Loop Summarization and Termination Analysis*

Aliaksei Tsitovich[1], Natasha Sharygina[1],
Christoph M. Wintersteiger[2], and Daniel Kroening[2]

[1] Formal Verification and Security Group, University of Lugano, Switzerland
[2] Oxford University, Computing Laboratory, UK

Abstract. We present a technique for program termination analysis based on loop summarization. The algorithm relies on a library of abstract domains to discover well-founded transition invariants. In contrast to state-of-the-art methods it aims to construct a complete ranking argument for all paths through a loop at once, thus avoiding expensive enumeration of individual paths. Compositionality is used as a completeness criterion for the discovered transition invariants. The practical efficiency of the approach is evaluated using a set of Windows device drivers.

1 Introduction

The program termination problem has received increased interest in the recent past. In practice, termination analysis is at a point where industrial application of termination proving tools is feasible. This is possible through a series of improvements upon methods that prove program termination by constructing *well-founded ranking relations*.

Podelski and Rybalchenko propose *disjunctive* well-foundedness of transition invariants [1] as a means to improve the performance of termination proving, as well as to simplify synthesis of ranking relations. Based on their crucial discovery, the same authors together with Cook give an algorithm to verify program termination using iterative construction of transition invariants — the *Terminator* algorithm [2,3]. 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 to form 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.

The Terminator algorithm was successfully implemented in tools (e.g., TERMINATOR [3], ARMC [4], SATABS [5]) 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, taking up to 99% of the

* Supported by the Swiss National Science Foundation under grant no. 200020-122077 and by a Microsoft Software Engineering Innovation Foundation (SEIF) Award. Christoph M. Wintersteiger is now with Microsoft Research, Cambridge, UK.

P.A. Abdulla and K.R.M. Leino (Eds.): TACAS 2011, LNCS 6605, pp. 81–95, 2011.

runtime in practice [3,5]. The runtime required for ranking relation synthesis is negligible in comparison. A possible solution to this performance issue is *Compositional Termination Analysis (CTA)* [6]. 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 runtime, but introduces incompleteness since a transitive termination argument may not be found for each loop of a program. However, an experimental evaluation on Windows device drivers indicates that this case is rare in practice.

The complexity of the termination problem together with the observation that most loops in practice have (relatively) simple termination arguments suggests the use of light-weight static analysis for this purpose. In this paper, we propose a new technique for termination analysis, which extends a known algorithm for loop summarization [7] based on abstract interpretation [8]. The crucial difference between the previous approach and our proposal is the use of (disjunctively well-founded) *transition invariants* instead of *state invariants* during summarization. Furthermore, fixpoint computation of abstract transformers is avoided (but required by other methods, e.g., [9,10]).

Our algorithm constructs summaries for loops, starting from the inner-most loop in the control flow graph of the program. In case of nested loops, inner loops are replaced with (loop-free) summaries during verification. At any point during the analysis, the problem is therefore reduced to the analysis of a single loop. During construction of the loop summaries, our algorithm relies on a library of templates for abstract domains. These are used to construct candidates for transition invariants, which subsequently are verified to be actual disjunctively well-founded transition invariants by means of a safety checker and a satisfiability decision procedure. Due to the fact that the safety checker is employed to analyze only a single unwinding of a loop at any point, we gain large speedups compared to algorithms like Terminator or CTA. At the same time, the false-positive rate of our algorithm is very low in practice, which we demonstrate using an experimental evaluation on a diverse suite of C programs.

This paper is organized as follows: Section 2 introduces the theoretical background, Section 3 presents our new methods. Section 4 proposes an optimization that simplifies the selection of candidates for transition invariants. In Section 5 we give experimental evidence of the practicality of our approach. Section 6 relates this approach to size-change termination principle and discusses the other related work. Finally, Section 7 suggests future work and concludes.

2 Background

We formalize programs as *transition systems*.

Definition 1 (Transition System). *A transition system (program) P is a three tuple $\langle S, I, R\rangle$, where*

- S *is a (possibly infinite) set of states,*
- $I \subseteq S$ *is the set of initial states,*
- $R \subseteq S \times S$ *is the transition relation.*

A *computation* of a transition system is a (maximal) sequence of states $s_0, s_1, \ldots$ such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$ for all $i \geq 0$.

