Noname manuscript No.
(will be inserted by the editor)

Loop Summarization using State and Transition Invariants

Daniel Kroening · Natasha Sharygina · Stefano Tonetta · Aliaksei Tsitovich · Christoph M. Wintersteiger

the date of receipt and acceptance should be inserted later

Abstract This paper presents algorithms for program abstraction based on the principle of loop summarization, which, unlike traditional program approximation approaches (e.g., abstract interpretation), does not employ iterative fixpoint computation, but instead computes *symbolic abstract transformers* with respect to a set of abstract domains. This allows for an effective exploitation of problem-specific abstract domains for summarization and, as a consequence, the precision of an abstract model may be tailored to specific verification needs. Furthermore, we extend the concept of loop summarization to incorporate *relational abstract domains* to enable the discovery of transition invariants, which are subsequently used to prove termination of programs. Well-foundedness of the discovered transition invariants is ensured either by a separate decision procedure call or by using abstract domains that are well-founded by construction.

We experimentally evaluate several abstract domains related to memory operations to detect buffer overflow problems. Also, our light-weight termination analysis is demonstrated to be effective on a wide range of benchmarks, including OS device drivers.

Keywords Program Abstraction · Loop Summarization · Loop Invariants · Transition Invariants · Termination

D. Kroening
University of Oxford, UK

N. Sharygina
University of Lugano, Switzerland

S. Tonetta
Fondazione Bruno Kessler, Trento, Italy

A. Tsitovich
University of Lugano / Phonak AG, Switzerland

C. M. Wintersteiger
Microsoft Research, Cambridge UK

1 Introduction

Finding good abstractions is key to further extension of the applicability of formal methods to real problems in software and hardware engineering. Building a concise model that represents the semantics of a system with sufficient precision is the objective of research in this area.

Loops in programs are the Achilles' heel of program verification. Sound analysis of all program paths through loops requires either an explicit unwinding or an over-approximation (of an invariant) of the loop.

Unwinding is computationally too expensive for many industrial programs. For instance, loops greatly limit the applicability of bounded model checking (BMC) [8]. In practice, if the bound on the number of loop iterations cannot be pre-computed (the problem is undecidable by itself), BMC tools simply unwind the loop a finite number of times, thus trading the soundness of the analysis for scalability. Other methods rely on sufficiently strong loop invariants; however, the computation of such invariants is an art. Abstract interpretation [22] and counterexample-guided abstraction refinement (CEGAR) [12] use saturating procedures to compute over-approximations of the loop. For complex programs, this procedure may require many iterations until the fixpoint is reached or the right abstraction is determined. Widening is a remedy for this problem, but it introduces further imprecision, yielding spurious behavior.

Many approximation techniques furthermore assume that loops terminate, i.e., that every execution reaches the end of the loop after a finite number of iterations. Unfortunately, it is not the case for many real applications (some loops are even designed to be infinite). Thus, (non-) termination should be taken into account when constructing approximations.

In this paper, we focus on effective program abstraction and, to that end, we propose a *loop summarization* algorithm that replaces program fragments with summaries, which are symbolic abstract transformers. Specifically, for programs with no loops, an algorithm precisely encodes the program semantics into symbolic formulæ. For loops, abstract transformers are constructed based on problem-specific abstract domains. The approach does not rely on fixpoint computation of the abstract transformer and, instead, constructs the transformer as follows: an abstract domain is used to draw a candidate abstract transition relation, giving rise to a candidate *loop invariant*, which is then checked to be consistent with the semantics of the loop. The algorithm allows tailoring of the abstraction to each program fragment and avoids any possibly expensive fixpoint computation and instead uses a finite number of relatively simple consistency checks. The checks are performed by means of calls to a SAT or SMT-based decision procedure, which allows to check (possibly infinite) sets of states within a single query. Thus, the algorithm is not restricted to finite-height abstract domains.

Our technique is a general-purpose loop and function summarization method and supports relational abstract domains that allow the discovery of (disjunctively well-founded) *transition invariants* (relations between pre- and poststates of a loop). These are employed to address the problem of program termination. If a disjunctively well-founded transition invariant exists for a loop, we can conclude that it is terminating, i.e., any execution through a loop contains a finite number of loop iterations. *Compositionality* (transitivity) of transition invariants is exploited to limit the analysis to a single loop iteration, which in many cases performs considerably better in terms of run-time.

We implemented our loop summarization technique in a tool called LOOPFROG and we report on our experience using abstract domains tailored to the discovery of buffer overflows on a large set of ANSI-C benchmarks. We demonstrate the applicability of the approach to industrial code and its advantage over fixpoint-based static analysis tools.

Due to the fact that LOOPFROG only ever employs a safety checker to analyze loop bodies instead of unwindings, we gain large speedups compared to state-of-the-art tools that are based on path enumeration. At the same time, the false-positive rate of our algorithm is very low in practice, which we demonstrate in an experimental evaluation on a large set of Windows device drivers.

Outline This paper is structured as follows. First, in Section 2 we provide several basic definitions required for reading the paper. Then, Section 3 presents our method for loop summarization and sketches the required procedures. Next, in Section 4 the algorithm is formalized and a proof of its correctness is given. Background on termination and the algorithm extension to support termination proofs is presented in Section 5. Implementation details and an experimental evaluation are provided in Section 6. Related work is discussed in Section 7. Finally, Section 8 gives conclusions and highlights possible future research directions.

2 Loop Invariants

Informally, an invariant is a property that always holds for (a part of) the program. The notion of invariants of computer programs has been an active research area from the early beginnings of computer science. A well-known instance are Hoare's rules for reasoning about program correctness [37]. For the case of looping programs fragments, [37] refers to *loop invariants*, i.e., predicates that hold upon entry to a loop and after each iteration. As a result, loop invariants are guaranteed to hold immediately upon exit of the loop[1]. For example, "`p-a` $\leq$ `length(a)`" is a loop invariant for the loop in Figure 1.

Important types of loop invariants that are distinguished and used in this work are *state invariants* and *transition invariants.*

Formally, a program can be represented as a transition system $P = \langle S, I, R \rangle$, where:

- S is a set of states;
- $I \subseteq S$ is the set of initial states;
- $R \subseteq S \times S$ is the transition relation.

We use the relational composition operator $\circ$ which is defined for two binary relations $R_i \subseteq S \times S$ and $R_j \subseteq S \times S$ as

$$R_i \circ R_j := \left\{(s, s') \in S \times S \;\middle|\; \exists s'' \in S \,.\, (s, s'') \in R_i \wedge (s'', s') \in R_j\right\}.$$

To simplify the presentation, we also define $R^1 := R$ and $R^n := R^{n-1} \circ R$ for any relation $R : S \times S$.

[1]Here we only consider structured loops or loops for which the exit condition is evaluated before an iteration has changed the state of any variables.

```
p=a;
while(*p!=0){
 if(*p=='/')
  *p=0;
 p++;
}
```

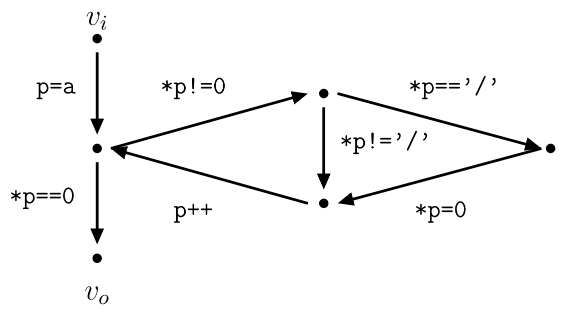


Fig. 1: The example of a program and its program graph

Note that a relation R is transitive if it is closed under relational composition, i.e., when $R \circ R \subseteq R$. The reflexive and non-reflexive transitive closures of R are denoted as R^* and R^+ respectively. The set of reachable states is then defined as $R^*(I) := \{s \in S \,|\, \exists s' \in I \;.\; (s', s) \in R^*\}$.

We now discuss two kinds of invariants, *state invariants* and *transition invariants*. For historical reasons, state invariants are often referred to simply as invariants; we will use the term "state invariant" when it is necessary to stress its difference to transition invariants.

Definition 1 (State Invariant) A *state invariant* V for program P represented by a transition system $\langle S, I, R\rangle$ is a superset of the reachable state space, i.e., $R^*(I) \subseteq V$.

In contrast to state invariants that represent the safety class of properties, transition invariants, introduced by Podelski and Rybalchenko [49], enable reasoning about liveness properties and, in particular, about program termination.

Definition 2 (Transition Invariant [49]) A *transition invariant* T for program P represented by a transition system $\langle S, I, R\rangle$ is a superset of the transitive closure of R restricted to the reachable state space, i.e., $R^+ \cap (R^*(I) \times R^*(I)) \subseteq T$.

State and transition invariants can be used together during loop summarization to preserve both safety- and liveness-related semantics of a loop in its abstraction. Our experiments with a static analyser tailored to buffer overflows focus on state invariants, while our experiments with termination analysis mainly use transition invariants.

3 Loop Summarization with State Invariants

Algorithm 1 presents an outline of loop summarization. The function Summarize traverses the control-flow graph of the program P and calls itself recursively for each block with nested loops. If a block is a loop without nested loops, it is summarized using the function SummarizeLoop and the resulting summary replaces the original loop in P'. Thereby, outer loops also become loop-free, which enables further progress.

Algorithm 1: Routines of loop summarization

```
1  SUMMARIZE($P$)
2  input: program $P$
3  output: Program summary
4  begin
5      foreach Block $B$ in CONTROLFLOWGRAPH($P$) do
6          if $B$ has nested loops then
7              $B$ :=SUMMARIZE($B$)
8          else if $B$ is a single loop then
9              $B$ :=SUMMARIZELOOP($B$)
10         endif
11     end foreach
12     return $P$
13 end

14 SUMMARIZELOOP($L$)
15 input: Single-loop program $L$ (over variable set $X$)
16 output: Loop summary
17 begin
18     $I := \top$
19     foreach Candidate $C$ in PICKINVARIANTCANDIDATES($L$) do
20         if ISSTATEINVARIANT($L$, $C$) then
21             $I := I \wedge C$
22         endif
23     end foreach
24     return "$X^{pre} := X;$ havoc($X$); assume($I(X^{pre}) \implies I(X)$);"
25 end

26 ISSTATEINVARIANT($L$, $C$)
27 input: Single-loop program $L$ (over variable set $X$), invariant candidate $C$
28 output: TRUE if $C$ is an invariant for $L$; FALSE otherwise
29 begin
30     return UNSAT($\neg(L(X, X') \wedge C(X) \Rightarrow C(X'))$)
31 end
```

The function SUMMARIZELOOP computes the summaries. A very imprecise over-approximation is to replace a loop with a program fragment that "havocs" the state by setting all variables that are (potentially) modified during loop execution to non-deterministic values. To improve the precision of these summaries, we strengthen them by means of (partial) loop invariants. SUMMARIZELOOP has two subroutines that are related to invariant discovery: PICKINVARIANTCANDIDATES, which returns a set of "invariant candidates" depending on an abstract interpretation selected for the loop and ISSTATEINVARIANT, which establishes whether a candidate is an actual loop invariant (a state invariant in this case).

Note that this summarization algorithm only uses state invariants and does not take loop termination into account. State invariants over-approximate the set of states that a loop can reach but do not provide any information about the progress of the loop. Thus, the summaries computed by the algorithm are always terminating program fragments. The abstraction is a sound over-approximation, but it may be too coarse for programs that contain unreachable paths. We address this issue in Section 5.

4 Summarization using Symbolic Abstract Transformers

In the following subsections we formally describe the steps of our summarization approach. We first present necessary notation and define summarization as an over-approximation of a code fragment. Next, we show that a precise summary can be computed for a loop-free code fragment, and we explain how a precise summary of a loop body is used to obtain information about the computations of the loop. Finally, we give a bottom-up summarization algorithm applicable to arbitrary programs.

4.1 Abstract interpretation

To formally state the summarization algorithm and prove its correctness we rely on *abstract interpretation* [22]. It constructs an abstraction of a program using values from an *abstract domain* by iteratively applying the instructions of a program to abstract values until a fixpoint is reached. Formally:

Definition 3 A program graph is a tuple
$G = \langle PL, E, pl_i, pl_o, \mathcal{L}, \mathcal{C} \rangle$, where

– PL is a finite non-empty set of vertices called *program locations*;
– $pl_i \in PL$ is the initial location;
– $pl_o \in PL$ is the final location;
– $E \subseteq PL \times PL$ is a non-empty set of edges; E^* denotes the set of *paths*, i.e., the set of finite sequences of edges;
– $\mathcal{L}$ is a set of elementary commands;
– $\mathcal{C} : E \to \mathcal{L}$ associates a command with each edge.

A program graph is often used as an intermediate modeling structure in program analysis. In particular, it is used to represent the control-flow graph of a program.

Example 1 To demonstrate the notion of a program graph we use the program fragment in Figure 1 as an example. On the left-hand side, we provide the program written in the programming language `C`. On the right-hand side, we depict its program graph.

Let S be the set of states, i.e., the set of valuations of program variables. The set of commands $\mathcal{L}$ consists of tests $\mathcal{L}_\mathcal{T}$ and assignments $\mathcal{L}_\mathcal{A}$, i.e., $\mathcal{L} = \mathcal{L}_\mathcal{T} \dot{\cup} \mathcal{L}_\mathcal{A}$, where:

– a test $q \in \mathcal{L}_\mathcal{T}$ is a predicate over S ($q \subseteq S$);
– an assignment $e \in \mathcal{L}_\mathcal{A}$ is a map from S to S.

A program P is then formalized as the pair $\langle S, G\rangle$, where S is the set of states and G is a program graph. We write $\mathcal{L}^*$ for the set of sequences of commands. Given a program P, the set $paths(P) \subseteq \mathcal{L}^*$ contains the sequence $\mathcal{C}(e_1), \ldots, \mathcal{C}(e_n)$ for every $\langle e_1, .., e_n\rangle \in E^*$.

The (concrete) semantics of a program is given by the pair $\langle A, \tau\rangle$, where:

- A is the set of *assertions* of the program, where each assertion $p \in A$ is a predicate over S ($p \subseteq S$);
  $A(\Rightarrow, \text{false}, \text{true}, \vee, \wedge)$ is a complete Boolean lattice;
- $\tau : \mathcal{L} \to (A \to A)$ is the predicate transformer.

