Explaining the Effectiveness of Small Refinement Heuristics in Program Verification with CEGAR

Tachio Terauchi

JAIST
terauchi@jaist.ac.jp

Abstract. Safety property (i.e., reachability) verification is undecidable for Turing-complete programming languages. Nonetheless, recent progress has lead to heuristic verification methods, such as the ones based on predicate abstraction with counterexample-guided refinement (CEGAR), that work surprisingly well in practice. In this paper, we investigate the effectiveness of the *small refinement heuristic* which, for abstraction refinement in CEGAR, uses (the predicates in) a small proof of the given counterexample's spuriousness [12, 3, 17, 22]. Such a method has shown to be quite effective in practice but thus far lacked a theoretical backing. Our core result is that the heuristic guarantees certain bounds on the number of CEGAR iterations, relative to the size of a proof for the input program.

1 Introduction

The safety property (i.e., reachability) verification problem asks, given a program, if an error state (e.g., given as a line number in the program text) is unreachable for every execution of the program. The problem is undecidable for Turing-complete programming languages (and intractably hard for many natural decidable fragments – e.g., PSPACE-complete for Boolean programs). Despite the staggering complexity, recent research has lead to heuristic verification methods that work surprisingly well in practice. They have been used to verify non-trivial real-world programs such as operating system device drivers [4, 11], and the yearly held software verification competition [1] shows an ever increasing variety of programs efficiently verified by the state-of-the-art automated software verifiers. By contrast, there has been comparatively less progress on explaining why such heuristics work. As a step toward bridging the gap, we present a theoretical explanation for the effectiveness of a heuristic used in predicate abstraction with counterexample-guided refinement (CEGAR).

CEGAR is a verification method that iteratively updates a finite set of predicates from some first-order logic (FOL) theory, called the *candidate predicate set*, until the candidate set forms a sufficient proof of the given program's safety (i.e., an inductive invariant). In each iteration, a process called *abstraction* checks if the current candidate set is sufficient, and if not, a counterexample is generated. A process called *refinement* infers a proof of the counterexample's safety (i.e.,

spuriousness) whereby the predicates in the proof are added as new candidates, and the iteration repeats.

In general, there is more than one proof that refutes the same counterexample, and which proof is inferred by the refinement process can significantly affect the performance of the overall verification process (cf. Section 2.2 for an example). A heuristic often used in practice is to infer a *small* proof (e.g., measured by the sum of the syntactic size of the predicates), and researchers have proposed methods for inferring a small proof of the given counterexample's spuriousness [12, 3, 17, 22].[1] This paper analyzes the effect such a *small refinement heuristic* has on the overall verification performance.

As safety property verification is undecidable, we cannot establish any (finitary) complexity result without some assumption on the input problem instances. In general, we will state our results relative to the size of a proof of safety for the input program, and we will try to show that the verification converges quickly assuming that the input program has a small proof (e.g., polynomial in the size of the program). The assumption captures the conventional wisdom that correct programs are often correct for simple reasons, per Occam's razor.

Overview of the Main Results. We formalize the small refinement heuristic to be a refinement process that infers a proof of size at most polynomial in that of the smallest proof for the given counterexample. Let CEGVERIF$_{\text{SR}}$ be a CEGAR verification with such a refinement process. Let $\mathit{mpfsize}(P)$ be the size of the smallest proof of safety for a program P. Our first main result is the following.

Theorem 1. CEGVERIF$_{\text{SR}}$ *converges in at most* $\mathit{exp}(\mathit{mpfsize}(P))$ *many CEGAR iterations given a program* P*.*

Theorem 1 implies that CEGVERIF$_{\text{SR}}$ is able to verify a program in an exponential number of iterations under the promise that the program has a polynomial size proof of safety. We prove Theorem 1 under a rather general setting.

Next, we consider a more concrete setting where a program is represented by a control flow graph (CFG) such that the program's proof is a Floyd-style node-wise inductive invariant [9], the abstraction process uses the Cartesian predicate abstraction, and the size of a proof is measured by the sum of the syntactic size of the predicates. We also assume that counterexamples are generated by unfolding each loop an arbitrary but the same number of times (copies of nested subloops are also unfolded the same number of times) which we show to be sufficient for the setting (cf. Theorem 9). Under such a setting, we show that CEGVERIF$_{\text{SR}}$ converges in at most $\mathit{poly}(\mathit{mpfsize}(P))^{\mathit{maxhds}(P)}$ many iterations where $\mathit{maxhds}(P)$ is the maximum number of loop entries per loop of the CFG (e.g., $\mathit{maxhds}(P) \leq 1$ for reducible CFGs) (Theorem 10). The result implies that, under such a setting, CEGVERIF$_{\text{SR}}$ is able to verify a program with a constant number of loop entries per loop in a polynomial number of iterations under the promise that the program has a polynomial size proof of safety.

[1] In this paper, a "proof" is a set of predicates. Note that this differs from the notion of proofs and proof sizes used in proof complexity [6].

The rest of the paper is organized as follows. We discuss related work next. Section 2 gives preliminary definitions pertaining to generic CEGAR verification and proves the first main result (Theorem 1). Section 3 gives additional preliminary definitions pertaining to CFG programs and proves the main result on CFG programs (Theorem 10). We discuss some limitations of our results in Section 4, and conclude the paper in Section 5. Appendix contains the proofs and extra materials omitted from the main body of the paper.

Related Work. Previous works on the small refinement heuristic [12, 3, 17, 22] have presented empirical evidences of the heuristic's effectiveness in the form of experiments. To our knowledge, this paper is the first work on a theoretical explanation for the heuristic's effectiveness.

A related but somewhat different heuristic proposed for CEGAR is the *stratified refinement* approach [13, 15, 20] where the proofs inferred in each refinement step is restricted to some finite set of proofs that is enlarged when the refinement fails to find a proof in the current set. The proof set is enlarged in such a way to ensure that the growing strata of proof sets eventually cover the underlying (possibly infinite) space of possible proofs. The stratified approach ensures an eventual convergence of CEGAR iterations under the promise that a proof of the program's safety exists in the underlying proof space, but the previous works give no results on the iteration bound.

2 Iteration Bound Under a Generic Setting

2.1 Preliminaries

Formulas and Predicates. We write finite sequences in boldface (e.g., $\boldsymbol{x}$ for $x_1, \ldots, x_n$). Let $\mathcal{T}$ be a FOL theory. A *term* a and *formula* ϕ in the signature of $\mathcal{T}$ is defined as follows.

$$\begin{array}{rrcl} \textit{term} & a & ::= & x \mid f(\boldsymbol{a}) \\ \textit{formula} & \phi & ::= & p(\boldsymbol{a}) \mid \neg\phi \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \phi \Rightarrow \phi' \mid \exists x.\phi \mid \forall x.\phi \end{array}$$

Here, x is a variable, f is an arity $|\boldsymbol{a}|$ function symbol, and p is an arity $|\boldsymbol{a}|$ predicate symbol of $\mathcal{T}$. As usual, $\Rightarrow$ binds weaker than $\wedge$ or $\vee$. For a formula ϕ, we write $\mathit{fv}(\phi)$ for the set of free variables in ϕ. A *predicate* in $\mathcal{T}$ is of the form $\lambda\boldsymbol{x}.\phi$ where ϕ is a formula such that $\{\boldsymbol{x}\} = \mathit{fv}(\phi)$. We often omit the explicit λ abstraction and treat a formula ϕ as the predicate $\lambda\boldsymbol{x}.\phi$ where $\{\boldsymbol{x}\} = \mathit{fv}(\phi)$. We overload $\mathcal{T}$ for the set of predicates in $\mathcal{T}$, and write $\mathcal{T}(\boldsymbol{x})$ for the set of $\mathcal{T}$-predicates of arity $|\boldsymbol{x}|$. We write $\models_{\mathcal{T}} \phi$ if ϕ is valid in $\mathcal{T}$. We write $\top$ for tautology and $\bot$ for contradiction. (Note that the term "predicate" is not limited to just atomic predicates as sometimes is in the literature on CEGAR.)

Generic CEGAR. To state Theorem 1 for a general setting, following the style of [20], we give a definition of CEGAR in terms of generic properties that abstraction and refinement processes are assumed to satisfy, but without specifying exactly when predicates form a proof of a program or how counterexamples are

```
1: procedure CegVerif(P)
2:   Cands := ∅;
4:   repeat
3:     match Abs(P, Cands) with
4:           safe → return safe
5:         | π → match Ref(π) with
6:                     unsafe → return unsafe
7:                   | F → Cands := Cands ∪ F
8:   end repeat
```

Fig. 1. CegVerif

generated and refuted (we concretize such notions for CFG programs in Section 3.1).

In general, a *counterexample* is an "unwound" slice of the given program and is itself considered a program (typically, without loops or recursion). Let γ range over programs and counterexamples. For a finite set of predicates $F \subseteq \mathcal{T}$, we write $F \vdash \gamma$ to mean that F is a proof of safety of γ. We often simply say that F is a *proof of* γ when $F \vdash \gamma$ (i.e., omitting "of safety"), and if γ is a counterexample in addition, we sometimes say that F *refutes* γ, or F is a proof of *spuriousness* of γ. We require the proof relation $\vdash$ to be monotonic on the predicate set, that is, if $F \vdash \gamma$ and $F \subseteq F'$, then $F' \vdash \gamma$ (i.e., having more predicates can only increase the ability to prove). We use F, F', F_1, etc. to range over finite sets of predicates.

