Proving Safety with Trace Automata and Bounded Model Checking⋆

Daniel Kroening[1], Matt Lewis[1], and Georg Weissenbacher[2]

[1] University of Oxford
[2] Vienna University of Technology

Abstract. Loop under-approximation enriches C programs with additional branches that represent the effect of a (limited) range of loop iterations. While this technique can speed up bug detection significantly, it introduces redundant execution traces which may complicate the verification of the program. This holds particularly true for tools based on Bounded Model Checking, which incorporate simplistic heuristics to determine whether all feasible iterations of a loop have been considered.
We present a technique that uses *trace automata* to eliminate redundant executions after performing loop acceleration. The method reduces the diameter of the program under analysis, which is in certain cases sufficient to allow a safety proof using Bounded Model Checking. Our transformation is precise—it does not introduce false positives, nor does it mask any errors. We have implemented the analysis as a source-to-source transformation, and present experimental results showing the applicability of the technique.

1 Introduction

Software verification can be loosely divided into two themes: finding bugs and proving correctness. These two goals are often at odds with one another, and it is rare that a tool excels at both tasks. This tension is well illustrated by the results of the 2014 Software Verification Competition, in which several of the best-performing tools were based on Bounded Model Checking (BMC) [5]. The BMC-based tools were able to quickly find bugs in the unsafe programs, but were unable to soundly prove safety for the remaining programs. Conversely, many of the sound tools had difficulty in detecting bugs in the unsafe programs.

The reasons for this disparity are rooted in the very nature of contemporary verification tools. Tools aiming at proof typically rely on over-approximating abstractions and refinement techniques to derive the loop invariants required (e.g., [14,23]). For certain classes of programs, invariants can be found efficiently using templates [4] or theorem provers [16]. For unsafe programs, however, any

attempt to construct a safety invariant must necessarily fail, triggering numerous futile refinement iterations before a valid counterexample is detected. Verifiers based on the BMC paradigm (such as CBMC [8]), on the other hand, are able to efficiently detect shallow bugs, but are unable to prove safety in most cases.

The key principle of this paper is that BMC is able to prove safety once the unwinding bound exceeds the reachability diameter of the model [5, 20]. The diameter of non-trivial programs is however in most cases unmanageably large. Furthermore, even when the diameter is small, it is often computationally expensive to determine, as the problem of computing the exact diameter is equivalent to a 2-QBF instance.

The contribution of this paper is a technique that reduces the diameter of a program in a way that the new, smaller diameter can be computed by means of a simple satisfiability check. The technique has two steps:

1. We first identify potentially deep program paths that can be replaced by a concise single-step summary called an *accelerator* [6, 7, 12].
2. We then remove those paths subsumed by the accelerators from the program using *trace automata* [13].

The resulting program preserves the reachable states of the original program, but is often very shallow, and consequently, we can obtain a sound verification result using BMC.

Our paper is organised as follows: We present a number of motivating examples and an outline of our approach in Section 2. Section 3 presents our notation, recapitulates the concept of a reachability diameter, and introduces a generalised notion of the under-approximating accelerators presented in [17, 19]. Section 4 describes the construction of accelerated programs and discusses the resulting reduction of the reachability diameter of the program. In Section 5, we introduce restricting languages and trace automata as a means to eliminate redundant transitions from accelerated programs. The experimental evaluation based on a selection of SV-COMP14 benchmarks is presented in Section 6. Finally, Section 7 briefly surveys related work.

2   Motivation

In this section we will discuss the differences between proving safety and finding bugs, with reference to some SV-COMP14 benchmarks, and informally demonstrate why our method is effective for both kinds of analyses.

The program in Figure 1, taken from the LOOPS category of SV-COMP14, proved challenging for many of the participating tools, with only 6 out of the 12 entrants solving it correctly. A proof of safety for this program using an abstract interpreter requires a relational domain to represent the invariant $x + y = N$, which is often expensive.

The program in Figure 2 resembles the one in Figure 1, except for the negated assertion at the end. This example is very easy for Bounded Model Checkers, which are able to discover a bug in a single unwinding by assigning $N = 1$.

A slight modification, however, illustrated in Figure 3, increases the number of loop iterations required to trigger the bug to 10^6, exceeding the capability of even the best BMC-based verification tools.

```
unsigned N := *;
unsigned x := N, y := 0;
while (x > 0) {
    x := x − 1;
    y := y + 1;
}
assert (y = N);
```

```
unsigned N = *;
unsigned x := N, y := 0;
while (x > 0) {
    x := x − 1;
    y := y + 1;
}
assert (y ≠ N);
```

```
unsigned N := 10^6;
unsigned x := N, y := 0;
while (x > 0) {
    x := x − 1;
    y := y + 1;
}
assert (y ≠ N);
```

Fig. 1. Safe program   **Fig. 2.** Unsafe program   **Fig. 3.** "Deep" bug

The relative simplicity of the program statements in Figures 1 to 3 makes them amenable to *acceleration* [6,7,12], a technique used to compute the effect of the repeated iteration of statements over integer linear arithmetic. Specifically, the effect of i loop iterations is that x is decreased and y is increased by i. Acceleration, however, is typically restricted to programs over fragments of linear arithmetic for which the transitive closure is effectively computable, thus restricting its applicability to programs whose semantics can be soundly modelled using unbounded integers. In reality, however, the scalar variables in Figures 1 to 3 take their values from the bounded subset $\{0, \ldots, (2^{32} − 1)\}$ of the positive integers $\mathbb{N}_0$. Traditional acceleration techniques do not account for integer overflows. To address this problem, we previously introduced *under-approximate acceleration*, bounding the acceleration to the interval in which the statements behave uniformly [17, 19].

The code snippet in Figure 4 represents an under-approximating accelerator for the loop bodies in Figures 1, 2, and 3. We introduce an auxiliary variable i representing a non-deterministic number of loop iterations. The subsequent assumption guarantees that the accelerated code reflects at least one iteration (and is optional in this example). The assumption that follows warrants the feasibility of the accelerated trace (in general, this condition may contain quantifiers [17, 19]). The effect of i iterations is encoded using the two assignment statements, which constitute the closed forms of the recurrence relations corresponding to the original assignments. The final assumption guarantees that i lies in the range in which the right-hand sides of the assignments behave linearly.

In general, under-approximating accelerators do not reflect all feasible iterations of the loop body. Accordingly, we cannot simply replace the original loop body. Instead, we add back the accelerator as an additional path through the loop, as illustrated in Figure 5.

