Demand-Driven Path-Sensitive Program Slicing

JOXAN JAFFAR   JORGE NAVAS   and   ANDREW E. SANTOSA

School of Computing
National University of Singapore
Republic of Singapore

Abstract

Program slicing is a technique to extract relevant parts of a program, and it is widely used in program debugging, parallelization, testing, reverse engineering, etc. This paper concerns *static slicing* and it follows the Weiser's definition of slicing that consists of computing what statements of the program might affect the value of some particular variable at a specified program point. We argue that although there is a broad variety of static slicing methods, it is commonly assumed that all paths are executable. However, this limitation may be completely unacceptable in, for instance, debugging and program understanding tasks since the slice is often quite big.

In this paper, we present a fully *path-sensitive* slicing that identifies infeasible paths in order to obtain accurate slices. Infeasible paths are detected by performing symbolic execution of the program. The major challenge is that in general there are exponentially many paths. Our method traverses the symbolic execution tree, in a post-order manner, and discovers an *interpolant* which generalizes the execution context of the tree. This enhances the likelihood that the *dependencies* computed for that tree can be *reused* in multiple contexts. Another key feature is that our algorithm stores, for each dependency, the executable path that defines it, that is, the *representative path formula* that gives rise to it. By doing so, the dependency information is accurate since it is only used if the new context demonstrates that the representative is executable. In fact, for loop-free programs, our algorithm computes *exactly* the statements relevant to the slicing criterion.

1. Introduction

Program slicing is a well-known technique that identifies the parts of a program that potentially affect the values of specified variables at some program point– the *slicing criterion*. Slicing was first developed to facilitate software debugging, but it has subsequently used for performing such diverse tasks as parallelization, software testing and maintenance, program comprehension, reverse engineering, program integration and differencing, and compiler tuning.

Static slicing was introduced by Weiser [39] who defined the *slice* of a program with respect to a program point p and a variable x as all statements of the program that might affect the value of x at point p. The Weiser's method is *flow-sensitive* (i.e., the analysis of statements depends on the order between them) and *context-sensitive* (i.e., the analysis of a called procedure is "sensitive" to the context in which it is called) although imprecise since it does not solve the *calling-context problem* (i.e., it includes unrealizable paths). Today, the most popular kind of static slicing is via *graph reachability*. The computation of a slice can be divided in two steps. In the first step, *Program Dependence Graphs* (PDGs) [32] are built, and then the algorithm produce slices by applying graph reachability analysis on them. Horwitz, Reps and Binkley [19, 20, 33] introduced first *System Dependence Graphs* (SDGs), which are an extension of PDGs in order to compute precise interprocedural slices, solving the *calling-context problem*. Since the Weiser's seminal work, several variants of slicing, which are not static, have been also proposed such as *dynamic slicing* [25, 26], *quasi-static slicing* [38], and *conditioned slicing* [5]. Moreover, there has been a large number of subsequent works (most of them by extending SDGs) for including more programming languages features such as unstructured control flow, composite data types and pointers, concurrency, and so on (see e.g. [3, 37, 41] for references).

In spite of the vast diversity of static slicing methods, it is commonly assumed that every program path is executable. However, according to Hedly et al. [17] 12.5% of all paths are not executable (i.e., there is no input for which the paths will be executed). Moreover, path feasibility tends to decay exponentially with an increasing number of predicates considered [42]. More importantly, in program slicing, the existence of infeasible paths has a great impact. Even if the best known algorithms are used, the sliced program is often quite imprecise because it is too big to be useful due to the consideration of all possible paths [1]. For instance, in the following program fragment, where S1 and S2 are arbitrary code[1]:

```
x = 0;          foo(y)
foo(x);         { if (y > 0) S1 else S2 }
```

S1 will never be executed. However, traditional static slicing methods would consider any fragment in S1 relevant to the criterion. The reason is that those methods do not exclude infeasible paths. The ability of discriminating statically which paths can be executed is called *path-sensitiveness* and can be used to improve the *precision* of static slices because spurious information from infeasible paths can be excluded from consideration.

Then, *why are traditional slicing methods path-insensitive?* The static identification of non-executable paths is an undecidable problem in the general case [40]. Even for loop-free programs the detection of infeasible paths is far from being trivial. For instance, in the following program fragment, where (∗) represents non-deterministic values, naïve approaches using symbolic execu-

---

[1] Today, software is often written in this way in which procedures are designed for general purposes and hence, they need to include many sanity checks.

2009/7/17

tion + theorem prover fail abruptly since the number of paths is 2^n:

```
if (*) S1 else S2
if (*) S3 else S4
...
if (*) Sn-1 else Sn
```

One well-known technique that mitigates this problem is *dynamic programming* (*DP*). *DP* solves combinatorial optimization problems exhibiting certain features. 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. Therefore, essential to *DP* is the notion of *memoization* of subproblem solutions already solved. Memoization is the identification of whether a solution can be reused or not. 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 [22].

Our approach. We present a technique that given a slicing criterion $\mathtt{C} \equiv \langle \mathtt{p}, \mathtt{x} \rangle^2$, discovers all statements of the program that might affect the value of $\mathtt{x}$ at program point $\mathtt{p}$. That is, we follow the Weiser's definition of program slicing. The novelty here is that our method is path-sensitive in the sense that it will detect infeasible paths not considering any relevant statement inferred from them. The major challenge is hence that in general there are exponentially many paths.

Our method considers a symbolic computation tree as a decision tree where a node has a conjunction of formulas (i.e. path formula) $\Psi \equiv \psi_1 \wedge \ldots \wedge \psi_i$, symbolically representing a set of states. $\psi_1, \ldots, \psi_i$ are constraints generated from each statement in the path from the root to that node. Its successor node has an incrementally larger conjunction representing a new decision. Ψ is satisfiable iff the path is feasible.

The algorithm performs a depth-first traversal of the tree, in a post-order manner. Whenever a path, Ψ, in the tree is traversed completely (either reaching the end of the program or becoming unsatisfiable), it enlarges the path formula Ψ by using the notion of *interpolation* [7] but still preserving the (un)satisfiability of Ψ:

- If Ψ is *satisfiable* then every state along the path can be generalized *true* since when we start from the state *true* traversing remaining computation path on Ψ, that is, the path Ψ_2 such that $\Psi = \Psi_1 \wedge \Psi_2$, Ψ_2 would still be satisfiable.

- If Ψ is *unsatisfiable* then every state along the path Ψ can be generalized as long as the unsatisfiability is preserved. That is, if $\Psi = \Psi_1 \wedge \Psi_2$, then we generalize the state after the execution of Ψ_1 into another state φ as long as $\varphi \wedge \Psi_2$ remains unsatisfiable.

Note that if Ψ is unsatisfiable, we have found an infeasible path and hence, our algorithm stops the traversal along that path without including any statement relevant to x in that path. In this case, our method also computes an interpolant that assures the false condition of the path. Together with the interpolant Ψ', our algorithm keeps track of dependencies relevant to the computation of criterion x. In our approach, there exists a *dependency* from the variable x to y if x is relevant to the computation of the value of y. Then, our algorithm starts propagating back the interpolants and updating the dependencies along the same path to ancestor states resulting in their possible generalization.

A key advantage of our method is that, during the traversal, our algorithm will not explore a subtree whenever it has been already explored under more general context. Therefore, the method provides an enhancement to the general method of *memoization*. We

---

argue that our method can compute very efficiently the dependencies due to the use of *interpolants* that generalize the context under the subtree's dependencies have been computed and it greatly increases the chance of being reused in different contexts. We also claim that our method is *demand-driven* since it only explores new subtrees only if new dependencies can arise from them.

Note that the use of interpolants is correct since our method only reuses the dependencies of a subtree whenever it is re-traversed with a less general context. Unfortunately, it does not preserve necessarily *precision*. When the analysis of a subtree reuses the dependencies from another subtree analyzed with a more general context, the solution may include more dependencies generated from paths that are possible with the more general context but not necessarily with a less general context. That is, some of those paths may be infeasible in the less general context.

To remedy this lack of precision, our algorithm also keeps track, for each dependency, the executable path that defines it which we will call the *representative path formula*. The key idea in order to achieve efficiency but without any loss of precision is that our algorithm only reuses a dependency if the new context demonstrates that its representative path holds. Otherwise, the dependency cannot be reused and a new traversal of the subtree is required.

Organization. The rest of this paper is organized as follows. In Section 2 we provide an informal overview of our approach showing several examples, and in Section 3 we introduce relevant concepts and definitions required for the rest of the paper. In Section 4, we present our slicing algorithm for loop-free programs which we refine in Section 4.1, and we demonstrate that it is optimal for straight-line programs. In Section 5, we extend the previous in order to support programs with loops, and finally, Section 6 concludes.

2. The Basic Idea

Our slicing method is best intuitively explained through several examples. Details will follow in subsequent sections.