Figure 1 shows the overview of the verification process. CegVerif takes as input the program P be be verified, and initializes the candidate predicate set *Cands* to $\emptyset$ (line 2). Then, it repeats the abstract-and-refine loop (lines 4-8) until convergence. Abs is the abstraction process which takes as input a program and a finite set of $\mathcal{T}$-predicates. For a program P and a finite $F \subseteq \mathcal{T}$, $\mathsf{Abs}(P, F)$ either returns safe, indicating that P has been proved safe using the predicates from F, or returns a counterexample. In the former case, the verification process halts, returning safe. Ref is the refinement process, which, given a counterexample π, either returns a proof of π's safety, or detects that π is irrefutable and returns unsafe. In the former case, the proof is added to the candidates, and in latter case, the verification halts by returning unsafe. For a run of CegVerif, the *number of CEGAR iterations* is defined to be the number of times the abstract-and-refine loop (lines 4-8) iterated.

We state the required assumptions on the abstraction and refinement processes. We require that if $F \vdash \gamma$ then $\mathsf{Abs}(\gamma, F) = \mathsf{safe}$, and that if $\mathsf{Abs}(P, F)$ returns a counterexample π then $\mathsf{Abs}(\pi, F) \neq \mathsf{safe}$ (i.e., Abs proves the safety of the program given a sufficient set of predicates, and otherwise returns a counterexample that it cannot prove safe with the given predicates). We also require that if $F \vdash P$ and $\mathsf{Abs}(P, F')$ returns π, then $F \vdash \pi$ (i.e., a proof for a program is also a proof for any counterexample of the program). We require that Abs is *sound* in that it only proves safe programs safe, that is, $\mathsf{Abs}(\gamma, F') = \mathsf{safe}$ only if

$\exists F \subseteq \mathcal{T}.F \vdash \gamma$. For the refinement process Ref, we require that $\mathsf{Ref}(\pi)$ returns F only if $F \vdash \pi$ (i.e., the returned proof is actually a proof for the counterexample), and that $\mathsf{Ref}(\pi)$ returns unsafe only if $\forall F \subseteq \mathcal{T}.F \nvdash \pi$ (i.e., only irrefutable counterexamples are detected irrefutable). Finally, we require Abs and Ref to halt on all inputs. We note that these assumptions are quite weak and satisfied by virtually any CEGAR verifiers.[2]

It is easy to see that CegVerif is sound in that it only proves safe programs safe.

Theorem 2 (Soundness). CegVerif(P) *returns* safe *only if* $\exists F \subseteq \mathcal{T}.F \vdash P$.

Note that Theorem 2 says nothing about how fast (or whether) CegVerif converges. The main results of the paper (Theorems 1 and 10) show that, when Ref is made to return a small proof of the given counterexample, CegVerif is guaranteed to converge in a number of iterations bounded by the size of a proof for the given program.

It is also easy to see that CegVerif is "complete" in the sense that it only detects unprovable programs unprovable.

Theorem 3 (Completeness). CegVerif(P) *returns* unsafe *only if* $\forall F \subseteq \mathcal{T}.F \nvdash P$.

Since the paper is only concerned with analyzing the behavior of CEGAR when given a provably safe program, in what follows, we disregard the situation where the given program is unprovable with the predicates from the background theory.

2.2 Main Result

To demonstrate the usefulness of the small refinement heuristic, we start with an example on which CegVerif (without the heuristic) may fail to converge despite the program having a small proof of safety (taken from [20]). Figure 2 shows the program P_{ex}. Here, `ndet()` returns a non-deterministic integer. The goal is to verify that the assertion failure is unreachable, that is, $a = b \Rightarrow y = x$ whenever line 9 is reached. We define a proof of the program to be the set of predicates that can be used as loop invariants at each of the loop heads (lines 3 and 6) and are sufficient to prove the unreachability (i.e., "safe" inductive invariant). For example, a possible proof is the singleton set $F_{inv} = \{a = b \Rightarrow y = x + z\}$.

```
1: int a = ndet(); int b = ndet();
2: int x = a; int y = b; int z = 0;
3: while (ndet()) {
4:   y++;z++;
5: }
6: while (z != 0) {
7:   y--;z--;
8: }
9: assert (a!=b || y=x);
```

Fig. 2. P_{ex}

[2] We do not impose $F \vdash P \Leftrightarrow \mathsf{Abs}(P, F) = \mathsf{safe}$ to allow modeling verifiers whose abstraction process can prove more from the same predicates than the refinement process (cf. Section 3.1).

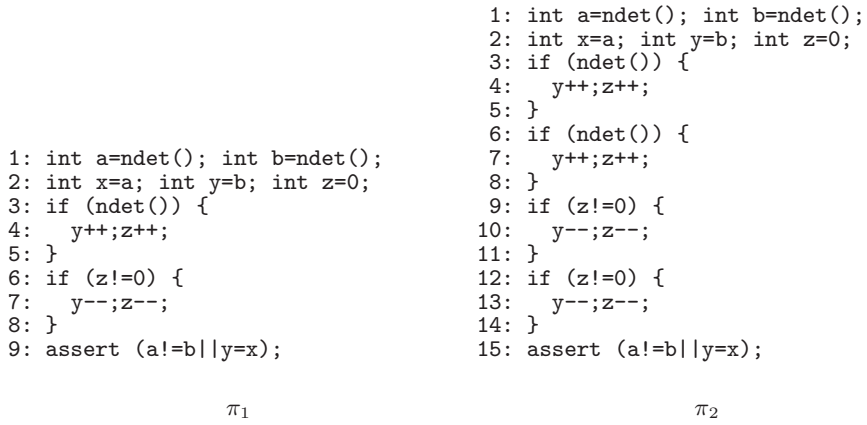

Fig. 3. Counterexamples of P_{ex}

Running CegVerif on the program, Abs may return the counterexample π_1 shown in Figure 3 in the first iteration, obtained by unfolding each loop once. Viewing the unfolded `if` statements at lines 3 and 6 as one-iteration loops, it can be seen that F_{inv} is a proof of π_1's safety. Thus, $\mathsf{Ref}(\pi_1)$ may return F_{inv}, which is also a proof for P_{ex} and would allow CegVerif to converge in the next iteration.

Unfortunately, the refinement process is not guaranteed to return F_{inv} but may choose any set of predicates that forms a proof of π_1's safety. For example, another possibility is $F_1 = \{\phi_0, \phi_1, \phi_0 \vee \phi_1\}$ where

$$\phi_0 \equiv x = a \wedge y = b \wedge z = 0$$
$$\phi_1 \equiv x = a \wedge y = b + 1 \wedge z = 1$$

Adding F_1 to the candidates is sufficient for proving the safety of π_1 but not that of P_{ex}, and so the abstraction process in the subsequent iteration would return yet another counterexample. For example, it may return π_2 shown in Figure 3 obtained by unfolding each loop twice. Then, Ref may choose the proof $F_2 = \{\bigvee F \mid F \subseteq \{\phi_0, \phi_1, \phi_2\}\}$ where $\phi_2 \equiv x = a \wedge y = b + 2 \wedge z = 2$ to prove the spuriousness of this new counterexample, which is still insufficient for proving P_{ex}. The abstract-and-refine loop may repeat indefinitely in this manner, adding to the candidates the predicates $F_k = \{\bigvee F \mid F \subseteq \{\phi_i \mid 0 \leq i \leq k\}\}$ where $\phi_i \equiv x = a \wedge y = b + i \wedge z = i$ in each k-th run of the refinement process.

Here, a key observation is that F_{inv} is a proof of safety for *every* counterexample π_1, π_2, ... of P_{ex}. Because the size of F_{inv} is "small" (under a suitable proof size metric – made more precise below), when using the small refinement heuristic, the refinement process would have to infer F_{inv} (or other small proof of P_{ex}) before producing a large number of incorrect proofs (such as F_1, F_2, ...).

Our first main result, Theorem 1, states that the above observation holds in general. The result can be proved for a rather generic notion of *proof size*. We

assumed that $size(\cdot)$ satisfies the following: there is a constant $c > 1$ such that for all $n \geq 0$, $|\{F \subseteq \mathcal{T} \mid size(F) \leq n\}| \leq c^n$ (i.e., there are at most c^n proofs of size less than or equal to n). We call such a proof size metric *generic*. The size of the smallest proof for γ, $mpfsize(\gamma)$, is defined to be $\min_{F \in \{F \subseteq \mathcal{T} \mid F \vdash \gamma\}} size(F)$.

Definition 1 (Small Refinement Heuristic). We define $\textsc{CegVerif}_{\mathrm{SR}}$ to be CegVerif with the refinement process Ref satisfying the following property: there is a polynomial f such that for all π and F, if $\mathsf{Ref}(\pi)$ returns F then $size(F) \leq f(mpfsize(\pi))$ (i.e., the refinement process returns proofs of size polynomially bounded in that of the smallest proof for the given counterexample).

We are now ready to prove Theorem 1.

Theorem 1. *Let the proof size metric be generic. Then,* $\textsc{CegVerif}_{\mathrm{SR}}$ *converges in at most* $exp(mpfsize(P))$ *many CEGAR iterations given a program* P*.*

We informally describe the intuition behind the result. First, as remarked above, a proof of a program is also a proof of any of its counterexamples. Therefore, under the small refinement heuristic, the inferred proof in each refinement step is at most polynomially larger than the smallest proof of the program. Then, the result follows from the definition of the generic proof size metric and the fact that the proofs inferred in the refinement process runs must be distinct.

3 Iteration Bound for CFG-Represented Programs

The exponential bound shown in Theorem 1 still seems to have a gap from the performance observed with using the small refinement heuristic in practice. The observation is the motivation for studying a more concrete setting such as CFG-represented programs.