The transformation preserves safety properties—that is to say, an accelerated program has a reachable, failing assertion iff the original program does. We can see that the failing assertion in Figure 5 is reachable after a single iteration of

```
unsigned i := *;          ⎫
assume (i > 0)            ⎬ iteration counter

assume(x > 0);            } feasibility check

x := x−i;                 ⎫
y := y+i;                 ⎬ acceleration

assume(¬underflow (x));   } iteration bound
```

Fig. 4. Accelerated loop body

```
unsigned N := 10^6, x := N, y := 0;
while (x > 0) {
    if (*) {
        i := *; assume (i > 0);
        x := x−i; y = y+i;
        assume (x ≥ 0);
    } else {
        x := x − 1; y := y + 1;
    }
}
assert (y ≠ N);
```

Fig. 5. Accelerated unsafe program

```
unsigned N := 10^6, x := N, y := 0;
if (x > 0) {
    x := x − 1; y := y + 1;
    if (x > 0) {
        x := x − 1; y := y + 1;
        if (x > 0) {
            x := x − 1;
            y := y + 1;
            assert (x ≤ 0);
        }
    }
}
assert (y = N);
```

Fig. 6. Unwinding ($k = 3$) of safe program with $N = 10^6$

```
      unsigned N := *, x := N, y := 0;
      bool g := *;
1:    while (x > 0) {
          if (*) {
              assume (¬g);
2:            i := *; x := x−i; y = y+i;
              assume (x ≥ 0);
3:            g := T;
          } else {
              x := x − 1; y := y + 1;
              assume (underflow (x));
              g := F;
          }
      }
4:    assert (y = N);
```

Fig. 7. Accelerated and instrumented safe program

the loop, by simply choosing $i = $ N. Since the accelerated program contains a feasible trace leading to a failed assertion, we can conclude that the original program does as well, despite having only considered a single trace of length 1.

While the primary application of BMC is bug detection, contemporary Bounded Model Checkers such as CBMC are able to prove safety in some cases. CBMC unwinds loops up to a predetermined bound k (see Figure 6). *Unwinding assertions* are one possible mechanism to determine whether further unwinding is required [8, 11]. The assertion ($x \leq 0$) in Figure 6 fails if there are feasible program executions traversing the loop more than three times. It is obvious that this assertion will fail for any $k < 10^6$.

Unfortunately, acceleration is ineffective in this setting. Since the accelerator in Figure 5 admits $i = 1$, we have to consider 10^6 unwindings before we

can establish the safety of the program in Figure 1 with $N = 10^6$. For a non-deterministically assigned N, this number increases to 2^{32}.

This outcome is disappointing, since the repeated iteration of the accelerated loop body is redundant. Furthermore, there is no point in taking the unaccelerated path through the loop (unless there is an impending overflow—which can be ruled out in the given program), since the accelerator *subsumes* this execution (with $i = 1$). Thus, if we eliminate all executions that meet either of the criteria above, we do not alter the semantics of the program but may reduce the difficulty of our problem considerably.

Figure 7 shows an accelerated version of the safe program of Figure 1, but instrumented to remove redundant traces. This is achieved by introducing an auxiliary variable g which determines whether the accelerator was traversed in the previous iteration of the loop. This flag is reset in the non-accelerated branch, which, however, in our example is never feasible. It is worth noting that every feasible trace through Listing 1 has a corresponding feasible trace through Listing 7, and vice versa.

The figure to the right shows an execution of the program in Figure 7: This trace is both feasible and safe—the assertion on line 4 is not violated. It is not too difficult to see that *every* feasible trace through the program in Figure 7 has the same length, which means that we can soundly reason about its safety considering traces with a single iteration of the loop, which is a tractable (and indeed, easy) problem.

Loc.	N	x	y	i	g
1	10^4	10^4	0	0	F
2	10^4	10^4	0	0	F
3	10^4	0	10^4	10^4	F
1	10^4	0	10^4	10^4	T
4	10^4	0	10^4	10^4	T

Since the accelerated and instrumented program in Figure 7 is safe, we can conclude that the original program in Figure 1 is safe as well.

We emphasise that our approach neither introduces an over-approximation, nor requires the explicit computation of a fixed point. In addition, it is not restricted to linear integer arithmetic and bit-vectors: our prior work can generate some non-linear accelerators and also allows for the acceleration of a limited class of programs with arrays [17, 19].

3 Notation and Basic Concepts

Let Stmts be the (infinite) set of statements of a simple programming language as defined in Table 1(a), where Exprs and $\mathbb{B}$-Exprs denote expressions and predicates over the program variables Vars, respectively. Stmts* denotes the Kleene closure of Stmts. The language comprises assignments $x := e$ (where $e \in$ Exprs, and $* \in$ Exprs denotes a non-deterministic value) and assumptions abbreviated by $[B]$ (with $B \in \mathbb{B}$-Exprs). Assertions are modeled using assumptions and error locations. For brevity, we omit array accesses. We assume that different occurrences of statements are distinguishable (using the program locations). The semantics is provided in Table 1(a) in terms of the weakest liberal precondition

Table 1. Program Statements and Traces

(a) Syntax and Semantics	(b) Transition Relations for Traces

$$\mathsf{stmt} ::= \mathtt{x} := e \mid [B] \mid \mathsf{skip}$$

$$(\mathtt{x} \in \mathsf{Vars},\ e \in \mathsf{Exprs},\ B \in \mathbb{B}\text{-}\mathsf{Exprs})$$

$$wlp(\mathtt{x} := e, P) \stackrel{\mathrm{def}}{=} P[e/\mathtt{x}]$$
$$wlp(\mathtt{x} := *, P) \stackrel{\mathrm{def}}{=} \forall \mathtt{x} . P$$
$$wlp([B], P) \quad \stackrel{\mathrm{def}}{=} B \Rightarrow P$$
$$wlp(\mathsf{skip}, P) \quad \stackrel{\mathrm{def}}{=} P$$

