A Path-Sensitive Control Flow Graph

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

Control Flow Graph (*CFG*) is a compact representation of all executable paths of a program and it is central to most program analyses. Unfortunately, the direct use of a CFG has two major sources of imprecision: (a) the existence of infeasible paths, and (b) the merging of states along incoming edges of a control-flow merge. Addressing these two problems is the path-sensitivity issue, and it is a folklore that path-sensitive analyses are more accurate.

In this paper, we present a method to systematically restructure a CFG in order to encode path-sensitivity into it. The path-sensitive CFG is generated from the symbolic execution tree by (a) removing infeasible paths, and (b) duplicating subgraphs when it is likely to produce more accurate results by the underlying analyzer. We use constraint solving to test path infeasibility. To produce a graph of manageable size, we systematically generalize the state associated with a tree node while preserving its reason of infeasibility. This increases the likelihood of subsuming other nodes. If a node is subsumed then it is merged together with its subsumer. Otherwise, our method splits the subsumer and the subsumed nodes. The above transformation is done offline and independent to the analysis. We finally present experimental data on real benchmarks that shows the efficiency and effectiveness of the approach.

1. Introduction

Static analysis using *abstract interpretation* [7] is an efficient and elegant way of extracting information about all possible executions of a program. It has been successfully used in compilers to decide whether some optimizations or transformations are applicable, for finding or explaining bugs, and also for proving the absence of bugs.

Given the program's *Control Flow Graph (CFG)* and a property of interest $\mathcal{P}$, a simple way to perform abstract interpretation consists of applying the *abstract transfer function* $T_{\mathcal{P}}$ on each node that represents basic blocks in the CFG. In presence of loops or recursive calls, a fixpoint computation and/or widening is needed. The main advantage of abstract interpretation is *efficiency* which is achieved by eliminating irrelevant details to the property of interest. Due to these efficiency reasons, abstract interpretation-based program analyses that use directly CFGs often incur in two kind of loss of accuracy: (1) consideration of *infeasible paths*, and (2) merging of different abstract states along incoming edges of a *control-flow merge*. Although these over-approximations are often precise enough for reasoning about the property, it is well understood that they may make unable compiler optimizations or arising false positives in the case of testing or verification.

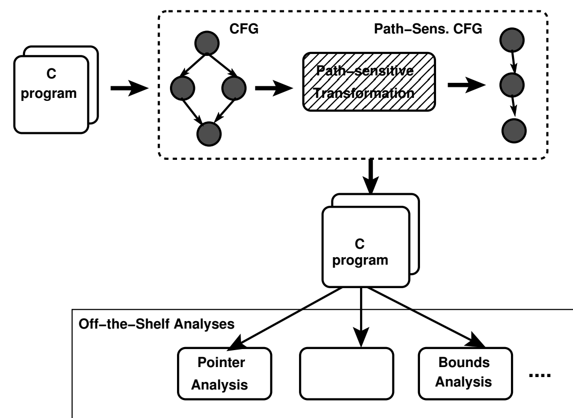

Figure 1. Transformation and analysis pipeline.

In this paper, we present a *systematic* and yet *practical symbolic execution*-based method to restructure the original program's CFG to alleviate those sources of imprecision by eliminating infeasible paths and by splitting explicitly merging points in case a loss of precision was possible during the abstract join operator. The result of this transformation is a CFG:

- encoded with path-sensitivity which might make an arbitrary *path-insensitive* program analysis more precise
- independent to the analysis, and thus, it can be built offline, and
- at the expense of a reasonable increase in the size with respect to the original CFG.

We say systematic here to mean that infeasibility is systematically checked, but within two fundamental limitations. First, loops are analyzed with a certain *loop-invariant* generated automatically by the algorithm in order to make finite the symbolic execution process. Secondly, we tacitly assume that infeasibility testing is really only applicable to predicates for which a reasonable theorem-prover exists. In this paper, we assume a theorem-prover that deals with linear arithmetic formulas over integer variables and array elements.

Our framework is informally schematized in Fig. 1. It is composed of a front-end compiler which takes the C program and produces its corresponding CFG. Next, our proposed transformation will build a path-sensitive CFG whose semantics preserves the semantics of the original CFG. Then, a back-end decompiler produces a new C program from the transformed CFG. Finally, we feed this path-sensitive C program into arbitrary *off-the-shelf* pro-

[Copyright notice will appear here once 'preprint' option is removed.]

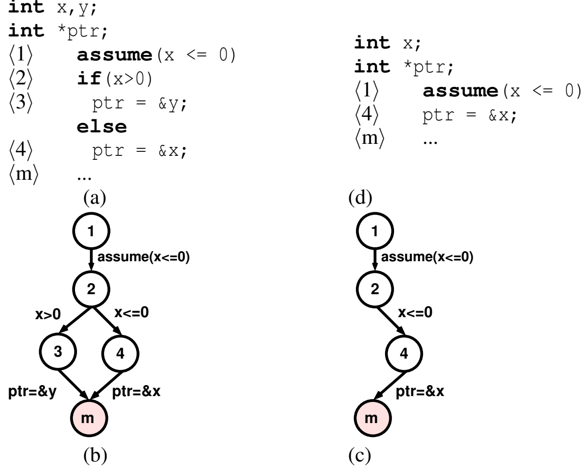

Figure 2. Deletion of Infeasible Paths

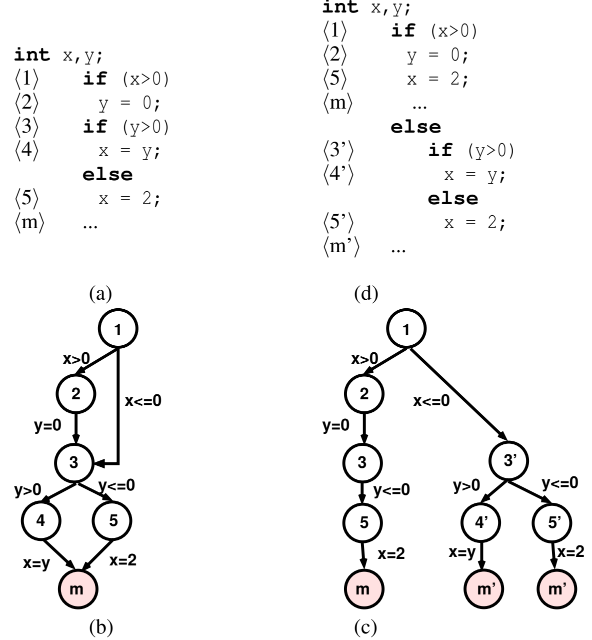

Figure 3. Node Splitting

gram analyses such as pointer analyses, slicers, bound analyzers, etc in the hope of obtaining more accurate results.

Our transformation component (dashed box in Fig. 1) restructures the original CFG as follows. It generates a symbolic execution tree where each path of the tree is a succession of nodes associated with each program point of the original CFG. Moreover, each node contains a conjunction of formulas $\Psi \equiv \psi_1 \wedge \ldots \wedge \psi_i$, symbolically representing a set of states. The edges contain the statements described in the CFG. Therefore, $\psi_1, \ldots, \psi_i$ are constraints generated from each statement in the path from the root to the node i. A main feature of this tree is that multiple paths may contain the same set of program points but with a different set of states, i.e. different *contexts*. Then, our method terminates each path Ψ whenever we are at the endpoint, or when Ψ is unsatisfiable (i.e., infeasible path). In either case, the algorithm records certain information about Ψ and backtracks to the next path.

A key observation is that multiple contexts allow us to transform the original CFG in two ways but still preserving the original semantics without any assumption about the property of interest:

1. Excluding all detected infeasible paths from the symbolic execution tree.
2. Splitting explicitly those merging points each time they are visited under a new context.

