariant. Notice that the recursive call to SymbolicExec in Line 34 possibly also returns a candidate loop invariants of the looping ancestors in $\overline{\mathcal{C}}_{\mathcal{T}}$ in case the path contains looping points. At Line 39 we update $\mathcal{G}_c$ to take into account this generalization. For example, assume that $\mathcal{G}_c$ is of the form $p(k,\tilde{x}) \wedge \Psi_c(\tilde{x}_P) \wedge \rho_c(\tilde{x}_P,\tilde{x})$, and $\mathcal{G}_P$, where $\overline{\mathcal{C}}_{\mathcal{T}} = \langle \mathcal{G}_P; \_\rangle :: \_$, is of the form $p(k_P,\tilde{x}_P) \wedge \Phi_P(\tilde{x}_P)$. APPLYINVARIANT replaces $\Psi_c(\tilde{x}_P)$ in $\mathcal{G}_c$ with the more general $\Psi_P(\tilde{x}_P)$ of $\mathcal{G}_P$, and the resulting state is assigned to $\mathcal{G}_c$. This results in weakening of reducts in Line 33, which may wake up a new transition which was infeasible previously but now under the weaker context $\mathcal{G}_c$ becomes feasible.

We now state the correctness of our algorithm.

THEOREM 1 (Soundness). *Assume set S of all solutions of a state $\mathcal{G}$ given a CLP program $\boldsymbol{P}$. SymbolicExec($\mathcal{G}$,nil) returns an answer tuple $\langle \_;\_;\sigma;\_\rangle$ such that $\otimes_{\varphi\in S} val(\varphi) \sqsubseteq \sigma$.*

To improve the efficiency of our algorithm, some heuristics can be applied such as prioritizing feasible reducts before infeasible reducts to obtain a more general abstraction, as feasible reducts have the potential to reach a looping point, which may result in generation of path-based invariant that weakens the context of some looping point.

7. Experimental Evaluation

We conducted our experiments implementing the classic Weiser's algorithm [26] for program slicing and plugging it into our framework. We performed experiments to address the following issues: (a) the effectiveness of our approach against a path-insensitive version by comparing the size of their slices, and (b) how efficiently our path-sensitive approach can compute program slices.

The implementation of our framework models the heap as an array. A flow-insensitive pointer analysis is then used to partition updates and reads into alias classes where each class is modeled by a different array. A theorem prover is used to decide linear arithmetic formulas over integer variables and array elements in order to check the satisfiability of formulas, computing interpolants, and computing individually invariants. External functions are modeled as having no side effects and returning an unknown value. The implementation of the slicing analysis covers loops, function calls, pointers, and arrays. Given a statement that involves pointers the sets *def* and *use* utilize the results of the pointer analysis. For instance, given the statement `*p =*q` the set *def* contains everything that might be pointed to by `p` and the set *use* includes everything that might be pointed by `q`.

Table 1 presents our experimental results on a set of C benchmarks. We used several instrumented device driver programs previously used as software model checking benchmarks: cdaudio, diskperf, floppy, and serial. In addition, we also considered mpeg, the mpeg-1 algorithm for compressing video, and fcron.2.9.5, a cron daemon. We consider for the criteria variables that may be of interest during debugging tasks. For the instrumented software model checking programs, we choose as slicing criteria the set of variables that appear in the safety conditions used for their verifica-

Program	LOC	Path-Insens		Path-Sens	
		Size Red	Time	Size Red	Time
mpeg	5K	4%	21s	43%	628s
diskperf	6K	32%	2s	57%	94s
floppy	8K	36%	9s	47%	263s
cdaudio	9K	23%	10s	52%	301s
serial	12K	39%	16s	50%	395s
fcron.2.9.5	12K	42%	32s	61%	832s
Mean		**23%**	**15s**	**51%**	**418s**

Table 1. Reduction in Slice Size and Analysis Times for Slicing on Intel 3.2Gz 2Gb.

tion in [15]. In the case of mpeg we choose a variable that contains the type of the video to be compressed. Finally, in fcron.2.9.5 we choose all the file descriptors opened and closed by the application.