The reflexive and non-reflexive transitive closures of R are denoted as R^* and R^+ respectively. The set of reachable states is $R^*(I)$. We also define the relational composition operator $\circ$ for two relations $R_1, R_2 : S \times S$ by

$$R_1 \circ R_2 := \{ (s, s') \mid \exists s''.(s, s'') \in R_1 \wedge (s'', s') \in R_2 \} \ .$$

Note that a relation R is transitive if it is closed under relational composition, i.e., when $R \circ R \subseteq R$.

2.1 Termination

A program is terminating if it does not allow infinite computations, which follows from well-foundedness of the transition relation (restricted to the reachable states). A *well-founded* relation is a relation that does not contain infinite descending chains or, more formally:

Definition 2 (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. $\forall X \subseteq S \ . \ X \neq \emptyset \implies \exists m \in X, \forall s \in S (s, m) \notin R$.*

The same does not hold true for the weaker notion of *disjunctive well-foundedness*. However, Podelski and Rybalchenko show that disjunctive well-foundedness of a *transition invariant* is equivalent to program termination:

Definition 3 (Disjunctive Well-foundedness [1]). *A relation T is* disjunctively well-founded (d.wf.) *if it is a finite union $T = T_1 \cup \ldots \cup T_n$ of well-founded relations.*

Definition 4 (Transition Invariant [1]). *A* transition invariant *T for program $P = \langle S, I, R \rangle$ is a superset of the transitive closure of R restricted to the reachable state space, i.e., $R^+ \cap (R^*(I) \times R^*(I)) \subseteq T$.*

The crucial theorem is as follows:

Theorem 1 (Termination [1]). *A program P is terminating iff there exists a d.wf. transition invariant for P.*

The Terminator algorithm [3] automates the 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 to $T = T \cup T'$

and the algorithm continues to search 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.

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

Definition 5 (Compositional Transition Invariant [1,6]). *A d.wf. transition invariant T is called* compositional *if it is also transitive, or equivalently, closed under composition with itself, i.e., when $T \circ T \subseteq T$.*

Podelski and Rybalchenko made an interesting remark regarding the compositionality (transitivity) of transition invariants: If T is transitive, it is sufficient 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 [1]. Therefore, compositionality of a d.wf. transition invariant implies program termination. This fact is exploited in *Compositional Termination Analysis* [6], which iteratively constructs a termination argument, similar to the Terminator algorithm. In contrast to Terminator however, the safety checker is not required to analyze complete loops. Instead, the algorithm checks an increasing number of unwindings of the loops in the program until a compositional transition invariant is established. This technique results in significant speed-ups in practice, but comes at a price: there is no guarantee that a compositional transition invariant can be found for every loop.

2.2 Loop Summarization

In the following section we present a method for static analysis based on a previously presented loop summarization algorithm [7]. This technique constructs sound program abstractions for the purpose of scalable static analysis. It replaces loops in a program by smaller loop-free program fragments that over-approximate the original behavior of the loop.

Algorithm 1 presents an outline of this procedure. 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 contains a non-nested loop, it is summarized using the function SummarizeLoop and the resulting summary replaces the original loop in P'. Consequently, any outer loop eventually becomes non-nested, which enables further progress.

The function SummarizeLoop computes the summaries. A simple over-approximation can be obtained by replacing a loop by a program fragment that 'havocs' the state, i.e., by setting all variables which are (potentially) modified during loop execution to non-deterministic values. To improve the precision of these summaries, they are strengthened by (partial) loop invariants. SummarizeLoop has two subroutines that are related to invariant discovery: 1) PickInvariantCandidates, which generates a set of 'invariant candidates' using a library of abstract domains, and 2) IsInvariant, which checks whether a candidate is an actual invariant for a given loop.

```
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     return P
11 end

12 SUMMARIZELOOP(L)
13 input: Single-loop program L (over variable set X)
14 output: Loop summary
15 begin
16     I := ⊤
17     foreach Candidate C(X) in PICKINVARIANTCANDIDATES(Loop) do
18         if ISINVARIANT(L, C) then
19             I := I ∧ C
20     return "X^pre := X; havoc(L); assume(I(X^pre) ⟹ I(X));"
21 end

22 ISINVARIANT(L, C)
23 input: Single-loop program L (over variable set X), invariant candidate C
24 output: TRUE if C is invariant for L; FALSE otherwise
25 begin
26     return UNSAT(L(X, X') ∧ C(X) ⇒ C(X'))
27 end
```

Algorithm 1. Basic routines of loop summarization

Note that this summarization algorithm does not preserve loop termination: the summaries computed by the algorithm are always terminating program fragments. This abstraction is a sound over-approximation, but it may be too coarse for programs that contain unreachable paths.

