Transition Power Abstractions for Deep Counterexample Detection*

Martin Blicha[1,3] (✉), Grigory Fedyukovich[2], Antti E.J. Hyvärinen[1], and Natasha Sharygina[1]

[1] Università della Svizzera italiana, Lugano, Switzerland, first.last@usi.ch
[2] Florida State University, Tallahassee, FL, USA, grigory@cs.fsu.edu
[3] Charles University, Prague, Czech Republic

Abstract. While model checking safety of infinite-state systems by inferring state invariants has steadily improved recently, most verification tools still rely on a technique based on bounded model checking to detect safety violations. In particular, the current techniques typically analyze executions by unfolding transitions one step at a time, and the slow growth of execution length prevents detection of deep counterexamples before the tool reaches its limits on computations. We propose a novel model-checking algorithm that is capable of both proving unbounded safety and finding long counterexamples. The idea is to use Craig interpolation to guide the creation of symbolic abstractions of *exponentially longer sequences of transitions*. Our experimental analysis shows that on unsafe benchmarks with deep counterexamples our implementation can detect faulty executions that are at least an order of magnitude longer than those detectable by the state-of-the-art tools.

Keywords: Model checking · Transition systems · Craig interpolation · Model-based projection.

1 Introduction

Model checking [17] is a very successful technique widely used for formal verification of hardware and software. While its ultimate goal is to *prove* safety, the ability to discover and report counterexamples primarily contributes to its industrial success. The algorithm that paved the way for the adaptation in the industry, *bounded model checking* (BMC) [9], still remains one of the most successful techniques today for detecting counterexamples. A typical BMC algorithm searches for counterexamples reachable in a finite number of steps, and if nothing is found, it increases the search limits and restarts. This philosophy has been largely adopted by most modern model-checking algorithms based on reachability

* The first author is partially funded by the project 20-07487S of the Czech Science Foundation. The first, third, and forth authors are partially funded by the Swiss National Science Foundation project 200021_185031. The second author is partially funded by the gift from Amazon Web Services.

© The Author(s) 2022
D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 524–542, 2022.
https://doi.org/10.1007/978-3-030-99524-9_29

analysis as one of the advantages of this approach is that it finds the shortest counterexample (if one exists). However, it also results in scalability issues. Specifically, in modern software systems, it is not uncommon that a program must iterate through a certain loop thousands of times (or more) before it reaches some error state. These *deep* counterexamples pose problems for reachability-based algorithms that rely on unrolling the bounds of the system's transition relation one transition at a time.

An important class of loops present in software systems are *multi-phase* loops [44]. A multi-phase loop, in short, is a loop with a conditional (branch) in its body such that the conditional exhibits a fixed number of phase transitions during the execution of the loop. A phase is a sequence of iterations during which the conditional has the same value. Multi-phase loops are notoriously challenging to analyze. When they are safe, they typically require disjunctive invariants. On the other hand, an unsafe multi-phase loop may admit only deep counterexamples if only later phases reveal the unsafe behavior.

In this paper we present a novel model-checking algorithm that is able to find counterexamples of much greater depth than state-of-the-art algorithms. At the same time, it is able to prove system safe under certain conditions and is competitive also on a general set of benchmarks. We build upon the large body of work on SMT-based model checking [1,3,4,8,14,15,25,28,30,37,38] and use Craig interpolation [18,35] for computing abstractions. However, we shift the focus from *state* abstractions—which is the widespread approach—to *transition* abstractions [40].

Our algorithm works on transition systems and it builds a sequence of abstract relations that gradually summarize (in an over-approximating way) an increasing number of steps of the transition relation. One important feature is that the summarized number of steps increases exponentially, not linearly. Another important feature is that all the abstract relations are expressed only over state and next-state variables, i.e., they do not require multiple copies of state variables to capture multiple steps of the transition relation. This sequence of abstract relations is used to refute the existence of bounded reachability paths in the system. If existence of a path cannot be refuted in the current abstraction, either the abstraction is strengthened to refute such path, or the path is shown to be real. The precise mechanics of building and refining the sequence of abstract relations are explained in Section 4. Our experiments demonstrate that our algorithm improves the ability to detect deep counterexamples in the multi-phase loop programs up to two orders of magnitude compared to the state-of-the-art. Furthermore, it enables the detection of bugs left undiscovered by the other tools.

The main contributions of the paper are the following:

- A novel model-checking algorithm for safety properties of transition system based on a sequence of relations over-approximating exponentially increasing number of steps of transition relation.
- Proof of correctness of the algorithm and its termination for unsafe systems.

- Implementation and experimental evaluation of the proposed algorithm demonstrating its capabilities of finding deep counterexamples in challenging benchmarks containing multi-phase loops.

The rest of the paper is organized as follows. The necessary background is given in Section 2, and a motivating example is given in Section 3. Section 4 describes our novel algorithm, and Section 5 presents the experimental results. We discuss the related work in Section 6 and conclude in Section 7.