3.1 Preliminaries

Graphs. A finite directed graph $G = (V, E)$ consists of a finite set of nodes V and edges $E \subseteq V \times V$. We write $v(G)$ for V and $e(G)$ for E. For an edge $e = (v, v')$, we write $sc(e)$ for v (the *source* of e) and $tg(e)$ for v' (the *target* of e). Since we only work with finite directed graphs, in what follows, we omit the adjectives and simply write *graphs*.

We write $G \setminus E$ for $(v(G), e(G) \setminus E)$, and $G \setminus V$ for $(V', e(G) \cap V' \times V')$ where $V' = v(G) \setminus V$. For $E \subseteq v(G) \times v(G)$, we write $G \cup E$ for $(v(G), e(G) \cup E)$. A *path* of G is a finite sequence nodes $\varpi = v_1 v_2 \dots v_n$ such that $(v_i, v_{i+1}) \in e(G)$ for each $i \in \{1, \dots, n-1\}$. We write $|\varpi|$ for the length of ϖ and $\varpi(i)$ for the i-th node visited (i.e., $\varpi = \varpi(1)\varpi(2)\dots\varpi(|\varpi|)$).

CFG Programs. We consider programs represented by control flow graphs (also known as *control flow automata* [5]). Formally, a *control flow graph* (CFG) is a tuple (G, T, v_{ini}, v_{err}) where G is a graph, $v_{ini} \in v(G)$ is the *initial node*, $v_{err} \in v(G)$ is the *error node*, and $T : e(G) \to \mathcal{T}(\boldsymbol{x}, \boldsymbol{x}')$ is the *transition*

relation where $|\boldsymbol{x}| = |\boldsymbol{x}'|$. Roughly, each $v \in v(G)$ represents a program location (e.g., a basic block), and for $e \in E$, $T(e)(\boldsymbol{x}, \boldsymbol{x}')$ expresses the state transition from the location $sc(e)$ to $tg(e)$ where $\boldsymbol{x}$ (resp. $\boldsymbol{x}'$) represents the values of the program variables before (resp. after) the transition. A *path* of the CFG is a path of G. Without loss of generality, we assume that v_{ini} has no incoming edges, v_{err} has no outgoing edges, and all nodes are reachable from v_{ini}. Let Vals be the set of values. The set of states of the program is $\mathsf{States} = \mathsf{Vals}^{|\boldsymbol{x}|}$. The set of states reached from $S \subseteq \mathsf{States}$ by taking an edge $e \in e(G)$ is defined to be $Post_T[e](S) = \{s' \in \mathsf{States} \mid s \in S \wedge \models_{\mathcal{T}} T(e)(s, s')\}$. The set of states reached from $S \subseteq \mathsf{States}$ by taking a path ϖ, $Post^*_T[\varpi](S)$, is defined inductively as $Post^*_T[v_1 v_2 \varpi](S) = Post^*_T[v_2 \varpi](Post_T[(v_1, v_2)](S))$ and $Post^*_T[v](S) = Post^*_T[\varepsilon](S) = S$.

We say that a program $P = (G, T, v_{ini}, v_{err})$ is *safe* if for all paths ϖ of P such that $\varpi(1) = v_{ini}$ and $\varpi(|\varpi|) = v_{err}$, $Post^*_T[\varpi](\mathsf{States}) = \emptyset$. In what follows, we often implicitly assume that $|\boldsymbol{x}, \boldsymbol{x}'|$ is the arity of the predicates in the range of the transition relation of the CFG being discussed where $\boldsymbol{x}$ and $\boldsymbol{x}'$ are distinct variables such that $|\boldsymbol{x}| = |\boldsymbol{x}'|$.

We note that the class of CFG programs is already Turing complete when $\mathcal{T}$ is the set of quantifier-free predicates in the theory of linear rational arithmetic (QFLRA – *quantifier-free theory of linear rational arithmetic*) (and is equivalent to Boolean programs when $\mathcal{T}$ is propositional), taking the reachable states from States as the computation result. Checking the safety of CFG programs when $\mathcal{T} = \mathsf{QFLRA}$ is undecidable.

Loops. We review the notion of loops in a CFG [2]. A *loop decomposition* of G is a set $\{L_0, \ldots, L_n\}$ with each $L_i = (G_i, hds_i)$ satisfying 1.) $G_0 = G$, 2.) each G_i, G_j are either disjoint or one is a subgraph of the other, 3.) each hds_i, hds_j for $i \neq j$ are disjoint, and 4.) each L_i satisfies the following:

- G_i is a non-empty subgraph of G;
- $G_i \setminus (\{e \in e(G_i) \mid tg(e) \in hds_i\} \cup \bigcup_{j \in Sub(i)} e(G_j)) \cup \bigcup_{j \in Sub(i)} \mathit{flatten}(G_i, G_j)$ is acyclic;
- $hds_i = \{v \in v(G_i) \mid v' \notin v(G_i) \wedge (v', v) \in e(G)\}$; and
- G_i is strongly connected except for G_0

where $\mathit{flatten}(G_i, G_j) = \{(sc(e_1), tg(e_2)) \mid e_1 \in e(G_i) \wedge e_2 \in e(G_i) \wedge tg(e_1) \in v(G_j) \wedge tg(e_2) \in v(G_j)\}$ (i.e., the edges formed by "flattening" G_j in G_i) and $Sub(i) = \{j \mid G_j \text{ is a proper subgraph of } G_i\}$.

Roughly, each loop L_i is a subgraph of G comprising the "back edges" $B_i = \{e \in e(G_i) \mid tg(e) \in hds_i\}$ that take the control flow back to one of the loop entries, and the "loop body" $G_i \setminus B_i$ that is acyclic when the nested subloops are flattened. Note that the loop entries hds_i are the nodes in G_i with incoming edges from nodes outside of the loop. A loop decomposition forms a tree with L_0 as the root and nested subloops as children.[3]

[3] In the literature, L_0 is typically not treated as a loop, and a loop decomposition forms a forest.

A graph is *rooted* if there is a node with no incoming edges and from which every node in the graph is reachable. Clearly, the graph underlying a CFG is rooted. The following is a folk theorem (cf. Appendix A.4 for a proof).

Theorem 4. *A rooted graph has a loop decomposition.*

We define a loop decomposition of a CFG (G, T, v_{ini}, v_{err}) to be a loop decomposition of G. In the following, we assume that each CFG P is associated with its loop decomposition $loops(P)$ (e.g., constructed by the algorithm given in Appendix A.4). We write $maxhds(P)$ for $\max_{(\_,hds)\in loops(P)} |hds|$. We remark that $maxhds(P) \leq 1$ if P is reducible [2].

Proofs and Counterexamples for CFG Programs. Informally, a counterexample of a CFG is an acyclic CFG obtained by *unfolding* the loops of the CFG. To formalize loop unfolding, we introduce a simple graph grammar below.

$$\begin{aligned} instr &::= v_0 \mapsto (\phi_1 : v_1, \phi_2 : v_2, \ldots, \phi_n : v_n) \\ t &::= \emptyset \mid \{instr\} \mid t_1 \cup t_2 \mid loop\ (h_1, \ldots, h_m)\ t \end{aligned}$$

Here, in each *instruction* $v_0 \mapsto (\phi_1 : v_1, \ldots, \phi_n : v_n)$, $v_1, \ldots, v_n$ are distinct and $\phi_i \in \mathcal{T}(\boldsymbol{x}, \boldsymbol{x'})$ for each $i \in \{1, \ldots, n\}$. We call v_0 the *source* and $v_1, ..., v_n$ the *targets* of the instruction. Roughly, $v_0 \mapsto (\phi_1 : v_1, \ldots, \phi_n : v_n)$ expresses the set of edges $\{(v_0, v_1), \ldots, (v_0, v_n)\}$ such that the transition relation of (v_0, v_i) is ϕ_i. A *term* t expresses the CFG comprising the edges represented by the instructions occurring in t. The *loop* annotations mark subparts of the CFG that correspond to loops, so that $loop\ (\boldsymbol{h})\ t$ expresses a loop with $\boldsymbol{h}$ being the entry nodes and t representing the union of the loop body, the back edges, and the edges from the loop body to the outer loops. We require the sources of the instructions occurring in a term to be distinct. We refer to Appendix B for the formal correspondence between CFGs and graph grammar terms. In what follows, we equate a CFG with the corresponding graph grammar term.

We overload t for the set of instructions occurring in t. We let $sc(v_0 \mapsto (\phi_1 : v_1, \ldots, \phi_n : v_n)) = \{v_0\}$, and $tg(v_0 \mapsto (\phi_1 : v_1, \ldots, \phi_n : v_n)) = \{v_1, \ldots, v_n\}$. We let $sc(t) = \bigcup_{instr \in t} sc(instr)$, and $tg(t) = \bigcup_{instr \in t} tg(instr)$. We let $v(t) = sc(t) \cup tg(t)$. For sets of nodes V_1 and V_2, we write $t\langle V_1, V_2 \rangle$ for t with each source occurrence of $v \in V_1$ replaced by $`v$ and each target occurrence of $v \in V_2$ replaced by $`v$. For instance, for $t = \{v_0 \mapsto (\top : v_1, \top : v_2), v_1 \mapsto (\top : v_0, \top : v_3)\}$, $t\langle\{v_0\}, \{v_1, v_3\}\rangle = \{`v_0 \mapsto (\top : `v_1, \top : v_2), v_1 \mapsto (\top : v_0, \top : `v_3)\}$. We write $t \backslash V$ for t with each instruction $v_0 \mapsto (v_1 : \phi_1, \ldots, v_n : \phi_n)$ replaced by $v_0 \mapsto (v_{i1} : \phi_{i1}, \ldots, v_{im} : \phi_{im})$ where $\{v_{i1}, \ldots, v_{im}\} = \{v_1, \ldots, v_n\} \backslash V$, treating an instruction with empty targets as $\emptyset$. For $\boldsymbol{h} = h_1, \ldots, h_n$, we write $`\boldsymbol{h}$ for $`h_1, \ldots, `h_n$.