3 Loop Summarization with Transition Invariants

In this section, we introduce a method that allows *transition* invariants to be included as a strengthening of loop summaries. This increases the precision of loop summaries and enables construction of termination proofs over summaries.

According to Definition 4, a binary relation T is a transition invariant for a program P if it contains R^+ (restricted to the reachable states). However, the transitivity of T is also a sufficient condition when T is only a superset of R:

Theorem 2. *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 16 of Algorithm 1, where candidate invariants are selected. Clearly, we need to allow selection of transition invariants here, i.e., invariant candidates now have the form $C(X, X')$, where X' is the post-state 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, as required by IsInvariant. A transition invariant for this program is required to cover L^+, which, according to Theorem 2, is implied by $L \subseteq C$ and transitivity of C. The original invariant check in IsInvariant 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 to include a check for compositionality. The resulting procedure is Algorithm 2.

```
1  SummarizeLoop-TI(L)
2  input: Single-loop program L with a set of variables X
3  output: Loop summary
4  begin
5      T := ⊤
6      foreach Candidate C(X, X') in PickInvariantCandidates(Loop) do
7          if IsInvariant(L, C) ∧ IsCompositional(C) then
8              T := T ∧ C
9      return "X^pre := X; havoc(L); assume(T(X^pre, X));"
10 end
```

Algorithm 2. Loop summarization with transition invariants

The additional compositionality (transitivity) check at line 7 of Algorithm 2 corresponds to a check for satisfiability of

$$\exists s_i, s_j, s_k \in S \,.\, \neg \left(C(s_i, s_j) \wedge C(s_j, s_k) \Rightarrow C(s_i, s_k)\right) \;, \tag{1}$$

which may be decided by a suitable decision procedure, e.g., an SMT solver. Of course, this check may be omitted if the selected invariant candidates are compositional by construction.

Termination. The changes to the summarization algorithm allow for termination checks during summarization through application of Theorem 1, 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 3, d.wf.-ness of a candidate relation T requires establishing well-foundedness of each of its disjuncts. This can be done

by an explicit encoding of the well-foundedness criteria of Definition 2. However, the resulting formula contains quantifiers, which severly limits the applicability of existing decision procedures.

4 Selection of Candidate Invariants

In this section, we propose a set of specialized candidate relations, which we find to be useful in practice. We focus on transition invariants for machine-level integers for programs implemented in low-level languages like ANSI-C.

In contrast to other work on termination proving with abstract domains (e.g., [10]), we do not aim at general domains like Octagons or Polyhedra, as they are not designed for termination and the required d.wf.-ness and compositionality checks can be costly. Instead we prefer domains that

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

Note that very similar criteria are applied in termination provers based on the size-change termination principle. This connection is discussed in more detail in Sec. 6.1.

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), complicating termination arguments. If termination of the loop depends only on machine-level integers, there is however a way to simplify the argument:

Observation 3. *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. T is a transition invariant, i.e., it holds for all pairs $(k_1, k_2) \in K \times K$. Thus it is total. Non-empty finite totally-ordered sets always have a least element and, therefore, T is well-founded. □

A total strict order relation is also transitive, which gives rise to a criterion weaker than Theorem 1:

Corollary 1. *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.

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.

Table 1. Templates used to generate transition invariant candidates

#	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	$sum(x', y') < sum(x, y)$ $sum(x', y') > sum(x, y)$	The sum of all numeric loop variables is strictly decreasing (increasing).
4	$max(x', y') < max(x, y)$ $max(x', y') > max(x, y)$ $min(x', y') < min(x, y)$ $min(x', y') > min(x, y)$	The 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 increasing or decreasing for one of loop variables while the remaining ones are not updated.

5 Evaluation

We have implemented the algorithm described in the previous section in a new version of the static analyzer LoopFrog [11]. The tool operates on program models produced by the Goto-CC model extractor; ANSI-C programs are considered as the primary target.

We implemented a number of domains based on strict orders, thus, following Corollary 1, additional checks for compositionality and d.wf.-ness of candidate relations are not required. The domains are listed in Table 1.