The potential benefits from transformation 1 are quite obvious. The elimination of spurious paths may eliminate some imprecision in the execution of program analyses. Consider the code snippet in Fig. 2(a) and its corresponding CFG in Fig. 2(b). Our symbolic traversal explained so far will detect that the edge $2 \rightarrow 3$ is infeasible since the constraints $x \leq 0 \wedge x > 0$ are unsatisfiable. Then, the edge and all its successors up to next merging point ⟨m⟩ can be safely removed. The resulting CFG is shown in Fig. 2(c). Finally, our framework generates another C program shown in Fig. 2(d) as the original but without the infeasible path and eliminating the redundant check of the edge $2 \rightarrow 4$. Then, assume we would like to run an Andersen-like pointer analysis on the program in Fig 2(a). The analysis will report that `ptr`,`&x`, and `&y` may point to the same memory location. However, the same analysis on the transformed program cannot infer that `ptr` and `&y` are aliased producing a more accurate result.

On the other hand, if transformation 2 is applicable then it means that there exists at least one symbolic execution which is possible in one context but not in another. If both contexts were merged then some imprecision might arise. Consider now the contrived program in Fig. 3(a). A depth-first symbolic traversal on the CFG in Fig. 3(b) will infer first the edge $3 \rightarrow 4$ is infeasible (i.e., $x > 0 \wedge y = 0 \wedge y > 0$ unsatisfiable). More importantly, since the node 3 is traversed twice we can split it into two nodes 3 and $3'$ and make a copy of all its successors up to next merging point as illustrated in Fig. 3(c). Finally, we can again obtain a C transformed program that behaves as the original in Fig. 3(d). Now, suppose a backward slicer on program point ⟨5⟩ and variable `x`. On the original program, the slicer will not able to delete any statement. However, the same slicer and same criteria on the transformed program will able to slice away the statement `y=0` at program point ⟨2⟩.

Notice that even if transformations 1 and 2 are applied exhaustively it does not necessarily imply that the accuracy of the underlying program analysis will be improved. Nevertheless, our experimental evaluation in Sec. 6 supports that these two transformations are quite effective in practice and the accuracy of the target program analyses can be significantly improved.

Unfortunately, a simple enumeration of all contexts is exponential. Therefore, the idea of breaking up nodes each time a new context is encountered is not plausible.

The main contribution of this paper is to make practical these two program transformations by using the well-known concept of *interpolation* [8]. Essentially, during the symbolic execution process explained above we weaken or generalize the formula Ψ associated to a path as long as we preserve its unsatisfiability. For instance, assume that the path is infeasible. Then, an interpolant is a formula $\overline{\Psi}$ such that $\Psi \Rightarrow \overline{\Psi}$ and $\overline{\Psi} \Rightarrow$ *false*. This generalization, $\overline{\Psi}$, reduces the likelihood of considering other paths with a less general context since new contexts that still imply the generalized state will be *subsumed*. The key advantage is that the traversal can be stopped and it avoids performing redundant infeasibility detection and node splitting. On the other hand, if the path is feasible then its most general interpolant can be trivially computed (i.e., *true*) since there is no infeasibility to preserve. To operate at any level in the tree, the algorithm must propagate during the backtracking process the

interpolants computed by the children states to ancestors. It is worth mentioning then that our method *interpolates trees* rather than just a path. Hence, the use of interpolants is twofold:

- reduction of the search space by generalizing the states associated with already visited subtrees in the hope that other subtrees with a less general state will be subsumed. This optimization avoids fully enumeration of all contexts.
- consequently, there is no need of applying the node splitting transformation on the subsumed subtrees. This optimization avoids a naive duplication of all nodes with multiple contexts which would produce an exponential size of the transformed CFG.

More specifically, our mechanism to split nodes can be now described as follows. Consider a program point k already visited whose interpolant is Ψ_1 for a context c_1. If k is again visited under a different context c_2 associated with the formula Ψ_2 and the entailment $\Psi_2 \Rightarrow \Psi_1$ holds there is no need to split nodes. Otherwise, we have found a merge node with two different contexts c_1 and c_2 which do not share the same pattern of infeasible paths. That is, the context c_2 contains at least one feasible path which was infeasible under context c_1. In this case, our algorithm will produce a new copy (new node in the CFG) of k and hence, the merge point is split. Thus, our technique either will subsume paths with a less general context in order to avoid full enumeration and unnecessary splitting of nodes or if the path cannot be subsumed then it is a good indicator that we should make a copy of the node where the subsumption did not take place. This is the core of our approach to be scalable and be able to build an effective path-sensitive CFG independent from the property of interest.

The use of subsumption to avoid traversal is *sound* in the sense that the semantics of a subsumed subtree is a subset of the semantics of the original interpolated subtree. Therefore, the exclusion of the subsumed subtrees does not affect the correctness of the program analysis at hand. That is, the underlying program analysis cannot produce incorrect results. However, the opposite does not necessarily hold. That is, the subsumer subtree may contain far more paths than the subtree with a less general context.

The consequence is that it is possible that our technique cannot produce a more precise CFG even in cases where a customized path-sensitive analysis might produce a more precise result on the original CFG. It is important to understand that our approach aims at *assisting* program analyses to produce better results. However, and more importantly, it is clearly not our objective to compete directly with those analyses.

Organization. The rest of this paper is organized as follows. Section 2 describes the most relevant works to our approach. In Section 3 we provide an informal overview of our approach showing several examples, and in Section 4 we introduce relevant concepts and definitions required for the rest of the paper. Section 5 presents our interpolation-based algorithm which takes a rule-based transition system describing a CFG and produces another transition system encoded with path-sensitivity. Section 6 shows the effectiveness and practicality of our approach with a set of C real benchmarks and several off-the-shelf program analyses, and finally, Section 7 concludes.

2. Related Work

The significance of infeasible paths is well understood in many software engineering fields. Since the general problem of detecting infeasible paths is undecidable [25], most approaches attempt to solve this problem focused on a particular analysis and/or performing an unsystematic checking of infeasible paths.

For instance, Bodik et al. [3–5] describe several dataflow analyses improved by detecting infeasible paths through branch correlation. Suhendra et al. [23] present an improved WCET analysis by eliminating infeasible paths detected using conflict sets. Gutzmann et al. [11] present a path-sensitive points-to analysis but the sensitivity is limited to eliminate paths that are infeasible for all contexts. In all these cases, the detection of infeasible paths is partial. Moreover, it might not be straightforward to adapt those analyses in order to provide path-sensitivity to an arbitrary program analysis like ours. Snelting et al. [20] describe a backward slicer that refines a sliced program by eliminating dependencies between nodes that are defined on non-executable paths. They use *interval analysis* and *BDDs* to overcome the potential combinatorial explosion. Recently, another approach using pattern recognition [19] has been presented. Infeasible paths are partially detected by searching for common code patterns.

In Fischer et al. [10] authors use the path-sensitiveness inherent in CEGAR (*CounterExample Guided Abstration Refinement*) to improve precision of dataflow analyses. The fundamental difference is that CEGAR needs a target in order to find counterexamples. However, in many program analyses such as points-to, dependencies, etc. the information has to be inferred at any program point and hence, there is no a particular target. Therefore, the approach is limited to verification and not general analysis.

The closest work to ours has been recently presented in [24]. Here, Thakur and Govindarajan propose a method to build a path-sensitive CFG, but there are two essential differences with our approach: the path-sensitiveness is partial (e.g., based on computing conflict sets), and more importantly, they need to first run a particular dataflow analysis to find out if a merging point is to be split or not, therefore it is dependent on target analysis.

Finally, the interpolation symbolic-execution method employed in this paper has been similarly applied on a search tree of a CLP goal in pursuit of a target property in [17], and also focused on a finite domain for an optimization problem in [16]. In this paper, there are two major differences. Firstly, we generalize the proof tree produced implicitly in [17] to a graph which constitutes our path-sensitive CFG. Secondly, and more importantly, we generalize the application beyond verification.

3. The Basic Idea

We illustrate the principles underlying our method which make it possible to transform the original C program in Fig. 4(a) into the C program in Fig. 5(b), and enhance the likelihood that subtrees with less general state can be avoided, making our approach practical.

The naïve symbolic execution of the original program is shown in Fig. 4(b). The representation is a directed graph, with nodes labeled as program points and edges between two locations labeled by the instruction that executes when control moves from the source to the destination. On the other hand, *infeasible* transitions are represented with a (red) cross on the corresponding edge. In Fig. 5(a), we show a smaller symbolic execution tree computed by our method that potentially avoids the exponential behavior of the naïve approach even though it detects infeasible paths and splits nodes in order to finally produce a C program that increases the accuracy of analyses.