An *abstract interpretation* is a pair $\langle \hat{A}, t\rangle$, where $\hat{A}$ is a complete lattice of the form $\hat{A}(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$, and $t : \mathcal{L} \to (\hat{A} \to \hat{A})$ is a predicate transformer. Note that $\langle A, \tau\rangle$ is a particular abstract interpretation called the *concrete interpretation.* In the following, we assume that for every command $c \in \mathcal{L}$, the function $t(c)$ (predicate transformer for command c) is monotone (which is the case for all natural predicate transformers). Given a predicate transformer t, the function $\tilde{t} : \mathcal{L}^* \to (\hat{A} \to \hat{A})$ is recursively defined as follows:

$$\tilde{t}(p)(\phi) = \begin{cases} \phi & \text{if } p \text{ is empty} \\ \tilde{t}(e)(t(q)(\phi)) & \text{if } p = q; e \text{ for a } q \in \mathcal{L},\ e \in \mathcal{L}^*. \end{cases}$$

Example 2 We continue using the program in Figure 1. Consider an abstract domain where abstract state is a four-tuple $\langle p_a, z_a, s_a, l_a\rangle$. The first member, p_a is the offset of the pointer p from the base address of the array a (i.e., $p - a$ in our example), the Boolean z_a holds if a contains the zero character, the Boolean s_a holds if a contains the slash character, l_a is the index of the first zero character if present. The predicate transformer t is defined as follows:

$t(p = a)(\phi) = \phi[p_a := 0]$ for any assertion ϕ;
$t(*p \,!= 0)(\phi) = \phi \wedge (p_a \neq l_a)$ for any assertion ϕ;
$t(*p == 0)(\phi) = \phi \wedge z_a \wedge (p_a \geq l_a)$ for any assertion ϕ;
$t(*p ==' /')(\phi) = \phi \wedge s_a$ for any assertion ϕ;
$t(*p \,!=' /')(\phi) = \phi$ for any assertion ϕ;
$t(*p = 0)(\phi) = \begin{cases} \phi[z_a := \text{true}, l_a := p_a] & \text{if } \phi \Rightarrow (p_a < l_a) \\ \phi[z_a := \text{true}] & \text{otherwise;} \end{cases}$
$t(p{+}{+})(\phi) = \phi[p_a := p_a + 1]$ for any assertion ϕ.

(We used $\phi[x := v]$ to denote an assertion equal to ϕ apart from the variable x that takes value v.)

Given a program P, an abstract interpretation $\langle \hat{A}, t\rangle$ and an element $\phi \in \hat{A}$, we define the *Merge Over all Paths* $MOP_P(t, \phi)$ as

$$MOP_P(t, \phi) := \bigsqcup_{\pi \in paths(P)} \tilde{t}(\pi)(\phi)\ .$$

Given two complete lattices $\hat{A}(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $\hat{A}'(\sqsubseteq', \bot', \top', \sqcup', \sqcap')$, the pair of functions $\langle \alpha, \gamma\rangle$, with $\alpha : \hat{A} \to \hat{A}'$ and $\gamma : \hat{A}' \to \hat{A}$ is a *Galois connection* iff α and γ are monotone and satisfy:

$$\begin{aligned} \text{for all } \phi \in \hat{A} :&\quad \phi \sqsubseteq \gamma(\alpha(\phi)) \\ \text{for all } \phi' \in \hat{A}' :&\quad \alpha(\gamma(\phi')) \sqsubseteq' \phi'\ . \end{aligned}$$

An abstract interpretation $\langle \hat{A}, t\rangle$ is a *correct over-approximation* of the concrete interpretation $\langle A, \tau\rangle$ iff there exists a Galois connection $\langle \alpha, \gamma\rangle$ such that for all $\phi \in \hat{A}$ and $p \in A$, if $p \Rightarrow \gamma(\phi)$, then $\alpha(MOP_P(\tau, p)) \sqsubseteq MOP_P(t, \phi)$ (i.e., $MOP_P(\tau, p) \Rightarrow \gamma(MOP_P(t, \phi))$).

4.2 Computing Abstract Transformers

In order to implement abstract interpretation for a given abstract domain, an algorithmic description of the abstract predicate transformer $t(p)$ for a specific command $p \in \mathcal{L}$ is required. These transformers are frequently hand-coded for a given programming language and a given domain. Reps et al. describe an algorithm that computes the *best possible* (i.e., most precise) abstract transformer for a given finite-height abstract domain automatically [52]. Graf and Saïdi's algorithm for constructing predicate abstractions [30] is identified as a special case.

The algorithm presented by Reps et al. has two inputs: a formula $F_{\tau(q)}$, which represents the concrete semantics $\tau(q)$ of a command $q \in \mathcal{L}$ symbolically, and an assertion $\phi \in \hat{A}$. It returns the image $t(q)(\phi)$ of the predicate transformer $t(q)$. The formula $F_{\tau(q)}$ is passed to a decision procedure, which is expected to provide a satisfying assignment to the variables. The assignment represents one concrete transition $(p, p') \in A \times A$. The transition is abstracted into a pair $(\phi, \phi') \in \hat{A} \times \hat{A}$, and a blocking constraint is added to remove this satisfying assignment. The algorithm iterates until the formula becomes unsatisfiable. An instance of the algorithm for the case of predicate abstraction is the implementation of SatAbs described in [15]. SatAbs uses a propositional SAT-solver as decision procedure for bit-vector arithmetic. The procedure is worst-case exponential in the number of predicates, and thus, alternatives have been explored. In [42, 39] a symbolic decision procedure generates a symbolic formula that represents the set of all solutions. In [43], a first-order formula is used and the computation of all solutions is carried out by an SMT-solver. In [9], a similar technique is proposed where BDDs are used in order to efficiently deal with the Boolean component of $F_{\tau(q)}$.

4.3 Abstract summarization

The idea of summarization is to replace a code fragment, e.g., a procedure of the program, by a *summary*, which is a representation of the fragment. Computing an exact summary of a program (fragment) is undecidable in the general case. We therefore settle for an over-approximation. We formalize the conditions the summary must fulfill in order to have a semantics that over-approximates the original program.

We extend the definition of a correct over-approximation (from Section 4.1) to programs. Given two programs P and P', we say that P' is a *correct over-approximation* of P iff for all $p \in A(\Rightarrow, \text{false}, \text{true}, \vee, \wedge)$, $MOP_P(\tau, p) \Rightarrow MOP_{P'}(\tau, p)$.

Definition 4 (Abstract Summary) Given a program P, and an abstract interpretation $\langle \hat{A}, t\rangle$ with a Galois connection $\langle \alpha, \gamma\rangle$ with $\langle A, \tau\rangle$, we denote the *abstract summary* of P by $Sum_{\langle \hat{A}, t\rangle}(P)$. It is defined as the program $\langle U, G\rangle$, where U denotes the universe of program variables and $G = \langle \{v_i, v_o\}, \{\langle v_i, v_o\rangle\}, v_i, v_o, \{a\}, \mathcal{C}\rangle$

and $\{a\}$ together with $\mathcal{C}(\langle v_i, v_o\rangle) = a$, where a is a new (concrete) command such that $\tau(a)(p) = \gamma(MOP_P(t, \alpha(p)))$.

Lemma 1 *If $\langle \hat{A}, t\rangle$ is a correct over-approximation of $\langle A, \tau\rangle$, the abstract summary $Sum_{\langle \hat{A},t\rangle}(P)$ is a correct over-approximation of P.*

Proof Let $P' = Sum_{\langle \hat{A},t\rangle}(P)$. For all $p \in A$,

$$\begin{aligned} MOP_P(\tau, p)) &\Rightarrow \gamma(MOP_P(t, \alpha(p)))\text{[by def. of correct over-approx.]} \\ &= \tau(a)(p)\text{[by Definition 4]} \\ &= MOP_{P'}(\tau, p)\ \text{[MOP over a single-command path]}. \end{aligned}$$

The following sections discuss our algorithms for computing abstract summaries. The summarization technique is first applied to particular fragments of the program, specifically to loop-free (Section 4.4) and single-loop programs (Section 4.5). In Section 4.6, we use these procedures as subroutines to obtain the summarization of an arbitrary program. We formalize code fragments as *program sub-graphs.*

Definition 5 Given two program graphs $G = \langle V, E, v_i, v_o, \mathcal{L}, \mathcal{C}\rangle$ and $G' = \langle V', E', v'_i, v'_o, \mathcal{L}', \mathcal{C}'\rangle$, G' is a *program sub-graph* of G iff $V' \subseteq V$, $E' \subseteq E$, and $\mathcal{C}'(e) = \mathcal{C}(e)$ for every edge $e \in E'$.

4.4 Summarization of loop-free programs

Obtaining $MOP_P(t, \phi)$ is as hard as assertion checking on the original program. Nevertheless, there are restricted cases where it is possible to represent $MOP_P(t, \phi)$ using a symbolic predicate transformer.

Let us consider a program P with a finite number of paths, in particular, a program whose program graph does not contain any cycles. A program graph $G = \langle V, E, v_i, v_o, \mathcal{L}, \mathcal{C}\rangle$ is *loop-free* iff G is a directed acyclic graph.

In the case of a loop-free program P, we can compute a precise (not abstract) summary by means of a formula F_P that represents the concrete behavior of P. This formula is obtained by converting P to static single assignment (SSA) form, whose size is linear in the size of P (this step is beyond the scope of this work; see [14] for details).

Example 3 We continue the running example (Fig. 1). The symbolic transformer of the loop body P' is represented by:

$$((*p =' /' \wedge a' = a[*p = 0]) \vee (*p \neq' /' \wedge a' = a)) \wedge (p' = p + 1)\ .$$

Recall the abstract domain from Ex. 2. We can deduce that:

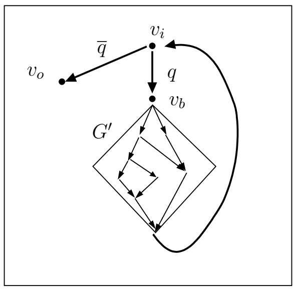
1. if $m < n$, then $MOP_{P'}(t, (p_a = m \wedge z_a \wedge (l_a = n) \wedge \neg s_a)) = (p_a = m+1 \wedge z_a \wedge l_a = n \wedge \neg s_a)$
2. $MOP_{P'}(t, z_a) = z_a$.

This example highlights the generic nature of our technique. For instance, case 1 of the example cannot be obtained by means of predicate abstraction because it requires an infinite number of predicates. Also, the algorithm presented in [52] cannot handle this example because assuming the string length has no a-priori bound, the lattice of the abstract interpretation has infinite height.

4.5 Summarization of single-loop programs

We now consider a program that consists of a single loop.

Definition 6 A program $P = \langle U, G\rangle$ is a *single-loop program* iff $G = \langle V, E, v_i, v_o, \mathcal{L}, \mathcal{C}\rangle$ and there exists a program sub-graph G' and a test $q \in \mathcal{L}_{\mathcal{T}}$ such that



- $G' = \langle V', E', v_b, v_i, \mathcal{L}', \mathcal{C}'\rangle$ with
  - $V' = V \setminus \{v_o\}$,
  - $E' = E \setminus \{\langle v_i, v_o\rangle, \langle v_i, v_b\rangle\}$,
  - $\mathcal{L}' = \mathcal{L} \setminus q$,
  - $\mathcal{C}'(e) = \mathcal{C}(e)$ for all $e \in E'$,
  - G' is loop-free.
- $\mathcal{C}(\langle v_i, v_b\rangle) = q$, $\mathcal{C}(\langle v_i, v_o\rangle) = \overline{q}$.

We first record the following simple lemma.

Lemma 2 *Given a loop-free program P, and an abstract interpretation $\langle \hat{A}, t\rangle$, if $MOP_P(t, \psi) \sqsubseteq \psi$, then, for all repetitions of loop-free paths of a program P, i.e., for all $\pi \in (paths(P))^*$, $\tilde{t}(\pi)(\psi) \sqsubseteq \psi$.*

Proof If $MOP_P(t, \psi) = \bigsqcup_{\pi \in paths(P)} \tilde{t}(\pi)(\psi) \sqsubseteq \psi$, then, for all paths $\pi \in paths(P)$, $\tilde{t}(\pi)(\psi) \sqsubseteq \psi$. Thus, by induction on repetitions of loop-free paths, for all paths $\pi \in (paths(P))^*$, $\tilde{t}(\pi)(\psi) \sqsubseteq \psi$.

The following can be seen as the "abstract interpretation analog" of Hoare's rule for `while` loops.

Theorem 1 *Given a single-loop program P with guard q and loop body P', and an abstract interpretation $\langle \hat{A}, t\rangle$, let ψ be an assertion satisfying $MOP_{P'}(t, t(q)(\psi)) \sqsubseteq \psi$ and let $\langle \hat{A}, t_\psi\rangle$ be a new abstract interpretation s.t.*

$$MOP_P(t_\psi, \phi) = \begin{cases} t(\overline{q})(\psi) & \textit{if } \phi \sqsubseteq \psi \\ \top & \textit{elsewhere.} \end{cases}$$

If $\langle \hat{A}, t\rangle$ is a correct over-approximation, then $\langle \hat{A}, t_\psi\rangle$ is also a correct over-approximation.

Proof If $\phi \sqsubseteq \psi$,

$$\begin{aligned}
\alpha(MOP_P(\tau, p)) &\sqsubseteq MOP_P(t, p) \qquad [\langle \hat{A}, t\rangle \text{ is a correct over-approximation}] \\
&= \bigsqcup_{\pi \in paths(P)} \tilde{t}(\pi)(\phi) \qquad [\text{by definition of } MOP] \\
&\sqsubseteq \bigsqcup_{\pi \in paths(P)} \tilde{t}(\pi)(\psi) \qquad [\phi \sqsubseteq \psi] \\
&= \bigsqcup_{\pi \in (q;\pi')^*, \pi' \in paths(P')} \tilde{t}((\pi'; \overline{q})^*)(\psi) \quad [P \text{ is a single-loop program}] \\
&= \bigsqcup_{\pi \in (q;\pi')^*, \pi' \in paths(P')} \tilde{t}((\overline{q})^*)(\tilde{t}((\pi')^*)(\psi)) \qquad [\text{by definition of } \tilde{t}] \\
&\sqsubseteq t(\overline{q})(\bigsqcup_{\pi \in (q;\pi')^*, \pi' \in paths(P')} \tilde{t}((\pi')^*)(\psi)) \qquad [t \text{ is monotone}] \\
&\sqsubseteq t(\overline{q})(\psi) \qquad [\text{by Lemma 2}]
\end{aligned}$$

Otherwise, trivially $\alpha(MOP_P(\tau, p)) \sqsubseteq \top = MOP_P(t_\psi, \phi)$.