2 Background

Safety problem We work with a standard symbolic representation of transition systems using the language of first-order logic. Given a set of variables X, we denote as X' the primed copy of X, i.e., $X' = \{x' \mid x \in X\}$. X is a set of *state* variables and X' is a set of *next-state* variables. The formulas are interpreted with respect to some background theory $\mathcal{T}$; in our examples and benchmarks we work with the theory of linear real or integer arithmetic (LRA and LIA in the terminology of satisfiability modulo theories (SMT) [6,7]). We say that a formula in the language of $\mathcal{T}$ over X is a *state* formula and a formula over $X \cup X'$ is a *transition* formula. We identify state formulas with a set of states where they hold and we freely move between these two representations. Similarly, we identify transition formulas with binary relations over the set of states. The identity relation $Id(x, x')$ corresponds to the transition formula $x = x'$.

Transition system is a pair $\langle Init, Tr \rangle$ where $Init$ is a state formula representing the initial states of the system and Tr is a transition formula representing the transition relation of the system. A *safety problem* is a triple $\langle Init, Tr, Bad \rangle$ where $\langle Init, Tr \rangle$ is a transition system and Bad is a state formula representing bad states.

When we only need to distinguish state and next-state variables, but not the individual state variables, for simplicity we only use the lower-case x, x' and not X, X'. These can be viewed as variables representing tuples. We also often need to refer to next-next-state variables, which we denote as x''.

We use $\circ$ to represent *concatenation* of relations. For example, given two relations $R_1(x, y)$ and $R_2(y, z)$ then $R = R_1 \circ R_2$ is a relation over x, z such that $R(x, z) \iff \exists y : R_1(x, y)$ and $R_2(y, z)$. In transition systems we can define relations that represent multiple steps of a transition relation. For example $Tr^2(x, x'') \equiv Tr(x, x') \circ Tr(x', x'')$ relates pair of states (s, t) such that t is reachable from s in *exactly* two steps of the transition relation Tr. We also write that $(s, t) \in Tr^2$. Existence of a counterexample (a path from some initial to some bad state) of a fixed length l can be encoded as a satisfiability check of formula

$$Init(x^{(0)}) \wedge Tr(x^{(0)}, x^{(1)}) \wedge Tr(x^{(1)}, x^{(2)}) \wedge \ldots \wedge Tr(x^{(l-1)}, x^{(l)}) \wedge Bad(x^{(l)}),$$

where $x^{(i)}$ is a state variable shifted i steps, "with i primes". A satisfying assignment determines $l + 1$ states such that the first one is an initial state, the

last one is a bad state, and each successor can be reached from its predecessor by one step of the transition relation Tr. If there is no satisfying assignment then no path of l steps from $Init$ to Bad exists.

Craig interpolation [18] Given an unsatisfiable formula $A \wedge B$, an interpolant I is a formula over the shared symbols of A and B such that $A \implies I$ and $I \wedge B$ is unsatisfiable. We denote as $Itp(A, B)$ an interpolation procedure that computes an interpolant for unsatisfiable $A \wedge B$. Various interpolation procedures exist, for propositional logic [31,42,34,19] as well as for different first-order theories [36,16,2,11].

3 Motivating example

Throughout the paper we demonstrate our approach on a family of C-like programs with a multi-phase loop (generalized from [44] where `N=50`) and an unsafe assertion. The use of parameter `N` (should not be confused with a nondeterministic variable) demonstrates the scale of search of counterexamples of different lengths. We have experimentally evaluated how various tools perform on this example in Section 5. The program source code and the corresponding transition system are given in Figure 1.

```
x=0; y=N;
while(x < 2N){
  x = x + 1;
  if(x > N)
    y = y + 1;
}
assert(y != 2N);
```

$$\begin{aligned} Init(x,y) &\equiv x = 0 \wedge y = N \\ Tr(x,y,x',y') &\equiv x < 2N \wedge x' = x + 1 \\ &\quad \wedge y' = ite(x' > N, y+1, y) \\ Bad(x,y) &\equiv x \geq 2N \wedge y = 2N \end{aligned}$$

Fig. 1: An example of unsafe multi-phase loop

Since the assertion is placed after the loop, any counterexample requires finding a complete unrolling of the loop, i.e., all `2N` iterations (or `2N` steps in the corresponding transition system). Interestingly, even a linear growth of `N` results in the exponential growth of complexity of search of counterexamples. Because of the control-flow divergence in each iteration of the loop, the number of possible program paths (that a verifier explores) doubles with each increment of counter `x`. Our technique allows finding the counterexamples for any `N` drastically more efficiently.

```
    input  : transition system S = ⟨Init, Tr, Bad⟩
    global : TPA sequence S (lazily initialized to true)
    Function CheckSafetyTPA(⟨Init, Tr, Bad⟩):
1       S[0] ← Id ∨ Tr
2       if Sat?[Init(x) ∧ S[0](x, x′) ∧ Bad(x′)] then return UNSAFE
3       n ← 0
4       while TRUE do
5           res ← IsReachable(n, Init, Bad)
6           if res ≠ ∅ then return UNSAFE
7           n ← n + 1
8       end
```

Algorithm 1: Main procedure for checking safety

4 Finding deep counterexamples with transition power abstractions

Our main procedure for detecting safety violation—given in Algorithm 1—follows the typical scheme of bounded model checking where in each iteration the reachability of *Bad* is checked within certain *bounded* number of steps and the bound gradually increases. This scheme has also been adopted by other model checking algorithms, such as Spacer [30] and interpolation-based model checking [20,34,45], which further support a generalization/adaptation of the proof of bounded safety to a proof of unbounded safety.