Interpolation-based Symbolic Execution. Let us focus on Fig. 5(a). Our symbolic execution-based method starts traversing the graph in a depth-first manner, $\langle 0\rangle$-$\langle 1\rangle$-$\langle 3\rangle$-$\langle 4\rangle$-$\langle 5\rangle$ reaching the node 6:1 with the formula $\Psi_{6:1} \equiv p > 0 \wedge z = 2 \wedge x > 0 \wedge y = 0 \wedge y > 0$. Since the formula is unsatisfiable, we have found an infeasible path. Thus, our algorithm stops traversing the path and generates an *interpolant*. Given two formulas Ψ and Φ, an interpolant is a formula $\overline{\Psi}$ whose variables are variables of both Ψ and Φ, and Ψ

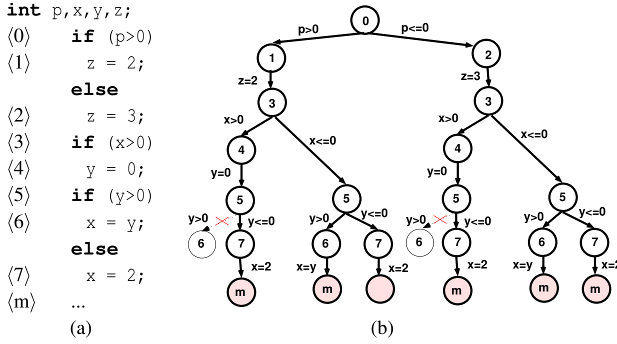

Figure 4. A Program and Its Naïve Symbolic Tree

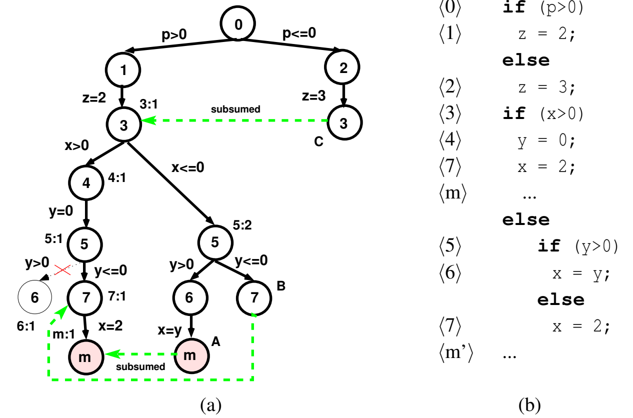

Figure 5. Interpolation-Based Execution Tree of Program in Fig. 4(a) and Output C Program

$\Rightarrow \overline{\Psi}$ 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 $\overline{\Psi}_{6:1} \Rightarrow$ *false*. Here, the only possible interpolant is *false*. Since 6:1 is infeasible, the algorithm backtracks to 5:1 (which has the formula $\Psi_{5:1} \equiv p > 0 \wedge z = 2 \wedge x > 0 \wedge y = 0$) while generating another interpolant $\overline{\Psi}^1_{5:1}$ for 5:1 such that $\Psi_{5:1} \Rightarrow \overline{\Psi}^1_{5:1}$ and $\overline{\Psi}^1_{5:1} \Rightarrow (y > 0 \Rightarrow \overline{\Psi}_{6:1})$. A suitable choice for $\overline{\Psi}^1_{5:1}$ is $y \leq 0$ which is exactly the *weakest liberal precondition* (*wlp*) [9] of the guard $y > 0$ wrt . the postcondition $\overline{\Psi}_{6:1} \equiv$ *false*. We note that the wlp is the most general interpolant possible. Ideally, the weaker the interpolant the better as more nodes can be subsumed by a weaker condition potentially resulting in a smaller graph. In practice, however, for efficiency we adopt a less general interpolant (e.g., the *constraint deletion* or *slackening* technique in [17]).

Next the algorithm reaches the terminal node m:1 through the prefix path ⟨0⟩-⟨1⟩-⟨3⟩-⟨4⟩-⟨5⟩-⟨7⟩, and with formula $\Psi_{m:1}$ being the conjunction of constraints along the path. Again, the algorithm produces an interpolant. The interpolant $\overline{\Psi}_{m:1}$ here satisfies $\Psi_{m:1} \Rightarrow \overline{\Psi}_{m:1}$ and $\overline{\Psi}_{m:1} \Rightarrow$ *true*. A suitable formula for $\overline{\Psi}_{m:1}$ here is *true*, since as mentioned above, the weaker the interpolant the better. By again interpolating (computing the wlp) through the path we obtain the interpolants $\overline{\Psi}_{7:1} \equiv$ *true* and $\overline{\Psi}^2_{5:1} \equiv$ *true*, respectively at nodes 7:1 and 5:1. The final abstraction of 5:1 is the conjunction $\overline{\Psi}^1_{5:1} \wedge \overline{\Psi}^2_{5:1}$, hereby named $\overline{\Psi}_{5:1}$, which is $y \leq 0$. We note here that the objective of applying conjunction is to ensure that all the infeasibilities found in the subtree of the node are preserved. In our example, the only infeasibility found is the one at 6:1. Any execution satisfying $\overline{\Psi}_{5:1}$ does not visit 6:1.

Propagating the interpolant backward from 5:1 to 4:1 results in $\overline{\Psi}_{4:1} \equiv$ *true*, as $\overline{\Psi}_{4:1}$ satisfies $(y = 0 \Rightarrow \overline{\Psi}_{5:1})$, which is *true*. Continuing we obtain *true* as $\overline{\Psi}^1_{3:1}$, the first candidate abstraction at 3:1.

The first opportunity to subsume a subtree appears during the traversal of the path ⟨0⟩-⟨1⟩-⟨3⟩-⟨5⟩, as the program points of 5:2 and 5:1 are the same. The formula $\Psi_{5:2}$ is $p > 0 \wedge z = 2 \wedge x \leq 0$. Then, our algorithm tests if $\Psi_{5:2} \Rightarrow \overline{\Psi}_{5:1}$. Unfortunately, the entailment does not hold. Therefore, our algorithm must traverse the subtree of 5:2. In our example, as is often the case, this traversal is necessary as at 5:2, there exists a path with suffix ⟨5⟩-⟨6⟩-⟨m⟩ which does not exist in 5:1. From this path, an analysis might discover new information. Moreover, this a good indicator for splitting the node ⟨5⟩ into two copies: 5:1 and 5:2 in our path-sensitive CFG.

Nevertheless, there are nodes below 5:2 which can be subsumed. We denote subsumed trees by (green) dotted edges and the label "subsumed". Here we elaborate the subsumption of the node labeled with B. When B is visited, node 7:1 with the same program point is already memoed with $\overline{\Psi}_{7:1} \equiv$ *true* as its abstracted symbolic state. Here Ψ_B (the conjunction of the constraints along the path to B) satisfies *true* and B is therefore subsumed by 7:1. Here B inherits the abstracted symbolic state of 7:1, which is *true*. In our path-sensitive CFG, subsumed nodes are joined (i.e., no split) with the subsumer, and therefore B also inherits the identifier 7:1. The state of A is similarly abstracted to *true* due to subsumption by m:1.

Both the candidate abstractions $\overline{\Psi}^1_{5:2}$ and $\overline{\Psi}^2_{5:2}$ propagated from the children of 5:2 are *true*, and therefore the abstracted state at 5:2 is $\overline{\Psi}_{5:2} \equiv$ *true*. Propagating this interpolant backward results in second candidate abstraction at 3:1 be $\overline{\Psi}^2_{3:1} \equiv$ *true* and therefore $\overline{\Psi}_{3:1} \equiv \overline{\Psi}^1_{3:1} \wedge \overline{\Psi}^2_{3:1} \equiv$ *true*.

We next discuss C, reached through the prefix path ⟨0⟩-⟨2⟩-⟨3⟩. The formula $\Psi_C \equiv p \leq 0 \wedge z = 3$. Since Ψ_C entails $\overline{\Psi}_{3:1}$ (i.e., C is subsumed by 3:1), we do not need to traverse its subtree. Such subsumptions potentially result in exponentially smaller graph wrt the original symbolic execution tree.