In other words, if we apply the predicate transformer of the test q and then the transformer of the loop body P' to the assertion ψ, and we obtain an assertion at least as strong as ψ, then ψ is a state invariant of the loop. If a stronger assertion ϕ holds before the loop, the predicate transformer of $\overline{q}$ applied to ϕ holds afterwards.

Theorem 1 gives rise to a summarization algorithm. Given a program fragment and an abstract domain, we heuristically provide a set of formulas, which encode that a (possibly infinite) set of assertions ψ are invariant (for example, $x' = x$ encodes that every ψ defined as $x = c$, with c a value in the range of x, is an invariant). We apply a decision procedure to check if the formulas are unsatisfiable (see IsStateInvariant(L,C) in Algorithm 1).

The construction of the summary is then straightforward: given a single-loop program P, an abstract interpretation $\langle \hat{A}, t\rangle$, and a state invariant ψ for the loop body, let $\langle \hat{A}, t_\psi\rangle$ be the abstract interpretation as defined in Theorem 1. We denote the summary $Sum_{\langle \hat{A}, t_\psi\rangle}(P)$ by `sls`$(P, \hat{A}, t_\psi)$ (Single-Loop Summary).

Corollary 1 *If $\langle \hat{A}, t\rangle$ is a correct over-approximation of $\langle A, \tau\rangle$, then* `sls`$(P, \hat{A}, t_\psi)$ *is a correct over-approximation of P.*

Example 4 We continue the running example from Figure 1. Recall the abstract domain in Ex. 2. Let P' denote the loop body of the example program and let q denote the loop guard. By applying the symbolic transformer from Ex. 3, we can check that the following conditions hold:

1. $MOP_{P'}(t, t(q)(\phi)) \sqsubseteq \phi$ for any assertion $((p_a \le l_a) \wedge z_a \wedge \neg s_a)$.
2. $MOP_{P'}(t, t(q)(\phi)) \sqsubseteq \phi$ for the assertion z_a.

Thus, we summarize the loop with the following predicate transformer:

$$(z_a \rightarrow z'_a) \wedge (((p_a \le l_a) \wedge z_a \wedge \neg s_a) \rightarrow ((p'_a = l'_a) \wedge z'_a \wedge \neg s'_a)) \ .$$

Algorithm 2: Arbitrary program summarization

```
1  SUMMARIZE($P$)
   input  : program $P = \langle U, G\rangle$
   output: over-approximation $P'$ of $P$
2  begin
3      $\langle T, >\rangle$ :=sub-graph dependency tree of $P$;
4      $P_r := P$;
5      for each $G'$ such that $G > G'$ do
6          $\langle U, G''\rangle$:=SUMMARIZE($\langle U, G'\rangle$);
7          $P_r := P_r$ where $G'$ is replaced with $G''$;
8          update $\langle T, >\rangle$;
9      end for
10     if $P_r$ is a single loop then
11         $\langle \hat{A}, t\rangle$ := choose abstract interpretation for $P_r$;
           /* Choice of abstract interpretation defines set of
           candidate assertions $\psi$, which are checked to hold in the
           next step.                                              */
12         $\psi$ := test state invariant candidates for $t$ on $P_r$;
13         $P'$ := SlS($P_r, \hat{A}, t_\psi$);
           /* Those $\psi$ that hold on $P_r$ form the single-loop summary
           (SlS).                                                  */
14     endif
15     else
           /* $P_r$ is loop-free                                   */
16         $P' := Sum_{\langle A,\tau\rangle}(P_r)$;
17     endif
18     return $P'$
19 end
```

4.6 Summarization for programs with multiple loops

We now describe an algorithm for over-approximating a program with multiple loops that are possibly nested. Like traditional algorithms (e.g., [56]), the dependency tree of program fragments is traversed bottom-up, starting from the leaves. The code fragments we consider may be function calls or loops. We treat function calls as arbitrary sub-graphs (see Definition 5) of the program graph, and do not allow recursion. We support irreducible graphs using loop simulation [2].

Specifically, we define the *sub-graph dependency tree* of a program $P = \langle U, G\rangle$ as the tree $\langle T, >\rangle$, where

- the set of nodes of the tree are program sub-graphs of G;
- for $G_1, G_2 \in T$, $G_1 > G_2$ iff G_2 is a program sub-graph of G_1 with $G_1 \neq G_2$;
- the root of the tree is G;
- every leaf is a loop-free or single-loop sub-graph;
- every loop sub-graph is in T.

Algorithm 2 takes a program as input and computes its summary by following the structure of the sub-graph dependency tree (Line 3). Thus, the algorithm is called recursively on the sub-program until a leaf is found (Line 5). If it is a single loop, an abstract domain is chosen (Line 11) and the loop is summarized as described in Section 4.5 (Line 13). If it is a loop-free program, it is summarized with a symbolic transformer as described in Section 4.4 (Line 16). The old sub-program is then replaced with its summary (Line 7) and the sub-graph dependency tree is updated (Line 8). Eventually, the entire program is summarized.

Theorem 2 SUMMARIZE(P) *is a correct over-approximation of* P*.*

Proof We prove the theorem by induction on the structure of the sub-graph dependency tree.

In the first base case (P_r is loop-free), the summary is precise by construction and is thus a correct over-approximation of P.

In the second base case (P_r is a single loop), by hypothesis, each abstract interpretation chosen at Line 11 is a correct over-approximation of the concrete interpretation. Thus, if P is a single-loop or a loop-free program, P' is a correct over-approximation of P (resp. by Theorem 1 and by definition of abstract summary).

In the inductive case, we select a program subgraph G' and we replace it with G'', where $\langle U, G''\rangle$=Summarize($\langle U, G'\rangle$). Through the induction hypothesis, we obtain that $\langle U, G''\rangle$ is a correct over-approximation of $\langle U, G'\rangle$. Thus, for all $p \in A$, $MOP_{\langle U,G'\rangle}(\tau, p) \Rightarrow MOP_{\langle U,G''\rangle}(\tau, p)$. Note that G'' contains only a single command g.

We want to prove that for all $p \in A$, $MOP_P(\tau, p) \Rightarrow MOP_{P'}(\tau, p)$ (for readability, we replace the subscript "$(\pi_i; \pi_g; \pi_f) \in paths(P)$, $\pi_g \in paths(\langle U, G'\rangle)$, and $\pi_i \cap G' = \emptyset$" with $*$ and "$\pi \in paths(P)$, and $\pi \cap G' = \emptyset$" with $**$):

$$MOP_P(\tau, p) = \bigsqcup_{\pi \in paths(P)} \tilde{\tau}(\pi)(p) \quad \text{[by definition of } MOP\text{]}$$
$$= \bigsqcup_{*} \tilde{\tau}(\pi_f)(\tilde{\tau}(\pi_g)(\tilde{\tau}(\pi_i)(p))) \cup \bigsqcup_{**} \tilde{\tau}(\pi)(p) \quad [G' \text{ is a subgraph}]$$
$$\Rightarrow \bigsqcup_{*} \tilde{\tau}(\pi_f)(MOP_{\langle U,G''\rangle}(\tilde{\tau}, (\tilde{\tau}(\pi_i)(p)))) \cup \bigsqcup_{**} \tilde{\tau}(\pi)(p) \; [G'' \text{ is an over-approx.}]$$
$$= \bigsqcup_{*} \tilde{\tau}(\pi_f)(MOP_{\langle U,(G'';\pi_i)\rangle}(\tau, p)) \cup \bigsqcup_{**} \tilde{\tau}(\pi)(p) \quad \text{[by definition of } MOP\text{]}$$
$$= \bigsqcup_{\pi \in paths(P)} MOP_{\pi[g/\pi_g]}(\tau, p) \quad \text{[by induction on length of paths]}$$
$$= \bigsqcup_{\pi \in paths(P')} MOP_{\pi}(\tau, p) \quad \text{[by definition of } \pi'\text{]}$$
$$= MOP_{\pi'}(\tau, P) \quad \text{[by definition of } MOP\text{]}$$

The precision of the over-approximation is controlled by the precision of the symbolic transformers. However, in general, the computation of the best abstract transformer is an expensive iterative procedure. Instead, we use an inexpensive syntactic procedure for loop-free fragments. Loss of precision only happens when summarizing loops, and greatly depends on the abstract interpretation chosen in Line 11.

Note that Algorithm 2 does not limit the selection of abstract domains to any specific type of domain, and that it does not iterate the predicate transformer on the program. Furthermore, this algorithm allows for *localization* of the summarization procedure, as a new domain may be chosen for every loop. Once the domains are fixed, it is also straightforward to monitor the progress of the summarization, as the number of loops and the cost of computing the symbolic transformers are known—another distinguishing feature of our algorithm.

The summarization can serve as an over-approximation of the program. It can be trivially analyzed to prove unreachability, or equivalently, to prove assertions.

4.7 Leaping counterexamples

Let P' denote the summary of the program. The program P' is a loop-free sequence of symbolic summaries for loop-free fragments and loop summaries. A *counterexample* for an assertion in P' follows this structure: for loop-free program fragments, it is identical to a concrete counterexample. Upon entering a loop summary, the effect of the loop body is given as a single transition in the counterexample: we say that the counterexample *leaps* over the loop.

Example 5 Consider the summary from Ex. 4. Suppose that in the initial condition, the buffer a contains a null terminating character in position n and no $'/'$ character. If we check that, after the loop, p_a is greater than the size n, we obtain a counterexample with $p_a^0 = 0, p_a^1 = n$.

The *leaping counterexample* may only exist with respect to the abstract interpretations used to summarize the loops, i.e., the counterexample may be spurious in the concrete interpretation. Nevertheless, leaping counterexamples provide useful diagnostic feedback to the programmer, as they show a (partial) path to the violated assertion, and contain many of the input values the program needs to violate the assertion. Furthermore, spurious counterexamples can be eliminated by combining our technique with counterexample-guided abstraction refinement, as we have an abstract counterexample. Other forms of domain refinement are also applicable.

5 Termination Analysis using Loop Summarization

In this section we show how to employ loop summarization with transition invariants to tackle the problem of program termination.

5.1 The termination problem

The termination problem (also known as the uniform halting problem) is has roots in Hilbert's *Entscheidungsproblem* and can be formulated as follows:

In finite time, determine whether a given program always finishes running or could execute forever.

Undecidability of this problem was shown by Turing [57]. This result sometimes gives rise to the belief that termination of a given program can never be proven. In contrast, numerous algorithms that prove termination of many realistic classes of programs have been published, and termination analysis now is at a point where industrial application of termination proving tools *for specific programs* is feasible.

One key fact underlying these methods is that termination may be reduced to the construction of *well-founded ranking relations* [58]. Such a relation establishes an order between the states of the program by *ranking* each of them, i.e., by assigning a natural number to each state such that for any pair of consecutive states s_i, s_{i+1} in any execution of the program, the rank decreases, i.e., $rank(s_{i+1}) < rank(s_i)$. The existence of such an assignment ensures well-foundedness of the given set of transitions. Consequently, a program is terminating if there exists a *ranking function* for every program execution.

Podelski and Rybalchenko proposed *disjunctive* well-foundedness of transition invariants [49] as a means to improve the degree of automation of termination provers. Based on this discovery, the same authors together with Cook gave an algorithm to verify program termination using iterative construction of transition invariants—the *Terminator* algorithm [17, 18]. This algorithm exploits the relative simplicity of ranking relations for a single path of a program. It relies on a safety checker to find previously unranked paths of a program, computes a ranking relation for each of them individually, and disjunctively combines them in a global (disjunctively well-founded) termination argument. This strategy shifts the complexity of the problem from ranking relation synthesis to safety checking, a problem for which many efficient solutions exist (mainly by means of reachability analysis based on Model Checking).

The Terminator algorithm was successfully implemented in tools (e.g., the Terminator [18] tool, ARMC [50], SatAbs [21]) and applied to verify industrial code, most notably, Windows device drivers. However, it has subsequently become apparent that the safety check is a bottleneck of the algorithm, consuming up to 99% of the run-time [18, 21] in practice. The runtime required for ranking relation synthesis is negligible in comparison. A solution to this performance issue is *Compositional Termination Analysis (CTA)* [40]. This method limits path exploration to several iterations of each loop of the program. Transitivity (or *compositionality*) of the intermediate ranking arguments is used as a criterion to determine when to stop the loop unwinding. This allows for a reduction in run-time, but introduces incompleteness since a transitive termination argument may not be found for each loop of a program. However, experimental evaluation on Windows device drivers confirmed that this case is rare in practice.

The complexity of the termination problem together with the observation that most loops have, in practice, (relatively) simple termination arguments suggests the use of light-weight static analysis for this purpose. In particular, we propose a termination analysis based on the loop summarization algorithm described in Section 3. We build a new technique for termination analysis by 1) employing an abstract domain of (disjunctively well-founded) transition invariants during summarization and 2) using a compositionality check as a completeness criterion for the discovered transition invariant.

5.2 Formalism for reasoning about termination

As described in Section 2, we represent a program as a transition system $P = \langle S, I, R \rangle$, where:

- S is a set of states;
- $I \subseteq S$ is the set of initial states;
- $R \subseteq S \times S$ is the transition relation.

We also make use of the notion of (disjunctively well-founded) transition invariants (Definition 2, page 4) introduced by Podelski and Rybalchenko [49].

Definition 7 (Well-foundedness) A relation R is *well-founded (wf.)* over S if for any non-empty subset of S, there exists a minimal element (with respect to R), i.e., $\forall X \subseteq S \,.\, X \neq \emptyset \implies \exists m \in X, \forall s \in X, (s,m) \notin R$.

The same does not hold for the weaker notion of *disjunctive well-foundedness.*

Definition 8 (Disjunctive Well-foundedness [49]) A relation T is *disjunctively well-founded (d.wf.)* if it is a finite union $T = T_1 \cup \cdots \cup T_n$ of well-founded relations.

The main result of the work [49] concludes program termination from the existence of disjunctively well-founded transition invariant.

Theorem 3 (Termination [49]) *A program P is terminating iff there exists a d.wf. transition invariant for P.*

This result is applied in the Terminator[2] algorithm [18], which automates construction of d.wf. transition invariants. It starts with an empty termination condition $T = \emptyset$ and queries a safety checker for a counterexample—a computation that is not covered by the current termination condition T. Next, a ranking relation synthesis algorithm is used to obtain a termination argument T' covering the transitions in the counterexample. The termination argument is then updated as $T := T \cup T'$ and the algorithm continues to query for counterexamples. Finally, either a complete (d.wf.) transition invariant is constructed or there does not exist a ranking relation for some counterexample, in which case the program is reported as non-terminating.