Let rewriting contexts be defined as follows.

$$C ::= [\,] \mid C \cup t \mid loop\ (h_1, \ldots, h_m)\ C$$

Loop unfolding is defined by the following rewriting rules.

$$\begin{aligned} &C[loop\ (\boldsymbol{h})\ t] \dashrightarrow C[t\langle v(t) \backslash \{\boldsymbol{h}\}, sc(t)\rangle \cup loop\ (`\boldsymbol{h})\ t\langle\{\boldsymbol{h}\}, \{\boldsymbol{h}\}\rangle] \\ &C[loop\ (\boldsymbol{h})\ t] \dashrightarrow C[t \backslash \{\boldsymbol{h}\}] \end{aligned}$$

The first rule unfolds the loop once, whereas the second rule "closes" the loop by removing the back edges. We formalize the *counterexamples* of a CFG P, $\mathsf{cex}(P)$, to be the acyclic CFGs obtained by applying the above rewriting an arbitrary number of times, that is, $\mathsf{cex}(P) = \{\pi \mid P \dashrightarrow^* \pi \wedge \pi \text{ is acyclic}\}$.

A proof of a CFG program or counterexample is a set of predicates that forms a Floyd-style node-wise inductive invariant [9]. More formally, we say that $\sigma : v(G) \to \mathcal{T}(\boldsymbol{x})$ is a *node-wise inductive invariant* of $\gamma = (G, T, v_{ini}, v_{err})$, written $\sigma \vdash_{cfg} \gamma$, if 1.) $\sigma(v_{ini}) = \top$, 2.) $\sigma(v_{err}) = \bot$, and 3.) for each $e \in e(G)$, $\models_{\mathcal{T}} \sigma(sc(e))(\boldsymbol{x}) \wedge T(e)(\boldsymbol{x}, \boldsymbol{x}') \Rightarrow \sigma(tg(e))(\boldsymbol{x}')$. We say that F is a *proof* of γ, written $F \vdash_{cfg} \gamma$, if there exists $\sigma : v(G) \to F \cup \{\bot, \top\}$ such that $\sigma \vdash_{cfg} \gamma$. The following are immediate from the definition of $\vdash_{cfg}$.

Theorem 5 (Soundness of $\vdash_{cfg}$). *If $F \vdash_{cfg} \gamma$ then γ is safe.*

Theorem 6 (Monotonicity of $\vdash_{cfg}$). *If $F \vdash_{cfg} \gamma$ and $F \subseteq F'$, then $F' \vdash_{cfg} \gamma$.*

Theorem 7. *Suppose $F \vdash_{cfg} P$. Then, for any $\pi \in \mathsf{cex}(P)$, $F \vdash_{cfg} \pi$.*

Theorems 5, 6 and 7 justify us to use $\vdash_{cfg}$ for the proof relation $\vdash$ of CegVerif (cf. Section 2.1).

Next, we concretize the abstraction process and counterexample generation for CFGs. We present the Cartesian predicate abstraction, a form of predicate abstraction used in CEGAR, as the abstraction process. Then, we present a counterexample set that is *sound* for the abstraction process to generate and refute (made precise below). We note that, because our result is shown for the Cartesian predicate abstraction, it also holds for stronger abstraction processes such as the Boolean predicate abstraction.

We write $F^{\wedge}$ for the $\wedge$-closure of F (i.e., $\{\bigwedge F' \mid F' \subseteq F\}$) and $F^{\wedge\vee}$ for the $\wedge\vee$-closure of F (i.e., $\{\bigvee F' \mid F' \subseteq F^{\wedge}\}$). We write $F_{\bot}$ for $F \cup \{\bot\}$. For $\gamma = (G, T, v_{ini}, v_{err})$, $\sigma : v(G) \to \mathcal{T}$, and $F \subseteq \mathcal{T}$, we say that σ is a *F-Cartesian predicate abstraction node-wise inductive invariant* of γ, written $\sigma \vdash^{F}_{crt} \gamma$, if 1.) $\sigma(v_{ini}) = \top$, 2.) $\sigma(v_{err}) = \bot$, and 3.) for each $e \in e(G)$, we have $\sigma(sc(e)) = \bigvee F_1$, $\sigma(tg(e)) = \bigvee F_2$ for some $F_1 \subseteq {F_{\bot}}^{\wedge}$ and $F_2 \subseteq {F_{\bot}}^{\wedge}$ such that for each $\phi \in F_1$, there is $\psi \in F_2$ where $\models_{\mathcal{T}} \phi(\boldsymbol{x}) \wedge T(e)(\boldsymbol{x}, \boldsymbol{x}') \Rightarrow \psi(\boldsymbol{x}')$. We write $F \vdash_{crt} \gamma$ if there exists σ such that $\sigma \vdash^{F}_{crt} \gamma$. Clearly, $F \vdash_{cfg} \gamma$ implies $F \vdash_{crt} \gamma$, and $F \vdash_{crt} \gamma$ implies $F^{\wedge\vee} \vdash_{cfg} \gamma$.

The *Cartesian predicate abstraction* abstraction process $\mathsf{Abs}^{crt}(\cdot, F)$ is an abstract interpretation [7] over the finite lattice $(F^{\wedge\vee}, \models_{\mathcal{T}} \cdot \Rightarrow \cdot)$ with the abstract state transformer $\alpha^{F}(Post)$ that computes the strongest cube over $F_{\bot}$ implied in the next state (from the initial abstract state $\top$):

$$\alpha^{F}(Post)_{T}[e](\phi) = \bigwedge\{\psi \in F_{\bot} \mid \models_{\mathcal{T}} \phi(\boldsymbol{x}) \wedge T(e)(\boldsymbol{x}, \boldsymbol{x}') \Rightarrow \psi(\boldsymbol{x}')\}$$

The abstraction process guarantees 1.) $F \vdash_{crt} \gamma$ if and only if $\mathsf{Abs}^{crt}(\gamma, F)$ returns safe, and 2.) if $\mathsf{Abs}^{crt}(P, F)$ returns a counterexample π, then $F \nvdash_{crt} \pi$.

We show that Abs^{crt} satisfies the requirements of the abstraction process given in Section 2.1.

Theorem 8. Abs^{crt} *satisfies the requirements for* Abs*. That is,*

- *If* $F \vdash_{cfg} \gamma$ *then* $\mathsf{Abs}^{crt}(\gamma, F) = \mathsf{safe}$*;*
- *If* $\mathsf{Abs}^{crt}(\gamma, F)$ *returns* π *then* $\mathsf{Abs}^{crt}(\pi, F) \neq \mathsf{safe}$*;*
- *If* $F \vdash_{cfg} P$ *and* $\mathsf{Abs}^{crt}(P, F')$ *returns* π *then* $F \vdash_{cfg} \pi$*; and*
- *If* $\mathsf{Abs}^{crt}(\gamma, F) = \mathsf{safe}$ *then* $\exists F' \subseteq \mathcal{T}.F' \vdash_{cfg} \gamma$*.*

We say that the set of counterexamples $\mathcal{X}(P) \subseteq \mathsf{cex}(P)$ is *sound* for Abs if $\mathsf{Abs}(P, F) \neq \mathsf{safe}$ implies that there exists $\pi \in \mathcal{X}(P)$ such that $F \nvdash_{cfg} \pi$ (i.e., generating and refuting only the counterexamples from $\mathcal{X}(P)$ is sufficient for verifying P). We say $\mathcal{X}$ is sound for Abs if $\mathcal{X}(P)$ is sound for Abs for each P. Let $\mathsf{cex}^{syn}(P) \subseteq \mathsf{cex}(P)$ be the set of counterexamples obtained by unfolding the loops $loops(P)$ an arbitrary but the same number of times for each loop (copies of nested subloops are also unfolded the same number of times – cf. Appendix C for the formal definition). We show that cex^{syn} is sound for Abs^{crt}.

Theorem 9. cex^{syn} *is sound for* Abs^{crt}*.*

Theorems 8 and 9 justify us to use Abs^{crt} with cex^{syn} as the counterexample generator for the abstraction process of CegVerif (cf. Section 2.1).

We note that, in the setting described above, a counterexample can be a general dag-shaped CFG. While early CEGAR verifiers often restricted the counterexamples to paths [4, 11], more recent verifiers also use dag counterexamples, and researchers have proposed methods for inferring small refinements from such counterexamples [8, 10, 19, 16, 22]. Also, we note that, when $\mathcal{T}$ is QFLRA, checking the provability for a dag CFG γ (i.e., checking if $\exists F \subseteq \mathcal{T}.F \vdash_{cfg} \gamma$) is decidable, whereas it is undecidable for an arbitrary (i.e., cyclic) CFG [14].[4]

3.2 Main Result

We define the *syntactic size* of F to be the sum of the syntactic sizes of the predicates, that is, $size(F) = \sum_{\phi \in F} |\phi|$ where $|\phi|$ is the number of (logical and non-logical) symbols in ϕ. We state the main result.