The distinguishing feature of our approach is that it increases the bound for the safety check *exponentially* in the number of iterations, while other approaches do this linearly. That is, in the nth iteration, traditional algorithms check bounded safety up to n steps; but our approach does up to 2^{n+1} steps. However, we do *not* unroll the transition relation an exponential number of times. Instead, we maintain a sequence of transition formulas (i.e., each formula contains only *two* copies of the state variables) where each element over-approximates twice as many steps of transition relation Tr as its predecessor. We call this sequence a *Transition Power Abstraction* (TPA) sequence.

4.1 TPA sequence for bounded reachability queries

The core of our approach lies in *creating and refining* a sequence of relations $ATr^{\leq 0}, ATr^{\leq 1}, \ldots, ATr^{\leq n}, \ldots$ where each relation over-approximates *twice* as many transition steps of a transition relation Tr as its predecessor. Formally, we require that n^{th} relation $ATr^{\leq n}$ satisfies:

$$Id(x,x') \vee Tr(x,x') \vee Tr^2(x,x') \vee \ldots \vee Tr^{2^n}(x,x') \implies ATr^{\leq n}(x,x') \quad (1)$$

The base for constructing a TPA sequence is $ATr^{\leq 0} \equiv Id \vee Tr$. Thus, $ATr^{\leq 0}$ is *not* an over-approximation, but a precise relation capturing true reachability in either 0 or 1 steps.

Our check for bounded safety is based on a procedure that answers *bounded reachability queries*: Given a set of *source* and *target* states, is any target state reachable from some source state in up to 2^{n+1} steps (for $n \geq 0$)? The procedure *uses* the TPA sequence to answer such queries and, at the same time, it *extends* the sequence and *refines* its existing elements.

Given two sets of states, *Source* and *Target*, and n^{th} element of the current TPA sequence $ATr^{\leq n}$, the following SMT query is issued:

$$Sat?[Source(x) \wedge ATr^{\leq n}(x, x') \wedge ATr^{\leq n}(x', x'') \wedge Target(x'')]. \tag{2}$$

If query (2) is unsatisfiable, it means that there is no *intermediate* state that would be reachable from *Source* using one step of $ATr^{\leq n}$ and, at the same time, can reach *Target* in yet another step of $ATr^{\leq n}$. Since one step of $ATr^{\leq n}$ over-approximates reachability (using Tr) in 0 to 2^n steps, this means that no path of length $\leq 2^{n+1}$ exists from *Source* to *Target*. Thus, the procedure can immediately conclude that no state from *Target* is reachable from any state in *Source* in $\leq 2^{n+1}$ steps.

Additionally, it is also possible to learn new information about the reachability in $\leq 2^{n+1}$ steps in the form of an interpolant between $ATr^{\leq n}(x, x') \wedge ATr^{\leq n}(x', x'')$ and $Source(x) \wedge Target(x'')$. The properties of interpolation guarantee that the interpolant contains only variables x, x'' (i.e., it does not contain x'), it over-approximates $ATr^{\leq n} \circ ATr^{\leq n}$, and it does not relate any source state with a target state. The relation defined by such an interpolant satisfies condition (1) for the $n+1^{\text{st}}$ element of TPA sequence and the current TPA sequence can be refined by conjoining the interpolant (after renaming of variables) to its $n+1^{\text{st}}$ element.

If query (2) is satisfiable, there exists some *intermediate* state m that can be reached from *Source* by one step of $ATr^{\leq n}$ and that can reach *Target* by yet another step of $ATr^{\leq n}$. If $n = 0$, the procedure returns and reports the answer "reachable" as $ATr^{\leq 0}$ is *precise*, not over-approximating. Otherwise, such an intermediate state m can be seen as a potential point on the path from *Source* to *Target*, and this path can be shown to be real if there exist two real paths: from *Source* to m and from m to *Target*. The existence of these two real paths can be checked in a recursive manner.

4.2 Algorithm for bounded reachability checks

The pseudocode for the procedure is given in Algorithm 2. We first explain the steps in more detail and demonstrate a run of the algorithm on our example from Section 3. We then prove the correctness and termination of Algorithm 2 from which follow the correctness of Algorithm 1 and its termination for unsafe systems.