5.3 Compositional termination analysis

Podelski and Rybalchenko [49] remarked an interesting fact regarding the compositionality (transitivity) of transition invariants: If T is transitive, it is enough to show that $T \supseteq R$ instead of $T \supseteq R^+$ to conclude termination, because a compositional and d.wf. transition invariant is well-founded, since it is an inductive transition invariant for itself [49]. Therefore, compositionality of a d.wf. transition invariant implies program termination.

To comply with the terminology in the existing literature, we define the notion of compositionality for transition invariants as follows:

[2]The Terminator algorithm is referred to as Binary Reachability Analysis (BRA), though BRA is only a particular technique to implement the algorithm (e.g., [21]).

Definition 9 (Compositional Transition Invariant [49, 40]) A d.wf. transition invariant T is called *compositional* if it is also transitive, or equivalently, closed under composition with itself, i.e., when $T \circ T \subseteq T$.

This useful property did not find its application in termination analysis until 2010. To understand its value we need to look closer at the transitive closure of program's transition relation R. The safety checker in the Terminator algorithm verifies that a candidate transition invariant T indeed includes R^+ restricted to the reachable states. Note that the (non-reflexive) transitive closure of R is essentially an unwinding of program loops:

$$R^+ = R \cup (R \circ R) \cup (R \circ R \circ R) \cup \ldots = \bigcup_{i=1}^{\infty} R^i \; .$$

Thus, instead of searching for a d.wf. transition invariant that is a superset of R^+, we can therefore decompose the problem into a series of smaller ones. We can consider a series of loop-free programs in which R is unwound k times, i.e., the program that contains the transitions in $R^1 \cup \ldots \cup R^k$. As was shown in [40], if there is a d.wf. T_k with $\bigcup_{j=1}^{k} R^j \subseteq T_k$ and T_k is also transitive, then T_k is a compositional transition invariant for P.

This idea results in an algorithm that constructs d.wf. relations T_i for incrementally deep unwindings of P until it finally finds a transitive T_k, which proves termination of P. In [40], this algorithm was named *Compositional Termination Analysis* (CTA).

5.4 From Terminator via CTA to a light-weight static analysis

Terminator is a complete algorithm (relative to completeness of the ranking procedure). Note that CTA is not even complete relative to the completeness of the ranking procedure for terminating programs even if they are finite-state. This is due to the fact that T is not guaranteed to ever become transitive, even if it contains R^+.

The Terminator strategy can be seen as a “proof by construction”: it explicitly builds the valid terminating argument for every path in a program. CTA combines “proof by construction” with a “proof by induction”: it first tries to construct a base step and then check the inductiveness. Inductive proofs are hard to find, and the implementation reported in [40] can only compute very simple inductive arguments. However, as it was shown in [40], for loops in industrial applications such as Windows device drivers, that CTA performs considerably better than Terminator.

This observation suggests an even more light-weight proof strategy—from a mix of “proof by construction” with “proof by induction” to a pure “proof by induction”: we propose to replace ranking synthesis-based transition invariant discovery with abstract domain-based transition invariant discovery. Instead of a complex base case, we “guess” several variants of the loop using lightweight static analysis methods and then check if the inductive argument happens to hold as well. This method is of course incomplete, but avoids expensive path enumeration inside the safety checker. We apply a variant of our loop summarization algorithm with specific *relational* domains for this purpose.

5.5 Loop summarization with transition invariants

We introduce a method that allows *transition* invariants to be included for strengthening of loop summaries. This increases the precision of the summaries by allowing loop termination to be taken into account.

According to Definition 2, a binary relation T is a transition invariant for a program P if it contains R^+ (restricted to the reachable states). Note, however, that transitivity of T is also a sufficient condition when T is only a superset of R:

Theorem 4 *A binary relation T is a transition invariant for the program $\langle S, I, R\rangle$ if it is transitive and $R \subseteq T$.*

Proof From transitivity of T it follows that $T^+ \subseteq T$. Since $R \subseteq T$ it follows that $R^+ \subseteq T$.

This simple fact allows for an integration of transition invariants into the loop summarization framework by a few adjustments to the original algorithm. Consider line 19 of Algorithm 1 (page 5), where candidate invariants are selected. Clearly, we need to allow selection of transition invariants here, i.e., invariant candidates now take the relational form $C(X, X')$, where X' is the post-state of a single iteration of L.

What follows is a check for invariance of C over $L(X, X')$, i.e., a single unwinding of the loop. Consider the temporary (sub-)program $\langle S, S, L\rangle$ to represent the execution of the loop from a non-deterministic entry state. A transition invariant for this program is required to cover L^+, which, according to Theorem 4, is implied by $L \subseteq C$ and transitivity of C. The original invariant check in IsTransitionInvariant establishes $L \subseteq C$, when the check for unsatisfiability receives the more general formula $L(X, X') \wedge C(X, X')$ as a parameter. The summarization procedure furthermore requires a slight change to include a check for compositionality. The resulting procedure is Algorithm 3.

5.6 Termination checks

The changes to the summarization algorithm allow for termination checks during summarization through application of Theorem 3, which requires a transition invariant to be disjunctively well-founded. This property may be established by allowing only disjunctively well-founded invariant candidates, or it may be checked by means of decision procedures (e.g., SMT solvers where applicable).

According to Definition 8, the well-foundedness of each of the disjuncts of a candidate relation T must be established in order to ensure that it is d.wf. This can be done by an explicit encoding of the well-foundedness criteria given in Definition 7. However, the resulting formula contains quantifiers. As a consequnce, the obtained decision problem is frequently beyond the capabilities of state-of-the-art solvers.

Algorithm 3: Loop summarization with transition invariants.

```
SummarizeLoop-TI($L$)
input: Single-loop program $L$ with a set of variables $X$
output: Loop summary
begin
    $T := \top$
    foreach Candidate $C$ in PickInvariantCandidates($L$) do
        if IsTransitionInvariant($L$, $C$) $\wedge$ IsCompositional($C$) then
            $T := T \wedge C$
        endif
    end foreach
    return "$X^{pre} := X$; havoc($X$); assume($T(X^{pre}, X)$);"
end

IsTransitionInvariant($L$, $C$)
input: Single-loop program $L$ (with entry state $X$ and post-iteration state
    $X'$), invariant candidate $C$
output: TRUE if $C$ is a transition invariant for $L$; FALSE otherwise
begin
    return Unsat($\neg(L(X, X') \wedge C(X, X') \Rightarrow C(X, X'))$)
end

IsCompositional($C$)
input: Invariant candidate $C(X, X')$
output: TRUE if $C$ is compositional; FALSE otherwise
begin
    return Unsat($\neg\left(C(X, Y) \wedge C(Y, X') \Rightarrow C(X, X')\right)$)
end
```

5.7 The difference between Terminator, CTA and loop summarization-based termination

The complexity of establishing well-foundedness of a transition invariant hints at the explanation of a major difference between our new algorithm and Terminator/CTA. The latter construct the transition invariant using the abstraction-refinement loop such that it is already disjunctively well-founded, while we allow any transition invariant to be discovered, though, later it needs to be checked for well-foundedness. Note that even if the discovered transition invariant is not well-founded, it is still a valid transition invariant and can therefore be used to improve the precision of summaries.

However, the research in size-change termination for functional languages [5][3] suggests that a small set of templates for ranking relations is enough to cover many classes of programs. Besides, the expensive well-foundedness check can be completely omitted if we employ specialized abstract domains that produce only

[3]We discuss the relation of our method to size-change termination in Section 7.2.1.

well-founded candidates for transition invariants. This is the approach we take in the following section.

Example 6 Consider the program in Figure 2. The symbolic transformer for the loop body is: $\phi_L := x' = x + 1$. Also consider the relation "$>$" for a pair x' and x as a candidate relation T_c. T_c is indeed a transition invariant if the following formula is unsatisfiable:

$$x < 255 \land x' = x + 1 \land \neg(x' > x) .$$

```
int x = 0;
while(x<255)
  x++;
```

Fig. 2: An example of a terminating loop with a strictly increasing iterator

The formula is UNSAT, i.e., the invariant holds, and $x' > x$ is added to the symbolic transformer as a transition invariant. Since the relation is compositional and d.wf. (we explain the reason for this later), the loop is marked as terminating.

5.8 Invariant candidate selection

We now propose a set of specialized candidate relations, which we find useful in practice, as demonstrated in the following section. We focus on transition invariants for machine-level integers (i.e., finite integers with overflows) for a bit-precise analysis of programs implemented in low-level languages such as ANSI-C.

In contrast to other work on termination proving with abstract domains (e.g., [6]), we do not aim for general domains like Octagons and Polyhedra. Although fast in computation, they are not designed for termination and d.wf. and compositionality checks for them can be costly. Instead we focus on domains that

- generate few, relatively simple candidate relations;
- allow for efficient d.wf. and compositionality checks.

Arithmetic operations on machine-level integers usually allow overflows, e.g., the instruction $i = i+1$ for a pre-state $i = 2^k - 1$ results in a post-state $i' = -2^{k-1}$ (when represented in two's-complement). If termination of the loop depends on machine-level integers, establishing well-foundedness of a relation over it is not straightforward, as increasing/decreasing sequences of numbers of this kind can be affected by overflow/underflow. However, we can use the following theorem to simplify the discovery of a d.wf. transition invariant.

Theorem 5 *If $T : K \times K$ is a strict order relation for a finite set $K \subseteq S$ and is a transition invariant for the program $\langle S, I, R\rangle$, then T is well-founded.*

Proof If T is a transition invariant, then for all pairs $(k_1, k_2) \in K \times K$. Thus, it is total over K. Non-empty finite totally-ordered sets always have a least element and, therefore, T is well-founded.

The proof uses the fact that, when checking T for being a transition invariant, we implicitly enumerated all the pairs of pre- and post-states to discover if any of them violates the order.

A total strict-order relation is also transitive, which allows for an alternative (stronger) criterion than Theorem 3:

Corollary 2 *A program terminates if it has a transition invariant T that is also a finite strict-order relation.*

This corollary allows for a selection of invariant candidates that ensures (disjunctive) well-foundedness of transition invariants. An explicit check is therefore not required. An example of such a candidate appears in Example 6.

Note that strictly ordered and finite transition invariants exist for many programs in practice: machine-level integers or strings of fixed length have a finite number of possible distinct pairs and strict natural or lexicographical orders are defined for them as well.

6 Experimental Evaluation

This section first presents LOOPFROG—a tool that implements loop summarization and serves as a basis for our experiments. Next, we evaluate the implementation to detect buffer overflows in ANSI-C programs. We finally demonstrate the applicability of loop summarization to termination analysis.

6.1 The LOOPFROG tool

The theoretical concept of symbolic abstract transformers is implemented and put to use by our tool LOOPFROG. Its architecture is outlined in Figure 3. As input, LOOPFROG receives a model file, extracted from software sources by GOTO-CC[4]. This model extractor features full ANSI-C support and simplifies verification of software projects that require complex build systems. It mimics the behavior of the compiler, and thus ‘compiles’ a model file using the original settings and options. Switching from compilation mode to verification mode is thus frequently achieved by changing a single option in the build system. As suggested by Figure 3, all other steps are fully automated.

The resulting model contains a control flow graph and a symbol table, i.e., it is an intermediate representation of the original program in a single file. For calls to system library functions, abstractions containing assertions (pre-condition checks) and assumptions (post-conditions) are inserted. Note that the model also can contain the properties to be checked in the form of assertions (calls to the ASSERT function).

Preprocessing The model, instrumented with assertions, is what is passed to the first stage of LOOPFROG. In this preprocessing stage, the model is adjusted in various ways to increase performance and precision. First, irreducible control flow graphs are rewritten according to an algorithm due to Ashcroft and Manna [2]. Like a compiler, LOOPFROG inlines small functions, which increases the model size, but also improves the precision of subsequent analysis. Thereafter, it runs a field-sensitive pointer analysis. The information obtained this way is used to generate assertions over pointers, and to eliminate pointer variables in the program where possible. LOOPFROG automatically adds assertions to verify the correctness of pointer operations, array bounds, and arithmetic overflows.

[4]`http://www.cprover.org/goto-cc/`

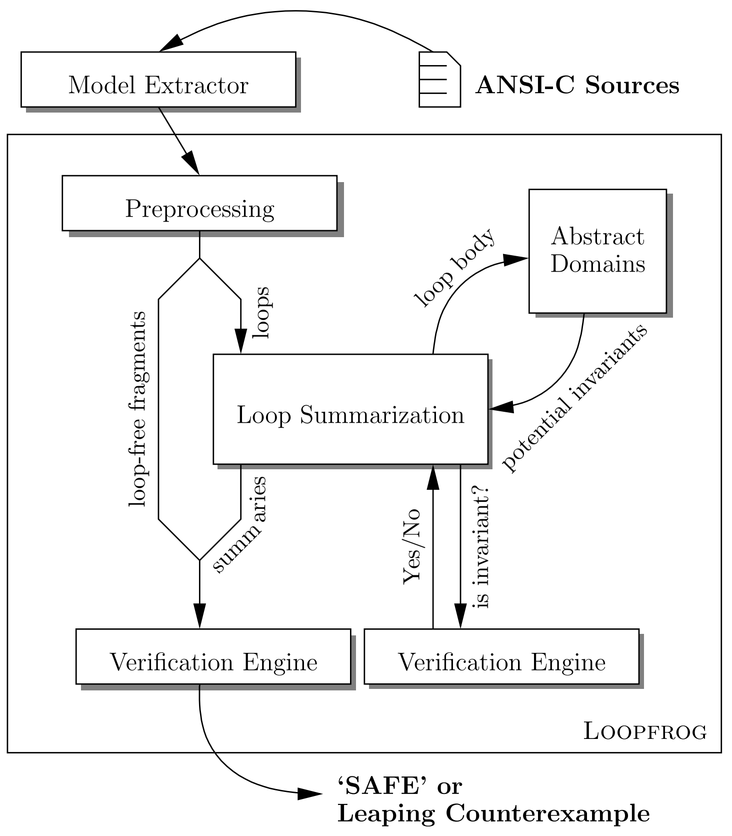


Fig. 3: Architecture of Loopfrog

Loop summarization Once the preprocessing is finished, Loopfrog starts to replace loops in the program with summaries. These are shorter, loop-free program fragments that over-approximate the original program behavior. To accomplish this soundly, all loops are replaced with a loop-free piece of code that "havocs" the program state, i.e., it resets all variables that may be changed by the loop to unknown values. Additionally, a copy of the loop body is kept, such that assertions within the loop are preserved.