The full set of experimental results is available on-line. Here, we only report the results for the two most illustrative schemata:

- LoopFrog 1: domain #3 in Table 1. 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 1. Expresses strict decreasing (increasing) for every numeric variable of a loop. This generates twice as many simple strict orders as there are variables in a loop;

As reference points we use termination provers built upon the CBMC/SatAbs framework [12]. This tool implements both Compositional Termination Analysis (CTA) [6] and the Terminator algorithm [2] (referred to as SatAbs+T in all tables). For both the default ranking function synthesis methods were enabled; for more details see [5].

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

- The SNU real-time benchmark suite that contains small C programs used for worst-case execution time analysis[1];
- The Powerstone benchmark suite as an example set of C programs for embedded systems [13];
- The Verisec 0.2 benchmark suite [14];
- Windows device drivers (from the Windows Device Driver Kit 6.0).

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

The results for SNU and Power-Stone are presented in Tables 2 and 3. Each table reports the number of loops that were proven as terminating (T), potentially non-terminating (NT) and time-out (TO) for each of the compared techniques. The time column contains the wall-clock time spend for the analysis of completed loops; time-outs are not included in the total time.

The results for the Verisec 0.2 benchmark suite are given in the aggregated form in Table 4. The suite consists a large number of stripped C programs that correspond to known security bugs. Although each program has only very few loops, the benchmark set offers a large variety of loop types and is therefore interesting for termination analysis.

The aggregated data on experiments with Windows device drivers is provided in Table 5. The benchmarks are grouped according to the harness used upon extraction of a model with Goto-CC.[2] We omit results of the Terminator algorithm for this benchmark set, as a corresponding comparison was already reported previously [6].

Discussion. Note that direct comparison of the runtime of LoopFrog with that of 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 the ranking synthesis method. Our loop summarization technique, on the other hand, is a static analysis that only aims at conservative abstractions. In particular, it does not try to prove unreachability of a loop or of preconditions that lead to non-termination[3].

The timing information provided here serves as a reference that allows to compare efforts of achieving the same result. In summary, the three techniques can be compared as follows:

- LoopFrog spends time enumerating invariant candidates, provided by the chosen abstract domain, and has to check just one loop iteration. Compositionality and d.wf. checks are not required for the domains we use.

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

[2] The groups in Table 5 have varying numbers of benchmarks/loops as we omit the benchmarks without loops.

[3] In future, we plan to use a loop-free stem to prove unreachability of certain loop preconditions.

Table 2. SNU real-time benchmark suite

Benchmark	Method	T	NT	TO	Time
adpcm-test 18 loops	LoopFrog 1	13	5	0	470.052
	LoopFrog 2	17	1	0	644.092
	CTA	13	3	2	260.982+
	SatAbs+T	12	2	4	165.673+
bs 1 loop	LoopFrog 1	0	1	0	0.05
	LoopFrog 2	0	1	0	0.118
	CTA	0	1	0	12.218
	SatAbs+T	0	1	0	18.469
crc 3 loops	LoopFrog 1	1	2	0	0.17
	LoopFrog 2	2	1	0	0.255
	CTA	1	1	1	0.206+
	SatAbs+T	2	1	0	13.878
fft1k 7 loops	LoopFrog 1	2	5	0	0.356
	LoopFrog 2	5	2	0	0.668
	CTA	5	2	0	141.176
	SatAbs+T	5	2	0	116.81
fft1 11 loops	LoopFrog 1	3	8	0	3.68
	LoopFrog 2	7	4	0	4.976
	CTA	7	4	0	441.937
	SatAbs+T	7	4	0	427.355
fibcall 1 loop	LoopFrog 1	0	1	0	0.04
	LoopFrog 2	0	1	0	0.016
	CTA	0	1	0	0.335
	SatAbs+T	0	1	0	0.309
fir 8 loops	LoopFrog 1	2	6	0	2.897
	LoopFrog 2	6	2	0	8.481
	CTA	6	2	0	2817.08
	SatAbs+T	6	1	1	236.702+
insertsort 2 loops	LoopFrog 1	0	2	0	0.054
	LoopFrog 2	1	1	0	0.063
	CTA	1	1	0	226.446
	SatAbs+T	1	1	0	209.12
jfdctint 3 loops	LoopFrog 1	0	3	0	5.612
	LoopFrog 2	3	0	0	0.05
	CTA	3	0	0	1.24
	SatAbs+T	3	0	0	0.975
lms 10 loops	LoopFrog 1	3	7	0	2.863
	LoopFrog 2	6	4	0	10.488
	CTA	6	4	0	2923.12
	SatAbs+T	6	3	1	251.031+
ludcmp 11 loops	LoopFrog 1	0	11	0	96.726
	LoopFrog 2	5	6	0	112.808
	CTA	3	5	3	3.256+
	SatAbs+T	3	8	0	94.657
matmul 5 loops	LoopFrog 1	0	5	0	0.148
	LoopFrog 2	5	0	0	0.086
	CTA	3	2	0	1.969
	SatAbs+T	3	2	0	2.152
minver 17 loops	LoopFrog 1	1	16	0	2.574
	LoopFrog 2	16	1	0	7.664
	CTA	14	1	2	105.26+
	SatAbs+T	14	1	2	87.088+
qsort-exam 6 loops	LoopFrog 1	0	6	0	0.671
	LoopFrog 2	0	6	0	3.96
	CTA	0	5	1	45.918+
	SatAbs+T	0	5	1	2530.58+
select 4 loops	LoopFrog 1	0	4	0	0.548
	LoopFrog 2	0	4	0	3.561
	CTA	0	3	1	32.599+
	SatAbs+T	0	3	1	28.12+