Theorem 10. *Let the proof size metric be syntactic,* $\vdash_{cfg}$ *be the proof relation, and* Abs^{crt} *be the abstraction process with* cex^{syn} *as the counterexample generator. Then,* $\text{CegVerif}_{\text{SR}}$ *converges in* $poly(mpfsize(P))^{maxhds(P)}$ *many CEGAR iterations given a CFG program* P*.*

We informally describe the intuition behind the result by analyzing the behavior of $\text{CegVerif}_{\text{SR}}$ on the program P_{ex} from Figure 2, treated as a CFG program. Figure 4 (a) shows the CFG representation of the program. Here, node 1 is the initial node, node 4 is the error node, and the transition relation is as follows.

$$\phi_1(\boldsymbol{x}, \boldsymbol{x'}) \equiv x' = a' \wedge y' = b' \wedge z' = 0$$
$$\phi_2(\boldsymbol{x}, \boldsymbol{x'}) \equiv z' = z + 1 \wedge y' = y + 1 \wedge x' = x \wedge a' = a \wedge b' = b$$
$$\phi_3(\boldsymbol{x}, \boldsymbol{x'}) \equiv x' = x \wedge y' = y \wedge z' = z \wedge a' = a \wedge b' = b$$
$$\phi_4(\boldsymbol{x}, \boldsymbol{x'}) \equiv z \neq 0 \wedge z' = z - 1 \wedge y' = y - 1 \wedge x' = x \wedge a' = a \wedge b' = b$$
$$\phi_5(\boldsymbol{x}, \boldsymbol{x'}) \equiv z = 0 \wedge a = b \wedge x \neq y$$

[4] The decidability holds for any theory with effective interpolation (cf. Appendix D).

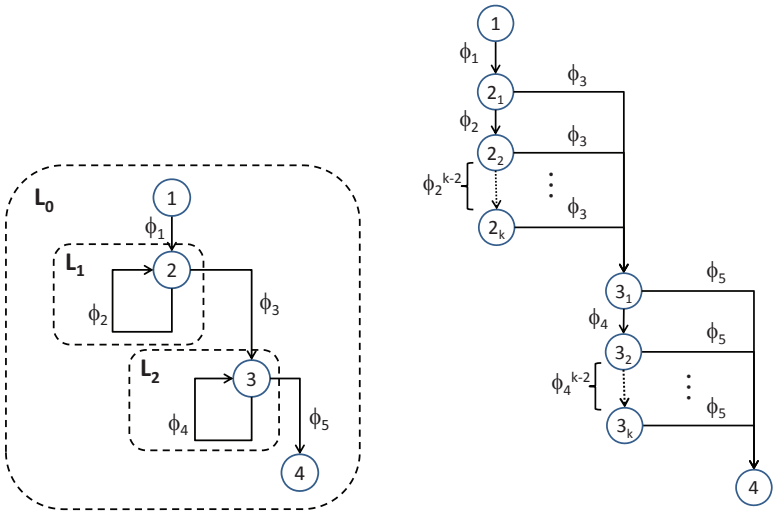

Fig. 4. (a) CFG representation of P_{ex} and (b) CFG representation of $\mathsf{cex}^{syn}(P_{ex})$

where $\boldsymbol{x} = x, y, z, a, b$ and $\boldsymbol{x'} = x', y', z', a', b'$. P_{ex} has two non-root loops L_1 and L_2 such that L_1 corresponds to the first loop (lines 3-5) and L_2 corresponds to the second loop (lines 6-8) of Figure 2. An element of $\mathsf{cex}^{syn}(P_{ex})$ is a CFG of the form shown in Figure 4 (b) where each loop is unfolded k times for some $k > 0$. Nodes $2_1, \ldots, 2_k$ (resp. $3_1, \ldots, 3_k$) are the copies of the entry node of L_1 (resp. L_2) created by the unfolding.

Recall that the small refinement heuristic returns a proof of size polynomially bounded in the size of the smallest proof of the given counterexample. Let f be the polynomial factor (i.e., Ref returns proofs of size at most $f(\mathit{mpfsize}(\pi))$ given a counterexample π). Then, because F_{inv} is a proof of any counterexample of P_{ex}, a proof returned by Ref in a run of $\textsc{CegVerif}_{\textsc{SR}}(P_{ex})$ would be of size at most $f(\mathit{size}(F_{inv}))$. Consider the counterexample $\pi_n \in \mathsf{cex}^{syn}(P_{ex})$ obtained by unfolding each loop n times where $n > f(\mathit{size}(F_{inv}))$. It can be seen that *any* proof F of π_n such that $\mathit{size}(F) \leq f(\mathit{size}(F_{inv}))$ must form a Cartesian predicate abstraction node-wise inductive invariant of P_{ex} (i.e., $F \vdash_{crt} P_{ex}$). This is because for such a F to *not* be a $\vdash_{crt}$-proof of P_{ex}, it must assign distinct predicates to the unfolded loop heads of each loop, but that is not possible when the proof size must be at most $f(\mathit{size}(F_{inv}))$.

More generally, as in Theorem 1, the first key observation used to prove Theorem 10 is the fact that a proof for the program is a proof for any counterexample of the program. As before, this implies that the small refinement heuristic ensures that the proofs inferred in the refinement are only polynomially larger than the smallest proof for the program. Then, we use the observation

that, for a CFG program, if a counterexample is "sufficiently large", then *any* small proof for the counterexample becomes a Cartesian predicate abstraction node-wise inductive invariant for the program, and so CEGAR will have to halt in the next iteration. Finally, the iteration bound follows from the number of counterexamples in cex^{syn} that are within such a sufficiently large size.

We remark that it is actually sufficient for the proof size metric to only satisfy $size(F) \geq |F|$ for Theorem 10 to hold (i.e., at least the number of predicates in F). Also, as remarked before, the result also holds for stronger abstraction processes such as the Boolean predicate abstraction. Meanwhile, as we shall further discuss in Section 4, a limitation of the result is that it only considers counterexamples where all loops are unfolded the same number of times. This implies that the abstraction process may generate large counterexamples in relatively early stages of the verification, especially when the program contains nested loops (cf. Appendix C).

We end the section by showing that the proof size metric is crucial to the result. That is, under the generic proof size metric (cf. Section 2.2), we can only guarantee an exponential bound (which is ensured by the generic result of Theorem 1), even when the rest of the setting is left unchanged from Theorem 10.

Theorem 11. *Let the proof size metric be generic, $\vdash_{cfg}$ be the proof relation, and Abs^{crt} be the abstraction process with cex^{syn} as the counterexample generator. Then, there exists a CFG program P with a proof $F \vdash_{cfg} P$ on which* CegVerif$_{\mathrm{SR}}$ *may take $exp(size(F))$ many CEGAR iterations to converge.*

4 Limitations

While we believe that the paper's results are a step toward understanding the effectiveness of the small refinement heuristic, we still have ways to go to get the whole picture. Below, we discuss some limitations of our work.

First, we do not account for the cost of the abstraction process and the refinement process (i.e., we only show bounds on the number of CEGAR iterations). For instance, the running time of the abstraction process typically grows as the number of candidate predicates grows. Also, previous work has suggested that inferring a small proof of a counterexample may be computationally expensive (cf. Appendix F of [20]), and while there has been much progress on efficient algorithms for inferring a small proof of a counterexample [12, 3, 17, 22], explaining why such algorithms work well in practice seems to be no easier than explaining the efficiency of the overall verification process.[5]

Secondly, in our setting of CFG programs, the counterexample form is rather restricted. That is, we only consider counterexamples obtained by unfolding the

[5] In a sense, this paper poses and studies the question "assuming we have such algorithms for inferring small refinements, what can be said about the overall verification efficiency?". Note that a possible outcome of the study can be a negative result; for example, showing that inferring small refinements is hard because otherwise it would give an efficient algorithm to some provably hard verification problem.

program's loops, and unfolding only the same number of times for every (copy of) loops (i.e., cex^{syn}). By contrast, an actual CEGAR verifier is often more liberal about the counterexample forms, and for example, may allow arbitrary unfoldings or paths as counterexamples.[6]

5 Conclusion

We have presented a theoretical explanation for the effectiveness of the small refinement heuristic used in program verification with CEGAR, which, to our knowledge, is the first of its kind. Specifically, we have formalized the small refinement heuristic to be a refinement process that returns a proof whose size is polynomially bounded in that of the smallest proof for the given counterexample, and shown that CEGAR with such a refinement is guaranteed to converge in a number of iterations bounded by the size of a proof of the given program. We have presented the results under a rather generic setting of CEGAR, and under a more concrete setting for CFG-represented programs.

Acknowledgements. We thank the anonymous reviewers for useful comments. This work was supported by MEXT Kakenhi 26330082 and 25280023, and JSPS Core-to-Core Program, A.Advanced Research Networks.

References

1. International competition on software verification (SV-COMP). `http://sv-comp.sosy-lab.org/`.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
3. A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 313–329. Springer, 2013.
4. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In J. Launchbury and J. C. Mitchell, editors, *POPL*, pages 1–3. ACM, 2002.
5. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32. IEEE, 2009.
6. S. A. Cook. The complexity of theorem-proving procedures. In M. A. Harrison, R. B. Banerji, and J. D. Ullman, editors, *STOC*, pages 151–158. ACM, 1971.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *POPL*, pages 238–252. ACM, 1977.
8. J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. *JSAT*, 5(1-4):27–56, 2008.

[6] It is easy to reduce any CFG program to an equivalent one whose unfoldings/paths would coincide with cex^{syn} (e.g., by encoding program locations in transition relation – cf. Appendix E). But, such a reduction is likely to affect the cost of the abstraction process and the refinement process.