While this is already enough to prove some simple properties, much higher precision is required for more complex ones. As indicated in Fig. 3, Loopfrog makes use of predefined abstract domains to achieve this. Every loop body of the model is passed to a set of abstract domains, through each of which a set of potential invariants of the loop is derived (heuristically).

The choice of the abstract domain for the loop summarization has a significant impact on the performance of the algorithm. A carefully selected domain generates fewer invariant candidates and thus speeds up the computation of a loop summary. The abstract domain has to be sufficiently expressive to retain enough of the semantics of the original loop to show the property.

#	**Constraint**	**Meaning**
1	ZT_s	String s is zero-terminated
2	$L_s < B_s$	Length of s (L_s) is less than the size of the allocated buffer (B_s)
3	$0 \leq i \leq L_s$	Bounds on integer variables i (i is non-negative, i is bounded by buffer size, etc.) k is an arbitrary integer constant.
4	$0 \leq i$	
5	$0 \leq i < B_s$	
6	$0 \leq i < B_s - k$	
7	$0 < \mathit{offset}(p) \leq B_s$	Pointer offset bounds
8	$\mathit{valid}(p)$	Pointer p points to a valid object

Table 1: Examples of abstract domains tailored to buffer-overflow analysis.

Checking invariant candidates All potential invariants obtained from abstract domains always constitute an abstract (post-)state of a loop body, which may or may not be correct in the original program. To ascertain that a potential invariant is an actual invariant, Loopfrog makes use of a verification engine. In the current version, the symbolic execution engine of CBMC [14] is used. This engine allows for bit-precise, symbolic reasoning without abstraction. In our context, it always gives a definite answer, since only loop-free program fragments are passed to it. It is only necessary to construct an intermediate program that assumes a potential invariant to be true, executes a loop body once and then checks if the potential invariant still holds. If the verification engine returns a counterexample, we know that a potential invariant does not hold; in the opposite case it can be a loop invariant and it is subsequently added to a loop summary, since even after the program state is havoced, the invariant still holds. Loopfrog starts this process from an innermost loop, and thus there is never an intermediate program that contains a loop. In case of nested loops, the inner loop is replaced with a summary before the outer loop is analyzed. Owing to this strategy and the small size of fragments checked (only a loop body), small formulas are given to the verification engine and an answer is obtained quickly.

Verifying the abstraction The result, after all loops have been summarized, is a loop-free abstraction of the input program. This abstract model is then handed to a verification engine once again. The verification time is much lower than that required for the original program, since the model does not contain loops. As indicated by Fig. 3, the verification engine used to check the assertions in the abstract model may be different from the one used to check potential invariants. In Loopfrog, we choose to use the same engine (CBMC).

6.2 An abstract domain for safety analysis of string-manipulating programs

In order to demonstrate the benefits of our approach to static analysis of programs with buffer overflows, the first experiments with Loopfrog were done with a set of abstract domains that are tailored to buffer-related properties. The constrains of the domains are listed in Table 1.

We also make use of string-related abstract domains instrumented into the model similar to the approach by Dor et al. [25]: for each string buffer s, a Boolean value z_s and integers l_s and b_s are tracked. The Boolean z_s holds if s contains the zero character within the buffer size b_s. If so, l_s is the index of the first zero character, otherwise, l_s has no meaning.

The chosen domains are instantiated according to variables occurring in the code fragment taken into account. To lower the number of template instantiations, the following simple heuristics can be used:

1. Only variables of appropriate type are considered (we concentrate on string types).
2. Indices and string buffers are combined in one invariant only if they are used in the same expression, i.e., we detect instructions which contain $p[i]$ and build invariants that combine i with all string buffers pointed to by p.

As shown in the next section these templates have proven to be effective in our experiments. Other applications likely require different abstract domains. However, new domain templates may be added quite easily: they usually can be implemented with less than a hundred lines of code.

6.3 Evaluation of loop summarization applied to static analysis of buffer-intensive programs

In this set of experiments we focus on ANSI-C programs: the extensive buffer manipulations in programs of this kind often give rise to buffer overruns. We apply the domains from Table 1 to small programs collected in benchmarks suites and to real applications as well. All data was obtained on an 8-core Intel Xeon with 3.0 GHz. We limited the run-time to 4 hours and the memory per process to 4 GB. All experimental data, an in-depth description of Loopfrog, the tool itself, and all our benchmark files are available on-line for experimentation by other researchers[5].

6.3.1 Evaluation on the benchmark suites

The experiments are performed on two recently published benchmark sets. The first one, by Zitser et al. [59], contains 164 instances of buffer overflow problems, extracted from the original source code of `sendmail`, `wu-ftpd`, and `bind`. The test cases do not contain complete programs, but only those parts required to trigger the buffer overflow. According to Zitser et al., this was necessary because the tools in their study were all either unable to parse the test code, or the analysis used disproportionate resources before terminating with an error ([59], pg. 99). In this set, 82 tests contain a buffer overflow, and the rest represent a fix of a buffer overflow.

We use metrics proposed by Zitser et al. [59] to evaluate and compare the precision of our implementation. We report the *detection rate* $R(d)$ (the percentage of correctly reported bugs) and the *false positive rate* $R(f)$ (the percentage of incorrectly reported bugs in the fixed versions of the test cases). The *discrimination*

[5]`http://www.cprover.org/loopfrog/`

rate $R(\neg f|d)$ is defined as the ratio of test cases on which an error is correctly reported, while it is, also correctly, not reported in the corresponding fixed test case. Using this measure, tools are penalized for not finding a bug, but also for not reporting a fixed program as safe.

The results of a comparison with a wide selection of static analysis tools[6] are summarized in Table 2. Almost all of the test cases involve array bounds violations. Even though Uno, Archer and BOON were designed to detect these type of bugs, they hardly report any errors. BOON abstracts all string manipulation using a pair of integers (number of allocated and used bytes) and performs flow-insensitive symbolic analysis over collected constraints. The three tools implement different approaches for the analysis. BOON and Archer perform a symbolic analysis while UNO uses Model Checking. Archer and UNO are flow-sensitive, BOON is not. All three are interprocedural. We observe that all three have a common problem—the approximation is too coarse and additional heuristics are applied in order to lower the false positive rate; as a result, only few of the complex bugs are detected. The source code of the test cases was not annotated, but nevertheless, the annotation-based Splint tool performs reasonably well on these benchmarks. LOOPFROG and the implementation of the Interval Domain are the only entrants that report all buffer overflows correctly (a detection rate of $R(d) = 1$). With 62%, LOOPFROG also has the highest discrimination rate among all the tools. It is also worth to note that our summarization technique performs quite well when only few relational domains are used (the second line of Table 2). The third line in this table contains the data for a simple interval domain, not implemented in LOOPFROG, but as a abstract domain used in SATABS model checker as a part of pre-processing; it reports almost all checks as unsafe.

	$R(d)$	$R(f)$	$R(\neg f\|d)$
LOOPFROG	1.00	0.38	0.62
$=, \neq, \leq$	1.00	0.44	0.56
Interval Domain	1.00	0.98	0.02
Polyspace	0.87	0.50	0.37
Splint	0.57	0.43	0.30
Boon	0.05	0.05	0
Archer	0.01	0	0
Uno	0	0	0
LOOPFROG [41]	1.00	0.26	0.74
$=, \neq, \leq$[41]	1.00	0.46	0.54

Table 2: Effectiveness of various static analysis tool in Zitser et al. [59] and Ku et al. [41] benchmarks: detection rate $R(d)$, false positive rate $R(f)$, and discrimination rate $R(\neg f|d)$.

The second set of benchmarks was proposed by Ku et al. [41]. It contains 568 test cases, of which 261 are fixed versions of buffer overflows. This set partly overlaps with the first one, but contains source code of a greater variety of applications, including the Apache HTTP server, Samba, and the NetBSD C system library. Again, the test programs are stripped down, and are partly simplified to enable current model checkers to parse them. Our results on this set confirm the results obtained using the first set; the corresponding numbers are given in the last two lines of Table 2. On this set the advantage of selecting property-specific domains is clearly visible, as a 20% increase in the discrimination rate over the simple relational domains is witnessed. Also, the performance of LOOPFROG is much better

[6]The data for all tools but LOOPFROG, "$=, \neq, \leq$", and the Interval Domain is from [59].

				Time				Assertions		
Suite	**Program**	**Instructions**	**# Loops**	**Summari-zation**	**Checking Assertions**	**Total**	**Peak Memory**	**Total**	**Passed**	**Violated**
freecell-solver	aisleriot-board-2.8.12	347	26	10s	295s	305s	111MB	358	165	193
freecell-solver	gnome-board-2.8.12	208	8	0s	3s	4s	13MB	49	16	33
freecell-solver	microsoft-board-2.8.12	168	4	2s	9s	11s	32MB	45	19	26
freecell-solver	pi-ms-board-2.8.12	185	4	2s	10s	13s	33MB	53	27	26
gnupg	make-dns-cert-1.4.4	232	5	0s	0s	1s	9MB	12	5	7
gnupg	mk-tdata-1.4.4	117	1	0s	0s	0s	3MB	8	7	1
inn	encode-2.4.3	155	3	0s	2s	2s	6MB	88	66	22
inn	ninpaths-2.4.3	476	25	5s	40s	45s	49MB	96	47	49
ncompress	compress-4.2.4	806	12	45s	4060s	4106s	345MB	306	212	94
texinfo	ginstall-info-4.7	1265	46	21s	326s	347s	127MB	304	226	78
texinfo	makedoc-4.7	701	18	9s	6s	16s	28MB	55	33	22
texinfo	texindex-4.7	1341	44	415s	9336s	9757s	1021MB	604	496	108
wu-ftpd	ckconfig-2.5.0	135	0	0s	0s	0s	3MB	3	3	0
wu-ftpd	ckconfig-2.6.2	247	10	13s	43s	57s	27MB	53	10	43
wu-ftpd	ftpcount-2.5.0	379	13	10s	32s	42s	37MB	115	41	74
wu-ftpd	ftpcount-2.6.2	392	14	8s	24s	32s	39MB	118	42	76
wu-ftpd	ftprestart-2.6.2	372	23	48s	232s	280s	55MB	142	31	111
wu-ftpd	ftpshut-2.5.0	261	5	1s	9s	10s	13MB	83	29	54
wu-ftpd	ftpshut-2.6.2	503	26	27s	79s	106s	503MB	232	210	22
wu-ftpd	ftpwho-2.5.0	379	13	7s	23s	30s	37MB	115	41	74
wu-ftpd	ftpwho-2.6.2	392	14	8s	27s	35s	39MB	118	42	76
wu-ftpd	privatepw-2.6.2	353	9	4s	17s	22s	32MB	80	51	29

Table 3: Large-scale evaluation of LOOPFROG on the programs from wu-ftpd, texinfo, gnupg, inn, and freecell-solver tools suites.

if specialized domains are used, simply because there are fewer candidates for the invariants.

The leaping counterexamples computed by our algorithm are a valuable aid in the design of new abstract domains that decrease the number of false positives. Also, we observe that both test sets include instances labeled as unsafe that LOOPFROG reports to be safe (1 in [59] and 9 in [41]). However, by manual inspection of the counterexamples for these cases, we find that our tool is correct, i.e., that the test cases are spurious.[7] For most of the test cases in the benchmark suites, the time and memory requirements of LOOPFROG are negligible. On average, a test case finishes within a minute.

6.3.2 Evaluation on real programs

We also evaluated the performance of LOOPFROG on a set of large-scale benchmarks, that is, complete un-modified program suites. Table 3 contains a selection of the results.

These experiments clearly show that the algorithm scales reasonably well in both memory and time, depending on the program size and the number of loops contained. The time required for summarization naturally depends on the complexity of the program, but also to a large degree on the selection of (potential) invariants. As experience has shown, unwisely chosen invariant templates may generate many useless potential invariants, each requiring a test by the SAT-solver.

In general, the results regarding the program assertions shown to hold are not surprising; for many programs (e.g., texindex, ftpshut, ginstall), our selection of

[7] We exclude those instances from our benchmarks.

			Loopfrog		Interval Domain	
Suite	Benchmark	Total	Failed	Ratio	Failed	Ratio
bchunk	bchunk	96	8	0.08	34	0.35
freecell-solver	make-gnome-freecell	145	40	0.28	140	0.97
freecell-solver	make-microsoft-freecell	61	30	0.49	58	0.95
freecell-solver	pi-make-microsoft-freecell	65	30	0.46	58	0.89
gnupg	make-dns-cert	19	5	0.26	19	1.00
gnupg	mk-tdata	6	0	0.00	6	1.00
inn	encode	42	11	0.26	38	0.90
inn	ninpaths	56	19	0.34	42	0.75
ncompress	compress	204	38	0.19	167	0.82
texinfo	makedoc	83	46	0.55	83	1.00
wu-ftpd	ckconfig	1	1	1.00	1	1.00
wu-ftpd	ftpcount	61	7	0.11	47	0.77
wu-ftpd	ftpshut	63	13	0.21	63	1.00
wu-ftpd	ftpwho	61	7	0.11	47	0.77

Table 4: Comparison between Loopfrog and an interval domain: The column labeled 'Total' indicates the number of properties in the program, and 'Failed' shows how many of the properties were reported as failing; 'Ratio' is Failed/Total.

string-specific domains proved to be quite useful. It is also interesting to note that the results on the ftpshut program are very different on program versions 2.5.0 and 2.6.2: This program contains a number of known buffer-overflow problems in version 2.5.0, and considerable effort was spent on fixing these bugs for the 2.6.2 release; an effort clearly reflected in our statistics. Just like in this benchmark, many of the failures reported by Loopfrog correspond to known bugs and the leaping counterexamples we obtain allow us to analyze those faults. Merely for reference we list CVE-2001-1413 (a buffer overflow in ncompress) and CVE-2006-1168 (a buffer underflow in the same program), for which we are easily able to produce counterexamples.[8] On the other hand, some other programs (such as the ones from the freecell-solver suite) clearly require different abstract domains, suitable for heap structures other than strings. The development of suitable domains and subsequent experiments, however, are left for future research.

6.3.3 Comparison with the interval domain

To highlight the applicability of Loopfrog to large-scale software and to demonstrate its main advantage, we present a comparative evaluation against a simple interval domain, which tracks the bounds of buffer index variables, a technique often employed in static analysers. For this experiment, Loopfrog was configured to use only two abstract domains, which capture the fact that an index is within the buffer bounds (#4 and #5 in Table 1). As apparent from Table 4, the precision of Loopfrog in this experiment is far superior to that of the simple interval analysis.