Table 3. PowerStone benchmark suite

Benchmark	Method	T	NT	TO	Time
adpcm 11 loops	LoopFrog 1	8	3	0	59.655
	LoopFrog 2	10	1	0	162.752
	CTA	8	3	0	101.301
	SatAbs+T	6	2	3	94.449+
bcnt 2 loops	LoopFrog 1	0	2	0	2.634
	LoopFrog 2	0	2	0	2.822
	CTA	0	2	0	0.79
	SatAbs+T	0	2	0	0.299
blit 4 loops	LoopFrog 1	0	4	0	0.155
	LoopFrog 2	3	1	0	0.047
	CTA	3	1	0	5.945
	SatAbs+T	3	1	0	3.672
compress 18 loops	LoopFrog 1	5	13	0	3.134
	LoopFrog 2	6	12	0	33.924
	CTA	5	12	1	698.996+
	SatAbs+T	7	10	1	474.361+
crc 3 loops	LoopFrog 1	1	2	0	0.152
	LoopFrog 2	2	1	0	0.208
	CTA	1	1	1	0.328+
	SatAbs+T	2	1	0	14.583
engine 6 loops	LoopFrog 1	0	6	0	2.397
	LoopFrog 2	2	4	0	9.875
	CTA	2	4	0	16.195
	SatAbs+T	2	4	0	4.877
fir 9 loops	LoopFrog 1	2	7	0	5.993
	LoopFrog 2	6	3	0	21.592
	CTA	6	3	0	2957.06
	SatAbs+T	6	2	1	193.911+
g3fax 7 loops	LoopFrog 1	1	6	0	1.565
	LoopFrog 2	1	6	0	6.047
	CTA	1	5	1	256.899+
	SatAbs+T	1	5	1	206.847+
huff 11 loops	LoopFrog 1	3	8	0	24.368
	LoopFrog 2	8	3	0	94.613
	CTA	7	3	1	16.353+
	SatAbs+T	7	4	0	52.323
jpeg 23 loops	LoopFrog 1	2	21	0	8.366
	LoopFrog 2	16	7	0	32.9
	CTA	15	8	0	2279.13
	SatAbs+T	15	8	0	2121.36
pocsag 12 loops	LoopFrog 1	3	9	0	2.07
	LoopFrog 2	9	3	0	6.906
	CTA	9	3	0	10.392
	SatAbs+T	7	3	2	1557.57+
ucbqsort 15 loops	LoopFrog 1	1	14	0	0.789
	LoopFrog 2	2	13	0	2.059
	CTA	2	12	1	71.729+
	SatAbs+T	9	5	1	51.084+
v42 12 loops	LoopFrog 1	0	12	0	82.836
	LoopFrog 2	0	12	0	2587.22
	CTA	0	12	0	73.565
	SatAbs+T	1	11	0	335.688

Table 4. Aggregated data on Verisec 0.2 suite

Benchmark group	Method	T	NT	TO	Time
244 loops in 160 programs	LoopFrog 1	33	211	0	11.381
	LoopFrog 2	44	200	0	22.494
	CTA	34	208	2	1207.62+
	SatAbs+T	40	204	0	4040.53