First, we model the C program into a transition system. Such modeling has been presented in various works [14, 21, 10] and is informally shown here. Consider the C program fragment in Fig. 1(a). The program points are enclosed in angle brackets. Its transition system is shown in Fig. 1(b). For instance, the transition $t_1(a,b,c,x,y,z) \mapsto t_2(a,b,c',x,y,z), c' = 0$ represents that the system state switches from program point 1 to 2 and the constraint $\Psi \equiv c' = 0$ is the statement executed. The values of the rest of variables remain the same.
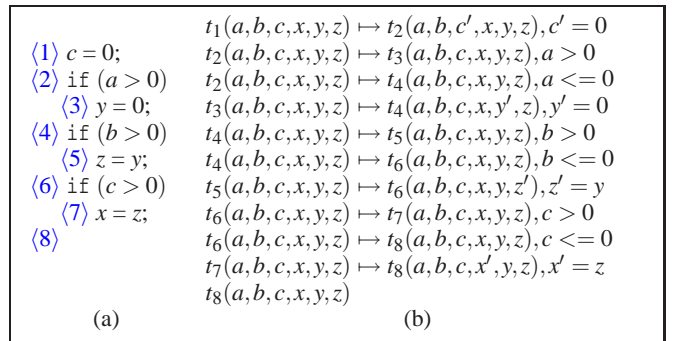

$$
\begin{array}{ll}
& t_1(a,b,c,x,y,z) \mapsto t_2(a,b,c',x,y,z), c' = 0 \\
\langle 1 \rangle\ c = 0; & t_2(a,b,c,x,y,z) \mapsto t_3(a,b,c,x,y,z), a > 0 \\
\langle 2 \rangle\ \mathtt{if}\ (a > 0) & t_2(a,b,c,x,y,z) \mapsto t_4(a,b,c,x,y,z), a <= 0 \\
\quad \langle 3 \rangle\ y = 0; & t_3(a,b,c,x,y,z) \mapsto t_4(a,b,c,x,y',z), y' = 0 \\
\langle 4 \rangle\ \mathtt{if}\ (b > 0) & t_4(a,b,c,x,y,z) \mapsto t_5(a,b,c,x,y,z), b > 0 \\
\quad \langle 5 \rangle\ z = y; & t_4(a,b,c,x,y,z) \mapsto t_6(a,b,c,x,y,z), b <= 0 \\
\langle 6 \rangle\ \mathtt{if}\ (c > 0) & t_5(a,b,c,x,y,z) \mapsto t_6(a,b,c,x,y,z'), z' = y \\
\quad \langle 7 \rangle\ x = z; & t_6(a,b,c,x,y,z) \mapsto t_7(a,b,c,x,y,z), c > 0 \\
\langle 8 \rangle & t_6(a,b,c,x,y,z) \mapsto t_8(a,b,c,x,y,z), c <= 0 \\
& t_7(a,b,c,x,y,z) \mapsto t_8(a,b,c,x',y,z), x' = z \\
& t_8(a,b,c,x,y,z) \\
\quad\quad\text{(a)} & \quad\quad\quad\quad\text{(b)}
\end{array}
$$

Figure 1: A Program Fragment and Its Transition System

Using a traditional algorithm to slice the program with respect to variable $\mathtt{x}$ at program point $\langle 8 \rangle$, we obtain everything from the original program. However, a closer inspection reveals that the assignment of z to x at point $\langle 7 \rangle$ is not executable since it is defined

in an infeasible path. Our algorithm produces the most precise slice here, which is the empty slice.

Infeasible Paths and Interpolation. We start by illustrating the principles underlying our method which make it possible to slice all statements of program in Fig. 1(a), and enhance the likelihood that the dependencies computed in a context can be reused in multiple contexts, making our approach practical. The naïve symbolic execution of the transition system is shown on the left side in Fig. 2. The representation is a directed tree, with nodes labeled as P#C (P is the corresponding program point and C is the context) and edges between two locations labeled by the instruction that executes when control moves from the source to the destination. We represent conditional statements with *diamond* nodes, basic block of statements with *box* nodes, and terminal nodes with *ellipses*. On the other hand, *feasible* transitions are denoted by (black) solid edges, and *infeasible* transitions by (red) edges with a dot in the arrow head. On the right side, in Fig. 2, we show the derivation tree computed by our method that avoids the exponential behavior of the naïve approach while it detects the infeasible path.
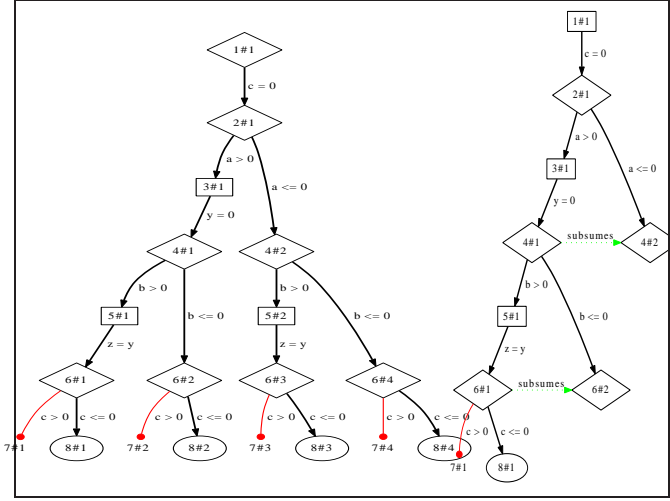

Figure 2: Interpolation and Infeasible Paths

Our method starts traversing the tree in a depth-first manner, reaching the node 7#1 with the path formula $\Psi_{7\#1} \equiv c = 0 \wedge a > 0 \wedge y = 0 \wedge b > 0 \wedge z = y \wedge c > 0$. Since the formula is unsatisfiable, we have found an infeasible path. Thus, our algorithm does not keep traversing the path and backtracks to the immediate ancestor. More importantly, it generates at 7#1 an interpolant. Intuitively, an interpolant is simply a formula Ψ' that generalizes another formula Ψ that represents the state of a node in the tree. Therefore, at node 7#1 our algorithm needs to generalize $\Psi_{7\#1}$ preserving the falsity of the path since the path is unsatisfiable. We just generate the interpolant, $\Psi'_{7\#1} \equiv false$. In addition, we store, for each node n, a set $\mathcal{D}_n^C$ that keeps track of the *dependencies* among variables. Informally, $x \in \mathcal{D}_n^C$ if variable x is relevant to the computation of values of the slicing criterion C at the node n. Therefore, the algorithm stores at 7#1 the set $\mathcal{D}_{7\#1}^x = \emptyset$, since the path is infeasible and there is no way that any variable can affect the value of x. Later, the algorithm will reach the terminal node 8#1. Again, the algorithm needs to generate an interpolant that should be as general as possible. Since the formula path is satisfiable, it generates $\Psi'_{8\#1} \equiv true$, which is the most general possible interpolant. Furthermore, the algorithm adds the criterion variable x. That is, $\mathcal{D}_{8\#1}^x = \{x\}$.

We next use the interpolants of 7#1 and 8#1 to produce the interpolant of 6#1. We first compute the *weakest precondition* [12] *wrt* the predicate c > 0 and the postcondition $\Psi'_{7\#1} \equiv false$, which

is $c \leq 0$. Similarly, for c $\leq$ 0 and the postcondition $\Psi'_{8\#1} \equiv true$, that in this case is *true*. [3] Then, the algorithm needs to *join* at 6#1 the interpolants propagated from its children, 7#1 and 8#1. The final interpolant $\Psi'_{6\#1} \equiv c \leq 0 \wedge true \equiv c \leq 0$ is the conjunction of these candidate interpolants

Moreover, in this case, the algorithm simply propagates the dependencies to the ancestor. Informally, we explain the rationale behind the backward propagation of the dependencies among variables for a transition from node k to k' with an arbitrary assignment $x_1 = x_2$. Assume that x_1 is relevant to the criterion C at node k', i.e., $\mathcal{D}_{k'}^C = \{x_1\}$. Then, the propagation of relevance from variable to variable is deduced from the assumption that if x_1 might be relevant to C, x_2 might be relevant to x_1, and hence, relevant to C. In addition, the assignment *kills* the value of x_1 before the assignment. Thus, x_1 is not anymore relevant to the criterion, obtaining $\mathcal{D}_k^C = \{x_2\}$. After this propagation, the algorithm needs to *combine* at 6#1 the dependencies propagated back from children by applying the union, obtaining $\mathcal{D}_{6\#1}^x = \{x\}$.