$$[\![\mathsf{stmt}]\!] \stackrel{\mathrm{def}}{=} \neg wlp(\mathsf{stmt}, \bigvee_{\mathtt{x} \in \mathsf{Vars}} \mathtt{x} \neq \mathtt{x}')$$
$$\mathsf{id} \stackrel{\mathrm{def}}{=} [\![\mathsf{skip}]\!]$$
$$[\![\mathsf{stmt}_1 \cdot \mathsf{stmt}_2]\!] \stackrel{\mathrm{def}}{=} [\![\mathsf{stmt}_1]\!] \circ [\![\mathsf{stmt}_2]\!]$$
$$[\![\mathsf{stmt}^n]\!] \stackrel{\mathrm{def}}{=} [\![\mathsf{stmt}]\!]^n,$$
$$\mathsf{stmt}^0 \stackrel{\mathrm{def}}{=} \varepsilon,$$
$$\mathsf{stmt}^n \stackrel{\mathrm{def}}{=} \mathsf{stmt} \cdot (\mathsf{stmt}^{(n-1)})$$
where
$$[\![\mathsf{stmt}]\!]^0 \stackrel{\mathrm{def}}{=} \mathsf{id},$$
$$[\![\mathsf{stmt}]\!]^n \stackrel{\mathrm{def}}{=} [\![\mathsf{stmt}]\!] \circ ([\![\mathsf{stmt}]\!]^{(n-1)})$$

wlp as defined in [25]. (The weakest liberal precondition $wlp(\mathtt{stmt}, P)$ yields the weakest condition under which $\mathtt{stmt}$ either does not terminate or establishes P.) Programs are represented using control flow automata.

Definition 1 (CFA). *Let* Stmts *be as introduced above in Table 1(a). A* control flow automaton *P is a directed graph* $\langle V, E, v_0 \rangle$*, where V is a finite set of vertices,* $\mathsf{Stmts}_P \subseteq \mathsf{Stmts}$ *is a finite set of statements,* $E \subseteq (V \times \mathsf{Stmts}_P \times V)$ *is a set of edges, and* $v_0 \in V$ *is the initial vertex. We write* $v \stackrel{\mathsf{stmt}}{\longrightarrow} u$ *if* $\langle u, \mathsf{stmt}, v \rangle \in E$.

A program state σ is a total function assigning a value to each program variable in Vars. States denotes the set of program states (not to be confused with vertices of a CFA). A transition relation $T \subseteq \mathsf{States} \times \mathsf{States}$ associates states with their successor states. Given Vars, let Vars' be a corresponding set of primed variables encoding successor states. The symbolic transition relation for a statement or trace is a predicate over $\mathsf{Vars} \cup \mathsf{Vars}'$ and can be derived using wlp as indicated in Table 1(b) (cf. [9]). We write $\langle \sigma, \sigma' \rangle \in [\![\mathsf{stmt}]\!]$ if $[\![\mathsf{stmt}]\!]$ evaluates to true under σ and σ' (i.e., $\sigma, \sigma' \models [\![\mathsf{stmt}]\!]$). The composition $[\![\pi_1]\!] \circ [\![\pi_2]\!]$ of two relations $[\![\pi_1]\!]$ and $[\![\pi_2]\!]$ is the relation $\{\langle \sigma, \sigma' \rangle \mid \exists \sigma'' . \langle \sigma, \sigma'' \rangle \in [\![\pi_1]\!]$ and $\langle \sigma'', \sigma' \rangle \in [\![\pi_2]\!]\}$, symbolically represented as a predicate over $\mathsf{Vars} \cup \mathsf{Vars}'$. A trace π is *feasible* if there exist states σ, σ' such that $\langle \sigma, \sigma' \rangle \in [\![\pi]\!]$.

Given a CFA $P \stackrel{\mathrm{def}}{=} \langle V, E, v_0 \rangle$, a trace $\pi \stackrel{\mathrm{def}}{=} \mathsf{stmt}_i \cdot \mathsf{stmt}_{i+1} \cdots \mathsf{stmt}_n$ (where $v_{j-1} \stackrel{\mathsf{stmt}_j}{\longrightarrow} v_j$ for $i < j \leq n$ and $\pi \in \mathsf{Stmts}_P^*$) of length $|\pi| = n - i + 1$ is *looping* (with head v_i) iff $v_i = v_n$, and *accepted* by the CFA iff $v_i = v_0$. We use $\mathcal{L}_P$ (where $\mathcal{L}_P \subseteq \mathsf{Stmts}_P^*$) to denote the set of all traces that are accepted by the CFA P. Abusing our notation, we write $v_i \stackrel{\pi}{\longrightarrow} v_j$ to denote path starting at v_i and ending at v_j and corresponding to the trace π.

A state σ is *reachable* from an initial state σ_0 iff there exists a trace π accepted by the CFA such that $\langle \sigma_0, \sigma \rangle \in [\![\pi]\!]$. The reachability diameter [5, 20] of a transition relation is the smallest number of steps required to reach all reachable states:

Definition 2 (Reachability Diameter). *Given a CFA with initial state σ_0, the* reachability diameter *is the smallest n such that for every state σ reachable from σ_0 there exists a feasible trace π of length at most n accepted by the CFA with $\langle \sigma_0, \sigma \rangle \in [\![\pi]\!]$.*

To show that a CFA does not violate a given safety (or reachability) property, it is sufficient to explore all feasible traces whose length does not exceed the reachability diameter. In the presence of looping traces, however, the reachability diameter of a program can be infinitely large.

Acceleration [6,7,12] is a technique to compute the reflexive transitive closure $[\![\pi]\!]^* \stackrel{\text{def}}{=} \bigcup_{i=0}^{\infty} [\![\pi]\!]^i$ for a looping trace π. Equivalently, $[\![\pi]\!]^*$ can be expressed as $\exists i \in \mathbb{N}_0 \,.\, [\![\pi]\!]^i$. The aim of acceleration is to express $[\![\pi]\!]^*$ in a decidable fragment of logic. In general, this is not possible, even if $[\![\pi]\!]$ is defined in a decidable fragment of integer arithmetic such as Presburger arithmetic. For octagonal relations $[\![\pi]\!]$, however, the transitive closure is $[\![\pi]\!]^*$ is Presburger-definable and effectively computable [6, 12].