To evaluate scalability, we applied other verification techniques to this example. CBMC [14] tries to unwind all the loops, but fails, reaching the 2 GB memory limit. The same behavior is observed using SatAbs [16], where the underlying model checker (SMV) hits the memory limit.

[8]The corresponding bug reports may be obtained from `http://cve.mitre.org/`.

#	**Constraint**	**Meaning**
1	$i' < i$ $i' > i$	A numeric variable i is strictly decreasing (increasing).
2	$x' < x$ $x' > x$	Any loop variable x is strictly decreasing (increasing).
3	$sum(x', y') < sum(x, y)$ $sum(x', y') > sum(x, y)$	Sum of all numeric loop variables is strictly decreasing (increasing).
4	$max(x', y') < max(x, y)$ $max(x', y') > max(x, y)$ $min(x', y') < min(x, y)$ $min(x', y') > min(x, y)$	Maximum or minimum of all numeric loop variables is strictly decreasing (increasing).
5	$(x' < x \wedge y' = y)\vee$ $(x' > x \wedge y' = y)\vee$ $(y' < y \wedge x' = x)\vee$ $(y' > y \wedge x' = x)$	A combination of strict increase or decrease for one of loop variables while the remaining ones are not updated.

Table 5: Templates of abstract domains used to draw transition invariant candidates

6.4 Evaluation of loop summarization applied to termination analysis

For a proof of concept we have implemented loop termination analysis within our static analyzer Loopfrog. As before, the tool operates on the program models produced by Goto-CC model extractor; ANSI-C programs are the primary experimental target.

We implemented a number of domains based on strict-order numeric relations, thus, following Corollary 2, additional checks for compositionality and d.wf.-ness of candidate relations are not required. The domains are listed in Table 5. Here we report the results for the two most illustrative schemata:

- Loopfrog 1: domain #3 in Table 5. Expresses the fact that a sum of all numeric variables of a loop is strictly decreasing (increasing). This is the fastest approach, because it generates very few (but large) invariant candidates per loop.
- Loopfrog 2: domain #1 in Table 5. Expresses that numeric variables are strictly decreasing (increasing). Generates twice as many simple strict-order relations as there are variables in a loop.

As a reference point, we used a termination prover built upon the CBMC and SatAbs [16] framework. This tool implements Compositional Termination Analysis (CTA) [40] and the Binary Reachability Analysis used in the Terminator algorithm [17]. For both the default ranking function synthesis methods were enabled—templates for relations on bit-vectors with SAT-based enumeration of coefficients; for more details see [21].

We experimented with a large number of ANSI-C programs including:

Benchmark	Method	T	NT	TO	Time	
adpcm 11 loops	Loopfrog 1	8	3	0	59.66	
	Loopfrog 2	10	1	0	162.75	
	CTA	8	3	0	101.30	
	Terminator	6	2	3	94.45	+
bcnt 2 loops	Loopfrog 1	0	2	0	2.63	
	Loopfrog 2	0	2	0	2.82	
	CTA	0	2	0	0.79	
	Terminator	0	2	0	0.30	
blit 4 loops	Loopfrog 1	0	4	0	0.16	
	Loopfrog 2	3	1	0	0.05	
	CTA	3	1	0	5.95	
	Terminator	3	1	0	3.67	
compress 18 loops	Loopfrog 1	5	13	0	3.13	
	Loopfrog 2	6	12	0	33.92	
	CTA	5	12	1	699.00	+
	Terminator	7	10	1	474.36	+
crc 3 loops	Loopfrog 1	1	2	0	0.15	
	Loopfrog 2	2	1	0	0.21	
	CTA	1	1	1	0.33	+
	Terminator	2	1	0	14.58	
engine 6 loops	Loopfrog 1	0	6	0	2.40	
	Loopfrog 2	2	4	0	9.88	
	CTA	2	4	0	16.20	
	Terminator	2	4	0	4.88	
fir 9 loops	Loopfrog 1	2	7	0	5.99	
	Loopfrog 2	6	3	0	21.59	
	CTA	6	3	0	2957.06	
	Terminator	6	2	1	193.91	+
g3fax 7 loops	Loopfrog 1	1	6	0	1.57	
	Loopfrog 2	1	6	0	6.05	
	CTA	1	5	1	256.90	+
	Terminator	1	5	1	206.85	+
huff 11 loops	Loopfrog 1	3	8	0	24.37	
	Loopfrog 2	8	3	0	94.61	
	CTA	7	3	1	16.35	+
	Terminator	7	4	0	52.32	
jpeg 23 loops	Loopfrog 1	2	21	0	8.37	
	Loopfrog 2	16	7	0	32.90	
	CTA	15	8	0	2279.13	
	Terminator	15	8	0	2121.36	
pocsag 12 loops	Loopfrog 1	3	9	0	2.07	
	Loopfrog 2	9	3	0	6.91	
	CTA	9	3	0	10.39	
	Terminator	7	3	2	1557.57	+
qurt 2 loops	Loopfrog 1	0	2	0	3.56	
	Loopfrog 2	1	1	0	11.67	
	CTA	1	1	0	30.77	
	Terminator	0	0	2	0.00	
ucbqsort 15 loops	Loopfrog 1	1	14	0	0.79	
	Loopfrog 2	2	13	0	2.06	
	CTA	2	12	1	71.73	+
	Terminator	9	5	1	51.08	+
v42 12 loops	Loopfrog 1	0	12	0	82.84	
	Loopfrog 2	0	12	0	2587.22	
	CTA	0	12	0	73.57	
	Terminator	1	11	0	335.69	

Table 6: Powerstone benchmark suite

Columns 3 to 5 state number of loops proven to terminate (T), possibly non-terminate (NT) and time-out (TO) for each benchmark. Time is computed only for loops noted in T and NT; '+' is used to denote testcases cases where at least one time-outed loop occurred.

Benchmark	**Method**	**T**	**NT**	**TO**	**Time**	
adpcm-test 18 loops	Loopfrog 1	13	5	0	470.05	
	Loopfrog 2	17	1	0	644.09	
	CTA	13	3	2	260.98	+
	Terminator	12	2	4	165.67	+
bs 1 loop	Loopfrog 1	0	1	0	0.05	
	Loopfrog 2	0	1	0	0.12	
	CTA	0	1	0	12.22	
	Terminator	0	1	0	18.47	
crc 3 loops	Loopfrog 1	1	2	0	0.17	
	Loopfrog 2	2	1	0	0.26	
	CTA	1	1	1	0.21	+
	Terminator	2	1	0	13.88	
fft1k 7 loops	Loopfrog 1	2	5	0	0.36	
	Loopfrog 2	5	2	0	0.67	
	CTA	5	2	0	141.18	
	Terminator	5	2	0	116.81	
fft1 11 loops	Loopfrog 1	3	8	0	3.68	
	Loopfrog 2	7	4	0	4.98	
	CTA	7	4	0	441.94	
	Terminator	7	4	0	427.36	
fir 8 loops	Loopfrog 1	2	6	0	2.90	
	Loopfrog 2	6	2	0	8.48	
	CTA	6	2	0	2817.08	
	Terminator	6	1	1	236.70	+
insertsort 2 loops	Loopfrog 1	0	2	0	0.05	
	Loopfrog 2	1	1	0	0.06	
	CTA	1	1	0	226.45	
	Terminator	1	1	0	209.12	
jfdctint 3 loops	Loopfrog 1	0	3	0	5.61	
	Loopfrog 2	3	0	0	0.05	
	CTA	3	0	0	1.24	
	Terminator	3	0	0	0.98	
lms 10 loops	Loopfrog 1	3	7	0	2.86	
	Loopfrog 2	6	4	0	10.49	
	CTA	6	4	0	2923.12	
	Terminator	6	3	1	251.03	+
ludcmp 11 loops	Loopfrog 1	0	11	0	96.73	
	Loopfrog 2	5	6	0	112.81	
	CTA	3	5	3	3.26	+
	Terminator	3	8	0	94.66	
matmul 5 loops	Loopfrog 1	0	5	0	0.15	
	Loopfrog 2	5	0	0	0.09	
	CTA	3	2	0	1.97	
	Terminator	3	2	0	2.15	
minver 17 loops	Loopfrog 1	1	16	0	2.57	
	Loopfrog 2	16	1	0	7.66	
	CTA	14	1	2	105.26	+
	Terminator	14	1	2	87.09	+
qsort-exam 6 loops	Loopfrog 1	0	6	0	0.67	
	Loopfrog 2	0	6	0	3.96	
	CTA	0	5	1	45.92	+
	Terminator	0	5	1	2530.58	+
qurt 1 loop	Loopfrog 1	0	1	0	8.02	
	Loopfrog 2	1	0	0	13.82	
	CTA	1	0	0	55.65	
	Terminator	0	0	1	0.00	
select 4 loops	Loopfrog 1	0	4	0	0.55	
	Loopfrog 2	0	4	0	3.56	
	CTA	0	3	1	32.60	+
	Terminator	0	3	1	28.12	+
sqrt 1 loop	Loopfrog 1	0	1	0	0.60	
	Loopfrog 2	1	0	0	5.10	
	CTA	1	0	0	15.28	
	Terminator	0	0	1	0.00	

Table 7: SNU real-time benchmarks suite

Benchmark	Method	T	NT	TO	Time
jhead 8 loops	Loopfrog 1	1	7	0	23.78
	Loopfrog 2	4	4	0	78.93
	CTA	3	5	0	42.38
	Terminator	2	4	2	208.78 +

Table 8: Jhead-2.6 utility

	Method	T	NT	TO	Time
244 loops in 160 benchmarks	Loopfrog 1	33	211	0	11.38
	Loopfrog 2	44	200	0	22.49
	CTA	34	208	2	1207.62 +
	Terminator	40	204	0	4040.53

Table 9: Aggregated data on Verisec 0.2 suite

Benchmark	Method	T	NT	TO	Time
bchunk 9 loops	Loopfrog 1	3	6	0	1.67
	Loopfrog 2	3	6	0	31.16
	CTA	3	6	0	53.03
	Terminator	4	5	0	91.13

Table 10: Bchunk 1.2.0 utility

– The SNU real-time benchmark suite that contains small C programs used for worst-case execution time analysis[9];
– The Powerstone benchmark suite as an example set of C programs for embedded systems [53];
– The Verisec 0.2 benchmark suite [41];
– The Jhead 2.6 utility;
– The Bchunk 1.2.0 utility;
– Windows device drivers (from the Windows Device Driver Kit 6.0).

All experiments were run on an Ubuntu server equipped with Dual-Core 2 GHz Opteron 2212 CPU and 4 GB of memory. The analysis was set to run with a timeout of 120 minutes for all loops at once (Loopfrog) or of 60 minutes per loop (CTA and Terminator).

The results for Powerstone, SNU, Jhead and Bchunk are presented in Tables 6, 7, 8 and 10. Each table in columns 3 to 5 reports the quantity of loops that were proven as terminating (T), potentially non-terminating (NT) and time-out (TO) for each of the compared techniques.

The time in column 6 is computed only for loops noted in T and NT; loops with timeout are not included in the total time. Instead, '+' is used to denote the cases where at least one time-out occurred.

The results for the Verisec 0.2 benchmark suite are given in aggregated form in Table 9. The suite consists of a large number of stripped C programs that correspond to known security bugs. Although each program has very few loops, the variety of loop types is fairly broad and, thus, is interesting for analysis.

[9]`http://archi.snu.ac.kr/realtime/benchmark/`

Benchmark group	**Method**	**T**	**NT**	**TO**	**Time**
SDV FLAT DISPATCH HARNESS 557 loops in 30 benchmarks	Loopfrog 1	135	389	33	1752.1
	Loopfrog 2	215	201	141	10584.4
	CTA	166	160	231	25399.5
SDV FLAT DISPATCH STARTIO HARNESS 557 loops in 30 benchmarks	Loopfrog 1	135	389	33	1396.0
	Loopfrog 2	215	201	141	9265.8
	CTA	166	160	231	28033.3
SDV FLAT HARNESS 635 loops in 45 benchmarks	Loopfrog 1	170	416	49	1323.0
	Loopfrog 2	239	205	191	6816.4
	CTA	201	186	248	31003.2
SDV FLAT SIMPLE HARNESS 573 loops in 31 benchmarks	Loopfrog 1	135	398	40	1510.0
	Loopfrog 2	200	191	182	6814.0
	CTA	166	169	238	30292.7
SDV HARNESS DRIVER CREATE 9 loops in 5 benchmarks	Loopfrog 1	1	8	0	0.1
	Loopfrog 2	1	8	0	0.2
	CTA	1	8	0	151.8
SDV HARNESS PNP DEFERRED IO REQUESTS 177 loops in 31 benchmarks	Loopfrog 1	22	98	57	47.9
	Loopfrog 2	66	54	57	617.4
	CTA	80	94	3	44645.0
SDV HARNESS PNP IO RE-QUESTS 173 loops in 31 benchmarks	Loopfrog 1	25	94	54	46.6
	Loopfrog 2	68	51	54	568.7
	CTA	85	86	2	15673.9
SDV PNP HARNESS SMALL 618 loops in 44 benchmarks	Loopfrog 1	172	417	29	8209.5
	Loopfrog 2	261	231	126	12373.2
	CTA	200	177	241	26613.7
SDV PNP HARNESS 635 loops in 45 benchmarks	Loopfrog 1	173	426	36	7402.2
	Loopfrog 2	261	230	144	13500.2
	CTA	201	186	248	41566.6
SDV PNP HARNESS UNLOAD 506 loops in 41 benchmarks	Loopfrog 1	128	355	23	8082.5
	Loopfrog 2	189	188	129	13584.6
	CTA	137	130	239	20967.8
SDV WDF FLAT SIMPLE HAR-NESS 172 loops in 18 benchmarks	Loopfrog 1	27	125	20	30.3
	Loopfrog 2	61	91	20	202.0
	CTA	73	95	4	70663.0

Table 11: Aggregated data of the comparison between Loopfrog and CTA on Windows device drivers

The aggregated data on experiments with Windows device drivers is provided in Table 11. The benchmarks are grouped according to the harness used upon extraction of a model with Goto-CC. Note that we skip the benchmarks where no loops are detected. Thus, the groups in Table 11 may differ in the numbers of benchmarks/loops. Furthermore, we do not report Terminator results here, as it was shown in [40] that CTA outperforms it on this benchmark set.