This 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 in order to (hopefully) reuse the dependencies whenever a new context is explored. For instance, assume the transition from node 2#1 to 4#2 which is a new context for $\langle 4 \rangle$. The path formula at 4#2 is $\Psi_{4\#2} \equiv c = 0 \wedge a \leq 0$. Before exploring the subtree, our method tests if $\Psi_{4\#2}$ is *subsumed* by $\Psi'_{4\#1}$. That is, if $\Psi_{4\#2} \models \Psi'_{4\#1}$. The solver answers 'yes' since $\Psi'_{4\#1} \equiv c \leq 0$. Therefore, the algorithm does not need to explore the subtree and reuse the dependencies $\mathcal{D}_{4\#1}^x$. In Fig.2, we denote subsumed transitions by (green) dotted edges and labeled with "subsumes". Note that $\Psi_{4\#2} \models \Psi'_{4\#1}$ succeeds due to the use of the interpolant $\Psi'_{4\#1}$. If we had used the original context, $\Psi_{4\#1}$, which is $a > 0, c = 0, y = 0$, the subsumption would fail.

Finally, we show how dependencies and the slice are updated in relevant transitions. Columns Post-State and Pre-State are the set of dependencies before and after the backward propagation, respectively. Column S_x contains the set of statements included in the slice. Informally, we include the statement s defined in transition from k to k' in S_x if there exists a variable v whose value changes at statement s, and $v \in \mathcal{D}_{k'}^x$.

Transition	Post-State	Pre-State	S_x
5#1 $\xrightarrow{z = y}$ 6#1	$\mathcal{D}_{6\#1}^x = \{x\}$	$\mathcal{D}_{5\#1}^x = \{x\}$	$\emptyset$
3#1 $\xrightarrow{y = 0}$ 4#1	$\mathcal{D}_{4\#1}^x = \{x\}$	$\mathcal{D}_{3\#1}^x = \{x\}$	$\emptyset$
1#1 $\xrightarrow{c = 0}$ 2#1	$\mathcal{D}_{2\#1}^x = \{x\}$	$\mathcal{D}_{1\#1}^x = \{x\}$	$\emptyset$

Programs without Infeasible Paths. Even though there are in general infeasible paths, it is often the case that many subtrees do not contain them, or contain few of them. In these relevant subcases, it is important to note that our method can, in principle, obtain "ideal" interpolants. This essentially means that the number of nodes that need to be considered is linear in the size of the program fragment in question.

Consider now the same program in Fig.1 but without the assignment c=0 at program point $\langle 1 \rangle$. Using our algorithm to slice the program *wrt* x, we will obtain the original program. The reason now is that all paths are executable.

A computation tree built without interpolation is shown on the left side in Fig.3. The tree computed by our algorithm is shown

---

[3] In general, the interpolant obtained need not to be the weakest. In our approach, we use effective polynomial algorithms for approximating the weakest precondition [23].

on the right. Our approach generates the *true* interpolant at each node with a negligible cost and hence, all new contexts will be subsumed. As a consequence, the size of the program state is *linear wrt* to the size of the program. Therefore, the key idea illustrated with this program fragment is that our *demand-driven* approach[4] only increases the program state if those new states can improve the precision of the analysis. Otherwise, if no infeasible paths, our approach does not enlarge the program state, and runs proportionally to traditional slicing methods.
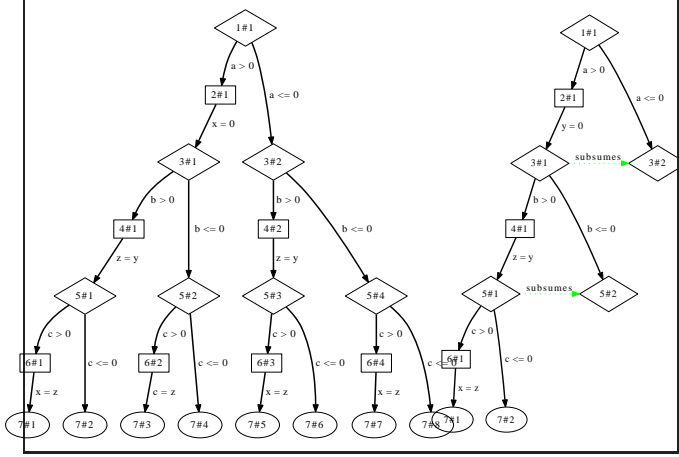
Figure 3: Linear Size of the Program State without Infeasible Paths

We also show the slice obtained wrt x for this program fragment. Note that we only show the slice wrt to assignments in the original program:[5]

Transition	Post-State	Pre-State	S_X
6#1 $\xrightarrow{x=z}$ 7#1	$\mathcal{D}^x_{7\#1} = \{x\}$	$\mathcal{D}^x_{6\#1} = \{z\}$	{ x=z }
4#1 $\xrightarrow{z=y}$ 5#1	$\mathcal{D}^x_{5\#1} = \{z\}$	$\mathcal{D}^x_{4\#1} = \{y\}$	{ z=y, x=z }
2#1 $\xrightarrow{y=0}$ 3#1	$\mathcal{D}^x_{3\#1} = \{y\}$	$\mathcal{D}^x_{2\#1} = \{\emptyset\}$	{ y=0, z=y, x=z }

Representative path formulas. While subsuming a node may save search space and yet preserve the correctness of the analysis, subsuming a node does not preserve the *accuracy* of the analysis. It is here where we balance efficiency (subsumption) and accuracy (representative paths).

Consider the C program fragment in Fig. 4(a). The slice *wrt* to x at program point $\langle 8 \rangle$ obtained by our algorithm so far, is the original program. However, a more detailed analysis of the program discovers that the statement at program point $\langle 4 \rangle$ does not affect the computation of x. The reason is that y can only affect x if statement at $\langle 7 \rangle$ is executed, but $\langle 7 \rangle$ is unreachable because it is located at an infeasible path whenever the program reaches $\langle 4 \rangle$.

The derivation tree computed by our algorithm, explained so far, is shown on the left side in Fig. 5. At node 6#1 the algorithm has joined the solutions from its children $\mathcal{D}^x_{8\#1} = \{y\}$ and $\mathcal{D}^x_{7\#1} = \{x\}$ obtaining $\mathcal{D}^x_{6\#1} = \{x,y\}$. Then, the process continues normally until a new context has been found for $\langle 6 \rangle$, 6#2. Since the interpolant stored for 6#1 is *true* (because no infeasible paths were found), the new context will be subsumed, and $\mathcal{D}^x_{6\#1}$ will be reused. The rows

---

[4] Our technique can be also considered *lazy* since new paths are only explored if they can provide new relevant information.

[5] We will explain in Sec. 4 how to consider branch statements. Our solution is very similar to Weiser's [39].
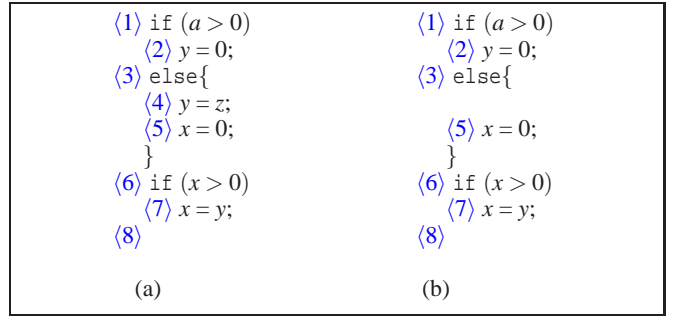

(a) (b)

Figure 4: Another Program and its Precise Slice

labeled by l) and r) contain the result of our algorithm for the left and right subtree rooted at 1#1, respectively:

	Transition	Post-State	Pre-State	S_X
l)	7#1 $\xrightarrow{x=y}$ 8#1	$\mathcal{D}^x_{8\#1} = \{x\}$	$\mathcal{D}^x_{7\#1} = \{y\}$	{ x=y }
	2#1 $\xrightarrow{y=0}$ 3#1	$\mathcal{D}^x_{3\#1} = \{x,y\}$	$\mathcal{D}^x_{2\#1} = \{x\}$	{ y=0 , x=y }
r)	5#1 $\xrightarrow{y=z}$ 6#2	$\mathcal{D}^x_{6\#2} = \{x,y\}$	$\mathcal{D}^x_{5\#1} = \{x,z\}$	{ y=z }
	4#1 $\xrightarrow{x=0}$ 5#1	$\mathcal{D}^x_{5\#1} = \{x,z\}$	$\mathcal{D}^x_{4\#1} = \{z\}$	{ x=0,y=z }

Therefore, the slice includes the statements y = 0, x = y, x = 0, y = z, and also, their corresponding branch statements. That is, the slice obtained is the original program fragment.