9. R. W. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics*, volume 19, pages 19–32, 1967.
10. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Refining abstract interpretations. *Information Processing Letters*, 110(16):666–671, 2010.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In N. D. Jones and X. Leroy, editors, *POPL*, pages 232–244. ACM, 2004.
12. K. Hoder, L. Kovács, and A. Voronkov. Playing in the grey area of proofs. In J. Field and M. Hicks, editors, *POPL*, pages 259–272. ACM, 2012.
13. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006.
14. N. Kobayashi. Personal communication, 2012.
15. K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2008.
16. P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 347–363. Springer, 2013.
17. C. Scholl, F. Pigorsch, S. Disch, and E. Althaus. Simple interpolants for linear arithmetic. In *DATE*, pages 1–6. IEEE, 2014.
18. R. Tarjan. Testing flow graph reducibility. In *STOC*, pages 96–107, 1973.
19. T. Terauchi. Dependent types from counterexamples. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 119–130. ACM, 2010.
20. T. Terauchi and H. Unno. Relaxed stratification: A new approach to practical complete predicate refinement. In J. Vitek, editor, *ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 610–633. Springer, 2015.
21. H. Unno and N. Kobayashi. Dependent type inference with interpolants. In A. Porto and F. J. López-Fraguas, editors, *PPDP*, pages 277–288. ACM, 2009.
22. H. Unno and T. Terauchi. Inferring simple solutions to recursion-free Horn clauses via sampling. In C. Baier and C. Tinelli, editors, *TACAS*, volume 9035 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2015.

A Proofs

A.1 Proof of Theorem 2

Theorem 2. $\textsc{CegVerif}(P)$ *returns* safe *only if* $\exists F \subseteq \mathcal{T}.F \vdash P$.

Proof. Immediate from the fact that $\mathsf{Abs}(P, F)$ returns safe only if $\exists F \subseteq \mathcal{T}.F \vdash P$. □

A.2 Proof of Theorem 3

Theorem 3. $\textsc{CegVerif}(P)$ *returns* unsafe *only if* $\forall F \subseteq \mathcal{T}.F \nvdash P$.

Proof. We show the contrapositive. Suppose $F \vdash P$. Then $F \vdash \pi$ for any π returned by $\mathsf{Abs}(F', P)$ for some F'. Hence, for any counterexample π passed to Ref in the run of $\textsc{CegVerif}(P)$, $F \vdash \pi$. Therefore, by the property of Ref that it returns unsafe only if given a counterexample π such that $\forall F \subseteq \mathcal{T}.F \nvdash \pi$, it follows that $\textsc{CegVerif}(P)$ does not return unsafe. □

A.3 Proof of Theorem 1

The following lemma states that each run of the refinement process returns a new proof.

Lemma 1 (Progress). *Suppose* $\mathsf{Abs}(P, F)$ *returns* π *and* $\mathsf{Ref}(\pi)$ *returns* F'. *Then,* $F' \not\subseteq F$.

Proof. Because $\mathsf{Ref}(\pi)$ returns F', we have $F' \vdash \pi$. Also, because $\mathsf{Abs}(P, F)$ returns π, we have $\mathsf{Abs}(\pi, F) \neq \mathsf{safe}$, and so $F \nvdash \pi$. Therefore, $F \neq F'$, and by the monotonicity of $\vdash$, it follows that $F' \not\subseteq F$. □

Theorem 1. *Let the proof size metric be generic. Then,* CegVerif$_{\mathrm{SR}}$ *converges in at most* $exp(mpfsize(P))$ *many CEGAR iterations given a program* P.

Proof. Let F_P be a proof of P such that $mpfsize(P) = size(F_P)$. Then, for any counterexample π of P (i.e., $\mathsf{Abs}(P, F')$ returns π for some F'), $F_P \vdash \pi$, and therefore $mpfsize(\pi) \leq size(F_P)$. There are at most $c^{f(mpfsize(P))}$ proofs of size at most $f(size(F_P))$. Therefore, by Lemma 1, CegVerif$_{\mathrm{SR}}$ converges in at most $exp(mpfsize(P))$ many CEGAR iterations. □

A.4 Proof of Theorem 4

Algorithm 1 Loop Decomposition Construction

Input: Rooted graph G
Output: Loop Decomposition of G

```
1: decomp := {(G, ∅)};
2: Process(G);
3: return decomp
4: procedure Process(G')
5:     {G_1, ..., G_m} = Scc(G');
6:     for each cyclic G_i do
7:         hds = {v ∈ v(G_i) | v' ∉ v(G_i) ∧ (v', v) ∈ e(G)};
8:         decomp := decomp ∪ {(G_i, hds)};
9:         Process(G_i \ hds)
10:    end for
11: end procedure
```

Theorem 4. *A rooted graph has a loop decomposition.*

The result is well known for reducible graphs [18].[7] Below, we provide a proof for the general (i.e., possibly irreducible) case.

[7] For a reducible graph, each loop has only one entry node.

Proof. Algorithm 1 builds the loop decomposition *decomp* by first initializing it to contain just $L_0 = (G, \emptyset)$, and then calling the recursive procedure PROCESS on the given graph. In PROCESS, the helper procedure SCC returns the set of strongly connected components of the given graph.

Because G is rooted, *hds* in the **for each** block of PROCESS is always non-empty, and therefore, the recursive call to PROCESS is on a smaller graph, and the algorithm terminates. Then, by induction on the recursive call depth, it follows that the constructed *decomp* is a loop decomposition of G. □

A.5 Proof of Theorem 5

Theorem 5. *If $F \vdash_{cfg} \gamma$ then γ is safe.*

Proof. Let $\gamma = (G, T, v_{ini}, v_{err})$. Suppose $\sigma \vdash_{cfg} \gamma$ where $\sigma : v(G) \to F \cup \{\bot, \top\}$. Let ϖ be a path of γ such that $\varpi(1) = v_{ini}$ and $\varpi(|\varpi|) = v_{err}$. Let $\sigma_\varpi : \{1, \ldots, |\varpi|\} \to F \cup \{\bot, \top\}$ be such that $\sigma_\varpi(i) = \sigma(\varpi(i))$ for each $i \in \{1, \ldots, |\varpi|\}$. Then, for any prefix ϖ' of ϖ, by induction on the length of ϖ', we have $Post^*_T[\varpi'](\mathsf{States}) \subseteq \{s \in \mathsf{States} \mid \models_{\mathcal{T}} \sigma_\varpi(|\varpi'|)(s)\}$. Therefore, $Post^*_T[\varpi](\mathsf{States}) \subseteq \{s \in \mathsf{States} \mid \models_{\mathcal{T}} \sigma_\varpi(|\varpi|)(s)\} = \emptyset$. □

A.6 Proof of Theorem 6

Theorem 6. *If $F \vdash_{cfg} \gamma$ and $F \subseteq F'$, then $F' \vdash_{cfg} \gamma$.*

Proof. Immediate from the definition of $\vdash_{cfg}$. □

A.7 Proof of Theorem 7

Theorem 7. *Suppose $F \vdash_{cfg} P$. Then, for any $\pi \in \mathsf{cex}(P)$, $F \vdash_{cfg} \pi$.*

Proof. Let $P = (G, T, v_{ini}, v_{err})$. Let $\pi \in \mathsf{cex}(P)$ where $\pi = (G_\pi, T_\pi, v_{ini}, v_{err})$. Suppose $\sigma \vdash_{cfg} P$ where $\sigma : v(G) \to F \cup \{\bot, \top\}$ Let $\sigma_\pi : v(G_\pi) \to F \cup \{\bot, \top\}$ such that $\sigma_\pi(v') = \sigma(v)$ where $v' \in v(G_\pi)$ is the copy of the node $v \in v(G)$ created by the unfolding. Then, from the construction of π, we have $\sigma_\pi \vdash_{cfg} \pi$. □

A.8 Proof of Theorem 8

Theorem 8. Abs^{crt} *satisfies the requirements for* Abs. *That is,*

(1) If $F \vdash_{cfg} \gamma$ then $\mathsf{Abs}^{crt}(\gamma, F) = \mathsf{safe}$;
(2) If $\mathsf{Abs}^{crt}(\gamma, F)$ returns π then $\mathsf{Abs}^{crt}(\pi, F) \neq \mathsf{safe}$;
(3) If $F \vdash_{cfg} P$ and $\mathsf{Abs}^{crt}(P, F')$ returns π then $F \vdash_{cfg} \pi$; and
(4) If $\mathsf{Abs}^{crt}(\gamma, F) = \mathsf{safe}$ then $\exists F' \subseteq \mathcal{T}. F' \vdash_{cfg} \gamma$.

Proof. We show each (1)-(4).

(1) Immediate from the fact that $F \vdash_{cfg} \gamma$ implies $F \vdash_{crt} \gamma$.
(2) Suppose $\mathsf{Abs}^{crt}(\gamma, F)$ returns π. Then, $F \nvdash_{crt} \pi$. Hence, we have $\mathsf{Abs}^{crt}(\pi, F) \neq \mathsf{safe}$.
(3) Suppose $F \vdash_{cfg} P$ and $\mathsf{Abs}^{crt}(P, F')$ returns π. Then, $\pi \in \mathsf{cex}(P)$, and by Theorem 7, we have $F \vdash_{cfg} \pi$.
(4) Immediate from the fact that $F \vdash_{crt} \gamma$ implies $F^{\wedge\vee} \vdash_{cfg} \gamma$.