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

Table 5. Aggregated data on Windows device drivers

Benchmark group	Method	T	NT	TO	Time
SDV FLAT DISPATCH HARNESS 557 loops in 30 benchmarks	LoopFrog 1	135	389	33	1752.08
	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	1396.01
	LoopFrog 2	215	201	141	9265.81
	CTA	166	160	231	28033.3
SDV FLAT HARNESS 635 loops in 45 benchmarks	LoopFrog 1	170	416	49	1323
	LoopFrog 2	239	205	191	6816.37
	CTA	201	186	248	31003.2
SDV FLAT SIMPLE HARNESS 573 loops in 31 benchmarks	LoopFrog 1	135	398	40	1510
	LoopFrog 2	200	191	182	6813.99
	CTA	166	169	238	30292.7
SDV HARNESS DRIVER CREATE 9 loops in 5 benchmarks	LoopFrog 1	1	8	0	0.135
	LoopFrog 2	1	8	0	0.234
	CTA	1	8	0	151.846
SDV HARNESS PNP DEFERRED IO REQUESTS 177 loops in 31 benchmarks	LoopFrog 1	22	98	57	47.934
	LoopFrog 2	66	54	57	617.41
	CTA	80	94	3	44645
SDV HARNESS PNP IO REQUESTS 173 loops in 31 benchmarks	LoopFrog 1	25	94	54	46.568
	LoopFrog 2	68	51	54	568.705
	CTA	85	86	2	15673.9
SDV PNP HARNESS SMALL 618 loops in 44 benchmarks	LoopFrog 1	172	417	29	8209.51
	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	7402.23
	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	8082.51
	LoopFrog 2	189	188	129	13584.6
	CTA	137	130	239	20967.8
SDV WDF FLAT SIMPLE HARNESS 172 loops in 18 benchmarks	LoopFrog 1	27	125	20	30.281
	LoopFrog 2	61	91	20	201.96
	CTA	73	95	4	70663

- CTA spends time 1) unwinding loop iterations, 2) discovering a ranking function for each unwound program fragment and 3) checking compositionality of a discovered relation.
- Terminator spends time 1) enumerating paths through the loop and 2) discovering a ranking function for each path.

The techniques can greatly vary in the time required for a particular loop or 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. Given a large number of candidates, this behavior results in an advantage for Terminator on loops that cannot be shown to terminate (`huff` and `engine` in Table 3). However, we observe in almost every other test that the LoopFrog technique is generally cheaper (often orders of magnitude) in computational effort required to discover a valid termination argument.

The comparison demonstrates some weak points of iterative analysis:

- Enumeration of paths through the loop can require many iterations or even can be non-terminating for infinite state systems (as are many realistic programs).

– Ranking procedures often fail to produce a ranking argument; the same time if successful, a simpler relation could be sufficient as well.
– CTA suffers from the fact that the search for a compositional transition invariant sometimes results in exponential growth of the loop unrolling depth.

LoopFrog does not suffer from at least the first of these problems: 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 domains over implementing complex ranking function methods. The third issue is transformed into the generation of suitable candidate invariants, which, in general, may result in a large number candidates, which slow the procedure down. However, as we can control the ordering of the candidates by prioritizing some domains over the others, simple ranking arguments can be expected to be discovered early.

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

6 Related Work

Although the field of program termination analysis is mature (the first results date back to Turing [15]), recent years have seen a tremendous increase in practical applications of termination proving. Two directions of research contributed to the efficacy of termination provers in practice:

– the size-change termination principle (SCT) presented by Lee, Jones and Ben-Amram [16], and
– transition invariants by Podelski and Rybalchenko [1],

where the former 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 connections is a recent publication by Heizmann et al. [17].

6.1 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., $G; G = G$.[4]

[4] In this discussion we omit introducing the notation necessary for a formal description of SCT; see Lee et al. [16,17] for more detail.

Lee et al. [16] 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 1: if an SC-graph is well-founded then there exists a d.wf. transition invariant. Indeed, Heizmann et al. identify the SCT termination criterion as strictly stronger than the argument via transition invariants [17]. An intuitive argument is that SC-graphs abstract from the reachability of states in a program, i.e., arguments based on SC-graphs require termination of all paths irrespectively 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 [18]. The domains #1–4 in Table 1 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 costly 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 Sec. 4 result in SC-graphs that are idempotent and well-founded by construction.