For the sake of discussion, let us consider that the node 6#2 is not subsumed by 6#1. The derivation tree is shown on the right in Fig.5. The key observation is that the new subtree rooted at 6#2 contains an infeasible path if x > 0. This infeasible path forces to not add the variable y at 6#2 (i.e., $\mathcal{D}^x_{6\#2} = \{x\}$ rather than $\mathcal{D}^x_{6\#2} = \{x,y\}$). The results of the slicing algorithm for the right subtree are shown:

	Transition	Post-State	Pre-State	S_X
r)	5#1 $\xrightarrow{y=z}$ 6#2	$\mathcal{D}^x_{6\#2} = \{x\}$	$\mathcal{D}^x_{5\#1} = \{x\}$	{ $\emptyset$ }
	4#1 $\xrightarrow{x=0}$ 5#1	$\mathcal{D}^x_{5\#1} = \{x\}$	$\mathcal{D}^x_{4\#1} = \{\emptyset\}$	{ x=0 }

The slice is presented in Fig. 4(b).The novelty is that we obtain a more precise slice since the statement y = z is not included.
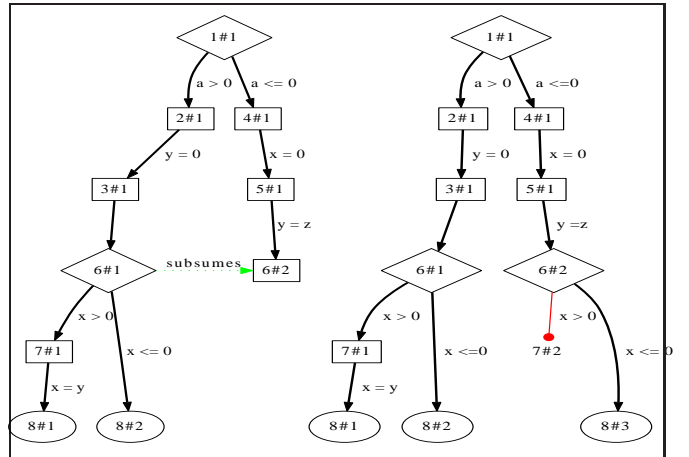

Figure 5: Representative Paths

The program in Fig. 4(a) 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 strength the condition that decides whether a node can be subsumed or not

in order to be precise. Hence, given the set of dependencies $\mathcal{D}_{k}^{C}$ our algorithm will also store, for each variable x in $\mathcal{D}_{k}^{C}$, the path formula Φ_x that defines it. We call this formula Φ_x, the *representative path formula*. Informally, a node n with context Ψ_n is subsumed by k with interpolant Ψ'_k and dependencies $\mathcal{D}_{k}^{C}$ if $\Psi_n \models \Psi'_k$ (as before), and for all x in $\mathcal{D}_{k}^{C}$ each representative path formula Φ_x is possible with the new context Ψ_n. That is, $\Phi_x \wedge \Psi_n$ is satisfiable.

Coming back to the tree on the right in Fig. 5, the context at node 6#2 is $\Psi_{6\#2} \equiv a \leq 0 \wedge x = 0 \wedge y = z$. The interpolant at 6#1 is $\Psi'_{6\#1} \equiv true$. It is straightforward to see that $\Psi_{6\#2} \models \Psi'_{6\#1}$. In addition, we test that, for each variable in the dependency set, their representatives still hold. That is, both $(a \leq 0 \wedge x = 0 \wedge y = z) \wedge (x \leq 0)$ and $(a \leq 0 \wedge x = 0 \wedge y = z) \wedge (x > 0 \wedge x = y)$ are satisfiable. Here, $\Phi_x \equiv x \leq 0$ is the formula that defines x, and $\Phi_y \equiv x > 0 \wedge x = y$ is the formula that defines y. The first formula holds, but the second is unsatisfiable because $(x = 0 \wedge x > 0)$ does not have solution. Therefore, the algorithm must explore the node 6#2 obtaining the most precise.

2.1 Related Work

The significance of infeasible paths detection is well understood in many software engineering fields. Since the general problem of detecting infeasible paths is undecidable [40], most approaches attempt to solve this problem in an *idealistic* or *incomplete* fashion: symbolic execution + theorem prover (e.g., [16, 4, 9]), based on heuristics [15, 42], and more recently, pattern recognition [31].

In program slicing, Korel and Laski [25, 26] were who first proposed the notion of *dynamic slicing*. A dynamic slice is a part of a program that affects the value of a variable in a particular execution. As a consequence, the slice is much smaller than static slices and, not surprisingly, without infeasible paths. However, dynamic approaches are only useful if all input can fixed to particular values. *Quasi-static slicing* [38] is a slicing method between static and dynamic. It is used when some input values are fixed using an initial prefix while other can vary. *Constrained slicing* [13] is conceptually similar to [38] although the input values can be constrained by Boolean predicates. *Conditioned slicing* [5] generalizes [38, 13] since input values can be characterized by a first order logic formula. These hybrid approaches [38, 13, 5] have in common with our method that, given some constraints on the input values, they need to propagate some program context through executable paths. Hence, some infeasible paths may be detected. However, when fully unconstrained inputs are used they behave as traditional static slicing, and hence, they do not detect infeasible paths. *Path slicing* [24] takes as input a possibly infeasible path to a target location and eliminates all the operations irrelevant to the reachability of the target location. It is used to accelerate the process of generating interpolants. The closest to ours is probably Snelting et al. [36, 34, 35]. They assume a sliced program which is then refined by eliminating dependencies between nodes that are defined on non-executable paths. They use *interval analysis* and *BDDs* to overcome the potential combinatorial explosion in order to reduce the number of paths to be tested. We consider this work orthogonal to us in the sense that we could also to use those techniques to accelerate our interpolation method.

The interpolation method that we employ in this paper has been similarly applied by the authors to resource-constrained shortest path problem [22] and to safety verification of programs [23]. The notion of interpolation itself has been also successfully used in the software model checking community. Henzinger et.al. [18] and McMillan [30] employ interpolation for automated successive refinement of abstract domain in an abstract interpretation setting in order to prove a safety condition. McMillan also in [28] achieves unbounded model checking by using interpolation to successively

refine an abstract transition relation that is then subjected to an external bounded model checking [2] procedure. Techniques for generating interpolants, for use in state-of-the-art SMT solvers, are presented in [6]. The use of interpolants can also be seen in the area of theorem-proving [29]. To the best of our knowledge our approach is the first in using interpolation to enhance program analysis, and in particular, slicing.

3. Preliminaries

In this section, we introduce all the terminology and definitions required for the understanding in the rest of the paper.

3.1 Dependencies and Slices

We first define the *Control Flow Graph* (*CFG*) of a program. Given a function f of a C program, its CFG is the tuple $\langle N, E, b, e \rangle$, where N is the set of nodes denoting statements, with special nodes $b, e \in N$ denoting the first and last statements of the function. In the context of a C program, the value passing to the arguments of the function can be considered as the first statement, and the return statement can be considered as the last statement, assuming each function has exactly one occurrence of the return statement. E is the set of pairs of elements of N, where $(s_1, s_2) \in E$ if and only if the statement s_1 immediately precedes s_2 in the execution of the program. Assume that the CFG is written in *Static Single Assignment* (*SSA*) form [8] such that each assignment updates a fresh variable (program variable with a new *version number*). Given a slicing criterion $\langle k, \tilde{x} \rangle$, we assume the variables $\tilde{x}$ have the right version numbers at program point k.

A typical C program contains a number of functions, each with their own CFG. In order to construct a single CFG of the whole program, we perform function *inlining* in the following way. At every function call point, we replace the function call with an assignment to the formal arguments of the callee, that is, the start statement of the callee, and we include from the end statement of the callee to the statement next to the call in the caller. We introduce a fresh variable to pass the return value of the callee. The CFG of the program starts with the start statement of the function main and ends with its end statement.

We define *path* from s to t as a sequence of pairs (s_0, s_1),- $(s_2, s_3), \ldots, (s_{n-1}, s_n)$ where $s_i \in N$ for all $0 \leq i \leq n$, each pair is an element of E, $s = s_0$ and $t = s_n$. There is no s such that $(s, b) \in E$ or $(e, s) \in E$. The *inverse dominator* of a control statement s, denoted by $\mathsf{D}(s)$ is a statement on every path to the end statement [11]. $\mathsf{ND}(s)$ [39] denotes the set of statements along a path to the nearest inverse dominator $\mathsf{D}(s)$, excluding s and the inverse dominator itself.

Given an assignment statement s, $\mathsf{USE}(s)$ is the set of variables referenced in s, and $\mathsf{DEF}(s)$ is the variable defined in s.[6] If s is a control statement (if-then-else, while loops, etc.), $\mathsf{USE}(s)$ is the set of variables in the condition and $\mathsf{DEF}(s)$ is empty. Given a path from s to t, the relation $\mathsf{INFL}(s,t)$ holds when $\mathsf{DEF}(s) \in \mathsf{USE}(t)$. In a sense, $\mathsf{INFL}(s,t)$ identifies that the definition in s reaches t, and therefore t is *dependent* on s. Now, the relation $\mathsf{CINFL}(s,t)$ holds for a control statement s and any statement t if and only if t is included along a path from s to its nearest inverse dominator, and s is the nearest such statement from t. Here, when $\mathsf{CINFL}(s,t)$ holds, the control statement s determines whether t is executed or not. In this way, t is also dependent on s. The INFL and CINFL relations defined here formalize the notion of dependency treated in this paper.