Definition 3 (Accelerated Transitions). *Given a looping trace $\pi \in \mathcal{L}_P$, we say that a trace $\hat{\pi} \in \mathsf{Stmts}^*$ is an* accelerator *for π if $[\![\hat{\pi}]\!] \equiv [\![\pi]\!]^*$.*

An accelerator $\widetilde{\pi} \in \mathsf{Stmts}^*$ is under-approximating *if the number of iterations is bounded from above by a function $\beta : \mathsf{States} \to \mathbb{N}_0$ of the starting state σ:*

$$\langle \sigma, \sigma' \rangle \in [\![\widetilde{\pi}]\!] \quad \textit{iff} \quad \exists i \in \mathbb{N}_0 \,.\, i \leq \beta(\sigma) \wedge \langle \sigma, \sigma' \rangle \in [\![\pi]\!]^i$$

We require that the function β satisfies the following monotonicity condition:

$$\big(i \leq \beta(\sigma) \wedge \langle \sigma, \sigma' \rangle \in [\![\pi]\!]^i\big) \Rightarrow (\beta(\sigma') \leq \beta(\sigma) - i) \tag{1}$$

Intuitively, β (which is not necessarily explicit in the encoding of $\widetilde{\pi}$) restricts i to the range of iterations of π which can be accelerated accurately. We say that $\widetilde{\pi}$ is strictly under-approximating *if $[\![\widetilde{\pi}]\!] \subset [\![\hat{\pi}]\!]$.*

We introduced under-approximating accelerators for linear integer arithmetic and the theories of bit-vectors and arrays in [17] in order to accelerate the detection of counterexamples. Under-approximations are caused by transition relations that can only be accelerated within certain intervals, e.g., the range in which no overflow occurs in the case of bit-vectors, or in which no conflicting assignments to array elements are made. The bound function β restricts this interval accordingly.

Example 1. An under-approximating accelerator for the statement $\mathtt{x} := \mathtt{x} + 1$, where $\mathtt{x}$ is a 32-bit-wide unsigned integer, can be given as the statement sequence

$$\widetilde{\pi} \stackrel{\text{def}}{=} i := *; \, [\mathtt{x} + i < 2^{32}]; \, \mathtt{x} := \mathtt{x} + i$$

with transition relation $\exists i \,.\, \big(\mathtt{x} + i < 2^{32}\big) \wedge (\mathtt{x}' = \mathtt{x} + i)$. Note that β is implicit here and that the alphabet of $\widetilde{\pi}$ is not restricted to Stmts_P of the CFA P.

4 Diameter Reduction via Acceleration

In this section, we introduce a reachability-preserving program transformation that reduces the reachability diameter of a CFA. While a similar transformation is used in [17] to detect counterexamples with loops, our goal here is to reduce the diameter in order to enable safety proofs (see Section 5).

Definition 4 (Accelerated CFA). *Let* $P \stackrel{\text{def}}{=} \langle V, E, v_0 \rangle$ *be a CFA over the alphabet* Stmts_P, *and let* $\pi_1, \ldots, \pi_k$ *be traces in* P *looping with heads* $v_1, \ldots, v_k \in V$, *respectively. Let* $\widehat{\pi}_1, \ldots \widehat{\pi}_k$ *be the (potentially under-approximating) accelerators for* $\pi_1, \ldots, \pi_k$. *Then the* accelerated CFA $\widehat{P} \stackrel{\text{def}}{=} \langle \widehat{V}, \widehat{E}, v_0 \rangle$ *for* P *is the CFA* P *augmented with non-branching paths* $v_i \xrightarrow{\widehat{\pi}_i} v_i$ $(1 \leq i \leq k)$.

A trace is *accelerated* if it traverses a path in $\widehat{P}$ that corresponds to an accelerator. A trace π_1 *subsumes* a trace π_2, denoted by $\pi_2 \preceq \pi_1$, if $[\![\pi_2]\!] \subseteq [\![\pi_1]\!]$. Accordingly, $\pi \preceq \widehat{\pi}$ and $\widetilde{\pi} \preceq \widehat{\pi}$ (which follows from Definition 3, since $[\![\pi]\!]^i \subseteq [\![\pi]\!]^*$ for any i). We extend the relation $\preceq$ to sets of traces: $\Pi_1 \preceq \Pi_2$ if $\left(\bigcup_{\pi \in \Pi_1} [\![\pi]\!] \right) \subseteq \left(\bigcup_{\pi \in \Pi_2} [\![\pi]\!] \right)$. A trace π is *redundant* if $\{\pi\}$ is subsumed by the set $\Pi \setminus \{\pi\}$ of other traces in the CFA.

Lemma 1. *Let* $\widetilde{\pi}$ *be an under-approximating accelerator for the looping trace* π. *Then* $\widetilde{\pi} \cdot \widetilde{\pi} \preceq \widetilde{\pi}$ *holds. Similarly, for an accelerator* $\widehat{\pi}$ *of* π *it holds that* $\widehat{\pi} \cdot \widehat{\pi} \preceq \widehat{\pi}$.

Proof. For accelerators that are not strictly under-approximating the claim holds trivially. Otherwise, we have

$$\langle \sigma, \sigma'' \rangle \in [\![\widetilde{\pi} \cdot \widetilde{\pi}]\!] \qquad \Leftrightarrow$$
$$\exists \sigma' . \exists i, j \in \mathbb{N}_0 . \left(\begin{matrix} \langle \sigma, \sigma' \rangle \in [\![\pi]\!]^i \wedge i \leq \beta(\sigma) & \wedge \\ \langle \sigma', \sigma'' \rangle \in [\![\pi]\!]^j \ \wedge j \leq \beta(\sigma') \end{matrix} \right)$$

If σ' exists, Condition 1 in Definition 3 guarantees that $(\beta(\sigma') \leq \beta(\sigma) - i)$, and therefore $\langle \sigma, \sigma'' \rangle \in [\![\widetilde{\pi} \cdot \widetilde{\pi}]\!]$ implies

$$\exists i, j \in \mathbb{N}_0 . \langle \sigma, \sigma'' \rangle \in [\![\pi]\!]^{i+j} \wedge \underbrace{i \leq \beta(\sigma) \wedge j \leq \beta(\sigma) - i}_{(i+j) \leq \beta(\sigma)} .$$

By replacing $i + j$ with a single variable i we arrive at the definition of $[\![\widetilde{\pi}]\!]$.

We emphasize that Lemma 1 holds for accelerators $\widehat{\pi}$ as well as (strictly) under-approximating accelerators $\widetilde{\pi}$ (cf. Definition 3). In the following, we use $\widehat{\pi}$ to denote accelerators as well as under-approximating accelerators unless explicitly stated otherwise.

The following theorem states that the transformation in Definition 4 preserves the reachability of states and never increases the reachability diameter.