Function `IsReachable` takes as input an integer $n \geq 0$, a set of source states, and a set of target states. The output is a subset of target states that are reachable in $\leq 2^{n+1}$ steps of transition relation Tr. The output set is empty if and only if no target state is reachable from any source state within the given bound.

```
input   : level n, source states Source, target states Target
output  : subset of Target reachable from Source within 2^{n+1} steps
global  : TPA sequence S
Function IsReachable(n, Source, Target):
1   while true do
2       ATr^{≤n} ← S[n]
3       query ← Source(x) ∧ ATr^{≤n}(x, x') ∧ ATr^{≤n}(x', x'') ∧ Target(x'')
4       sat_res ← Sat?[query]
5       if sat_res = UNSAT then
6           I ← Itp(ATr^{≤n}(x, x') ∧ ATr^{≤n}(x', x''), Source(x) ∧ Target(x''))
7           S[n + 1] ← S[n + 1] ∧ I[x'' ↦ x']
8           return ∅
9       else
10          if n = 0 then return QE(∃x, x' query)[x'' ↦ x]
11          Intermediate ← QE(∃x, x'' query)[x' ↦ x]
12          IntermediateReached ← IsReachable(n − 1, Source, Intermediate)
13          if IntermediateReached = ∅ then continue
14          TargetReached ← IsReachable(n − 1, IntermediateReached, Target)
15          if TargetReached = ∅ then continue
16          return TargetReached
17      end
18  end
```

Algorithm 2: Reachability query using TPA

The procedure loops until it computes a truly reachable subset of target states or proves all target states unreachable. In each iteration the procedure reads the current n^{th} element of the TPA sequence (line 2). Note that this will be different in each iteration as the TPA sequence will be updated in the recursive calls on lines 13 and 15. After that, a satisfiability query is constructed and passed to a decision procedure for the background theory $\mathcal{T}$ (lines 3 and 4). The satisfiability query represents a question whether or not there exists an intermediate state that would be reachable from *Source* using one step of $ATr^{\leq n}$ and, at the same time, can reach *Target* in yet another step of $ATr^{\leq n}$.

Query on line 4 is unsatisfiable. If the query is unsatisfiable then no target state can be reached from any source state in two steps of $ATr^{\leq n}$. It follows from Eq. (1) that no target state can be reached from any source state in $\leq 2^{n+1}$ steps. Before indicating the unreachability by returning $\emptyset$ (line 8), the function updates the TPA sequence to ensure termination (discussed later): The function computes an interpolant between $ATr^{\leq n}(x, x') \wedge ATr^{\leq n}(x', x'')$ and $Source(x) \wedge Target(x'')$ (line 6). After renaming variables, the interpolant is conjoined to the $n{+}1^{\text{st}}$ element of the TPA sequence. The following example demonstrates this part of the procedure on our motivating example.

Example 1. Consider the system from Figure 1 for $N = 3$. This system is not safe and the counterexample requires six steps of transition relation Tr.

After Algorithm 1 initializes the base element of TPA sequence to $(x' = x \wedge y' = y) \vee (x < 6 \wedge x' = x+1 \wedge y' = \mathit{ite}(x' > 3, y+1, y))$ it issues a reachability query $\texttt{IsReachable}(0, x = 0 \wedge y = 3, x \geq 6 \wedge y = 6)$ in the first iteration of its loop. This translates to a satisfiability check of the formula

$$\begin{aligned}
& x = 0 \wedge y = 3 \\
& \wedge ((x' = x \wedge y' = y) \vee (x < 6 \wedge x' = x + 1 \wedge y' = \mathit{ite}(x' > 3, y + 1, y))) \\
& \wedge ((x'' = x' \wedge y'' = y') \vee (x' < 6 \wedge x'' = x' + 1 \wedge y'' = \mathit{ite}(x'' > 3, y' + 1, y'))) \\
& \wedge x'' \geq 6 \wedge y'' = 6
\end{aligned}$$

on line 4 of Algorithm 2. This query is unsatisfiable, and $x'' \leq x + 2$ is a possible interpolant computed on line 6. After variable renaming, this interpolant refines $S[1]$, which becomes $x' \leq x + 2$. Then this call to IsReachable terminates and the main loop issues a new reachability query for $n = 1$. This yields a satisfiability query $x = 0 \wedge y = 3 \wedge x' \leq x + 2 \wedge x'' \leq x' + 2 \wedge x'' \geq 6 \wedge y'' = 6$. Again, this formula is unsatisfiable and a possible interpolant is $x'' \leq x+4$. The next element of the TPA sequence, $S[2]$ is refined to $x' \leq x + 4$.

For $n = 2$ (reachability within eight steps), the query on line 4 is satisfiable, and the procedure switches to checking if the counterexample from abstract transition is real or exists only due to a coarse abstraction.

Query on line 4 is satisfiable. If the query on line 9 is satisfiable, a concrete path of length $\leq 2^{n+1}$ cannot be ruled out at this point and the algorithm proceeds to recursively check the existence of one. In the base case $n = 0$ of the recursion, $ATr^{\leq 0}$ is not an over-approximation but a precise relation representing 0 or 1 steps of *Tr* and there exists a real path from *Source* to *Target*. The algorithm computes a state formula representing a truly reachable subset of *Target*. This is done by first using *quantifier elimination* (QE) to eliminate all except next-next state variables from the query (line 10) and then renaming the variables to state variables.[4]