---

[6] We consider that multiple assignments are transformed into a sequence of single assignments.

Given a criterion $C \equiv \langle k, \tilde{x} \rangle$ for a program P and s_k the statement at program point k, we define the following monotonic function:

$$
\begin{aligned}
F_P^C(S) \quad = \quad & \{s \in P | \mathsf{DEF}(s) \in \tilde{x}\} \cup \\
& \{s \in P | s \text{ is on a path to } s_k, \mathsf{DEF}(s) = \emptyset, \mathsf{USE}(s) \cap \tilde{x} \neq \emptyset\} \cup \\
& \{s \in P | \mathsf{INFL}(s,t), t \in S\} \cup \\
& \{s \in P | \mathsf{CINFL}(s,t), t \in S\}
\end{aligned}
$$

The *path-insensitive slice* of the program wrt. the criterion is the set of statements S which is the least solution to the equation $S = F_P^C(S)$, denoted $Slice_{Path-Ins}(P,C)$.

In addition, we can speak about the feasibility or infeasibility of paths. Given a statement s, $\mathsf{EXEC}(s)$ holds when there is a feasible path from b to s_k which includes s. For the *path-sensitive* slice of the program wrt. the criterion, we simply add the requirement $\mathsf{EXEC}(s)$ for every statement s in the solution. Therefore, given a program, its path-sensitive slice is the set

$$
Slice_{Path-Sens}(P,C) = Slice_{Path-Ins}(P,C) \cap \{s \in P | \mathsf{EXEC}(s)\}.
$$

3.2 Constraint Transition Systems

Internally, a program is represented as a set of *constraint transition systems*, one for each function of the program, which can be executed symbolically. Given a CFG the construction of an equivalent constraint transition system is straightforward. Informally, in constraint transition systems, nodes correspond to program counter values and edges between nodes are labeled by constraints. These constraints are obtained from the statement that executes. In the rest of the paper, we shall assume all definitions in Sec. 3.1 described over statements also for constraints. We now provide a formal description of *constraint transitions systems*.

We start by defining a language of first-order formulas. Let $\mathcal{V}$ denote an infinite set of variables, each of which has a type in the domains $\mathcal{D}_1, \cdots, \mathcal{D}_n$, let Σ denote a set of *functors*, and Π denote a set of *constraint symbols*. Functors represent program operations such as arithmetic operations and array assignments, while constraints represent conditionals in program statements such as arithmetic relations, in addition to equalities. There is a special collection of *final* variables. A *term*[7] is either a constant (0-ary functor) in Σ or of the form $f(t_1, \cdots, t_m)$, $m \geq 1$, where $f \in \Sigma$ and each t_i is a term, $1 \leq i \leq m$. A *primitive constraint* is of the form $\phi(t_1, \cdots, t_m)$ where ϕ is a $m-ary$ constraint symbol and each t_i is a term, $1 \leq i \leq m$. A *constraint* is constructed from primitive constraints using logical connectives in the usual manner. Where Ψ is a constraint, we write $\Psi(\tilde{x})$ to denote that Ψ possibly refers to variables in $\tilde{x}$.

A *substitution* θ simultaneously replaces each variable in a term or constraint e into some expression, and we write $e\theta$ to denote the result. A *renaming* is a substitution which maps each variable in the expression into a distinct variable. We write $[\tilde{x} \mapsto \tilde{y}]$ to denote such mappings.

A *grounding* is a substitution which maps each variable into a value in its domain. Where e is an expression containing a constraint Ψ, $[\![e]\!]$ denotes the set of its instantiations obtained by applying *all possible* groundings which satisfy Ψ.

We shall model computation by considering n *system variables* $v_1, \cdots, v_n$ with domains $\mathcal{D}_1, \cdots, \mathcal{D}_n$ respectively, and a program counter k ranging over program points.

DEFINITION 1 (States and Transitions). *A* ground state *is of the form* $(k, d_1, \cdots, d_n)$ *where* k *is a program point and* $d_i \in \mathcal{D}_i, 1 \leq i \leq n$, *are values for the system variables. A* transition *is a pair of states.*

DEFINITION 2 (Symbolic State). *A* symbolic state *(or simply, state)*, $\mathcal{G}$, *is of the form:*

$$
\mathcal{G} \equiv \langle k, \tilde{x}, \Psi(\tilde{x}) \rangle
$$

where k *is a program point,* $\tilde{x}$ *is a sequence of variables over system states, and* Ψ *is a constraint over some or all of the variables* $\tilde{x}$, *and possibly some additional variables. Finally, we write* Ψ_k *and* $\mathcal{G}_k$ *to indicate that* $\mathcal{G}$ *and* Ψ *correspond to program point* k.

DEFINITION 3 (Constraint Transition System, CTS). *A* constraint transition *is a defined as:*

$$
t_k(\tilde{x}) \mapsto t_{k_1}(\tilde{x_1}), \Psi(\tilde{x}, \tilde{x_1})
$$

where $(k, \tilde{x})$ *and* $(k_1, \tilde{x_1})$ *are system states, and* Ψ *is a constraint over* $\tilde{x}$ *and* $\tilde{x_1}$, *and possibly some additional auxiliary variables. Note that* Ψ *is the statement executed between program points* k *and* k_1. *A constraint transition system (CTS) is a finite set of constraint transitions.*

Clearly the variables in a constraint transition may be renamed freely because their scope is local to the transition. We thus say that a constraint transition is a *variant* of another if one is identical to the other when a renaming substitution is performed.

We say that a state is *false* if its constraint is unsatisfiable. We shall also the notation *false* to denote a *false* state. We say that a state is *final* if k is the final program point, one from which there are no transitions.

DEFINITION 4 (Transition Step, Sequence and Tree). *Let there be a CTS for a program, and let* $\mathcal{G} \equiv \langle k, \tilde{x}, \Psi \rangle$ *be a state for this. A* transition step *from* $\mathcal{G}$ *may be obtained providing* Ψ *is satisfiable. It is obtained using a variant* $t_k(\tilde{y}) \mapsto t_{k_1}(\tilde{y_1}), \Psi_1$ *of a transition in the CTS in which all the variables are fresh. The result is a state of the form* $t_{k_1}(\tilde{y_1}), \Psi, \tilde{x} = \tilde{y}, \Psi_1$ *We say that this new state is a false state if the constraint* $\Psi, \tilde{x} = \tilde{y}, \Psi_1$ *is unsatisfiable.*

A transition sequence *is a finite sequence of transition steps which terminate in either a final state or a false state. A* transition path *is a finite sequence of transitions corresponding to a transition sequence. Intuitively, a path denotes the "skeleton" of a sequence. A* transition tree *is defined from transition sequences in the obvious way.*

We shall impose a special condition on transition steps: if a step results in a final state, then the primary variables of the final state are the *final* variables. We say that a transition sequence or path is *successful* if it terminates in a final state; otherwise, the sequence or path is *false*.

3.3 Interpolants

We say that a state $\overline{\mathcal{G}}$ *subsumes* another state $\mathcal{G}$ if $\mathcal{G} \models \overline{\mathcal{G}}$ (i.e., $[\![\overline{\mathcal{G}}]\!] \supseteq [\![\mathcal{G}]\!]$). Equivalently, we say that $\overline{\mathcal{G}}$ is a *generalization* of $\mathcal{G}$. We write $\mathcal{G}_1 \equiv \mathcal{G}_2$ if $\mathcal{G}_1$ and $\mathcal{G}_2$ are generalizations of each other. Note that if $\overline{\mathcal{G}}$ is a *generalization* of $\mathcal{G}$, then there is a constraint Ψ such that $\overline{\mathcal{G}} \wedge \Psi \equiv \mathcal{G}$.

DEFINITION 5 (Interpolant). *A state* $\mathcal{G}_1$ *is an interpolant for a state* $\mathcal{G}$ *if* $\mathcal{G}_1$ *subsumes* $\mathcal{G}$, *and every path in the tree for* $\mathcal{G}$ *has a corresponding path[8] in the tree for* $\mathcal{G}_1$. *In the base case where* $\mathcal{G}$ *is a final state or false state, then* $\mathcal{G}_1$ *is any generalizing state of* $\mathcal{G}$ *which is known to be safe.*

4. Exact Path-sensitive Slicing Algorithm

In this section, we describe our algorithm to compute the precise slice of a loop-free program wrt to a criterion $\mathcal{C}$. For clarity, we will

---

[7] In this paper, we shall only be using simple integer terms and constraints as examples. In general, we can code data structures such as arrays and pointers.