Theorem 1. *Let* P *be a CFA and* $\widehat{P}$ *a corresponding accelerated CFA as in Definition 4. Then the following claims hold:*

1. *Every trace in P is subsumed by at least one trace in $\widehat{P}$.*
2. *Let π_1 be an accelerated trace accepted by $\widehat{P}$, and let $\langle \sigma_0, \sigma \rangle \in [\![\pi_1]\!]$. Then there exists a trace π_2 accepted by P such that $\langle \sigma_0, \sigma \rangle \in [\![\pi_2]\!]$.*

Proof. Part 1 of the theorem holds because P is a sub-graph of $\widehat{P}$. For the second part, assume that $\widehat{\pi}_1, \ldots \widehat{\pi}_k$ are the accelerators occurring in π_1. Then there are $i_1, \ldots, i_k \in \mathbb{N}$ such that $\pi_2 \stackrel{\text{def}}{=} \pi_1[\pi_1^{i_1}/\widehat{\pi}_1] \cdots [\pi_k^{i_k}/\widehat{\pi}_k]$ and $\langle \sigma_0, \sigma \rangle \in [\![\pi_2]\!]$.

The diameter of a CFA is determined by the longest of the shortest traces from the initial state σ_0 to all reachable states [20]. Accordingly, the transformation in Definition 4 results in a reduction of the diameter if it introduces a shorter accelerated trace that results in the redundancy of this longest shortest trace. In particular, acceleration may reduce an infinite diameter to a finite one.

5   Checking Safety with Trace Automata

Bounded Model Checking owes its industrial success largely to its effectiveness as a bug-finding technique. Nonetheless, BMC can also be used to prove safety properties if the unwinding bound exceeds the reachability diameter. In practice, however, the diameter can rarely be determined statically. Instead, *unwinding assertions* are used to detect looping traces that become infeasible if expanded further [8]. Specifically, an unwinding assertion is a condition that fails for an unwinding bound k and a trace $\pi_1 \cdot \pi_2^k$ if $\pi_1 \cdot \pi_2^{k+1}$ is feasible, indicating that further iterations may be required to exhaustively explore the state space.

In the presence of accelerators, however, unwinding assertions are inefficient. Since $\widehat{\pi} \cdot \widehat{\pi} \preceq \widehat{\pi}$ (Lemma 1), repeated iterations of accelerators are redundant. The unwinding assertion for $\pi_1 \cdot \widehat{\pi}_2$, however, fails if $\pi_1 \cdot \widehat{\pi}_2 \cdot \widehat{\pi}_2$ is feasible, suggesting that further unwinding is required. Accordingly, the approximate diameter as determined by means of unwinding assertions for an accelerated program $\widehat{P}$ is the *same* as for the corresponding non-accelerated program P.

In the following, we present a technique that remedies the deficiency of unwinding assertions in the presence of accelerators by *restricting* the language accepted by a CFA.

Definition 5 (Restriction Language). *Let $\widehat{P}$ an accelerated CFA for P over the vocabulary $\mathsf{Stmts}_{\widehat{P}}$. For each accelerator $\widehat{\pi} \in \mathsf{Stmts}_{\widehat{P}}^+$, let $\pi \in \mathsf{Stmts}_P^+$ be the corresponding looping trace. The restriction language $\mathcal{L}_R$ for $\widehat{P}$ comprises all traces with a sub-trace characterised by the regular expression $(\pi \,|\, (\widehat{\pi} \cdot \widehat{\pi}))$ for all accelerators $\widehat{\pi}$ in $\widehat{P}$ with $\pi \preceq \widehat{\pi}$.*

The following lemma enables us to eliminate traces of an accelerated CFA $\widehat{P}$ that are in the restriction language $\mathcal{L}_R$.

Lemma 2. *Let $\widehat{P}$ be an accelerated CFA, and $\mathcal{L}_R$ be the corresponding restriction language. Let π_1 be a trace accepted by $\widehat{P}$ such that $\pi_1 \in \mathcal{L}_R$. Then there exists a trace π_2 which is accepted by $\widehat{P}$ such that $\pi_1 \preceq \pi_2$ and π_1 is not a sub-trace of π_2.*

Proof. The regular expression $(\pi \,|\, (\widehat{\pi}{\cdot}\widehat{\pi}))$ can match the trace π_1 for two reasons:

(a) The trace π_1 contains a sub-trace which is a looping trace π with a corresponding accelerator $\widehat{\pi}$ and $\pi \preceq \widehat{\pi}$. We obtain π_2 by replacing π with $\widehat{\pi}$.
(b) The trace π_1 contains the sub-trace $\widehat{\pi}{\cdot}\widehat{\pi}$ for some accelerator $\widehat{\pi}$. Since $\widehat{\pi}{\cdot}\widehat{\pi} \preceq \widehat{\pi}$ (Lemma 1), we replace the sub-trace with $\widehat{\pi}$ to obtain π_2.

Since the accelerator $\widehat{\pi}$ differs from the sub-trace it replaces in case (a), and $|\pi_2| < |\pi_1|$ in case (b), π_1 can not be contained in π_2.

Using Lemma 2 and induction over the number of traces and accelerators, it is admissible to eliminate all traces accepted by $\widehat{P}$ and contained in $\mathcal{L}_R$ without affecting the reachability of states:

Theorem 2. *Let $\mathcal{L}_{\widehat{P}}$ be the language comprising all traces accepted by an accelerated CFA $\widehat{P}$ and $\mathcal{L}_R$ be the corresponding restriction language. Then every trace $\pi \in \mathcal{L}_{\widehat{P}}$ is subsumed by the traces in $\mathcal{L}_{\widehat{P}} \setminus \mathcal{L}_R$.*

Notably, Definition 5 explicitly excludes accelerators $\widehat{\pi}$ that do not satisfy $\pi \preceq \widehat{\pi}$, a requirement that is therefore implicitly present in Lemma 2 as well as Theorem 2. The rationale behind this restriction is that strictly underapproximating accelerators $\widetilde{\pi}$ do not necessarily have this property. However, even if $\widetilde{\pi}$ does not subsume π in general, we can characterize the set of starting states in which it does:

$$\{\sigma \,|\, \langle \sigma, \sigma' \rangle \in [\![\pi]\!] \Rightarrow \langle \sigma, \sigma' \rangle \in [\![\widetilde{\pi}]\!]\} \tag{2}$$

In order to determine whether a looping path π is redundant, we presume for each accelerated looping trace π the existence of a predicate $\varphi_\pi \in \mathsf{Exprs}$ and an assumption statement $\tau_\pi \stackrel{\text{def}}{=} [\varphi_\pi]$ such that

$$[\![\, \tau_\pi \,]\!] \stackrel{\text{def}}{=} \{\langle \sigma, \sigma \rangle \,|\, \langle \sigma, \sigma' \rangle \in [\![\pi]\!] \Rightarrow \langle \sigma, \sigma' \rangle \in [\![\widetilde{\pi}]\!]\} \tag{3}$$