If the base case has not been reached yet ($n > 0$), the procedure first computes a set of candidate *intermediate* states by eliminating all except next-state variables from the query (line 11). Then, the procedure recursively calls itself to determine the existence of a path from *Source* to the newly computed intermediate set with the bound on length halved (line 12). This check has two possible outcomes. In case the recursive call returns $\emptyset$, none of the intermediate candidates is reachable (within 2^n steps). Moreover, $S[n]$ must have been strengthened (line 7) before the recursive call returned as to *not* relate any of the source states and intermediate candidates. The procedure then continues to the next iteration (line 13) where it tries to find new intermediate candidates or prove there are none anymore. In case the set returned on line 12 is non-empty, it represents a set of states reachable from *Source* within 2^n steps of *Tr*. The procedure proceeds to check the existence

[4] QE computes maximal reachable subsets. While this is convenient for proving termination of Algorithm 2, in practice quantifier elimination is a very expensive operation. Our implementation therefore supports also the use of *model-based projection* to efficiently under-approximate quantifier elimination (see Section 4.4).

of a path from these states to the target states (line 14). The reasoning here is the same as for the first recursive call: If *Target* is not reachable, the procedure attempts to find new intermediate candidates in a new iteration. Otherwise, real path from *Source* to *Target* exists and the computed truly reachable states are returned. The returned states are reachable with 2^{n+1} steps as both recursive calls check reachability within 2^n steps.

We continue Example 1 to illustrate this phase of Algorithm 2.

Example 2. Following Example 1, the algorithm is checking bounded reachability between *Init* and *Bad* for $n = 2$, i.e., within 8 steps. The issued satisfiability query is $x = 0 \wedge y = 3 \wedge x' \leq x + 4 \wedge x'' \leq x' + 4 \wedge x'' \geq 6 \wedge y'' = 6$. Eliminating all except next-state variables yields $x' \leq 4 \wedge x' \geq 2$. This results in the call $\texttt{IsReachable}(1, x = 0 \wedge y = 3, x \leq 4 \wedge x \geq 2)$. The satisfiability query issued next is $x = 0 \wedge y = 3 \wedge x' \leq x + 2 \wedge x'' \leq x' + 2 \wedge x'' \leq 4 \wedge x'' \geq 2$. This is again satisfiable and yields $x' \leq 2 \wedge x' \geq 0$ after quantifier elimination. Now we reach level 0 with a call $\texttt{IsReachable}(0, x = 0 \wedge y = 3, x \leq 2 \wedge x \geq 0)$. The constructed satisfiability query is again satisfiable and since we are at level 0, the procedure returns a set of states truly reachable from $x = 0 \wedge y = 3$ within 2 steps. These can be characterized as $(x = 0 \vee x = 1 \vee x = 2) \wedge y = 3$. The reachable states are reported to level 1 which issues reachability query for the second part: $\texttt{IsReachable}(0, (\mathrm{x} = 0 \vee \mathrm{x} = 1 \vee \mathrm{x} = 2) \wedge y = 3, x \leq 4 \wedge x \geq 0)$. This is also successful and returns reachable states $(x = 0 \vee x = 1 \vee x = 2 \vee x = 3 \vee x = 4) \wedge \mathrm{y} = 3$. These are states reachable from *Init* within 4 steps and they are reported to level 2. There, the second part of the counterexample is found in a similar way and the procedure concludes that *Bad* is truly reachable from *Init* within 8 steps.

The behaviour of the algorithm on these examples can be generalized for the system of Figure 1 for larger values of N. The length of the counterexample is $2N$ and let l denote $\lfloor log_2(2N) \rfloor$. The bounded safety will be quickly determined up to 2^l steps with l calls to `IsReachable` which all return $\emptyset$ in their first iteration. On the next iteration, for $n = l$, `IsReachable` will find the real counterexample, but it requires $O(2^l)$ recursive calls to find the counterexample of length in the interval $(2^l, 2^{l+1}]$.

4.3 Correctness and termination

We first prove correctness and termination of Algorithm 2 which then entails correctness of Algorithm 1 and its termination for unsafe systems. We prove the correctness of procedure `IsReachable` separately for the unreachable and the reachable case.

Lemma 1. *If* `IsReachable`*(n, Source, Target) returns $\emptyset$, then no state from Target can be reached from Source within 2^{n+1} steps.*

Proof. The proof relies on the invariant that S is always a TPA sequence, i.e., its elements satisfy the property of Eq. (1). This is obviously true when S is initialized in Algorithm 1. The only update of S happens in Algorithm 2 on line 7.

Consider an update on any level $k \leq n$. From the properties of interpolation, we know that $I(x, x'')$ (on line 6) over-approximates $ATr^{\leq k}(x, x') \wedge ATr^{\leq k}(x', x'')$, which represents two steps of the relation $ATr^{\leq k}$. Since $ATr^{\leq k}$ over-approximates $\leq 2^k$ steps of Tr, it follows that $I(x, x'')$ over-approximates $\leq 2^{k+1}$ steps of Tr. Thus, conjoining it to $ATr^{\leq k+1}$ preserves the condition of Eq. (1).

It follows from Eq. (1) that when the query on line 4 is unsatisfiable, there exists no path of length $\leq 2 \times 2^n = 2^{n+1}$ from any source state to any target state. □

Lemma 2. *If* `IsReachable`*(n, Source, Target) returns a non-empty set Res, then Res $\subseteq$ Target and every state in Res can be reached from some state in Source in $\leq 2^{n+1}$ steps.*

Proof. The proof is by induction on n.