[8] One that uses the same sequence of transitions.

present the algorithm in two steps. First, we show in Fig. 6 its main structure and basic operations. Then, we describe in Sec. 4.1 how to refine the algorithm in order to support representative path formulas leading to an exact slicing algorithm for loop-free programs.

Technically speaking, our algorithm does not compute directly the slice itself but a labeled computation tree from the symbolic execution of the program. Each node is annotated with some dependency information in such a way that we can postprocess the labeled computation tree in a straightforward manner in order to obtain the desired slice.

The algorithm assumes that it is given an empty table to store relationships between dependencies and states. In fact, the pair $\langle \mathcal{G}, \mathcal{D}_k^C \rangle$ lead to the concept of *summarization*. A summarization, Σ, is a partial description of the input-output behavior of a program fragment. Here, the pair $\langle \mathcal{G}, \mathcal{D}_k^C \rangle$ can be considered as a summarization because $\mathcal{D}_k^C$ describes the behavior in the sense of dependencies of the symbolic state $\mathcal{G}$. Two operations are provided to manipulate the table:

- memoed($\mathcal{G}$): tests if $\Sigma \equiv \langle \mathcal{G}', \mathcal{D}_k^C \rangle$ is in the memo table such that $\mathcal{G}'$ subsumes $\mathcal{G}$. If this is the case, it returns Σ.

- memoize(Σ): records the summarization Σ in the memo table.

The algorithm described in Fig. 6 comprises three other key operations: pre, w̌p, and join.

First, pre defines the variable dependencies of the current state, from the variable dependencies returned by a recursive call to solve.

$$\mathsf{pre}(\mathcal{D}_n^C, t_n(\tilde{x}) \mapsto t_{n'}(\tilde{x}'), \phi(\tilde{x}, \tilde{x}')) \overset{\text{def}}{=}$$
$$\{v \mid v \in \mathcal{D}_n^C, v \notin \mathsf{DEF}(\phi)\} \cup$$
$$\{v \mid s \in \phi, \mathsf{DEF}(s) \cap \mathcal{D}_n^C, \neq \emptyset, v \in \mathsf{USE}(s)\} \cup$$
$$\{v \mid \mathsf{DEF}(\phi) = \emptyset, \text{there is a transition } T \text{ in the slice}$$
$$\text{reachable from } n' \text{ s.t.}$$
$$\mathsf{CINF}(\phi, T), v \in \mathsf{USE}(\phi)\}$$

Given a transition $t_k(\tilde{x}) \mapsto t_{k'}(\tilde{x}'), \phi(\tilde{x}, \tilde{x}')$, and a state $\mathcal{G} \equiv \langle k, \tilde{x}, \Psi \rangle$, then:

$$wp(\mathcal{G}, \phi(\tilde{x}, \tilde{x}')) \overset{\text{def}}{=} \langle k, \tilde{x}, \forall \tilde{x}'.\phi(\tilde{x}, \tilde{x}') \longrightarrow \Psi[\tilde{x}'/\tilde{x}]\rangle$$

In general, it is well known that it is not practical to compute this function $wp()$ precisely. Thus, in practice, we just compute, w̌p, a precondition of the descendant $\mathcal{G}_i$ which is at least as general as the parent $\mathcal{G}$, as opposed to the weakest precondition. Our implementation uses an efficient algorithm based on constraint deletions presented recently in [23]. Finally, the algorithm needs to conjoin the dependencies from the descendants $\mathcal{D}_k^C(i), 1 \leq i \leq n$, at the parent:

$$\mathsf{join}(\mathcal{D}_k^C(1), \cdots, \mathcal{D}_k^C(n)) \overset{\text{def}}{=} \mathcal{D}_k^C(1) \cup \cdots \cup \mathcal{D}_k^C(n)$$

We are now ready to present the algorithm in Fig. 6. The input of the algorithm is a symbolic state $\mathcal{G}$ and the slicing criterion C. The algorithm performs in a post-order fashion a depth-first traversal of the computation tree of the given state $\mathcal{G}$. This process is most naturally implemented recursively. This recursive procedure has three base cases:

- SUBSUMED: the state $\mathcal{G}$ has been already explored with a more general context. That is, there exists another state $\mathcal{G}'$ that subsumes G. The subsumption test is done by memoed($\mathcal{G}$), and if it succeeds, it returns the summarization that contains $\mathcal{G}'$.

- INFEASIBLE: the context, Ψ, carried by $\mathcal{G}$ is unsatisfiable. That is, the path is not executable. The algorithm computes the interpolant, $\Psi' \equiv false$ and it does not add any variable to $\mathcal{D}_n^C$ since no variable can affect the computation of C.

```
solve(G ≡ ⟨k, x̃, Ψ⟩, CTS, C)
  • if  Σ = memoed(G ) then                          SUBSUMED
        return Σ
  • if  Ψ is unsatisfiable then                      INFEASIBLE
        return ⟨⟨k, x̃, false⟩, ∅⟩
  • if  k is final  then                             SUCCESSFUL
        return ⟨⟨k, x̃, true⟩, {C} ⟩
  • otherwise:                                       RECURSIVE
        for each rule i, 1 ≤ i ≤ n : Tᵢ ≡ tₖ(x̃) ↦ tₖ'(x̃'), φᵢ(x̃, x̃')
            ⟨Gᵢ, 𝒟ₖ'^C, (i)⟩ = solve(⟨k', x̃', Ψ ∧ φᵢ⟩, CTS, C)
            Ψ'ᵢ = w̌p(Gᵢ, φᵢ)
            𝒟ₖ^C(i) = pre(𝒟ₖ'^C, (i), Tᵢ)
        endfor
        Ψ' = Ψ'₁ ∧ ··· ∧ Ψ'ₙ
        𝒟ₖ^C = join(𝒟ₖ^C(1), ··· , 𝒟ₖ^C(n))
        memoize(Σ ≡ ⟨⟨k, x̃, Ψ'⟩, 𝒟ₖ^C⟩)
        return Σ
```

Figure 6: Slicing Algorithm for Loop-Free Programs

- SUCCESSFUL: the state $\mathcal{G}$ is successful because a feasible path has been found. Here, the algorithm returns the interpolant *true* and it adds the variable C in the dependencies (i.e., $\mathcal{D}_n^C = \{C\}$).

We now consider the recursive case. The algorithm represented by the recursive procedure solve, given in Fig 6, triggers all applicable transitions to create new descendant states from $\mathcal{G}$. Given the return values (i.e., summarizations $\Sigma_i \equiv \langle \mathcal{G}_i, \mathcal{D}_k^C, (i) \rangle$) of the recursive calls to those descendants, the algorithm:

1. Computes the interpolant, Ψ'_i of the state $\mathcal{G}$ from the child's interpolant $\mathcal{G}_i$ by an operation akin to weakest precondition propagation (w̌p). Then, the final interpolant, Ψ', is simply the *intersection* of the interpolants returned by the previous step.

2. Updates the dependencies among variables, $\mathcal{D}_k^C$, from the child's solution $\mathcal{D}_k^C, (i)$ by applying the backward transfer function (pre), and finally, it conjoins (join) all children solutions $\mathcal{D}_k^C(i)$, obtaining $\mathcal{D}_k^C$.

Finally, we show in Fig. 7 how to generate a sliced program from the summarizations stored by algorithm described in Fig. 6. We consider the procedure initial_goal which given the initial CTS returns an initial state. Note that $\mathcal{D}_k^C$ is part of the summarization and we consider it as a global variable in the algorithm. A transition of the form $t_k(\tilde{x}) \mapsto t_{k'}(\tilde{x}'), \phi(\tilde{x}, \tilde{x}')$ corresponding to an assignment statement is included in the slice if $\mathsf{DEF}(\phi) \cap \mathcal{D}_{k'}^C \neq \emptyset$. Moreover, if the transition corresponds to a conditional statement then it is included in the slice if $\mathsf{USE}(\phi) \cap \mathcal{D}_{k'}^C \neq \emptyset$.