Analogously, we can define the dual statement $\overline{\tau}_\pi \stackrel{\text{def}}{=} [\neg\varphi_\pi]$. Though both $[\![\tau_\pi]\!]$ and $[\![\overline{\tau}_\pi]\!]$ are non-total transition relations, their combination $[\![\tau_\pi]\!] \cup [\![\overline{\tau}_\pi]\!]$ is total. Moreover, it does not modify the state, i.e., $[\![\tau_\pi]\!] \cup [\![\overline{\tau}_\pi]\!] \equiv [\![\mathsf{skip}]\!]$. It is therefore evident that replacing the head v of a looping trace π with the subgraph $u \overset{\tau_\pi}{\underset{\overline{\tau}_\pi}{\rightleftarrows}} w$ (and reconnecting the incoming and outgoing edges of v to u and w, respectively) preserves the reachability of states. It does, however change the traces of the CFA. After the modification, the looping traces $\tau_\pi \cdot \pi$ and $\overline{\tau}_\pi \cdot \pi$ replace π. By definition of τ_π, we have $\tau_\pi \cdot \pi \preceq \widetilde{\pi}$. Consequently, if we accelerate the newly introduced looping trace $\tau_\pi \cdot \pi$, Definition 5 and therefore Lemma 2 as well as Theorem 2 apply.

The discriminating statement $\overline{\tau}_\pi$ for the example path $\mathtt{x} := \mathtt{x} + 1$ at the end of Section 3, for instance, detects the presence of an overflow. For this specific example, $\overline{\tau}_\pi$ is the assumption $[\mathtt{x} = 2^{32} - 1]$. In practice, however, the bit-level-accurate encoding of CBMC provides a mechanism to detect an overflow

after it happened. Therefore, we introduce statements $\overline{\tau}_\pi \overset{\text{def}}{=} [\mathsf{overflow}(\mathtt{x})]$ and $\tau_\pi \overset{\text{def}}{=} [\neg\mathsf{overflow}(\mathtt{x})]$ that determine the presence of an overflow at the end of the looping trace. The modification and correctness argument for this construction is analogous to the one above.

In order to recognize redundant traces, we use a *trace automaton* that accepts the restriction language $\mathcal{L}_R$.

Definition 6 (Trace Automaton). *A trace automaton T_R for $\mathcal{L}_R$ is a deterministic finite automaton (DFA) over the alphabet $\mathsf{Stmts}_{\widehat{P}}$ that accepts $\mathcal{L}_R$.*

Since $\mathcal{L}_R$ is regular, so is its complement $\overline{\mathcal{L}}_R$. In the following, we describe an instrumentation of a CFA $\widehat{P}$ which guarantees that every trace accepted by T_R and $\widehat{P}$ becomes infeasible. To this end, we construct a DFA T_R recognising $\mathcal{L}_R$, starting out with an ϵ-NFA which we then determinise using the subset construction [1]. While this yields (for a CFA with k statements) a DFA with $O(2^k)$ states in the worst case, in practice the DFAs generated are much smaller.

We initialise the set the vertices of the instrumented CFA $\tilde{P}$ to the vertices of $\widehat{P}$. We inline T_R by creating a fresh integer variable g in $\tilde{P}$ which encodes the state of T_R and is initialised to 0. For each edge $u \overset{s}{\longrightarrow} v \in \widehat{P}$, we consider all transitions $n \overset{s}{\longrightarrow} m \in T_R$. If there are no such transitions, we copy the edge $u \overset{s}{\longrightarrow} v$ into $\tilde{P}$. Otherwise, we add edges as follows:

- If m is an accepting state, we do not add an edge to $\tilde{P}$.
- Otherwise, construct a new statement sequence $l \overset{\text{def}}{=} [\mathsf{g} = n]$; s; $\mathsf{g} := m$ and add the path $u \overset{l}{\longrightarrow} v$ to $\tilde{P}$, which simulates the transition $n \overset{s}{\longrightarrow} m$.

Since we add at most a constant number of edges to $\tilde{P}$ for each transition in T_R, this construction's time and space complexity are both $\Theta(\|\widehat{P}\| + \|T_R\|)$. By construction, if a trace π accepted by CFA $\tilde{P}$ projected to $\mathsf{Stmts}_{\widehat{P}}$ is contained in the restriction language $\mathcal{L}_R$, then π is infeasible. Conceptually, our construction suppresses traces accepted by $\mathcal{L}_R$ and retains the remaining executions.

An example is shown in Figure 8. The CFA in Figure 8(a) represents an unaccelerated loop with a single path through its body. After adding an extra path to account for integer overflow, we arrive at the CFA in Figure 8(b). We are able to find an accelerator for the non-overflowing path, which we add to the CFA resulting in Figure 8(c). We use $\tilde{\pi}$ to represent the accelerator π for the corresponding path. Then the restriction language is represented by the regular expression $(\pi \,|\, \tilde{\pi} \cdot \tilde{\pi})$. The corresponding 4-state trace automaton is shown in Figure 8(d). By combining the trace automaton and the CFA using the algorithm outlined above, we obtain the restricted CFA in Figure 8(e).

In the restricted CFA $\tilde{P}$, looping traces π that can be accelerated and redundant iterations of accelerators are infeasible and therefore do not trigger the failure of unwinding assertions. A CFA is safe if all unwinding assertions hold and no safety violation can be detected for a given bound k. The reduction of the diameter achieved by acceleration (Section 4) in combination with the construction presented in this section enables us to establish the safety of CFAs in cases in which traditional BMC would have been unable to do so. Section 6 provides an experimental evaluation demonstrating the viability of our approach.
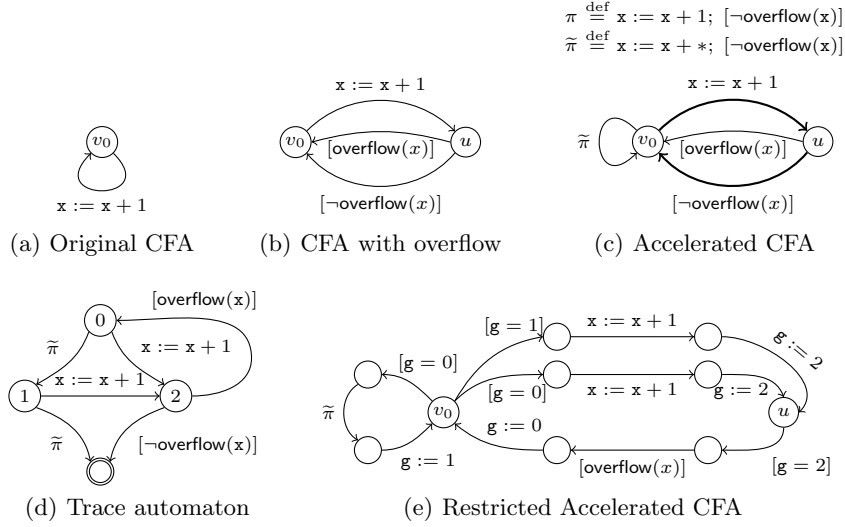
$$\pi \stackrel{\text{def}}{=} \mathbf{x} := \mathbf{x} + 1; \ [\neg\text{overflow}(\mathbf{x})]$$
$$\widetilde{\pi} \stackrel{\text{def}}{=} \mathbf{x} := \mathbf{x} + *; \ [\neg\text{overflow}(\mathbf{x})]$$