Base case: For $n = 0$ $ATr^{\leq 0}$ represents precise reachability in 0 or 1 step. It follows that if the query on line 4 is satisfiable, some target states are truly reachable from the set of source states in ≤ 2 steps. Moreover, the properties of QE guarantee that $Res = QE(\exists x, x' \; query)[x'' \mapsto x]$ is a subset of $Target(x)$ that are reachable from *Source* using $ATr^{\leq 0} \circ ATr^{\leq 0}$.

Inductive case: Suppose the claim holds for $n - 1$. If at level n the procedure returned a non-empty set, it must have been the case that the first recursive call (line 12) returned a non-empty set *IntermediateReached* of states truly reachable from *Source* in $\leq 2^n$ steps, by our induction hypothesis. Additionally, the second recursive call (line 14) also returned a non-empty set *TargetReached* that, according to our induction hypothesis, is a subset of *Target* truly reachable from *IntermediateReached* in $\leq 2^n$ steps. It follows that *TargetReached* is a subset of *Target* truly reachable from *Source* in $\leq 2^{n+1}$ steps. □

The correctness of procedure `IsReachable` extends naturally to the correctness of our main procedure.

Theorem 1 (Correctness). *If Algorithm 1 returns UNSAFE, then the system $\mathcal{S}$ is unsafe, i.e., some bad state is reachable from some initial state.*

Proof. The satisfiablity query on line 2 of Algorithm 1 checks reachability in 0 and 1 step. If this query is satisfiable, there exists a counterexample path of length 0 or 1 from some initial state to a bad state.

Otherwise, it enters the loop where UNSAFE is returned only if `IsReachable` returns non-empty set of states for some n. From the correctness of `IsReachable` it follows that the returned set is a subset of *Bad* that is reachable from *Init* in $\leq 2^{n+1}$ steps. Thus there exists a counterexample path in the system. □

Next, we want to show that if there exists a counterexample path in the system, our procedure will eventually report it. This boils down to the question of termination of a single call to `IsReachable`.

Lemma 3. *Assume that the satisfiability check (line 4) terminates, i.e., that the background theory $\mathcal{T}$ is decidable, and that $\mathcal{T}$ has procedures for interpolation and quantifier elimination.*[5] *Then a single call to* `IsReachable` *always terminates.*

[5] The linear arithmetic theories of our experiments satisfy these assumptions.

Proof. The proof proceeds by induction on level n. The base case ($n = 0$) trivially terminates after a single satisfiability query on line 4.

For the inductive case, consider the first iteration of the loop. If the query is unsatisfiable, the procedure terminates. If it is satisfiable, quantifier elimination yields a set of states $\mathit{Intermediate} = \{m \mid \exists s \in \mathit{Source}, \exists t \in \mathit{Target} : (s,m) \in \mathit{ATr}^{\leq n} \wedge (m,t) \in \mathit{ATr}^{\leq n}\}$. Now consider the first recursive call (line 12). By induction, it terminates. If it returns $\emptyset$, then, by properties of the interpolation, $\mathit{ATr}^{\leq n}$ has been strengthened such that $\forall s \in \mathit{Source}, \forall m \in \mathit{Intermediate} : (s,m) \notin \mathit{ATr}^{\leq n}$ now holds. Consequently, in the second iteration the query on line 4 must be unsatisfiable and the procedure terminates.

Now consider the situation where the recursive call on line 12 returned a non-empty set *IntermediateReached*. The procedure continues to the second recursive call (line 14), which also terminates, by induction. If the returned set *TargetReached* is non-empty, the procedure terminates (line 16). If it is empty, then no state reachable from *Source* in $\leq 2^n$ steps of *Tr* can reach any state in *Target* in another $\leq 2^n$ steps. Moreover, $\mathit{ATr}^{\leq n}$ has been strengthened so that now it does not relate any state from *IntermediateReached* with a state in *Target*. In the second iteration, the query on line 4 could still be satisfiable. However, the extracted *Intermediate* (of the second iteration) cannot contain states that are reachable from *Source* in $\leq 2^n$ steps. Thus first recursive call (line 12) in the second iteration must return $\emptyset$ and this is followed by an unsatisfiable query (line 4) in the third iteration and termination. □

The immediate consequence of Lemma 3 is that our main procedure will find a counterexample if one exists.

Theorem 2. *If there exists a counterexample in the system, Algorithm 1 terminates with UNSAFE result.*

4.4 Under-approximating QE with model-based projection

Model-based projection (MBP) [30] is a recent technique for under-approximating quantifier elimination for existentially quantified formulas. In short, given an existentially quantified formula $\exists x \phi(x, y)$, MBP is a function that maps each model of ϕ to a quantifier-free formula that implies $\exists x \phi(x, y)$ and is true in the model. Moreover, it is required that the function has a finite image (it produces only finitely many quantifier-free under-approximations) and the disjunction of the image is equal to the quantified formula. Efficient MBP for linear real and integer arithmetic was given in [30,10]. MBP has also been designed for algebraic datatypes [10], arithmetic signature of bit-vectors [23] and arrays[6] [29].