4.1 Adding Representative Path Formulas

This section extends the previous to the case where we desire not just a safe slice but, in some sense, the most precise slice, i.e., the *exact* slice. The algorithm presented in Fig.6 does not in general produce the exact slice for the input $\mathcal{G}$ and criterion C. Recall that the algorithm tests if there exists an entry $\Sigma \equiv \langle \mathcal{G}', \mathcal{D}_n^C \rangle$ in the memo table such that $\mathcal{G}'$ subsumes $\mathcal{G}$. If yes, the algorithm returns Σ. Since $\mathcal{G}'$ subsumes $\mathcal{G}$, the set of states represented by $\mathcal{G}'$ is a superset of those represented by $\mathcal{G}$. Hence, the algorithm in Fig.6 performs an over-approximation which may lead to a loss of precision.

```
slice(CTS, C)
    G = initial_goal(CTS)
    solve(G,CTS,C)
    S = ∅
    for each rule T ≡ t_k(x̃)↦t_{k'}(x̃'),φ(x̃,x̃')
        if T is assignment and DEF(φ)∩𝒟_k^C, ≠ ∅ then
            S = S∪{T}
        if T is conditional and USE(φ)∩𝒟_k^C, ≠ ∅ then
            S = S∪{T}
    endfor
    return S
```

Figure 7: Construction of the Sliced Program

Interestingly, we may only lose precision when a subsumed node inherits the solution from another which was computed in a more general context, and more concretely, considering a smaller number of infeasible paths. This situation was exemplified in Fig. 5, Sec.2. Our wish is to ensure that our algorithm performs exact propagation of the dependencies for loop-free programs. We will show in Sec. 5 how to support loops and still obtain precise slices.

```
solve(G ≡ ⟨k,x̃,Ψ⟩,CTS,C)
  • if Σ ≡ ⟨G',𝒟_k^C⟩ = memoed(G) and
    (∗)  ∀⟨x,Φ_x⟩ ∈ 𝒟_k^C,Ψ∧Φ_x is satisfiable
         return Σ
  • if Ψ is unsatisfiable then
         return ⟨⟨k,x̃,false⟩,∅⟩
  • if k is final then
    (∗)  return ⟨k,x̃,true,{⟨C,true⟩}⟩
  • otherwise:
         for each rule i,1 ≤ i ≤ n : T_i ≡ t_k(x̃)↦t_{k'}(x̃'),φ_i(x̃,x̃')
             ⟨G_i,𝒟_k^C,(i)⟩ = solve(⟨k',x̃',Ψ∧φ_i⟩,CTS,C)
             Ψ'_i = w̌p(G_i,φ_i)
    (∗)      𝒟_k^C(i) = pre(𝒟_k^C,(i),T_i)
         endfor
         Ψ' = Ψ'_1 ∧ ⋯ ∧ Ψ'_n
    (∗)  𝒟_k^C = join(𝒟_k^C(1),⋯,𝒟_k^C(n))
         memoize(Σ ≡ ⟨⟨k,x̃,Ψ'⟩,𝒟_k^C⟩)
         return Σ
```

Figure 8: Exact Slicing Algorithm for Loop-Free Programs

We now refine our algorithm and obtain the algorithm described in Fig. 8. New features are annotated with the symbol (∗). The first change is the definition of whether a symbolic state is subsumed by a previously computed summarization. We redefine $\mathcal{D}_k^C$ as a set of pairs $\langle x, \Phi_x \rangle$, where x is a variable and Φ_x its representative path formula. Then, a node n with context Ψ_n is subsumed by k with interpolant Ψ'_k and dependencies $\mathcal{D}_k^C$ if:

a) $\Psi_n \models \Psi'_k$ (as before), and

b) $\forall \langle x, \Phi_x \rangle \in \mathcal{D}_k^C, \Psi_n \wedge \Phi_x$ is satisfiable.

The major change, however, is the pre step. Essentially, the new definition is similar to the previous but adding the update of the representative path formulas:

$$\mathsf{pre}(\mathcal{D}_{n'}^C, t_n(\tilde{x}) \mapsto t_{n'}(\tilde{x}'), \phi(\tilde{x}, \tilde{x}')) \stackrel{\text{def}}{=}$$
$$MG(\{\langle v, \Phi_v \wedge \phi(\tilde{x}, \tilde{x}') \rangle \mid \langle v, \Phi_v \rangle \in \mathcal{D}_{n'}^C, v \notin \mathsf{DEF}(\phi)\} \cup$$
$$\{\langle v, \Phi_x \wedge \phi(\tilde{x}, \tilde{x}') \rangle \mid s \in \phi, \langle x, \Phi_x \rangle \in \mathcal{D}_{n'}^C, x \in \mathsf{DEF}(s),$$
$$v \in \mathsf{USE}(s)\} \cup$$
$$\{\langle v, true \rangle \mid \mathsf{DEF}(\phi) = \emptyset, \text{there is a transition } T \text{ in the slice}$$
$$\text{reachable from } n' \text{ s.t. } \mathsf{CINF}(\phi, T),$$
$$v \in \mathsf{USE}(\phi)\})$$

In the above the function MG includes only the most general pairs, where $\langle v, \varphi \rangle$ is not included if there is a pair $\langle v, \varphi' \rangle$ in the set such that $\varphi \rightarrow \varphi'$.

In addition to the above redefinition, we need also to modify the join step to consider the representative paths. The idea is that, for each variable, we choose arbitrarily a representative path from the set of candidates.

The following theorem statement says that our algorithm shown in Fig. 8 computes precise slices in the sense that for any statement included in the slice, there must exist a executable path that reaches it.

THEOREM 1 (Exact Slice for Loop-Free Programs). *Let CTS be a constraint transition system without loops, and C the slicing criterion. Let S be the sliced program constructed using the algorithm in Fig. 7 with the procedure* solve *described in Fig. 8 wrt C.*

Then, for every transition $s \in S$ there exists a path p_s containing s that affects C which terminates in a final state, i.e., p_s is feasible.

5. Path-Sensitive Slicing Algorithm with Loops

In Sec. 4.1 we have presented an exact slicing algorithm for loop-free programs. In this section, we show how to extend it in order to compute slices in the presence of loops.

The problem of obtaining an exact slice of a program with loops is undecidable [39]. The standard solution is to compute iteratively the algorithm described in Fig. 8, Sec. 4.1 until the dependencies reach a *fixpoint*. The result of this fixpoint is an over-approximation of the concrete set of states.

Another major issue is during the symbolic execution of loops. Problems can arise whenever the current path formula does not imply either the loop condition or its negation. This is the *loop invariant* problem which requires in general the determination of a *suitable* loop invariant.

The algorithm for slicing programs with loops is described in Fig. 9. This algorithm is similar to the previous but adding special support for loops denoted by the symbol (∗). We first need to add a global stack, $\mathcal{L}$, which allows us to detect when the algorithm has ended up with the body a loop. We assume that given a program point k, we can determine if a loop has been encountered or not. If this is the case, our method first "skips" the loop and it executes starting at the first transition after the loop (i.e., exit transition). The key here is to use the loop invariant I as the context to be propagated after the loop. When the recursive call to solve ends up, the algorithm *pushes* the entry program point into $\mathcal{L}$ and it starts analyzing the loop body. Again, the body loop will only consider I in the context. The procedure solve_fixp computes the dependencies for the loop bodies until a fixpoint is reached. After each fixpoint iteration the memo table is re-initialized to discard all the summarizations generated inside the loop. Moreover, the algorithm needs to restart the context to the original invariant I. When solve_fixp terminates, the algorithm *pops* the entry program point of the loop from $\mathcal{L}$ and it stores the summarization computed.

Loop invariant problem. We consider the fundamental problem of discovering loop invariants beyond the scope of this paper.

```
solve(𝒢 ≡ ⟨k,x̃,Ψ⟩, CTS, C)                    \* L: global stack *\
  • if  Σ ≡ ⟨𝒢′, 𝒟ₖᶜ⟩ = memoed(𝒢) and

        ∀⟨x,Φₓ⟩ ∈ 𝒟ₖᶜ, Ψ ∧ Φₓ is satisfiable
        return Σ

  • if  Ψ is unsatisfiable then
        return ⟨⟨k,x̃,false⟩,∅⟩

  • if  k is final  then
        return ⟨k,x̃,true,{⟨C,true⟩}⟩

  • (∗) if 𝒢 is a loop and k = k′ where k′ the top of L
    (∗)   return ⟨k,x̃,Ψ,∅⟩

  • (∗) if 𝒢 is a loop
    (∗)   let tₖ(x̃)↦t_{k_entry}(x̃_entry), φ_entry(x̃,x̃_entry)
    (∗)   let tₖ(x̃)↦t_{k_exit}(x̃_exit), φ_exit(x̃,x̃_exit)
    (∗)   let I  be the invariant for the loop
    (∗)   let Ψ_entry ≡ I ∧ φ_entry and Ψ_exit ≡ I ∧ φ_exit
    (∗)   let 𝒢_exit ≡ ⟨k_exit,x̃_exit,Ψ_exit⟩
    (∗)   ⟨_, 𝒟ₖᶜ′, ⟩ = solve(𝒢_exit,CTS,C)
    (∗)   let 𝒢_entry ≡ ⟨k_entry,x̃_entry,Ψ_entry⟩
    (∗)   push(L,k)
    (∗)   𝒟ₖᶜ = solve_fixp(𝒢_entry, 𝒟ₖᶜ′, I , CTS,C )
    (∗)   pop(L)
    (∗)   memoize(Σ ≡ ⟨⟨k,x̃,I⟩, 𝒟ₖᶜ⟩)
    (∗)   return Σ

  • otherwise:
        for each rule i, 1 ≤ i ≤ n : Tᵢ ≡ tₖ(x̃)↦t_{k′}(x̃′), φᵢ(x̃,x̃′)
            ⟨𝒢ᵢ,𝒟ₖᶜ′,(i)⟩ = solve(⟨k′,x̃′,Ψ ∧ φᵢ⟩,CTS,C)
            Ψ′ᵢ = w̌p(𝒢ᵢ,φᵢ)
            𝒟ₖᶜ(i) = pre(𝒟ₖᶜ′,(i),Tᵢ)
        endfor
        Ψ′ = Ψ′₁ ∧ ⋯ ∧ Ψ′ₙ
        𝒟ₖᶜ = join(𝒟ₖᶜ(1),⋯,𝒟ₖᶜ(n))
        memoize(Σ ≡ ⟨⟨k,x̃,Ψ′⟩,𝒟ₖᶜ⟩)
        return Σ
(∗) solve_fixp(𝒢 ≡ ⟨k,x̃,Ψ⟩, D_old ᶜ, I , CTS, C)
(∗)     fixpoint = false
(∗)     while (not fixpoint)
(∗)         ⟨_,D_new ᶜ⟩ = solve(𝒢,CTS,C)
(∗)         if (D_new ᶜ == D_old ᶜ)
(∗)             fixpoint = true
(∗)         D_old ᶜ = join(D_old ᶜ,D_new ᶜ)
(∗)         Ψ = I , cleanup_memo
(∗)     endwhile
(∗)     return D_new ᶜ
```