Loops. We explain now how our interpolation-based symbolic execution method handles loops. Here 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. 6(a) and our symbolic execution tree in Fig. 6(b). When node A is reached, we attempt to compute a set of abstractions of the state at node 1:1 that are invariant for the looping path ⟨0⟩-⟨1⟩-⟨2⟩-⟨3⟩-⟨5⟩-⟨1⟩. 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 ⟨1⟩-⟨2⟩-⟨3⟩-⟨5⟩-⟨1⟩. For example, $i = 0$ is not individually invariant, as the value of the variable `i` at A is 1, while $d = 1$ is individually invariant, as the value of `d` is 1 at A. The sought after set is therefore $\{c = 0, d = 1\}$, named $S_{1:1}$. When interpreted as conjunction, any subset of this set is invariant through the cycle from 1:1 to A. The strongest invariant here is $c = 0 \wedge d = 1$. We then stop the traversal at A and compute interpolants backward given $c = 0 \wedge d = 1$ at A.

We next traverse the else branch of the if conditional after applying the invariant computed to generalize $\Psi_{1:1}$ in this traversal. The branch is satisfiable under this abstraction, and we reach B with the formula $\Psi_B \equiv c = 0 \wedge d = 1 \wedge i < 10 \wedge a \geq 0 \wedge c' = 1 \wedge i' = i + 1$ (here we denote different variable versions using primes). As the value of `c` is 1 at B, 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

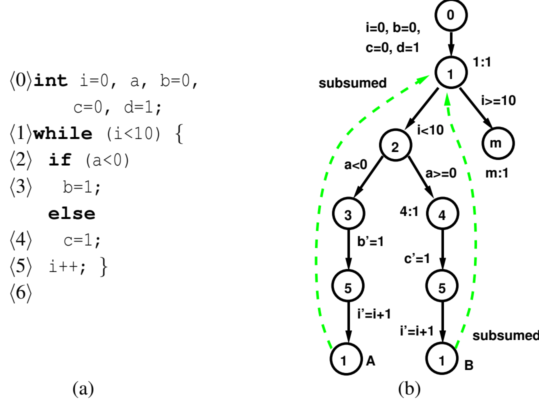

Figure 6. Loop Program and Its Interpolation-Based Execution Graph

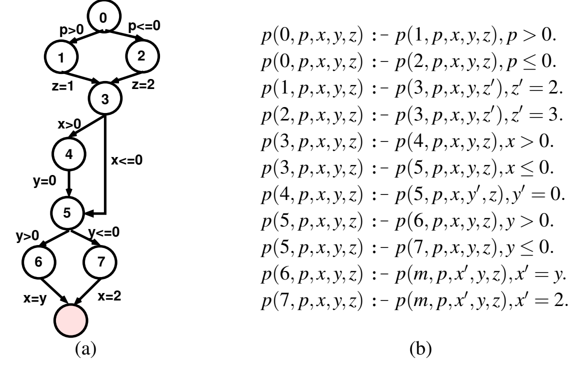

Figure 7. CFG and CLP Model for Program in Fig. 4(a)

if any removed, the remaining constraints are still invariant through the first cycle. In this way, at B we are sure 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 m:1 with the formula $\Psi_{\mathsf{m:1}} \equiv d=1 \wedge i \geq 10$.

Note that we always prioritize analyzing the loop body to obtain an invariant before executing the exit path. This requires an assumption that the program loops are *structured*, that is, each loop has only one entry and one exit point.

Building a Path-Sensitive CFG. Our algorithm, in general, produces a graph due to the existence of ancestor-descendant subsumptions (i.e., loops). The path-sensitive CFG can be produced in a straightforward manner from the interpolation-based symbolic execution graph by: eliminating infeasible nodes (e.g., node 6:1 in Fig. 5(a)), joining subsumed nodes to their subsumer nodes (e.g., A, B, and C in Fig. 5(a)), and finally, splitting nodes where subsumption did not hold (e.g., 5:1 and 5:2 in Fig. 5(a)). As post-processing, given any *if-then-else* statement in the original program with one of the branches always infeasible in our symbolic execution tree, we can replace the statement with just the body of the branch eliminating the redundant test.

Finally, for convenience, it may be desirable to produce a final C program from our path-sensitive CFG. We implemented such a translation for our experiments in Sec. 6. In this paper, we do not detail this translation as it is straightforward. The resulting C program for our example is shown in Fig. 5(b).

4. Preliminaries

We model a C program as a Control Flow Graph (CFG) and use the framework of *Constraint Logic Programming* (*CLP*) [14] to formalize a Control Flow Graph as a set of CLP rules. We then provide the operational semantics of a CLP program as the process of constructing a derivation tree.

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 tuple of terms.

A *rule* is of the form A :- $\tilde{B}, \phi$ 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. A *goal* G has exactly the same format as the body of a rule.

A *substitution* simultaneously replaces each variable in a term or constraint into some expression. We specify a substitution by the notation $[\tilde{E}/\tilde{X}]$, where $\tilde{X}$ is a sequence $X_1,\ldots,X_n$ of variables and $\tilde{E}$ a list $E_1,\ldots,E_n$ of expressions, such that X_i is replaced by E_i for all $1 \leq i \leq n$. Given a substitution θ, we write as $E\theta$ the application of the substitution to an expression E. A *renaming* is a substitution which maps variables into variables. A *grounding* is a substitution which maps each variable into a value in its domain.

Given a goal $\mathcal{G} \equiv p(k,\tilde{X}), \Psi(\tilde{X})$, $[\![\mathcal{G}]\!]$ is the set of the groundings θ of the primary variables $\tilde{X}$ such that $\tilde{\exists}\Psi(\tilde{X})\theta$ holds. We say that a goal $\overline{\mathcal{G}} \equiv p(k,\tilde{X}), \overline{\Psi}(\tilde{X})$ *subsumes* another goal $\mathcal{G} \equiv p(k',\tilde{X}'), \Psi(\tilde{X}')$ if $k=k'$ and $[\![\overline{\mathcal{G}}]\!] \supseteq [\![\mathcal{G}]\!]$. 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. We say that a sequence is *subsumed* if its last goal is subsumed by another goal in the sequence.

Let $\mathcal{G} \equiv (B_1,\cdots,B_n,\phi)$ and P denote a goal and program respectively. Let $R \equiv A$:- $C_1,\cdots,C_m,\phi_1$ denote a rule in P, written so that none of its variables appear in $\mathcal{G}$. Let $A=B$, where A and B are atoms, be shorthand for equations between their corresponding arguments. A *reduct* of $\mathcal{G}$ using R is of the form

$$(B_1,\cdots,B_{i-1},C_1,\cdots,C_m,B_{i+1},\cdots,B_n,B_i = A \wedge \phi \wedge \phi_1) \quad (1)$$

provided $B_i = A$.

A *derivation sequence* or *path* is a possibly infinite sequence of goals $\mathcal{G}_0, \mathcal{G}_1, \cdots$ where $\mathcal{G}_i, i>0$ is a reduct of $\mathcal{G}_{i-1}$. Given a sequence τ defined to be $\mathcal{G}_0,\mathcal{G}_1,\ldots,\mathcal{G}_n$, then $cons(\tau)$ is all the constraints of the goal $\mathcal{G}_n$. We say that a sequence is *feasible* if $cons(\tau)$ is satisfiable, and *infeasible* otherwise.

A *ground* derivation sequence is obtained from a derivation sequence by instantiating all of its variables into the domain of discourse in such a way that the constraints are evaluated to *true*.

DEFINITION 1 (Ground Traces). *Let* TRACES*($\mathcal{P}$, $\mathcal{G}$) be the set of all ground derivation sequences of the program $\mathcal{P}$ using the initial goal $\mathcal{G}$.*

A *derivation tree* for a goal, $\mathcal{G}$, has as branches all derivation sequences emanating from $\mathcal{G}$. We say a derivation tree is *closed* if all its leaf goals are either successful, infeasible, or subsumed by some other goal in the tree. A goal $\mathcal{G} \equiv p(k,\tilde{X}), \Psi(\tilde{X})$ is called *looping* if it is derived from another goal with the same k (called its *looping parent*) through one or more reduction steps.