Discussion Note that direct comparison of Loopfrog in time with iterative techniques like CTA and Terminator is not fair. The latter methods are complete at least for finite-state programs, relative to the completeness of ranking synthesis method (which is not complete by default in the current CTA/Terminator implementation for scalability reasons). Our loop summarization technique on the other hand is a static analysis which aims only for conservative abstractions. In particular, it does not try to prove unreachability of a loop or of preconditions that lead to non-termination.

The timing information provided here serves as a reference that allows to compare efforts of achieving the same result. Note that:

- Loopfrog spends time enumerating invariant candidates, provided by the chosen abstract domain, against a path of one loop iteration. Compositionality and d.wf. checks are not required for the chosen domains.
- CTA spends time 1) unwinding loop iterations, 2) discovering a ranking function for each unwounded path and 3) checking compositionality of a discovered relation.
- Terminator spends time 1) enumerating all paths through the loop and 2) discovering a ranking function for each path.

The techniques can greatly vary in time of dealing with a particular loop/program. CTA and Terminator give up on a loop once a they hit a path on which ranking synthesis fails. Loopfrog gives up on a loop if it runs out of transition invariant candidates to try. In a few tests this leads to an advantage for Terminator (`huff` and `engine` in Table 6), however, we observe in almost all other tests that the Loopfrog technique is generally cheaper (often in orders of magnitude) in computational efforts required for building a termination argument.

Tables 7, 6 and 8 show that loop summarization is able to prove termination for the same number of loops as CTA and Terminator, but does so with less resource requirements. In particular it demonstrates that a simple strict order relation for all numeric variables of the loop (Table 5, domain #1) is, in practice, as effective as CTA with default ranking functions. The results on the considerably larger Windows device drivers (Table 11) lead to similar conclusions.

The comparison demonstrates some weak points of the iterative analysis:

- Enumeration of all paths through the loop can require many iterations or even can be infinite for infinite state systems (as are most of realistic programs).
- The ranking procedures can often fail to produce a ranking argument; but if it succeeds, a very simple relation is often sufficient.
- The search for a compositional transition invariant sometimes results in an exponential growth of the number of loop unrollings (in case of CTA).

Loopfrog does not suffer from the first problem: the analysis of each loop requires a finite number of calls to a decision procedure. The second issue is leveraged by relative simplicity of adding new abstract domain over implementing complex ranking function method. The third issue is transformed into generation of suitable invariant candidates, which, in general, may generate many candidates, which slows the procedure down. However, we can control the order of candidates by prioritizing some domains over the others, and thus, can expect simple ranking arguments to be discovered first.

The complete results of these experiments as well as the Loopfrog tool are available at `www.verify.inf.usi.ch/loopfrog/termination`.

7 Related Work

This section is divided into two parts: the first covers research related to summarization while the second one relates our work to other termination analysis techniques.

7.1 Work related to loop summarization

The body of work on analysis using summaries of functions is extensive (see a nice survey in [29]) and dates back to Cousot and Halbwachs [23], and Sharir and Pnueli [54]. In a lot of projects, function summaries are created for alias analysis or points-to analysis, or are intended for the analysis of program fragments. As a result, these algorithms are either specialized to particular problems and deal with fairly simple abstract domains or are restricted to analysis of parts of the program. An instance is the summarization of library functions in [29]. In contrast, our technique aims at computing a summary for the entire program, and is applicable to complex abstract domains.

The same practical motivation, sound analysis of ANSI-C programs, drives our work and the work behind Frama-C project[10]. In particular, the PhD work of Moy [48] even targets, among others, the same set of benchmarks—Verisec [41] and Zitser's [59] test suites. To tackle them with the Frama-C tools, Moy employs a number of techniques that discover pre- and post-conditions for loops as well as loop invariants. He combines abstract interpretation-based invariant inference with weakest precondition-based iterative methods such as the Suzuki-Ishihata algorithm [55]. The latter one, induction iteration, applies weakest precondition computation to a candidate loop invariant iteratively until an inductive invariant is found. Thus, loop summarization can be seen as 1-step application of the induction-iteration method, in which "weakest precondition" is replaced with "strongest postcondition"[11].

Note that application of the Suzuki-Ishihata algorithm to string operation-intensive programs (as our benchmarks are) often leads to non-terminating iterative computation since there is no guarantee to obtain an inductive invariant from a candidate. To avoid this uncertainty, we are interested only in those candidates that can be proven to be an inductive invariant in a single step. We claim that a careful choice of candidates would contribute more to precision and scalability of analysis. In fact, our results on the aforementioned benchmark suites support this claim. We analyze Zitser's benchmark suite in a matter of seconds and are able to discharge 62% of bug-free instances, while Frama-C does not complete any of test cases within a 1 hour limit. When applied to a smaller programs of the Verisec test suite both tools are able to discharge 74% of bug-free test cases; LOOPFROG required almost no time for this analysis.

LOOPFROG shares a lot of its concept and architecture with Houdini, an annotation assistant for ESC/Java [27]. Houdini was first created as a helper to ESC/-Java; the goal was to lower the burden of manual program annotation (sufficient annotation is critical for the application of ESC/Java). Similar to loop summarization, Houdini "magically" guesses a set of candidate relations between program variables and then discharges or verifies them one by one using the ESC/Java as a refuter. Verified candidates are added to the program as annotations and are used later by the main ESC/Java chec in the same way as symbolic execution makes use of summaries when it runs over a loop-free program. However, there are also numerous differences between the two tools. Houdini is designed to be applied to any

[10] `http://frama-c.com/`

[11] However, the choice of transformer, i.e., "pre-condition" or "post-condition", is irrelevant if only one step is performed.

program module or a routine in a library while our summarization concentrates deliberately on loops. Houdini adds annotations to the program, while LOOPFROG replaces each loop with the summary, thus keeping the cost of analysis for every consecutive loop as low as for the inner-most one.

Houdini as well as LOOPFROG generate a lot of candidates that help to address buffer access checks. For instance, it generates 6 different comparison relations for each integral type and a constant in a program. While experimenting with LOOPFROG, we found such an abstract domain of arbitrary relations to be effective, though very expensive. The result are too many useless candidates. Therefore we prefer problem-tailored domains that generate fewer candidates.

Furthermore, as we show in Section 5, LOOPFROG extends candidates selection to those that relate two different valuations of the same program variable, e.g., before and after a loop iteration. This allows discovering not only safety, but also liveness-related loop invariants; in particular loop termination can be proven with the help of this addition.

A series of work by Gulwani et al. [33, 34] uses loop invariant discovery for the purpose of worst-case execution time (WCET) analysis. One of the approaches (reported as the most effective in practice) employs template-based generation of invariant candidates. Starting from the inner-most loop, a bound of the loop's maximal resources usage is computed. Therefore, it can be seen as a loop summarization with domains tuned for WCET-analysis rather then string-operations as in LOOPFROG.

The Saturn tool [1] computes a summary of a function with respect to an abstract domain using a SAT-based approach to improve scalability. However, summaries of loop-bodies are not created. In favor of scalability, Saturn simply unwinds loops a constant number of times, and thus, is unsound as bugs that require more iterations are missed.

SAT-solvers, SAT-based decision procedures, and constraint solvers are frequently applied in program verification. Notable instances are Jackson's Alloy tool [38] and CBMC [14]. The SAT-based approach is also suitable for computing abstractions, as, for example, in [1, 15, 52] (see detailed discussion in Sec. 4.2). The technique reported here also uses the flexibility of a SAT-based decision procedure for a combination of theories to compute loop summaries.

Our technique can be used for checking buffer overruns and class-string vulnerabilities. There exist a large number of static analysis tools focusing on these particular problems. In this respect, the principal difference of our technique is that it is a general purpose abstraction-based checker which is not limited to special classes of faults.

A major benefit of our approach is its ability to generate diagnostic information for failed properties. This is usually considered a distinguishing feature of model checking [13] and, sometimes, extended static checking [28], but rarely found in tools based on abstract interpretation. Most model checkers for programs implement a CEGAR approach [4, 36], which combines model checking with counterexample-guided abstraction refinement. The best-known instance is SLAM [4], and other implementations are BLAST [36], MAGIC [10], and SATABS [16], which implement predicate abstraction.

Recently, a number of projects applied counterexample-guided refinement to refine abstract domains other than predicate abstraction. Manevich et al. [47] formalize CEGAR for general powerset domains; Beyer et al. [7] integrate the TVLA system [46] into BLAST and use counterexamples to refine 3-valued structures to make shape analysis more scalable; Gulavani and Rajamani devised an algorithm for refining any abstract interpretations [31, 32] by combining widening with interpolation. Our procedure is also able to generate counterexamples with respect to the abstract domain and could be integrated into a CEGAR loop for automatic refinement. As LOOPFROG does not perform domain refinement, but instead computes an abstraction for a given domain using SAT, it is more closely related to the work on computing abstractions with abstract conflict driven clause learning (ACDCL) [26].

7.2 Work related to termination

Although the field of program termination analysis is relatively old and the first results date back to Turing [58], recent years have seen a tremendous increase in practical applications of termination proving. Two directions of research enabled the efficacy of termination provers in practice:

– transition invariants by Podelski and Rybalchenko [49], and
– the size-change termination principle (SCT) by Lee, Jones and Ben-Amram [44],

where the latter has its roots in previous research on termination of declarative programs. Until very recently, these two lines of research did not intersect much. The first systematic attempt to understand their common traits is a recent publication by Heizmann et al. [35].

7.2.1 Relation to size-change termination principle

Termination analysis based on the SCT principle usually involves two steps:

1. construction of an abstract model of the original program in the form of *size-change graphs* (SC-graphs) and
2. analysis of the SC-graphs for termination.

SC-graphs contain abstract program values as nodes and use two types of edges, along which values of variables *must decrease*, or *decrease or stay the same*. No edge between nodes means that none of the relations can be ensured. Graphs G which are closed under composition with itself, are called *idempotent*, i.e., $G; G = G$.[12]

Lee et al. [44] identify two termination criteria based on a size-change graph:

1. The SC-graph is well-founded, or
2. The idempotent components of an SC-graph are well-founded.

An SC-graph can be related to transition invariants as follows. Each sub-graph corresponds to a conjunction of relations, which constitutes a transition invariant. The whole graph forms a disjunction, resulting in a termination criterion very similar to that presented as Theorem 3: if an SC-graph is well-founded then there exists a d.wf. transition invariant. Indeed, Heizmann et al. [35] identify the SCT criterion as strictly stronger than the argument via transition invariants [49]. In

[12]In this discussion we omit introducing the notation necessary for a formal description of SCT; see Lee et al. [44, 35] for more detail.

other words, there are terminating programs for which there are no suitable SC-graphs that comply with the termination criteria above.

The intuition behind SCT being a stronger property comes from the fact that SC-graphs abstract from the reachability of states in a program, i.e., the SC-graph requires termination of all paths regardless of whether those paths are reachable or not. Transition invariants, on the other hand, require the computation of the reachable states of the program. In this respect, our light-weight analysis is closely related to SCT, as it havocs the input to individual loop iterations before checking a candidate transition invariant.

The domains of SC-graphs correspond to abstract domains in our approach. The initial inspiration for the domains we experimented with comes from a recent survey on ranking functions for SCT [5]. The domains #1–4 in Table 5 encode those graphs with only down-arcs. Domain #5 has down-arcs and edges that preserve the value. However, note that, in order to avoid well-foundedness checks, we omit domains that have mixed edge types.

Program abstraction using our loop summarization algorithm can be seen as construction of size-change graphs. The domains suggested in Section 5.8 result in SC-graphs that are idempotent and well-founded by construction.

Another relation to SCT is the second SCT criterion based on idempotent SC-components. In [35] the relation of idempotency to some notion in transition invariant-based termination analysis was stated as an open question. However, there is a close relation between the idempotent SC-components and compositional transition invariants (Definition 9, page 17) used here and in compositional termination analysis [40]. The d.wf. transition invariant constructed from idempotent graphs is also a compositional transition invariant.

7.2.2 Relation to other research in transition invariant-based termination

The work in Section 5 is a continuation of the research of transition invariants-based termination proving methods initiated by [49]. Methods developed on the basis of transition invariants rely on an iterative abstraction refinement-like construction of d.wf. transition invariants [17, 18, 40]. Our approach is different, because it aims to construct a d.wf. transition invariant without refinement. Instead of ranking function discovery for every non-ranked path, we use abstract domains that express ranking arguments for all paths at the same time.

Chawdhary et al. [11] propose a termination analysis using a combination of fixpoint-based abstract interpretation and an abstract domain of disjunctively well-founded relations. The abstract domain they suggest is of the same form as domain #5 in Table 5. However their technique attempts iterative computation of the set of abstract values and has a fixpoint detection of the form $T \subseteq R^+$, while in our approach it is enough to check $T \subseteq R$, combined with the compositionality criterion. This allows more abstract domains to be applied for summarization, as each check is less demanding on the theorem prover.

Dams et al. [24] present a set of heuristics that allow heuristic inference of candidate ranking relations from a program. These heuristics can be seen as abstract domains in our framework. Moreover, we also show how candidate relations can be checked effectively.

Cook et al. [20] use relational predicates to extend the framework of Reps et al.[51] to support termination properties during computation of inter-procedural program summaries. Our approach shares a similar motivation and adds termination support to abstract domain-based loop summarization. However, we concentrate on scalable non-iterating methods to construct the summary while Cook et al. [20] rely on a refinement-based approach. The same argument applies in the case of Balaban et al.'s framework [3] for procedure summarization with liveness properties support.

Berdine et al. [6] use the Octagon and Polyhedra abstract domains to discover invariance constraints sufficient to ensure termination. Well-foundedness checks, which we identify as an expensive part of the analysis, are left to iterative verification by an external procedure like in the Terminator algorithm [18] and CTA [40]. In contrast to these methods, our approach relies on abstract domains which are well-founded by construction and therefore do not require explicit checks.

Dafny, a language and a program verifier for functional correctness [45], employs a very similar method to prove a loop (or a recursive call) terminating. First, each Dafny type has an ordering, values are finite, and, except for integers, values are bounded from below. Second, Dafny offers a special `decrease` predicate. Now, if one can provide a tuple of variables, each bounded from below, for which `decrease` holds (a termination metric), then termination can be concluded. A termination metric can be given by a developer or guessed using predefined heuristics. Effectively, this method maps one to one to strict order relational domains used in Loopfrog.

It is interesting to note one particular case in Dafny: if an unbounded integer variable is used in a termination metric, then an additional invariant is required to bound the integer from below. Loopfrog usually deals with machine integers, which are bounded by design.

Altogether, successful application of the relatively simple predefined heuristics in Dafny and our experiments supports the main claim of Section 5: light-weight analysis based on simple heuristics is often sufficient to prove many loops terminating.