Figure 9: Slicing Algorithm with Loops

However, we propose here a simple method. Given the path formula $Ψ ≡ ψ_1 ∧ ⋯ ∧ ψ_n$ at the entry of the loop L, we keep just those constraints $ψ_i ∧ ⋯ ∧ ψ_j$ corresponding to variables whose values do not change in L[9].

Although this loop invariant method is very simple, it posses two major characteristics:

  • Propagates unrelated constraints along the loops. That is, if a loop does not contribute to a constraint $Ψ$, our method will keep $Ψ$ and propagate it through the rest of the program.

_____
[9] Of course, even though it is purely syntactic in presence of pointers, an alias analysis is required.

  • Propagates new context generated inside loop bodies. It is important to notice that although our algorithm in Fig. 9 needs to discard all new constraints collected in the analysis of the loop body from one fixpoint iteration to another, during a particular iteration those new constraints will be propagated. Note that loop bodies are arbitrarily large (including more nested loops) and hence, more precise slices can be obtained.

Finally, note that all slicing methods need to face this challenge in one way or another. Traditional static slicing (e.g., [39, 19, 33]) assumes the trivial *true* loop invariant. Other approaches that need to propagate some context [38, 13, 5, 34] either unroll loops or need ultimately also the use of a loop invariant.

6. Concluding Remarks

We have presented an algorithm for optimal program slicing. Our method consists of two parts. First, program paths that are infeasible detected by symbolic execution are eliminated from consideration. Thus, we ensure that statements included in the slice are defined in some executable path. Second, we make the approach practical by using interpolation in order to generalize the context of computation trees. To the best of our knowledge, our approach is the first program analysis that uses interpolation in reducing the search space.

Interpolation increases the chance for the reuse of the dependency information from one subtree to another in the program's computation tree. Moreover, we ensure the accuracy of the dependency analysis by testing, upon an attempt to reuse, the feasibility of the representative paths that give rise to the dependency. In programs without loops, we have demonstrated that our approach obtains, in principle, the minimal slice.

In the future, we shall extend our slicing method to define a general program analysis framework in which other program analyzes can compute more accurate information using the infeasible path information.

Limitations. There are some technical reasons why our algorithm may give imprecise results in real programs. The *solver* underlying, which is used for detecting infeasible paths and generating interpolants, may fail to decide whether a formula path is satisfiable or not. Another fundamental issue is the analysis of *loops*. In presence of loops, the accuracy of our approach will depend on the loop invariant. We have shown that simple loop invariants can be computed automatically to obtain reasonably precise slices.

Practicality of the approach. Although no experimental data is reported in this paper, there are some evidences to believe that our approach can be practical for real programs. For instance, in our initial results with our ongoing prototype implementation, we have tested the program statemate with 1238 LOC (from the Mälardalen benchmarks [27]) which is an extreme case because 30% of all paths are infeasible. Recall that if all paths are feasible our approach has a similar performance to other methods. The results were that without interpolation (i.e., naïve symbolic execution), the analysis explores 8629 nodes in the computation tree. Our algorithm only explores 262 nodes. Less specific but still it is worth to mention, interpolation has been a successful technique used, for similar efficiency purposes, in safety verification of systems. In counterexample-guided abstraction refinement approaches [18, 30], interpolants are generated whenever a spurious counterexample has been found in order to refine the predicate abstract domain. In [23], the interpolants are also generated whenever a final path is encountered. The interpolant is produced wrt the safety property while that, in our case, the interpolant is generated wrt *true*. Hence, the efficiency gains in proving safety may be more limited. More importantly, some of these systems have been able to

prove safety properties in programs up to hundreds of thousands of LOC [18].

On the other hand, the use of representative paths may limit the reuse of solutions. However, we believe that it is not likely to be high. Even so, one possible practical solution that seems promising is to use, for each variable, $k > 1$ representative paths, with k constant, in order to augment the likelihood of reusing.

References

[1] Leeann Bent, Darren C. Atkinson, and William G. Griswold. A comparative study of two whole program slicers for c. Technical report, La Jolla, CA, USA, 2001.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *5th TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[3] D.W. Binkley and K.B. Gallagher. Program slicing. *Advances in Computing, Academic Press.*, 43:1–50, 1996.

[4] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. *SIGSOFT Softw. Eng. Notes*, 22(6):361–377, 1997.

[5] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40, no. 11-12:595–607, 1998.

[6] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theo ries. In C. R. Ramakrishnan and J. Rehof, editors, *14th TACAS*, volume 4963 of *LNCS*, pages 397–412. Springer, 2008.

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

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[9] Sebastian Danicic, Chris Fox, and Chris Harman. Consit: A conditioned program slicer. In *In IEEE International Conference on Software Maintenance (ICSM'00*, pages 216–226. IEEE Computer Society Press, 2000.

[10] Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(3), 2001.

[11] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

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

[13] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 379–392, 1995.

[14] C. Flanagan. Automatic software model checking using CLP. In P. Degano, editor, *12th ESOP*, volume 2618 of *LNCS*, pages 189–203. Springer, 2003.

[15] István Forgács and Antonia Bertolino. Feasible test path selection by principal slicing. *SIGSOFT Softw. Eng. Notes*, 22(6):378–394, 1997.

[16] Allen Goldberg, T. C. Wang, and David Zimmerman. Applications of feasible path analysis to program testing. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 80–94, 1994.

[17] D. Hedley and M. A. Hennell. The causes and effects of infeasible paths in computer programs. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 259–266, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan.

[19] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, 1988.

[20] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[21] J. Jaffar, A. E. Santosa, and R. Voicu. Modeling systems in CLP. In M. Gabbrielli and G. Gupta, editors, *21st ICLP*, volume 3668 of *LNCS*, pages 412–413. Springer, 2005.

[22] 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.

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

[24] Ranjit Jhala and Rupak Majumdar. Path slicing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–47, 2005.

[25] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[26] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.

[27] Mälardalen WCET research group benchmarks. URL http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2006.

[28] K. L. McMillan. Interpolation and SAT-based model checking. In *15th CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[29] K. L. McMillan. An interpolating theorem prover. *TCS*, 345(1):101–121, 2005.

[30] K. L. McMillan. Lazy abstraction with interpolants. In T. Ball and R. B. Jones, editors, *18th CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[31] Minh Ngoc Ngo and Hee Beng Kuan Tan. Detecting large number of infeasible paths through recognizing their patterns. In *ESEC-FSE '07*, pages 215–224, 2007.

[32] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, 1984.

[33] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *In Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20. ACM Press, 1994.

[34] Torsten Robschink and Gregor Snelting. Efficient path conditions in dependence graphs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 478–488, 2002.

[35] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[36] Gregor Snelting and Abteilung Softwaretechnologie. Combining slicing and constraint solving for validation of measurement software. In *SAS*, pages 332–348, 1996.

[37] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[38] G. A. Venkatesh. The semantic approach to program slicing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 107–119, 1991.

[39] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449,

Abstractions from proofs. In *31st POPL*, pages 232–244. ACM Press, 2004.

1981.

[40] E.J Weyuker. The applicability of program schema results to programs. *Int. J. Computer and Information Sci., 8, 5*, 1979.

[41] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[42] D. Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. *SIGSOFT Softw. Eng. Notes*, 14(8):48–54, 1989.