We now formalize a Control Flow Graph (CFG) as a CLP program. Hence, here and in the rest of the paper, CFG and CLP program are synonymous. We define a CFG in a slightly different way than usual[1]. A CFG can be described as a pair $(\mathcal{N},\mathcal{E})$ where $\mathcal{N}$ is the set of *nodes* and $\mathcal{E}$ is the set of *edges*. A node denotes

[1] In a textbook definition of a CFG, nodes represent basic blocks and edges are used to represent jumps in the control flow.

$p(0,p,x,y,z)$:- $p(1,p,x,y,z), p>0.$
$p(0,p,x,y,z)$:- $p(2,p,x,y,z), p\leq 0.$
$p(1,p,x,y,z)$:- $p(3,p,x,y,z'), z'=2.$
$p(2,p,x,y,z)$:- $p(3',p,x,y,z'), z'=3.$
$p(3,p,x,y,z)$:- $p(4,p,x,y,z), x>0.$
$p(3,p,x,y,z)$:- $p(5',p,x,y,z), x\leq 0.$
$p(4,p,x,y,z)$:- $p(5,p,x,y',z), y'=0.$
$p(5,p,x,y,z)$:- $p(7,p,x,y,z), y\leq 0.$
$p(7,p,x,y,z)$:- $p(m,p,x',y,z), x'=2.$
$p(5',p,x,y,z)$:- $p(6,p,x,y,z), y>0.$
$p(5',p,x,y,z)$:- $p(7',p,x,y,z), y\leq 0.$
$p(6,p,x,y,z)$:- $p(m',p,x',y,z), x'=y.$
$p(3',p,x,y,z)$:- $p(3,p,x,y,z).$
$p(7',p,x,y,z)$:- $p(7,p,x,y,z).$
$p(m',p,x,y,z)$:- $p(m,p,x,y,z).$

Figure 8. Path-Sensitive Version of CLP Program in Fig. 7(b)

program points and edges denote both sequence of straight-line statements and jump conditions in the control flow. Our notion of CFG has as usual two designated blocks: *entry* and *exit*.

The translation from a CFG to CLP is straightforward. Let $\mathcal{G}=(\mathcal{N},\mathcal{E})$ be a graph, and let $e\equiv(\mathsf{pc}_i,\mathsf{pc}_j)\in\mathcal{E}$ be an edge such that $\mathsf{pc}_i,\mathsf{pc}_j\in\mathcal{N}$ and e is labeled with the statement or jump condition $\chi(\tilde{X},\tilde{X}')$ where $\tilde{X}$ and $\tilde{X}'$ represent the state of the program variables before and after the execution, respectively. Then an equivalent CLP rule is defined as follows:

$$p(\mathsf{pc}_i,\tilde{X}) \text{ :- } p(\mathsf{pc}_j,\tilde{X}'),\chi(\tilde{X},\tilde{X}').$$

An example of a CFG and its respective CLP program is given, respectively, in Fig. 7(a) and Fig. 7(b) for the C fragment described in Fig. 4(a). The details of how to generally perform a compilation of an underlying C program into CLP are beyond the scope of this paper. We refer readers to [17] and its references for more details. In this paper, we assume a sound compiler from C to CLP rules.

5. Algorithm

In this section, we present an algorithm which takes a program represented as a set CLP rules $\mathcal{P}$ (as program in Fig. 7(b)) and produces another CLP program $\mathcal{P}'$ (as program in Fig. 8). $\mathcal{P}'$ preserves all execution traces of $\mathcal{P}$, and is obtained from the execution tree by:

1. excluding all detected infeasible paths, and
2. explicitly splitting merging points in $\mathcal{P}$ if further execution differs wrt infeasible paths.

Symbolic states are goals of the form $\langle\mathsf{pc}:\mathsf{k},\tilde{X},\Psi(\tilde{X})\rangle$ where pc is a program point and k is a *context identifier*. We reserve pc_{init} and pc_{end} to denote the initial and the last program points, respectively. $\tilde{X}$ is a list of variables representing the variables of the underlying program, and $\Psi(\tilde{X})$ is a sequence of constraints.

We define the function $\mathsf{Out}_{\mathcal{P}}(\mathcal{G})$ to obtain the reducts of $\mathcal{G}$ in one symbolic step in program $\mathcal{P}$ using Eq. 1,Sec. 4. We denote by $\mathsf{Out}^{+}_{\mathcal{P}}(\mathcal{G})$ the set:

$$\{\mathcal{G}' \mid cons(\mathcal{G}') \text{ is satisfiable } \wedge \mathcal{G}'=\mathsf{Out}_{\mathcal{P}}(\mathcal{G})\}$$

Similarly, we denote by $\mathsf{Out}^{-}_{\mathcal{P}}(\mathcal{G})$ the set:

$$\{\mathcal{G}' \mid cons(\mathcal{G}') \text{ is unsatisfiable } \wedge \mathcal{G}'=\mathsf{Out}_{\mathcal{P}}(\mathcal{G})\}$$

In essence, $\mathsf{Out}^{+}_{\mathcal{P}}(\mathcal{G})$ ($\mathsf{Out}^{-}_{\mathcal{P}}(\mathcal{G})$) is the set of reducts of $\mathcal{G}$ such that, using the same rules, the reduction of $\mathsf{Out}_{\mathcal{P}}(\mathcal{G})$ is feasible (infeasible). In the algorithm, we prioritize recursive call with the arguments in $\mathsf{Out}^{+}_{\mathcal{P}}(\mathcal{G})$ over $\mathsf{Out}^{-}_{\mathcal{P}}(\mathcal{G})$ to use the abstraction produced by traversing $\mathsf{Out}^{+}_{\mathcal{P}}(\mathcal{G})$ that may make $\mathsf{Out}^{-}_{\mathcal{P}}(\mathcal{G})$ feasible. Executing infeasible transition earlier may result in less general abstraction and therefore less subsumption.

The algorithm described in Fig. 9 maintains globally a *memo table* $\mathcal{M}_{\mathcal{T}}$ that stores goals (states) of the form $\langle\mathsf{pc}:k,\tilde{X},\overline{\Psi}(\tilde{X})\rangle$ where $\overline{\Psi}(\tilde{X})$ is an interpolant that generalizes the state at program pc. It also maintains a set of edges $\mathcal{E}$ that induces the path-sensitive graph produced by the algorithm. Initially, $\mathcal{M}_{\mathcal{T}}$ and $\mathcal{E}$ are empty.

The procedure SymbolicExec, in Fig. 9, takes as inputs an initial state and an empty table called $\mathcal{C}_{\mathcal{T}}$. The purpose of $\mathcal{C}_{\mathcal{T}}$ is to record all potential looping parents of a given program point. The basis of SymbolicExec is to run $\mathcal{P}$ symbolically starting with the initial goal and calculating constraints on the program variable values, while subsuming unnecessary subtrees to avoid exponential explosion. For convenience, we define the procedure in a recursive manner. At any node with state $\mathcal{G}$, the procedure performs one symbolic step by triggering all applicable rules, $\mathcal{G}'$, and calling itself recursively in Line 18. Moreover, an edge from $\mathcal{G}$ to $\mathcal{G}'$ is added into the graph (Line 17). Since, the main purpose is to attempt subsumption at any node, each recursive call must return the child's interpolant, $\overline{\Psi}'(\tilde{X}')$. Note that SymbolicExec returns a pair rather than only the child's interpolant. The second component is the set of generalized states of looping parents (in general there are multiple states due to cascading loops), which is to be used to generalize the symbolic traversal of sibling paths. We detail the explanation later in this section.

The symbolic execution on $\mathcal{P}$ terminates normally due to three different reasons.

In the first, SUCCESS, the execution reaches the end of the path, pc_{end}. Since the path is satisfiable the interpolant can be fully generalized to *true* due to the lack of infeasibility.