□

A.9 Proof of Theorem 9

Theorem 9. cex^{syn} *is sound for* Abs^{crt}.

Proof. Suppose that $F \vdash_{cfg} \pi$ for all $\pi \in \mathsf{cex}^{syn}(P)$. It suffices to show that $\mathsf{Abs}(P, F) = \mathsf{safe}$.

Then, for some "sufficiently large" $\pi \in \mathsf{cex}^{syn}(P)$, we have $F \vdash_{cfg} \pi$. Therefore, by the construction of a Cartesian predicate abstraction node-wise inductive invariant from a proof of a sufficiently large counterexample (cf. Appendix A.10), we have $F \vdash_{crt} P$, and so $\mathsf{Abs}^{crt}(P, F) = \mathsf{safe}$. □

A.10 Proof of Theorem 10

Theorem 10. *Let the proof size metric be syntactic,* $\vdash_{cfg}$ *be the proof relation, and* Abs^{crt} *be the abstraction process with* cex^{syn} *as the counterexample generator. Then,* CegVerif$_{\mathrm{SR}}$ *converges in poly*$(mpfsize(P))^{maxhds(P)}$ *many CEGAR iterations given a CFG program* P.

Proof. Let F_P be a proof of $P = (G, T, v_{ini}, v_{err})$ such that $mpfsize(P) = size(F_P)$. Then, for any counterexample π of P (i.e., $\mathsf{Abs}^{crt}(P, F)$ returns π for some F), $F_P \vdash \pi$, and so $mpfsize(\pi) \leq size(F_P)$. Let $lim = (f(size(F_P)) + 2)^{maxhds(P)} + 1$ ("+2" accounts for $\{\bot, \top\}$). Let π_{lim} be a counterexample in $\mathsf{cex}^{syn}(P)$ obtained by unfolding the loops at least *lim* many times. We show that for any F such that $F \vdash_{cfg} \pi_{lim}$ and $size(F) \leq f(size(F_P))$, $F \vdash_{crt} P$. (We call such a counterexample π_{lim} *sufficiently large*.) Then, the result follows from the fact that there are only *lim* many counterexamples in $\mathsf{cex}^{syn}(P)$ that are obtained by unfolding the loops at most *lim* many times, and the fact that no counterexample is returned more than once by the refinement process in a run of CegVerif$_{\mathrm{SR}}$.

Let $\pi_{lim} = (G_{lim}, T_{lim}, v_{ini}, v_{err})$. Let $\sigma_{lim} : v(G_{lim}) \to F \cup \{\bot, \top\}$ be such that $size(F) \leq f(size(F_P))$ and $\sigma_{lim} \vdash_{cfg} \pi_{lim}$. Let $k \geq lim$ be the number of times the loops are unfolded in π_{lim}. We construct $\sigma : v(G) \to F^{\wedge\vee}$ such that $\sigma \vdash^{F}_{crt} P$ by "folding" the unfolded loops in a bottom up manner. We initialize $\sigma = \sigma_{lim}$, and $\gamma = \pi_{lim}$. We iteratively fold γ from leaf loops, while maintaining the property that $\sigma \vdash^{F}_{crt} \gamma$. Then, the result follows from the fact that γ becomes P at the root of the folding process.

Let γ and σ be the current CFG and its Cartesian predicate abstraction node-wise inductive invariant (i.e., $\sigma \vdash^{F}_{crt} \gamma$). Let $\gamma = C[t'_1 \cup t'_2 \cup \ldots t'_k \cup t_k \backslash \{\boldsymbol{h}_k\})]$ where

$\boldsymbol{h}_0 = \boldsymbol{h}$, $t_0 = t$, $\boldsymbol{h}_{i+1} = \text{`}(\boldsymbol{h}_i)$, ${t_i}' = t_i\langle v(t_i)\backslash\{\boldsymbol{h}_i\}, sc(t_i)\rangle$, and $t_{i+1} = t_i\langle \boldsymbol{h}_i, \boldsymbol{h}_i\rangle$ for each $i \in \{0, \ldots, k-1\}$. Let $\gamma' = C[loop\ (\boldsymbol{h})\ t]$. That is, γ' is obtained by folding the unfolded loop of γ. We construct $\sigma' : v(\gamma') \to F^{\wedge\vee}$ such that $\sigma' \vdash^F_{crt} \gamma'$ as follows. Because $ran(\sigma_{lim}) = F \cup \{\bot, \top\}$, for some $1 \leq i_1 < i_2 \leq (f(size(F_P)) + 2)^{|\boldsymbol{h}|} + 1 \leq k$, $\sigma_{lim}(h^{i_1}) = \sigma_{lim}(h^{i_2})$ for each $h \in \{\boldsymbol{h}\}$. By construction, $\sigma_{lim}(h^i) = \sigma(h^i)$ for each $i \in \{1, \ldots, k\}$ and $h \in \{\boldsymbol{h}\}$. We set $\sigma'(v) = \bigvee_{i=1}^{i_2} \sigma(v^i)$ for each $v \in sc(t)$, and $\sigma'(v) = \sigma(v)$ for each $v \in v(\gamma) \setminus sc(t)$. Then, it can be seen that $\sigma' \vdash^F_{crt} \gamma'$. □

A.11 Proof of Theorem 11

Theorem 11. *Let the proof size metric be generic, $\vdash_{cfg}$ be the proof relation, and Abs^{crt} be the abstraction process with cex^{syn} as the counterexample generator. Then, there exists a CFG program P with a proof $F \vdash_{cfg} P$ on which* CegVerif$_{\mathrm{SR}}$ *may take $exp(size(F))$ many CEGAR iterations to converge.*

Proof. We show that P_{ex} from Figure 4 is such a program. As remarked before, we have $F_{inv} \vdash_{cfg} P_{ex}$ where $F_{inv} = \{a = b \Rightarrow y = x + z\}$. For each $k > 0$, let $\pi_k \in \mathsf{cex}^{syn}(P_{ex})$ be the counterexample obtained by unfolding each loop k times, and let $F_k = \{\bigvee F \mid F \subseteq \{\phi_i \mid 0 \leq i \leq k\}\}$ where $\phi_i \equiv x = a \wedge y = b + i \wedge z = i$. Then, $F_k \vdash_{cfg} \pi_k$ for each $k > 0$, and $\bigcup_{i=1}^{j} F_i \nvdash_{crt} \pi_k$ for each $k > j > 0$.

Let $\ell > 0$. Define *size* as follows: $size(F_{inv}) = \ell$, $size(F_k) = \lfloor \log k \rfloor$ for $k \in \{1, \ldots, 2^\ell - 1\}$, and $size(F') = \ell + syntactic(F')$ for $F' \in \mathcal{P}(\mathcal{T}) \setminus (\{F_{inv}\} \cup \{F_k \mid 1 \leq k \leq 2^\ell - 1\})$ where $syntactic(F')$ is the syntactic size of F'. Note that *size* is a generic proof size metric. Then, CegVerif$_{\mathrm{SR}}$ takes $2^{size(F_{inv})}$ iterations to converge when Abs^{crt} returns the counterexamples $\pi_1, ...\pi_{2^\ell - 1}$ in the first $2^\ell - 1$ iterations and Ref returns the proofs $F_1, \ldots, F_{2^\ell - 1}$ before returning F_{inv}. □

B Correspondence of CFGs and CFG Graph Grammar

For an instruction $instr = v_0 \mapsto (\phi_1 : v_1, \ldots, \phi_n : v_n)$, we write $e(instr)$ for $\{(v_0, v_1), \ldots, (v_0, v_n)\}$, and $trans(instr)$ for the map T such that and $T((v_0, v_i)) = \phi_i$ for each $i \in \{1, \ldots, n\}$. For a term t, we write $e(t)$ for $\bigcup_{instr \in t} e(instr)$, and $trans(t)$ for $\bigcup_{instr \in t} trans(instr)$.

We define $loops(t)$ inductively as follows: $loops(\{instr\}) = \emptyset$, $loops(t_1 \cup t_2) = loops(t_1) \cup loops(t_2)$, and $loops(loop\ (\boldsymbol{h})\ t) = loops(t) \cup \{(G, \{\boldsymbol{h}\})\}$ where $G = (sc(t), e(t) \cap sc(t) \times sc(t))$. Then, a CFG (G, T, v_{ini}, v_{err}) is said to *correspond* to a term t when $v(G) = v(t)$, $e(G) = e(t)$, $T = trans(t)$, and $loops(G) = loops(t)$.

C Formal Definition of cex^{syn}

For a term t and $k > 0$, we define t^k be t with each loop annotation added the unfolding counter k. More formally, we inductively define counter annotation as follows: ${instr}^k = instr$, $(t_1 \cup t_2)^k = {t_1}^k \cup {t_2}^k$, and $(loop\ (\boldsymbol{h})\ t)^k = loop^k\ (\boldsymbol{h})\ (t^k)$.

We let the rewriting relation respect the counter annotation, so that each loop is unfolded the number of times it is annotated:

$$C[loop^k\ (\boldsymbol{h})\ t] \dashrightarrow_{ct} C[t\langle v(t)\backslash\{\boldsymbol{h}\}, sc(t)\rangle \cup loop^{k-1}\ (`\boldsymbol{h})\ t\langle\{\boldsymbol{h}\},\{\boldsymbol{h}\}\rangle]$$
$$C[loop^0\ (\boldsymbol{h})\ t] \dashrightarrow_{ct} C[t\backslash\{\boldsymbol{h}\}]$$

Note that in the first rule, the counter annotations in t are carried over to $t\langle v(t)\backslash\{\boldsymbol{h}\}, sc(t)\rangle$, and so the nested copies of loops created in the unfolding will also be unfolded the number of times as specified by the annotations in t.