Quantifier elimination in Algorithm 2 can be replaced by MBP in a straightforward way. On line 4, if the query is satisfiable, we obtain from the SMT solver a model witnessing the satisfiability. Then, on lines 10 and 11 we replace QE with MBP using the obtained model. It is easy to check that the proof of Lemma 2 remains valid with this change, and thus also the result of Theorem 1. In Section 5 we experimentally demonstrate the practical advantage of MBP over QE.

[6] MBP for arrays does not satisfy the finite image condition

4.5 Proving safety

Even though the main purpose of the TPA sequence is to help to quickly rule out bounded reachability queries, it can also be useful in another way. Specifically, an element of the TPA sequence may turn out to be a *transition invariant* with respect to transition relation Tr.

Definition 1 (transition invariant). *We say that* $R(x, x')$ *is a* transition invariant *if* $Tr^* \subseteq R$*, i.e.,* $\forall x, x' \; Tr^*(x, x') \implies R(x, x')$*, where* Tr^* *is the reflexive transitive closure of* Tr*.*

Note that our definition is slightly simpler than that of [40], as it only depends on the transition relation and not, for example, on the initial states of the system.

If we find a transition invariant that does not relate any initial state with a bad state, we can immediately conclude that the system is safe. We show one way how to detect if a member of the TPA sequence is a transition invariant using SMT query.

Lemma 4. *Assume that for some* n*,* $ATr^{\leq n} \circ Tr \subseteq ATr^{\leq n}$ *or that* $Tr \circ ATr^{\leq n} \subseteq ATr^{\leq n}$*. Then* $ATr^{\leq n}$ *is a transition invariant.*

Proof. We consider the case $ATr^{\leq n} \circ Tr \subseteq ATr^{\leq n}$ and show that $Tr^* \subseteq ATr^{\leq n}$. The other case is analogous. Take any two states s, s' such that s' is reachable from s, i.e., $(s, s') \in Tr^*$. We show that $(s, s') \in ATr^{\leq n}$ by induction on d, the length of the path from s to s'. If $d \leq 2^n$ then $(s, s') \in ATr^{\leq n}$ by Eq. (1). Assume now that $d > 2^n$. Then there exists a state t such that t can be reached from s in $d - 1$ steps and $(t, s') \in Tr$. By induction, we have that $(s, t) \in ATr^{\leq n}$ and $(s, s') \in ATr^{\leq n} \circ Tr$. By our assumption it follows that $(s, s') \in ATr^{\leq n}$. □

Note that when a call to `IsReachable` on line 5 in Algorithm 1 returns $\emptyset$, the $n{+}1^{\text{st}}$ element of TPA sequence $ATr^{\leq n+1}$ does not relate any initial and bad state. Thus we can check at this point for the conditions of Lemma 4, and, if satisfied, we can immediately conclude that no counterexample (of any length) exists in the system and report safety.

In fact, to detect that no counterexample exists, the assumptions of Lemma 4 can be relaxed a bit. We can consider the restriction of these relations to only initial or bad states. The notation $A \lhd R$ denotes a domain restriction of a binary relation R to a set A, i.e., $(x, y) \in A \lhd R$ iff $(x, y) \in R \wedge x \in A$. Similarly $R \rhd B$ denotes the codomain restriction, i.e., $(x, y) \in R \rhd B$ iff $(x, y) \in R \wedge y \in B$.

Lemma 5. *Assume that for some* n $Init \lhd ATr^{\leq n} \circ Tr \subseteq Init \lhd ATr^{\leq n}$*. Then* $Init \lhd Tr^* \subseteq Init \lhd ATr^{\leq n}$*. Similarly, if* $Tr \circ ATr^{\leq n} \rhd Bad \subseteq ATr^{\leq n} \rhd Bad$*, then* $Tr^* \rhd Bad \subseteq ATr^{\leq n} \rhd Bad$*.*

Proof. Same as the proof of Lemma 4, with appropriate restrictions.

Lemma 5 represents a weaker form of Lemma 4: it has a weaker assumption and a weaker conclusion. Nevertheless, the conclusion is still strong enough to ensure that no counterexample exists and conclude safety.

5 Experiments

We have implemented our TPA-based procedure (Algorithm 1) in our new CHC solver Golem[7]. Golem is built on top of the interpolating SMT solver OpenSMT [26]. In our experiments we used version 2.2.0 of OpenSMT[8].

To gauge the feasibility of our algorithm we performed a set of experiments. All experiments were conducted on a machine with AMD EPYC 7452 32-core processor and 8x32 GiB of memory. We compared our approach to the current state-of-the-art tools Eldarica 2.0.6 [25], IC3-IA 20.04.1 [15] and Z3 4.8.12 [39] (using both its BMC [9] and Spacer [30] engines), which were the top competitors in CHC-COMP 2020 and 2021 [43,21]. We used both versions of our algorithm in the experiments: using MBP (TPA-MBP) and QE (TPA-QE). The format of all the benchmarks is that of the constrained Horn clauses (CHCs) used in the CHC-COMP. Since IC3-IA's input format differs, all CHC benchmarks were translated to VMT format using the automated tool packaged with IC3-IA.[9]

The goal of the first experiment was to investigate the scalability of our algorithm with respect to the length of the counterexample and compare its performance to the state-of-the-art tools. We used the parametrized transition system from our motivating example in Section 3. The counterexample in this system has length $2N$ and we ran the tools on instances for N ranging from 1 to 511. The timeout was set to 300 seconds. Figure 2 shows the runtime of the tools for the given value of N.