The next case, SUBSUMED, 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, there is no need to continue with the execution. This is the core step to make our symbolic execution method practical. Since the set of states at the program point $\mathsf{pc}:\mathsf{k}$ is a *subset* of the states stored previously at $\mathsf{pc}:\mathsf{k}_{\mathcal{S}}$, the algorithm can ensure that the subsumed node cannot derive a symbolic path which was not traversed already under the context $\mathsf{k}_{\mathcal{S}}$. On the other hand, it is worth mentioning that the subsumption test is more likely to hold because the algorithm does not store simply the state $\Psi_{\mathcal{S}}(\tilde{X})$ at $\mathsf{pc}:\mathsf{k}_{\mathcal{S}}$. Instead, it stores a generalization of it, the interpolant $\overline{\Psi}(\tilde{X})$, which increases significantly the likelihood of subsumption. Furthermore, $\mathcal{E}$ must be updated by adding a *sibling-to-sibling* edge from the current state $\mathsf{pc}:\mathsf{k}$ to the subsumer $\mathsf{pc}:\mathsf{k}_{\mathcal{S}}$.

INFEASIBLE is triggered when the constraints along the path are unsatisfiable. To simplify the discussion, we assume a theorem prover that decides, say, the theory of linear arithmetic formulas over integer variables and array elements. Here, the execution also stops and returns the formula *false* as interpolant.

Another key operation of the algorithm is how to propagate back the interpolants from the descendants to their ancestors. The parent $\mathcal{G}$ receives all interpolants returned by its children $\mathcal{G}'$ through the recursive calls to SymbolicExec (Line 18). The idea is to compute a formula $\overline{\Psi}(\tilde{X})$ that generalizes $\Psi(\tilde{X})$ and still preserves the path infeasibility. Ideally, we would like to compute the *weakest precondition wrt* the constraints attached to the transition at hand and the interpolant provided by the descendants. In practice, we can derive $\overline{\Psi}(\tilde{X})$ efficiently (in polynomial time) using a less general interpolant provided by the greedy *constraint deletion* and *slackening* techniques in [17]. The function INTERP, used in Line 19, will compute the interpolant. The final interpolant for $\mathcal{G}$ will be the conjunction, also in Line 19, of all interpolants inferred for each $\mathcal{G}'$.

The above procedure needs to deal with loops in order to make the symbolic execution process finite resulting in finite graph. Our algorithm builds *on-the-fly* a loop invariant for each loop before visiting the transitions corresponding to the exit condition of the loop. For a given loop, we compute a path-based loop invariant in order to force *parent-child* subsumption. Since we would like to detect

```
ProgramTransform(P)
    INPUT   : program P expressed as CLP rules
    OUTPUT: program P' expressed as CLP rules
    // M_T: global memo table for performing subsumption
    // C_T:  table for identifying looping parents
    // E:    global set of edges (graph)
    M_T ← ∅, C_T ← ∅, E ← ∅
    SymbolicExec(P,⟨pc_init : 1, X̃, true⟩, C_T)
    P' ← GenPathSensProg(P,E)
```

```
SymbolicExec(P,G,C_T)
1:  switch(G : ⟨pc : k, X̃, Ψ(X̃)⟩)
SUCCESS
2:     case pc = pc_end:
3:        return ⟨true, ⊥⟩
SUBSUMED
4:     case ∃ ⟨pc : k_S, X̃, Ψ̄(X̃)⟩ ∈ M_T and Ψ(X̃) ⇒ Ψ̄(X̃):
5:        E ← E ∪ {(pc : k → pc : k_S)}
6:        return ⟨Ψ̄(X̃), ⊥⟩
INFEASIBLE
7:     case Ψ(X̃) is unsatisfiable:
8:        return ⟨false, ⊥⟩
LOOPING
9:     case ∃ G_P ≡ ⟨pc : k_P, X̃_P, Ψ_P(X̃_P)⟩ ∈ C_T:
10:       Φ(X̃) ← INVARIANT(Ψ_P(X̃_P), Ψ(X̃))
11:       E ← E ∪ {(pc : k → pc : k_P)}
12:       return ⟨Φ(X̃), ⟨pc : k_P, X̃_P, Φ(X̃_P)⟩⟩
RECURSIVE
13:  default:
14:       Ψ̄(X̃) ← true
15:       G_c ← G
16:       foreach G' ≡ ⟨pc' : k', X̃', Ψ(X̃)⟩ in Out⁺_P(G_c), Out⁻_P(G_c) do
17:           E ← E ∪ {(pc : k → pc' : k')}
18:           ⟨Ψ̄'(X̃'), G_P⟩ ← SymbolicExec(P, G', C_T ∪ {G})
19:           Ψ̄(X̃) ← Ψ̄(X̃) ∧ INTERP(G', Ψ̄'(X̃'))
20:           if (G_P ≠ ⊥) then G_c ← APPLYINVARIANT(G_c, G_P)
21:  M_T ← M_T ∪ {⟨pc : k, X̃, Ψ̄(X̃)⟩}
22:  if (G_P = ⟨pc : k_P, _, _⟩ ∧ k_P ≠ k) then return ⟨Ψ̄(X̃), G_P⟩
23:  else return ⟨Ψ̄(X̃), ⊥⟩
```

Figure 9. Algorithm

as many infeasible path as possible, the algorithm attempts to derive the *strongest loop invariant* possible. For correctness, our symbolic traversal must consider all paths under a *global* loop invariant. By "global" we mean here the final abstraction after all path-based loop invariants were computed. Therefore, SymbolicExec must return also each path-based loop invariant (in second component) which will *weaken* the state for further paths. More importantly, after a loop is analyzed the algorithm ensures that no feasible path (inside or outside the loop) considering the generalized state given by the loop invariant is detected as infeasible.

The LOOPING case forces parent-child subsumption to make the analysis of loops finite. Assume $\mathcal{G}_P \equiv \langle \mathsf{pc} : \mathsf{k}_P, \tilde{X}_P, \Psi_P(\tilde{X}_P)\rangle$ is a looping ancestor of $\mathcal{G} \equiv \langle \mathsf{pc} : \mathsf{k}, \tilde{X}, \Psi(\tilde{X})\rangle$ where $\mathcal{G}$ was derived from $\mathcal{G}_P$ via multiple reduction steps. Note the formula $\Psi(\tilde{X})$ is of the form $\Psi_P(\tilde{X}_P) \wedge \Psi'(\tilde{X}')$ and $\tilde{X} \equiv \tilde{X}_P \,.\, \tilde{X}'$ where $\Psi'(\tilde{X}')$ is the formula that contains all constraints collected through the loop path. Then the procedure INVARIANT derives, using a theorem prover, those constraints at $\mathsf{pc} : \mathsf{k}_P$ which remain invariant through the path. The constraints are obtained from the set of atomic constraints in $\exists var(\Psi_P) - \tilde{X}_P : \Psi_P$ (projection of Ψ_P onto the variables $\tilde{X}_P$, which

```
GenPathSensProg(P, E)
    INPUT   : the original CLP program P and a set of edges E
    OUTPUT: program P' expressed as CLP rules
1:  P' ← ∅
2:  foreach (pc_1 : k_1, pc_2 : k_2) ∈ E do
3:     r ← getCLPRule(P, pc_1, pc_2)
4:     pc_a ← genUniquePC(pc_1 : k_1)
5:     pc_b ← genUniquePC(pc_2 : k_2)
6:     if r is a guard and |Outgoing_E(pc_1 : k_1)| ≤ 1
7:        P' ← P' ∪ {genEmptyCLPRule(r, pc_a, pc_b)}
8:     else
9:        P' ← P' ∪ {genCLPRule(r, pc_a, pc_b)}
10:return P'
```

Figure 10. Algorithm (Continuation)

still is a correct symbolic description of the program state, as for all groundings θ_P of $\tilde{X}_P$, $(\exists var(\Psi_P) - \tilde{X}_P : \Psi_P)\theta_P$ is valid if and only if $\Psi_P\theta_P$ is satisfiable). Name this set $S_{\mathsf{pc}:\mathsf{k}_P}$. We remove from $S_{\mathsf{pc}:\mathsf{k}_P}$ any constraint φ such that $\varphi \wedge \Psi'(\tilde{X}') \not\Rightarrow \varphi[\tilde{X}/\tilde{X}_P]$. The remaining constraints are individually invariant through this path.