Another similarity to SCT relates to the second SCT criterion based on idempotent SC-components. In [17], the relation of idempotency to analyses using transition invariants was stated as an open question. We remark that there is a close relation between the idempotent SC-components and compositional transition invariants (Definition 5) used here and in compositional termination analysis [6]. The d.wf. transition invariant constructed from idempotent graphs is also a compositional transition invariant.

6.2 Relation to Other Research Using Transition Invariants

The work in this paper is a continuation of the research on proving termination using transition invariants initiated by Podelski and Rybalchenko [1]. Methods developed on the basis of transition invariants rely on an iterative, abstraction refinement-like construction of d.wf. transition invariants [2,3,5,6]. Our approach differs in that it aims to construct a d.wf. transition invariant without refinement. Instead of applying 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. [9] 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 1. However, their method performs an 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 sufficient to check $T \subseteq R$, combined with the compositionality criterion. This allows a richer set of abstract domains to be applied for summarization, as the resulting satisfiability problems are low-cost.

Dams et al. [19] present a set of heuristics for inferring 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 using SAT/SMT.

Cook et al. [20] use relational predicates to extend the framework of Reps et al. [21] to support termination properties during computation of inter-procedural program summaries. Our approach shares a similar motivation and adds termination support to loop summarization based on abstract domains. 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 [22] for procedure summarization with support for liveness properties.

Berdine et al. [10] 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 as in the TERMINATOR algorithm [3] and CTA [6]. In contrast to these methods, our approach relies on abstract domains that yield well-founded relations by construction and therefore do not require explicit checks.

7 Conclusion and Future Work

In this paper, we present an extension to a loop summarization algorithm such that it correctly handles termination properties while constructing a loop-free program over-approximation. To that end, we employ abstract domains that encode transition invariants, i.e., relations over pre- and post-states of the summarized loop. Termination of loops may be established at the same time, by checking disjunctive well-foundedness of the discovered transition invariants. We demonstrate the practicality of our approach on a large set of benchmarks including open-source programs and Windows device drivers.

Further research includes an investigation of abstract domains that allow effective summarization with termination support. We are especially interested in encoding forms of Size-Change-Graphs into schemata for generating candidate invariants.

References

1. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41. IEEE Computer Society, Los Alamitos (2004)
2. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
3. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, pp. 415–426. ACM, New York (2006)

4. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2006)
5. Cook, B., Kroening, D., Ruemmer, P., Wintersteiger, C.: Ranking function synthesis for bit-vector relations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 236–250. Springer, Heidelberg (2010)
6. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
7. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using abstract transformers. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 111–125. Springer, Heidelberg (2008)
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
9. Chawdhary, A., Cook, B., Gulwani, S., Sagiv, M., Yang, H.: Ranking abstractions. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 148–162. Springer, Heidelberg (2008)
10. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.: Variance analyses from invariance analyses. SIGPLAN Not. 42(1), 211–224 (2007)
11. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loopfrog: A static analyzer for ANSI-C programs. In: The 24th IEEE/ACM International Conference on Automated Software Engineering, pp. 668–670. IEEE Computer Society, Los Alamitos (2009)
12. Clarke, E., Kröning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
13. Scott, J., Lee, L.H., Arends, J., Moyer, B.: Designing the low-power M*CORE architecture. In: Proc. IEEE Power Driven Microarchitecture Workshop (1998)
14. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: ASE 2007, pp. 389–392. ACM Press, New York (2007)
15. Turing, A.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines, pp. 67–69. Univ. Math. Lab., Cambridge (1949)
16. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL, pp. 81–92. ACM, New York (2001)
17. Heizmann, M., Jones, N., Podelski, A.: Size-change termination and transition invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 22–50. Springer, Heidelberg (2010)
18. Ben-Amram, A.M., Lee, C.S.: Ranking functions for size-change termination II. Logical Methods in Computer Science 5(2) (2009)
19. Dams, D., Gerth, R., Grumberg, O.: A heuristic for the automatic generation of ranking functions. In: Workshop on Advances in Verification, pp. 1–8 (2000)
20. Cook, B., Podelski, A., Rybalchenko, A.: Summarization for termination: no return! Formal Methods in System Design 35(3), 369–387 (2009)
21. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Symposium on Principles of Programming Languages (POPL), pp. 49–61. ACM, New York (1995)
22. Balaban, I., Cohen, A., Pnueli, A.: Ranking abstraction of recursive programs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 267–281. Springer, Heidelberg (2005)