8 Conclusion and Future Work

The discovery of an appropriate abstraction is a fundamental step in establishing a successful verification framework. Abstraction not only reduces the computational burden of verification, but also makes it possible to analyze in a sound manner infinite-state software models.

The proposed loop summarization algorithm is a step towards understanding of what is a right abstraction, how it should be discovered and used to enable efficient analysis of programs by formal verification tools. The algorithm computes an abstract model of a program with respect to a given abstract interpretation by replacing loops and function calls in the control flow graph by their symbolic transformers. The run-time of the new algorithm is linear in the number of looping constructs in a program and a finite number of (relatively simple) decision procedure calls is used for discovery of every abstract symbolic transformer. Therefore, it addresses the problem of the high complexity of computing abstract fixpoints.

The procedure over-approximates the original program, which implies soundness of the analysis, but, as any other abstraction-based technique, it can introduce false positives on the consequent phase of analysis of the constructed abstract model. An additional benefit of the technique is its ability to generate *leaping counterexamples*, which are helpful for diagnosis of the error or for filtering spurious warnings. The conducted experimental evaluation within an analysis framework for buffer overflows demonstrates the best error-detection and error-discrimination rates when comparing to a broad selection of static analysis tools.

Loop summarization algorithm also can be effectively employed to perform light-weight program termination analysis. For a sequential program, termination of all loops is enough to conclude program termination, therefore we focused our analysis on individual loops. The new algorithm is based on loop summarization and employs relational abstract domains to discover transition invariants for loops. It uses compositionality of a transition invariant as a completeness criterion, i.e., that a discovered transition invariant holds for any execution through the loop. If such an invariant exists and it is (disjunctively) well-founded, then the loop is guaranteed to terminate. Well-foundedness can be checked either by an application of a quantifier-supporting decision procedure or be ensured by construction. In the latter case an abstract domain for producing candidates for transition invariants should be chosen appropriately.

Note, that, although this algorithm is incomplete (because a compositional transition invariant does not always exist and the ability to discover transition invariants is restricted by expressiveness of the selected abstract domains), our evaluation demonstrates its effectiveness. We applied our new termination analysis to numerous benchmarks including Windows device drivers and demonstrated high scalability as well as a level of precision that matches to the state-of-the-art path-based algorithms such as Terminator and CTA.

In contrast to other methods, our algorithm performs both loop summarization and transition invariant inference at the same time, thus, both safety- and liveness properties of loop semantics are preserved. Also, it utilizes a family of simple, custom abstract domains whereas other works in termination analysis often use off-the-shelf domains; it seems very interesting to note that simpler domains can go a long way in solving those problems, while keeping computational costs low.

For future research we would like to highlight two specific directions:

1) Problem-specific abstract domains for termination (liveness) analysis Additional relational abstract domains should be considered for termination (liveness) analysis in areas where it is appropriate. Possible applications include:

- Verification of liveness properties in protocol implementations with abstract domains used to reason about message ordering.
- Verification of liveness properties in concurrent programs with abstract domains employed to reason about the independence of termination from thread scheduling or the execution progress over all threads.

A recent attempt to build a bridge between transition invariants-based termination analysis and size-change termination by Heizmann et al. [35] suggests that the well-studied size-change graphs can be adopted as abstract domains.

As a stand-alone theoretical problem we see the definition of a class of systems and properties for which termination (liveness) can be proved by induction, i.e., by guessing and proving the base step (discovery of a transition invariant) and proving the inductive step (compositionality of a transition invariant).

2) Combined state/transition invariants abstract domains for conditional termination (liveness) Cook et. al. aim to compute a loop precondition that implies termination and use it to establish *conditional termination* [19]. A combination of state and transition invariants can be used for similar purposes.

We also plan to investigate whether a negative result of a transition invariant candidate check can be used to derive a counterexample for termination or a precondition. For instance, the failure to ensure the total order between all values of an iterator in a loop may be a hint that an integer overflow leads to non-termination.

Bibliography

1. Aiken A, Bugrara S, Dillig I, Dillig T, Hackett B, Hawkins P (2007) An overview of the Saturn project. In: Das M, Grossman D (eds) Workshop on Program Analysis for Software Tools and Engineering, ACM, pp 43–48
2. Ashcroft E, Manna Z (1979) The translation of 'go to' programs to 'while' programs. In: Classics in software engineering, Yourdon Press, Upper Saddle River, NJ, USA, pp 49–61
3. Balaban I, Cohen A, Pnueli A (2006) Ranking abstraction of recursive programs. In: Emerson E, Namjoshi K (eds) Verification, Model Checking, and Abstract Interpretation (VMCAI), Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol 3855, pp 267–281
4. Ball T, Rajamani SK (2001) The SLAM toolkit. In: Computer Aided Verification (CAV), Springer, Lecture Notes in Computer Science, vol 2102, pp 260–264
5. Ben-Amram AM, Lee CS (2009) Ranking functions for size-change termination II. Logical Methods in Computer Science 5(2)
6. Berdine J, Chawdhary A, Cook B, Distefano D, O'Hearn P (2007) Variance analyses from invariance analyses. In: Principles of Programming Languages (POPL), ACM, New York, NY, USA, POPL '07, pp 211–224
7. Beyer D, Henzinger TA, Théoduloz G (2006) Lazy shape analysis. In: Computer Aided Verification (CAV), Springer, Lecture Notes in Computer Science, vol 4144, pp 532–546
8. Biere A, Cimatti A, Clarke EM, Strichman O, Zhu Y (2003) Bounded model checking. Advances in Computers 58:118–149
9. Cavada R, Cimatti A, Franzén A, Kalyanasundaram K, Roveri M, Shyamasundar RK (2007) Computing predicate abstractions by integrating BDDs and SMT solvers. In: Formal Methods in Computer-Aided Design (FMCAD), IEEE Computer Society, pp 69–76
10. Chaki S, Clarke EM, Groce A, Jha S, Veith H (2004) Modular verification of software components in C. IEEE Transactions on Software Engineering 30(6):388–402
11. Chawdhary A, Cook B, Gulwani S, Sagiv M, Yang H (2008) Ranking abstractions. In: Drossopoulou S (ed) Programming Languages and Systems, Lecture Notes in Computer Science, vol 4960, Springer Berlin / Heidelberg, pp 148–162
12. Clarke E, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Emerson E, Sistla A (eds) Computer Aided Veri-

fication, Lecture Notes in Computer Science, vol 1855, Springer Berlin / Heidelberg, pp 154–169
13. Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press
14. Clarke EM, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: Jensen K, Podelski A (eds) Tools and Algorithms for Construction and Analysis of Systems (TACAS), Springer, Lecture Notes in Computer Science, vol 2988, pp 168–176
15. Clarke EM, Kroening D, Sharygina N, Yorav K (2004) Predicate abstraction of ANSI-C programs using SAT. Formal Methods in System Design 25(2-3):105–127
16. Clarke EM, Kroening D, Sharygina N, Yorav K (2005) SATABS: SAT-based predicate abstraction for ANSI-C. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS), Springer, Lecture Notes in Computer Science, pp 570–574
17. Cook B, Podelski A, Rybalchenko A (2005) Abstraction refinement for termination. In: International Symposium on Static Analysis (SAS), Springer, Lecture Notes in Computer Science, vol 3672, pp 87–101
18. Cook B, Podelski A, Rybalchenko A (2006) Termination proofs for systems code. In: Programming Language Design and Implementation (PLDI), ACM, pp 415–426
19. Cook B, Gulwani S, Lev-Ami T, Rybalchenko A, Sagiv M (2008) Proving conditional termination. In: Computer Aided Verification (CAV), Springer, Lecture Notes in Computer Science, vol 5123, pp 328–340
20. Cook B, Podelski A, Rybalchenko A (2009) Summarization for termination: no return! Formal Methods in System Design 35(3):369–387
21. Cook B, Kroening D, Ruemmer P, Wintersteiger CM (2010) Ranking function synthesis for bit-vector relations. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS), Springer, Lecture Notes in Computer Science, vol 6015, pp 236–250
22. Cousot P, Cousot R (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages (POPL), pp 238–252
23. Cousot P, Halbwachs N (1978) Automatic Discovery of Linear Restraints Among Variables of a Program. In: Principles of Programming Languages (POPL), pp 84–96
24. Dams D, Gerth R, Grumberg O (2000) A heuristic for the automatic generation of ranking functions. In: Proceedings of the Workshop on Advances in Verification (WAVE), pp 1–8
25. Dor N, Rodeh M, Sagiv S (2003) CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In: Programming Language Design and Implementation (PLDI), pp 155–167
26. D'Silva V, Haller L, Kroening D, Tautschnig M (2012) Numeric bounds analysis with conflict-driven learning. In: TACAS, Springer, pp 48–63
27. Flanagan C, Leino KRM (2001) Houdini, an annotation assistant for esc/java. In: Oliveira J, Zave P (eds) FME 2001: Formal Methods for Increasing Software Productivity, Lecture Notes in Computer Science, vol 2021, Springer Berlin / Heidelberg, pp 500–517
28. Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R (2002) Extended static checking for Java. In: Proceedings of the ACM SIGPLAN 2002

Conference on Programming language design and implementation, ACM, New York, NY, USA, PLDI '02, pp 234–245
29. Gopan D, Reps TW (2007) Low-level library analysis and summarization. In: Computer Aided Verification (CAV), Springer, Lecture Notes in Computer Science, vol 4590, pp 68–81
30. Graf S, Saidi H (1997) Construction of abstract state graphs with PVS. In: Computer Aided Verification (CAV), Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol 1254, pp 72–83
31. Gulavani BS, Rajamani SK (2006) Counterexample driven refinement for abstract interpretation. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS), Springer, Lecture Notes in Computer Science, vol 3920, pp 474–488
32. Gulavani BS, Chakraborty S, Nori AV, Rajamani SK (2010) Refining abstract interpretations. Information Processing Letters 110(16):666–671
33. Gulwani S, Lev-Ami T, Sagiv M (2009) A combination framework for tracking partition sizes. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, pp 239–251
34. Gulwani S, Mehra KK, Chilimbi T (2009) Speed: precise and efficient static estimation of program computational complexity. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, pp 127–139
35. Heizmann M, Jones N, Podelski A (2011) Size-change termination and transition invariants. In: Cousot R, Martel M (eds) Static Analysis, Lecture Notes in Computer Science, vol 6337, Springer Berlin / Heidelberg, pp 22–50
36. Henzinger TA, Jhala R, Majumdar R, Sutre G (2002) Lazy abstraction. In: Principles of Programming Languages (POPL), ACM, pp 58–70
37. Hoare T (1969) An axiomatic basis for computer programming. Communications of ACM 12(10):576–580
38. Jackson D, Vaziri M (2000) Finding bugs with a constraint solver. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp 14–25
39. Kroening D, Sharygina N (2006) Approximating predicate images for bit-vector logic. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS), Springer, Lecture Notes in Computer Science, vol 3920, pp 242–256
40. Kroening D, Sharygina N, Tsitovich A, Wintersteiger CM (2010) Termination analysis with compositional transition invariants. In: International Conference on Computer-Aided Verification (CAV), Springer, Edinburgh, UK, Lecture Notes in Computer Science, vol 6174
41. Ku K, Hart TE, Chechik M, Lie D (2007) A buffer overflow benchmark for software model checkers. In: Automated Software Engineering (ASE), ACM, pp 389–392
42. Lahiri SK, Ball T, Cook B (2005) Predicate abstraction via symbolic decision procedures. In: Computer Aided Verification (CAV), Springer, Lecture Notes in Computer Science, vol 3576, pp 24–38
43. Lahiri SK, Nieuwenhuis R, Oliveras A (2006) SMT Techniques for Fast Predicate Abstraction. In: Computer Aided Verification (CAV), Springer, Lecture Notes in Computer Science, vol 4144, pp 424–437

44. Lee CS, Jones ND, Ben-Amram AM (2001) The size-change principle for program termination. In: Principles of Programming Languages (POPL), ACM, vol 36, pp 81–92
45. Leino KRM (2010) Dafny: an automatic program verifier for functional correctness. In: Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning, Springer-Verlag, Berlin, Heidelberg, Lecture Notes in Computer Science, vol 6355, pp 348–370
46. Lev-Ami T, Sagiv S (2000) TVLA: A system for implementing static analyses. In: Static Analysis (SAS), Springer, Lecture Notes in Computer Science, vol 1824, pp 280–301
47. Manevich R, Field J, Henzinger TA, Ramalingam G, Sagiv M (2006) Abstract counterexample-based refinement for powerset domains. In: Program Analysis and Compilation (PAC), Springer, Lecture Notes in Computer Science, vol 4444, pp 273–292
48. Moy Y (2009) Automatic modular static safety checking for C programs. PhD thesis, Université Paris-Sud
49. Podelski A, Rybalchenko A (2004) Transition invariants. In: IEEE Symposium on Logic in Computer Science (LICS), IEEE Computer Society, pp 32–41
50. Podelski A, Rybalchenko A (2007) ARMC: The logical choice for software model checking with abstraction refinement. In: Practical Aspects of Declarative Languages (PADL), Springer, pp 245–259
51. Reps T, Horwitz S, Sagiv M (1995) Precise interprocedural dataflow analysis via graph reachability. In: POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, pp 49–61
52. Reps TW, Sagiv S, Yorsh G (2004) Symbolic implementation of the best transformer. In: Verification, Model Checking, and Abstract Interpretation (VMCAI), Springer, Lecture Notes in Computer Science, vol 2937, pp 252–266
53. Scott J, Lee LH, Chin A, Arends J, Moyer B (1999) Designing the m·core$^{\mathrm{tm}}$ m3 cpu architecture. In: International Conference on Computer Design (ICCD), pp 94–101
54. Sharir M, Pnueli A (1981) Two approaches to interprocedural data flow analysis. Program Flow Analysis: theory and applications, Prentice-Hall
55. Suzuki N, Ishihata K (1977) Implementation of an array bound checker. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM, New York, NY, USA, POPL '77, pp 132–143
56. Tarjan RE (1981) Fast algorithms for solving path problems. Journal of ACM 28(3):594–614
57. Turing AM (1936) On computable numbers, with an application to the Entscheidungsproblem. Proceedings London Mathematical Society 2(42):230–265
58. Turing AM (1949) Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines, Cambridge, pp 67–69
59. Zitser M, Lippmann R, Leek T (2004) Testing static analysis tools using exploitable buffer overflows from open source code. In: International Symposium on Foundations of Software Engineering, ACM, pp 97–106