The next step is to propagate the computed path-based loop 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 18 possibly also returns a weakened state, $\mathcal{G}_P$ of a looping ancestor, in case the path contains a loop. At Line 20 we update $\mathcal{G}_c$ to take into account this generalization. More specifically, $\mathcal{G}_c$ is of the form $\langle \mathsf{pc} : \mathsf{k}, \tilde{X}, \Psi_c(\tilde{X}_P) \wedge \Phi_c(\tilde{X}_P, \tilde{X})\rangle$, while $\mathcal{G}_P$ is of the form $\langle \mathsf{pc} : \mathsf{k}, \tilde{X}_P, \Psi_P(\tilde{X}_P)\rangle$. 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 16, which may wake up a new transition which was infeasible previously but now under the weaker condition $\mathcal{G}_c$ is feasible.

The algorithm finally records the interpolated state $\overline{\Psi}(\tilde{X})$ in Line 21. In Lines 22 and 23 we simply return that interpolant. As for the loop invariant, we simply propagate the abstraction information $\mathcal{G}_P$ (Line 22), unless the current state is looping and is abstracted by a descendant, in which case we remove the abstraction (Line 23) as it has no effect on ancestors.

So far, we have presented the procedure SymbolicExec which is the core of our program transformation. This procedure runs $\mathcal{P}$ symbolically using interpolation and subsumption while building a set of edges $\mathcal{E}$. The last step is to return a CLP program from $\mathcal{E}$ and this done by procedure GenPathSensProg in Fig. 10.

The final transformation is quite straightforward. We assume a function getCLPRule which given the original CLP program $\mathcal{P}$ and a pair of program counters $\mathsf{pc}_1, \mathsf{pc}_2$, it returns the CLP rule that matches with the program counters (without loss of generality, we assume that the rule is unique):

$$p(\mathsf{pc}_1, \tilde{X}) \;\text{:-}\; p(pc_2, \tilde{X}'), \chi(\tilde{X}, \tilde{X}').$$

We assume also a function genUniquePC that maps program counters and context identifiers to a new set of program counters. The set of counters returned by the function is disjoint from the input set. Finally, genCLPRule and genEmptyCLPRule generate a new CLP rule given the new program counters and the rest of information given by the original rule. The function genEmptyCLPRule is defined as genCLPRule but without attaching the original constraints $\chi(\tilde{X}, \tilde{X}')$ to the new rule. That is, genEmptyCLPRule is similar to *nop* operation.

The purpose of using genEmptyCLPRule is to avoid redundant statements in the transformed program. Given an edge, associated

with a jump condition, if the number of its outgoing edges is at most one (i.e., it is always feasible) (Line 6) then the test is redundant and hence, we can use genEmptyCLPRule rather than genCLPRule.

We now state the correctness of our transformation, in that the new CLP program is, in the sense of program executions, equivalent to the original.

THEOREM 1 (Equivalence). *Let $\mathcal{P}$ be a CLP program and $\mathcal{G}$ be an initial goal as defined in Sec. 4. Let $\mathcal{P}'$ be another program returned by ProgramTransform($\mathcal{P}$) in algorithm described in Fig. 9 and 10. Then,*

$$\text{TRACES}(\mathcal{P},\ \mathcal{G}) \equiv \text{TRACES}(\mathcal{P}',\ \mathcal{G})$$

Limitations. In this paper, we assume structured programs. By doing so, $\text{Out}^{+}_{\mathcal{P}}$ can prioritize those feasible transitions which correspond to the entry of the loop against the exit transitions. Otherwise, we may analyze paths out of the loop without having computed yet its loop invariant. The limitation of structured programs is not fundamental but simplifies considerably the description of our algorithm. We consider the analysis of unstructured programs as an interesting future work.

Furthermore, the algorithm described in Fig. 9 does not compute summaries for functions, and hence, all functions calls will be inlined. As a direct consequence, our path-sensitive CLP program will not contain function calls (only `main` function). It is well understood that inlining functions may generate programs exponentially bigger than the originals thought this potential limitation did not arise in our experiments. We consider the computation of summaries an orthogonal issue and also interesting future work.

6. Experimental Evaluation

It is folklore that path-sensitiveness produces more accurate analyses, and so performing analysis using our path-sensitive CFG generally produces better results as compared with using the traditional CFG. Thus the primary evaluation of this section is to show the *size increase* of the path-sensitive CFG over the original. We claim that this increase is manageable, especially when the intended analyzer is path-insensitive and, typically, very fast (i.e., almost linear time).

We then demonstrate the path-sensitive CFG in action using two different path-insensitive analyzers, one of them a well-known commercial tool. We did this by decompiling the path-sensitive CFG back into C, and then running the analyzers. The purpose here is to demonstrate that added *accuracy* can be obtained using existing off-the-shelf analyzers.

These two experiments serve to demonstrate practical enhancement of analysis accuracy. However, there is a different potential benefit of using our path-sensitive CFG: it can produce *speedup* for analyzers that are already (at least partially) path-sensitive. To demonstrate this, in our final experiment, we ran the program verification tool BLAST (here considering verification as a special case of analysis).

Throughout the experiments, we implemented our program transformation for C programs by modeling the heap as an array. Alias analysis is then used to partition updates and reads into alias classes where each class is modeled by a different array. We use the CLP($\mathcal{R}$) [15] system and its native constraint solver, and extend it for reasoning about arrays in order to check the satisfiability of formulas and computing interpolants. Function calls are inlined and external functions are modeled as having no side effects and returning an unknown value.

Our tool performs the full pipeline explained in Fig. 1, Sec. 1. That is, it takes a C program and translates it into its corresponding CLP program. Then, the algorithm described in Sec. 5 is applied in order to obtain another CLP program codified with path-sensitive information. The next step is to produce its equivalent C program. We omit the details since it is straightforward. Finally, we ran several off-the-shelf analyzers to produce our experiments.

We used as benchmarks several device driver examples previously used as software model checking benchmarks [12]: `cdaudio`, `diskperf`, `floppy`, `kbfiltr`, `serial`, and `tcas`. In addition, we also consider other three real programs: `statemate` from the Mälardalen WCET group [18]; `mpeg` from [1]; and finally, `susan_thin` from [2].

Program	LOC	Original CFG		Path-Sensitive CFG			
		R	Mem	R	Mem	Time	Inc
cdaudio	8921	1385	4.3	5759	17.9	7.7	4.2
diskperf	6024	660	1.1	3100	4.8	7.2	4.3
floppy	8579	1381	3.9	2922	8.2	5.1	2.1
kbfiltr	4930	557	0.5	1337	2.1	1.5	4.2
mpeg	1773	822	0.9	1581	1.7	3.9	1.8
serial	10380	3867	10.1	53389	138.1	212.2	13.6
statemate	1276	800	10.3	25054	324.1	37.5	31.4
susan_thin	2371	608	1.1	902	1.7	4.1	1.45
tcas	405	193	0.6	2432	7.2	3.7	12

Table 1. Size Increase and Timing on Intel 3.2Gz 2Gb

First, we demonstrate that our algorithm can produce a path-sensitive CFG of a manageable size in a reasonable amount of time for several real programs. The results are summarized in Table 1. The second column LOC represents the number of lines of codes excluding comments of the C original program. In the next two columns, labeled as Original CFG, we show the number of rules of the resulting CLP program (third column R) and the size of the CLP program in megabytes (fourth column Mem). Our translation from C to CLP collapses sequence of assignments into a single CLP rule. This is the reason of the lack of correlation between LOC and R. The same measures are shown for the path-sensitive CFG. The column Time represents the time in seconds spent by the algorithm to produce the path-sensitive CFG. Note that we do not include the timing for compiling/decompiling from/to C since those numbers are negligible. Finally, the column Inc shows the size increase of the path-sensitive CFG.

In summary, the size increases of the path-sensitive CLP programs are quite reasonable in most of the benchmark examples. The program `statemate` deserves special attention as its path-sensitive version is around 30 times bigger than the original. The reason is that `statemate` is automatically generated code by the STAtechart Real-time-Code generator STARC. The program has a large amount of infeasible paths resulting in inability to extensively generalize nodes in the execution tree resulting in less subsumption. Even so, the transformed program is still manageable and it is produced within a reasonable amount of time.