Table 1 compares the slicer algorithm making use of path-sensitivity provided by our framework (columns labelled with Path-Sens) against the same slicer coupled in the framework but without path-sensitivity (labelled with Path-Insens). Path-insensitivity is achieved by the following modifications in the framework: (1) all paths are feasible, and (2) reuse always summarizations. Those changes have similar effect to merge always the abstract states along incoming edges in a control-flow merging node. Clearly, we could have used a much faster off-the-shelf path-insensitive program slicer using, for example, using dependence graphs [17]. However, our objective here is to isolate the impact of path-sensitivity and hence, we decided to perform the comparison on a common platform to produce the fairest results.

The column LOC represents the number of lines of program without comments. The column Size Red shows the reduction in slice size (in %) wrt the original program size. The reduction size is computed using the formula $(1 - \frac{size\ of\ slice}{size\ of\ original}) \times 100$. By *size* we mean all executable statements in the program, excluding type declarations, unused functions, comments, and blank lines. The column Time reflects the running time of the analysis in seconds excluding the alias analysis phase and the production of the CFG since they are negligible. Finally, we summarize in row Mean the numbers of columns Size Red and Time by computing their *geometric* and *arithmetic* mean, respectively.

Our results described in Table 1 demonstrate the effectiveness of the path-sensitive slicer at least for the benchmarks and criteria used. Overall, the path-sensitive variant produces program slices roughly 30% smaller than its path-insensitive counterpart. We observed the largest reduction margin for the program mpeg ranging from 4% to 43%. The selected criteria here heavily affect the evaluation of many if statements. Path-sensitivity allows us to eliminate many branches and hence, all dependencies originated from them.

More importantly, these results exhibit that our path-sensitive framework is able to slice those programs in a reasonable amount of time. As expected, the slicing time is closely related to the size of the program. We investigated the slower times for the programs mpeg and fcron.2.9.5. We noticed that the reason in both cases was due to the naive fixpoint computation implemented in the slicing algorithm. The current implementation is not optimized and computes dependencies from scratch at each new iteration. This caused a significant slowdown due to the high frequency of fixpoint iterations required in both programs.

8. Conclusion

We presented a path-sensitive symbolic execution-based framework for program analysis. Its accuracy is essentially that of a full enumeration of all symbolic paths, up to loops. We then addressed the fundamental problem of a large enumeration space. We began with a formulation of the analysis in dynamic programming, a method which can be efficient whenever the reuse of subsolutions, which constitute part of bigger solutions, is often possible. Our main contribution was to define a condition, based on interpolation, that is weak enough to allow reuse, and another condition, witnesses, a condition that balances the first condition in such a way as to preserve accuracy. Finally, we demonstrated the practicality of our framework by experimenting with a driving example of program slicing. We showed that our approach can scale up to tens of thousand of lines of C code producing slices significantly smaller than a path-insensitive slicer.

References

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. In *PASTE '07*, pages 43–48, 2007.

[2] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *PLDI '98*, pages 72–84, 1998.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01*, pages 203–213.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software (extended abstract). In *PLDI'03*, pages 196–207.

[5] R. Bodík, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *PLDI '97*, pages 146–158.

[6] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. *FSE'97*, pages 361–377.

[7] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis. In *4th POPL*, pages 238–252. ACM Press, 1977.

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

[9] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI '02*, pages 57–68, 2002.

[10] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1976.

[11] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI '08*, pages 270–280, 2008.

[12] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verificatio n conditions. In *28th POPL*, pages 193–205. ACM Press, 2001.

[13] H Hampapuram, Y. Yang, and M. Das. Symbolic path simulation in path-sensitive dataflow analysis. In *PASTE'05*, pages 52–58, 2005.

[14] W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis via satisfiability modulo path programs. In *POPL'10*, pages 71–82, 2010.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, pages 232–244. ACM Press, 2004.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *29th POPL*, pages 58–70. ACM Press, 2002. SIGPLAN Notices 37(1).

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88*, pages 35–46.

[18] F. Ivancic, Z. Yang, M K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform. In *CAV/05*, pages 301–306.

[19] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.

[20] J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *23rd AAAI*, pages 297–303. AAAI Press, 2008.

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

[22] G. A. Kildall. A unified approach to global program optimization. In *POPL '73*, pages 194–206.

[23] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP'05*, pages 5–20, 2005.

[24] K. L. McMillan. Lazy abstraction with interpolants. In *CAV '06*, pages 123–136.

[25] A. Thakur and R. Govindarajan. Comprehensive path-sensitive data-flow analysis. In *CGO '08*, pages 55–63, 2008.

[26] M. Weiser. Program slicing. In *ICSE '81*, pages 439–449, 1981.