(a) Original CFA   (b) CFA with overflow   (c) Accelerated CFA

(d) Trace automaton   (e) Restricted Accelerated CFA

Fig. 8. Accelerating a looping path

6 Experimental Evaluation

We evaluate the effect of instrumenting accelerated programs with trace automata and determine the direct cost of constructing the automata as well as the impact of trace automata on the ability to find bugs on the one hand and prove safety on the other.

Our evaluation is based on the LOOPS category of the benchmarks from SV-COMP14 and a number of small but difficult hand-crafted examples[3]. Our hand-crafted examples require precise reasoning about arithmetic and arrays. The unsafe examples have deep bugs, and the safe examples feature unbounded loops. The SV-COMP14 benchmarks are largely arithmetic in nature. They often require non-trivial arithmetic invariants to be inferred, but rarely require complex reasoning about arrays. Furthermore, all bugs of the unsafe SV-COMP14 benchmarks occur within a small number of loop iterations.

In all of our experiments we used CBMC taken from the public SVN at r3849 to perform the transformation. Since CBMC's acceleration procedure generates assertions with quantified arrays, we used Z3 [24] version 4.3.1 as the backend decision procedure. All of the experiments were performed with a timeout of 30 s and very low unwinding limits. We used an unwinding limit of 100 for unaccelerated programs and an unwinding limit of 3 for their accelerated counterparts.

The version of CBMC we use has incomplete acceleration support, e.g., it is unable to accelerate nested loops. As a result, there are numerous benchmarks that it cannot accelerate. We stress that our goal here is to evaluate the effect of

---

[3] These examples have been accepted into SV-COMP15, where they can be found in the LOOP-ACCELERATION directory.

Table 2. Summary of experimental results

	#Benchmarks	CBMC #Correct	CBMC Time (s)	#Benchmarks accelerated	CBMC + Acceleration #Correct	CBMC + Acceleration Acceleration Time (s)	CBMC + Acceleration Checking Time (s)	CBMC + Acceleration + Trace Automata #Correct	CBMC + Acceleration + Trace Automata Acceleration Time (s)	CBMC + Acceleration + Trace Automata Checking Time (s)
SV-COMP14 safe	35	14	298.73	21	2	23.24	244.72	14	23.86	189.61
SV-COMP14 unsafe	32	20	394.96	18	11	15.79	197.94	12	16.51	173.74
Crafted safe	15	0	11.42	15	0	2.75	32.41	15	2.91	1.59
Crafted unsafe	14	0	9.03	14	14	2.85	12.24	14	2.95	2.55

adding trace automata to accelerated programs. Acceleration has already proven to be a useful technique for both bug-finding and proof [15, 17, 21, 26, 27] and we are interested in how well inlined trace automata can complement it.

Our experimental results are summarised in Table 2, and the full results are provided in our technical report [18]. We discuss the results in the remainder of this section.

Cost of Trace Automata. To evaluate the direct cost of constructing the trace automata, we direct the reader's attention to Table 2 and the columns headed "acceleration time". The first "acceleration time" column shows how long it took to generate an accelerated program without a trace automaton, whereas the second shows how long it took when a trace automaton was included. For all of these benchmarks, the additional time taken to build and insert the trace automaton is negligible. Detailed information about the size increase of the instrumented binary over the accelerated binary is provided in our technical report [18, Appendix B]. The average increase is about 15%, but the maximum increase is 77%. There is still room for optimisation, as we do not minimise the automata before inserting them.

Bug Finding. In the following, we evaluate the effectiveness of our technique for bug finding. The current state-of-the-art method for bug finding is BMC [3]. To provide a baseline for bug finding power, we start by evaluating the effect of just combining acceleration with BMC. We then evaluate the impact of adding trace automata, as compared to acceleration without trace automata. Our hypothesis is that adding trace automata has negligible impact on acceleration's ability to find bugs. The statistics we use to measure these effects are the number of bugs found and the time to find them. We measure these statistics for each of three techniques: BMC alone, acceleration with BMC, and our combination of acceleration, trace automata and BMC.

The results are summarised in Table 2. In SV-COMP14, almost all of the bugs occur after a small number of unwindings. In these cases, there are no deep

loops to accelerate so just using CBMC allows the same bugs to be reached, but without the overhead of acceleration (which causes some timeouts to be hit). In the crafted set the bugs are much deeper, and we can see the effect of acceleration in discovering these bugs – none of the bugs are discovered by CBMC, but each of the configurations using acceleration finds all 14 bugs.

In both of the benchmark sets, adding trace automata does not negatively impact the bug finding ability of acceleration. Indeed, for the crafted set the addition of trace automata significantly improves bug finding performance – the total time needed to find the 14 bugs is reduced from 12.31 s to 1.85 s.

Safety Proving. We evaluate the effectiveness of our technique for proving safety *with BMC*, the key contribution of this paper. Our two benchmark sets have very different characteristics with respect to the safety proofs required for their safe examples. As can be seen from Table 2, 14 of the SV-COMP14 benchmarks can be proved safe using just BMC. That is, they can be exhaustively proved safe after a small number of loop unwindings. For the 14 cases that were provable using just BMC, none had loops that could execute for more than 10 iterations.

Of the 35 safe SV-COMP14 benchmarks, 21 contained loops that could be accelerated by our implementation. Of these 21 cases, 14 were proved safe using trace automata. These are not the same 14 cases that were proved by CBMC, and notably 8 cases with unbounded loops are included, which would be impossible to prove safe with just BMC.