Program	T_{Orig}	$T_{Path\text{-}sens}$	Red
cdaudio	394	374	5%
diskperf	537	388	27%
floppy	548	404	26%
mpeg	350	257	26%
serial	8990	7051	22%
statemate	293	263	11%
susan_thin	5025	3001	40%
tcas	96	91	5%
Average			20.25%

Table 2. Accuracy Gains for WCET Analysis

Next we demonstrate the path-sensitive CFG in action using two different path-insensitive analyses and report their results. For

both cases, we use the C original program and the path-sensitive C version obtained by decompiling its corresponding CFG. Since both analyses are quite fast due to their path-insensitive nature, the running times are insignificant (less than few seconds), and hence, not displayed.

The first analysis is a simplified version of *Worst-Case Execution Time (*WCET*)* analysis. WCET analysis aims to compute the worst possible execution time of a program and it is usually performed at two different levels. *Low-level*, which is done on binary code, provides the execution time of basic blocks considering the effects of hardware level features such as cache, pipelining, branch prediction, etc. *High-level* analysis is often performed on source code, and it focuses on characterizing all possible executable paths.

We consider high-level WCET an useful representative analysis to illustrate the potential accuracy gains of our transformation since the main objective is to exclude infeasible paths. For this purpose, we implement a path-insensitive *path-based* algorithm *à* la [21, 22] with a simple timing model: the program is instrumented with a dedicated timing variable which is incremented after the execution of each statement. As usual, upon encountering a loop, it computes separately the WCET of the loop body and multiplies that value with a given number of loop iterations. In real WCET tools the number of loop iterations can sometimes be computed automatically. For simplicity, here we assume one iteration per loop.

We ran our prototype and the results are shown in Table 2. The column $\mathsf{T}_{\mathsf{Orig}}$ expresses the value of the dedicated timing variable after running the prototype on the C original program. Column $\mathsf{T}_{\mathsf{Path\text{-}Sens}}$ represents the same information but using the transformed C program. The smaller, the better. The column Red shows the percentage of reduction in the value of the timing variable if the path-sensitive program is used. Finally, we report a considerable reduction of 20.25% in average for the set of programs selected.

The program susan_thin shows the highest accuracy improvement (40%). Here, some input values are fixed and then the evaluation of many if statements depend on those values. Therefore, there are many infeasible paths. This is good example because the size increase of the path-sensitive CFG is very small wrt to the original (1.45 only of increase, Table 1), but accuracy gains are huge.

For the second analysis we use a popular commercial analyzer, *CodeSurfer* [6], which can perform program backward slicing. This technique aims at identifying the parts of a program that potentially affect the values of specified variables at some program point.

To increase the validity of our study, we choose several *realistic* slicing criteria and report the average of them. By "realistic" we mean potential slicing criteria used in practice. The results are shown in Table 3. The column $\mathsf{Reduct}_{\mathsf{Orig}}$ reflects the percentage of lines sliced away by CodeSurfer on the original program. Similarly, column $\mathsf{Reduct}_{\mathsf{Path\text{-}Sens}}$ expresses the same measure but using the transformed C program. Column Improv shows the percentage of improvement if the path-sensitive C program is used. In this case, the improvement average is around 21%.

Program	$\mathsf{Reduct}_{\mathsf{Orig}}$	$\mathsf{Reduct}_{\mathsf{Path\text{-}Sens}}$	Improv
`cdaudio`	50%	59%	9%
`diskperf`	53%	67%	14%
`floppy`	55%	66%	11%
`kbfiltr`	52%	60%	8%
`mpeg`	38%	69%	31%
`statemate`	15%	52%	37%
`tcas`	18%	56%	38%
Average			21%

Table 3. Accuracy Gains for Slicing Using *CodeSurfer*

A different potential benefit of our program transformation could be to produce speedup in other path-sensitive program analyses. This class of analyses needs often to deal with infeasible paths in order to obtain path-sensitiveness on their own. Therefore, the efficiency gains of using our CFG come from the fact that part of the false path detection phase and the computation of its interpolants (if any), which is an expensive task, can be precomputed by our method. Of course, since the spectrum of path-sensitive analyses is very broad and the goal of our method is to build a control flow graph offline, those gains will not always pay off.

To elaborate we focus on the verification tool BLAST [13], which implements the *CounterExample-Guided Abstraction Refinement* (CEGAR), a successful technique for proving safety in large programs. CEGAR methods start with a very coarse abstraction of the program and if the abstract model violates the safety condition, then the abstraction is refined using the counterexample found. Therefore, the core idea of abstraction refinement is to use the most general abstraction first, and refine later. This causes the exploration of *infeasible paths* which is already well understood as an important drawback.

We believe our method might mitigate those limitations in certain programs where the exploration of infeasible paths may cause problems. We ran the BLAST tool on both the original `tcas` program and its path-sensitive version. The `tcas` program is an implementation of a traffic collision avoidance system, a real-life safety-critical embedded system. The program is instrumented with ten safety conditions, of which five are violated. Table 4 illustrates the impact on performance using the path-sensitive version. The second and third columns show the number of predicates considered (BLAST uses predicate abstraction) and time in seconds spent by BLAST taking the original version of `tcas`. The fourth and fifth columns are also the number of predicates and time in seconds but this time using as input the path-sensitive version of `tcas` generated by our algorithm. The sixth column S illustrates the speedup of using the path-sensitive version.

Program	Original		Path-sensitive		
	P	T	P	T	S
`tcas-1a-safe`	38	27.8	12	4.0	6.9
`tcas-1b-safe`	34	28.9	14	4.5	6.4
`tcas-2a-safe`	39	22.8	11	3.6	6.3
`tcas-3b-safe`	15	4.4	12	3.6	1.2
`tcas-5a-safe`	34	18.1	10	3.4	5.3
`tcas-2b-unsafe`	39	39.3	6	0.9	43.6
`tcas-3a-unsafe`	16	5.5	2	0.4	13.7
`tcas-4a-unsafe`	20	4.2	2	0.3	14
`tcas-4b-unsafe`	24	7.5	4	0.4	18.7
`tcas-5b-unsafe`	28	12.9	6	0.74	17.4

Table 4. Speedup for the Verification Tool BLAST on Intel 1.33Gz 2Gb

Although most speedups are significant, the numbers that attract more attention are related to the unsafe conditions. The explanation to that is simple. Our path-sensitive program contains fewer infeasible paths that lead BLAST directly to the real error. However, if BLAST is run on the original program, the tools needs to explore those infeasible paths, refine more often (increasing the number of predicates), and hence, spent more time to find the error.

7. Conclusions

We developed a program transformation which takes the CFG of a program and restructures it in order to encode path-sensitivity into the CFG. One of the major features is that the transformation is done without any knowledge of the property of interest. Hence, our

transformation can be used, in principle, to enhance the accuracy of any CFG-based analysis.

Our method executes symbolically the CFG eliminating infeasible paths from it and duplicating subgraphs when it is likely to produce more precise results by the underlying analysis. We employ interpolation to make practical the symbolic execution and produce a path-sensitive CFG of manageable size. We also provide an extensive evaluation of our method using a set of real programs that shows our method can handle non-trivial programs and still produce significant accuracy gains.

References

[1] High level synthesis benchmark suite. http://mesl.ucsd.edu/spark-/benchmarks.shtml.

[2] Mibench version 1.0: a free, commercially representative embedded benchmark suite. http://www.eecs.umich.edu/mibench/.

[3] R. Bodík, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 146–158. ACM, 1997.

[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] Rastisalv Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *POPL '98*, pages 237–251.

[6] CodeSurfer. Grammatech Inc. http://www.grammatech.com/products/codesurfer/.

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

[10] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. Joining dataflow with predicates. In *ESEC/FSE-13*, pages 227–236, 2005.

[11] T. Gutzmann, J. Lundberg, and W. Lowe. Towards path-sensitive points-to analysis. In *SCAM '07*, pages 59–68, 2007.

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

[13] 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).

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

[15] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *J. Syst. Archit.*, 46(4):339–355, 2000.

[22] F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES '01*, pages 132–140. ACM, 2001.

[23] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC '06*, 2006.

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

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