Then, $\mathsf{cex}^{syn}(P)$ is defined to be the set of CFGs obtained by applying the above reduction to convergence from a term P^k for some $k > 0$. That is, $\mathsf{cex}^{syn}(P) = \{\pi \mid P^k \dashrightarrow^*_{ct} \pi \wedge k > 0 \wedge \pi \text{ contains no } loop\}$.

Example. Consider the program P below (left).

```
L1: while (b1) {
      if (b2) {
L2:     while (b3) { s1; }
      } else {
L3:     while (b4) { s2; }
      }
    }
```

P

```
if (b1) {
   if (b2) {
     if (b3) { s1; }if (b3) { s1; }
   } else {
     if (b4) { s2; }if (b4) { s2; }
   }
}
if (b1) {
   if (b2) {
     if (b3) { s1; }if (b3) { s1; }
   } else {
     if (b4) { s2; }if (b4) { s2; }
   }
}
```

π

P has three non-root loops: L_1, L_2, and L_3. The loops L_2 and L_3 are nested inside L_1. Assuming that the CFG-representation of P preserves the loop structure, the counterexample π obtained by unfolding the loops twice (i.e., $P^2 \dashrightarrow^*_{ct} \pi$) is shown above (right). Note that the outer loop L_1 is unfolded twice, and the inner loops L_2 and L_3 are each unfolded twice per each unfolding of L_1 (and so each four times total).

D Deciding Provability of Acyclic CFGs

We show an algorithm for deciding the provability of acyclic CFGs that is based on a reduction to recursion-free Horn-clause constraint solving [21, 19].

First, we define effective interpolation. Given $\mathcal{T}$-formulas ϕ and ψ, we say that a $\mathcal{T}$-formula θ is an *interpolant* of ϕ and ψ if $fv(\theta) \subseteq fv(\phi) \cap fv(\psi)$, $\models_{\mathcal{T}} \phi \Rightarrow \theta$, and $\models_{\mathcal{T}} \theta \Rightarrow \psi$. We say that $\mathcal{T}$ is *effectively interpolating* if there exists an algorithm $\mathcal{A}$ such that, given $\mathcal{T}$-formulas ϕ and ψ, $\mathcal{A}(\phi, \psi)$ returns an interpolant of ϕ and ψ if one exists, and otherwise returns No. By abuse of terminology, for

"quantifier-free theories", we restrict ϕ, ψ, θ to be quantifier-free formulas of $\mathcal{T}$ in the definition above. QFLRA is known to be effectively interpolating.[8]

Recursion-free Horn-clause Constraints. A *predicate variable application* is of the form $Q(\boldsymbol{x})$ where Q is a *predicate variable* of arity $|\boldsymbol{x}|$. A *Horn clause* hc is of the form $\phi \wedge \beta_1 \wedge \cdots \wedge \beta_n \Rightarrow \alpha$ where ϕ is a $\mathcal{T}$-formula, each β_i is a predicate variable application, and α is a predicate variable application or $\bot$. A *Horn-clause constraint set* (HCCS) $\mathcal{H}$ is a finite set of Horn clauses. We write $pv(\mathcal{H})$ for the predicate variables in $\mathcal{H}$. We say that $\mathcal{H}$ is *recursion-free* if $\{(Q, Q') \mid \psi \wedge \ldots Q(\boldsymbol{x}) \ldots \Rightarrow Q'(\boldsymbol{y}) \in \mathcal{H}\}$ is acyclic.

We say that $\rho : pv(\mathcal{H}) \to \mathcal{T}$ is a *solution* of $\mathcal{H}$, written $\rho \vdash \mathcal{H}$, if $\models_{\mathcal{T}} \rho(hc)$ for each $hc \in \mathcal{H}$, where $\rho(hc)$ is hc with each predicate variable application $Q(\boldsymbol{x})$ replaced by $\rho(Q)(\boldsymbol{x})$. Previous works [21, 19] have shown algorithms for solving recursion-free HCCSs for effectively interpolating $\mathcal{T}$.[9]

Reduction. We reduce provability of an acyclic CFG to solving a recursion-free HCCS. Let $\gamma = (G, T, v_{ini}, v_{err})$. For each $v \in v(G)$, let Q_v be a distinct predicate variable of arity $|\boldsymbol{x}|$. Let $\mathcal{H}_\gamma$ be the set of Horn clauses $\{Q_v(\boldsymbol{x}) \wedge T((v, v'))(\boldsymbol{x}, \boldsymbol{x}') \Rightarrow Q_{v'}(\boldsymbol{x}') \mid (v, v') \in e(G)\} \cup \{\top \Rightarrow Q_{v_{ini}}(\boldsymbol{x}), Q_{v_{err}}(\boldsymbol{x}) \Rightarrow \bot\}$. Because G is acyclic, $\mathcal{H}_\gamma$ is recursion-free. Also, by construction, we have $\exists \rho : pv(\mathcal{H}_\gamma) \to F \cup \{\bot, \top\}. \rho \vdash \mathcal{H}_\gamma$ if and only if $F \vdash_{cfg} \gamma$.

E Reducing CFG to one-loop CFG

Let $P = (G, T, v_{ini}, v_{err})$. We give a polynomial time transformation of P into a CFG $P^\dagger$ such that (1) $P^\dagger$ is safe if and only if P is safe, and (2) $\mathsf{cex}^{syn}(P^\dagger) = \mathsf{cex}(P^\dagger)$.

For each $v \in v(G)$, let ϕ_v be a $\mathcal{T}$-predicate of some fixed arity (i.e., $|\boldsymbol{pc}|$) satisfying $\models_{\mathcal{T}} \phi_v(\boldsymbol{pc}) \Rightarrow \neg\phi_{v'}(\boldsymbol{pc})$ for each $v, v' \in v(G)$ such that $v \neq v'$. The transformation works for any theories that have such predicates. For example, for QFLRA, it suffices to let $\phi_v = \lambda pc.pc = num(v)$ where $num(\cdot)$ assigns a distinct rational constant to each $v \in V$.

Let $\boldsymbol{y} = \boldsymbol{pc}, \boldsymbol{x}$ and $\boldsymbol{y'} = \boldsymbol{pc'}, \boldsymbol{x'}$ where $\boldsymbol{y}, \boldsymbol{y'}$ are distinct variables such that $|\boldsymbol{y'}| = |\boldsymbol{y}|$. We define $G^\dagger$ and $T^\dagger$ as follows.

$$
\begin{array}{l}
v(G^\dagger) = \{{v_{ini}}^\dagger, {v_{err}}^\dagger, {v_{loop}}^\dagger\} \\
e(G^\dagger) = \{({v_{ini}}^\dagger, {v_{loop}}^\dagger), ({v_{loop}}^\dagger, {v_{loop}}^\dagger), ({v_{loop}}^\dagger, {v_{err}}^\dagger)\} \\
T^\dagger(({v_{ini}}^\dagger, {v_{loop}}^\dagger)) = \lambda \boldsymbol{y}, \boldsymbol{y'}.\phi_{v_{ini}}(\boldsymbol{pc'}) \\
T^\dagger(({v_{loop}}^\dagger, {v_{loop}}^\dagger)) = \lambda \boldsymbol{y}, \boldsymbol{y'}. \bigvee_{e \in e(G)} \phi_{sc(e)}(\boldsymbol{pc}) \wedge T(e)(\boldsymbol{x}, \boldsymbol{x'}) \wedge \phi_{tg(e)}(\boldsymbol{pc'}) \\
T^\dagger(({v_{loop}}^\dagger, {v_{err}}^\dagger)) = \lambda \boldsymbol{y}, \boldsymbol{y'}.\phi_{v_{err}}(\boldsymbol{pc})
\end{array}
$$

[8] This differs from the notion of interpolants from mathematical logic which requires the non-logical symbols of θ to appear in ϕ and ψ. But, this definition is often used in the verification literature and is sufficient for the purpose.

[9] We remark that the converse also holds, that is, any algorithm for solving recursion-free HCCS gives an interpolation algorithm for the corresponding theory. This is because θ is an interpolant of ϕ and ψ if and only if $\{Q \mapsto \lambda \boldsymbol{x}.\theta\}$ is a solution of $\{\phi \Rightarrow Q(\boldsymbol{x}), Q(\boldsymbol{x}) \wedge \neg\psi \Rightarrow \bot\}$ where $\{\boldsymbol{x}\} = fv(\phi) \cap fv(\psi)$.

It is easy to see that $P^\dagger = (G^\dagger, T^\dagger, {v_{ini}}^\dagger, {v_{err}}^\dagger)$ satisfies the conditions (1) and (2) given above. We note that (2) follows because $P^\dagger$ has only one non-root loop. Also, it can be seen that the the syntactic size of the smallest proof of $P^\dagger$ is polynomial in the size of G, the syntactic sizes of ϕ_v's, and the syntactic size of the smallest proof of P. This follows because $\sigma \vdash_{cfg} P$ implies $\sigma^\dagger \vdash_{cfg} P^\dagger$ where $\sigma^\dagger({v_{ini}}^\dagger) = \top$, $\sigma^\dagger({v_{err}}^\dagger) = \bot$, and $\sigma^\dagger({v_{loop}}^\dagger) = \lambda \boldsymbol{y}. \bigvee_{v \in v(G)} \phi_v(\boldsymbol{pc}) \wedge \sigma(v)(\boldsymbol{x})$.