Additionally we were able to solve the SUM_ARRAY_TRUE benchmark (given as Figure 9) in 1.75 s. Of all the tools entered in SV-COMP14, the only tools to claim "safe" for this benchmark were BMC-based (even though analyses able to solve this problem exist, e.g. [10]), and as such do not generate safety proofs.

```
unsigned N := ∗, i;
int a[M], b[M], c[M];

for (i = 0; i < M; i := i + 1) {
    c[i] := a[i] + b[i];
}
for (i = 0; i < M; i := i + 1) {
    assert (c[i] = a[i] + b[i]);
}
```

Fig. 9. The SUM_ARRAYS benchmark from SV-COMP14

For the 7 cases where accelerators were produced but we were unable to prove safety, 5 are due to timeouts, 1 is a crash in CBMC and 1 is an "incomplete". The 5 timeouts are due to the complexity of the SMT queries we produce. For these timeout cases, we generate assertions which contain non-linear multiplication and quantification over arrays, which are very difficult for Z3 to solve. The

"incomplete" case (TREX03_TRUE) requires reasoning about accelerated paths that commute with each other, which we leave as future work.

7   Related Work

The diameter of a transition system was introduced in Biere et al.'s seminal paper on BMC [5] in the context of finite-state transition relations. For finite-state transition relations, approximations of the diameter can be computed symbolically by constraining the unwound transition relation to exclude executions that visit states repeatedly [20]. For software, however, this technique is ineffective. Baumgartner and Kühlmann use structural transformations of hardware designs to reduce the reachability diameter of a hardware design to obtain a complete BMC-based verification method [2]. This technique is not applicable in our context.

Trace automata are introduced in [13] as abstractions of safe traces of CFAs [14], constructed by means of interpolation. We use trace automata to recognize redundant traces.

Acceleration amounts to computing the transitive closure of a infinite state transition relation [6, 7, 12]. Acceleration has been successfully combined with abstract interpretation [26] as well as interpolation-based invariant construction [15]. These techniques rely on over-approximate abstractions to prove safety. We previously used acceleration and under-approximation to quickly find deep bugs [17, 19, 21, 22]. The quantified transition relations used to encode under-approximations pose an insurmountable challenge to interpolation-based refinement techniques [17, 19], making it difficult to combine the approach with traditional software model checkers.

8   Conclusion

The reduction of the reachability diameter of a program achieved by acceleration and loop under-approximation enables the rapid detection of bugs by means of BMC. Attempts to apply under-approximation to prove safety, however, have been disappointing: the simple mechanism deployed by BMC-based tools to detect that an unwinding bound is exhaustive is not readily applicable to accelerated programs.

In this paper, we present a technique that constrains the search space of an accelerated program, enabling BMC-based tools to prove safety using a small unwinding depth. To this end, we use *trace automata* to eliminate redundant execution traces resulting from under-approximating acceleration. Unlike other safety provers, our approach does not rely on over-approximation, nor does it require the explicit computation of a fixed point. Using unwinding assertions, the smaller diameter can be computed by means of a simple satisfiability check.

15

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
2. Baumgartner, J., Kuehlmann, A.: Enhanced diameter bounding via structural transformations. In: Design, Automation and Test in Europe (DATE). pp. 36–41. IEEE (2004)
3. Beyer, D.: Status Report on Software Verification (Competition Summary SV-COMP 2014). In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Lecture Notes in Computer Science, vol. 8413, pp. 373–388. Springer (2014)
4. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Programming Language Design and Implementation (PLDI). pp. 300–309. ACM (2007)
5. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer (1999)
6. Boigelot, B.: Symbolic Methods for Exploring Infinite State Spaces. Ph.D. thesis, Université de Liège (1999)
7. Bozga, M., Iosif, R., Konecný, F.: Fast acceleration of ultimately periodic relations. In: Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 6174, pp. 227–242. Springer (2010)
8. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 168–176. Springer (2004)
9. Dijkstra, E.W., et al.: From predicate transformers to predicates (April 1982), tuesday Afternoon Club Manuscript EWD821
10. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 6012, pp. 246–266. Springer (2010)
11. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 27(7), 1165–1178 (July 2008)
12. Finkel, A., Leroux, J.: How to compose Presburger-accelerations: Applications to broadcast protocols. In: Foundations of Software Technology and Theoretical Computer Science (FST-TCS). Lecture Notes in Computer Science, vol. 2556, pp. 145–156. Springer (2002)
13. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Static Analysis Symposium (SAS). Lecture Notes in Computer Science, vol. 5673, pp. 69–85. Springer (2009)
14. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Principles of Programming Languages (POPL), pp. 58–70. ACM (2002)
15. Hojjat, H., Iosif, R., Konecny, F., Kuncak, V., Ruemmer, P.: Accelerating interpolants. In: Automated Technology for Verification and Analysis (ATVA). Lecture Notes in Computer Science, vol. 7561, pp. 187–202. Springer (2012)
16. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Fundamental Approaches to Software Engineering (FASE). Lecture Notes in Computer Science, vol. 5503, pp. 470–485. Springer (2009)

17. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. In: Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 8044, pp. 381–396. Springer (2013)
18. Kroening, D., Lewis, M., Weissenbacher, G.: Proving safety with trace automata and bounded model checking. CoRR abs/1410.5764 (2014), http://arxiv.org/abs/1410.5764
19. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. Formal Methods in System Design (2015), (to appear)
20. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: Verification, Model Checking and Abstract Interpretation (VMCAI). Lecture Notes in Computer Science, vol. 2575, pp. 298–309. Springer (2003)
21. Kroening, D., Weissenbacher, G.: Counterexamples with loops for predicate abstraction. In: Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 4144, pp. 152–165. Springer (2006)
22. Kroening, D., Weissenbacher, G.: Verification and falsification of programs with loops using predicate abstraction. Formal Aspects of Computing 22, 105–128 (2010)
23. McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 4144, pp. 123–136. Springer (2006)
24. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 4963, pp. 337–340 (2008)
25. Nelson, G.: A generalization of Dijkstra's calculus. ACM Transactions on Programming Languages and Systems (TOPLAS) 11(4), 517–561 (1989)
26. Schrammel, P., Jeannet, B.: Logico-numerical abstract acceleration and application to the verification of data-flow programs. In: Static Analysis Symposium (SAS). Lecture Notes in Computer Science, vol. 6887, pp. 233–248. Springer (2011)
27. Schrammel, P., Melham, T., Kroening, D.: Chaining test cases for reactive system testing. In: Testing Software and Systems (ICTSS). Lecture Notes in Computer Science, vol. 8254, pp. 133–148. Springer (2013)