TPA-MBP was able to report all instances as unsafe, needing *less than two seconds* for each instance. Eldarica, IC3-IA and Z3-BMC exhibit relatively stable pattern where the performance decreases rapidly with increasing N. Z3-Spacer, on the other hand, exhibits a curious behaviour where it is able to solve most of the instances (even though it is slower than TPA-MBP by at least an order of magnitude), but on a relatively large number of instances it times out, and we were not able to understand the pattern on which instances this happens. Quick look at the instances for $N < 100$ suggests that on some instances its behaviour is much closer to that of IC3-IA. Finally, TPA-QE also shows an interesting pattern in its runtime where its performance drops considerably on every power of two, and then it slowly improves for larger N until the next power of two.

This first experiment showed very promising results for TPA-MBP which benefited from the fact that the reason why shorter counterexamples do not exist can be summarized relatively easily. It scaled exceptionally well compared to the state-of-the-art tools, as well as TPA-QE.

To confirm the results from the first experiment, we continued with the second set of benchmarks representing instances of our targeted type of problems. They represent assertions over *multi-phase* loops, which are known to be difficult to analyze by state-of-the-art techniques. We took 54 safe multi-phase benchmarks

[7] https://github.com/usi-verification-and-security/golem; commit 4ea1a53

[8] https://github.com/usi-verification-and-security/opensmt

[9] Full results of the experiments available at http://verify.inf.usi.ch/horn-clauses/tpa/experiments. Artifact available at https://doi.org/10.5281/zenodo.5815911

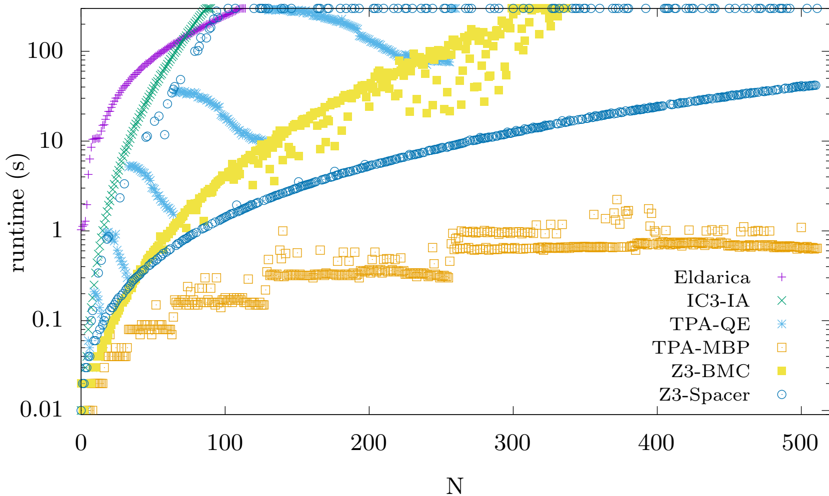

Fig. 2: Runtime for motivating example for N from 1 to 511 (log y-axis)

from CHC-COMP repository[10] and then for each benchmark created its unsafe version with a minor modification of the safety property.[11] In most cases this was done by negating one of the conjuncts of the property. In a few cases this resulted in a simple benchmark with a very short CEX (< 10 steps), but in most cases, the minimal counterexample is much larger, ranging from a few hundreds to a few tens of thousands of steps. There are even a few extremes where the minimal counterexample requires hundreds of thousands or even millions of steps.

With the timeout of 300 seconds, out of 54 benchmarks, TPA-QE solved 20 and TPA-MBP solved 35 benchmarks, beating the other tools among which Z3-Spacer performed the best, solving 20 benchmarks. The results are summarized in Figure 3 where the number of solved benchmarks by each tool is plotted against the time needed for their solving.

Overall, our tool solved 15 benchmarks that none of the other tools was able to solve and in general could be one or two orders of magnitude faster. There were two noticeable exceptions: benchmark 24 was uniquely solved by Z3 and benchmark 39 was uniquely solved by IC3-IA (for benchmark numbering, see the link in footnote 11). We found out that in the latter case our tool suffered from incompleteness in the decision procedure of OpenSMT for integer arithmetic, while in the former case the interpolation used by our algorithm was not producing good abstractions and we suffered from the need for frequent refinements.

We also examined the solved benchmarks for the length of the minimal counterexample they admit. The results are in line with the observations from our first experiments: Other tools could only solve benchmarks with a counterexample

[10] https://github.com/chc-comp/aeval-benchmarks

[11] Benchmarks available at https://github.com/blishko/chc-benchmarks.

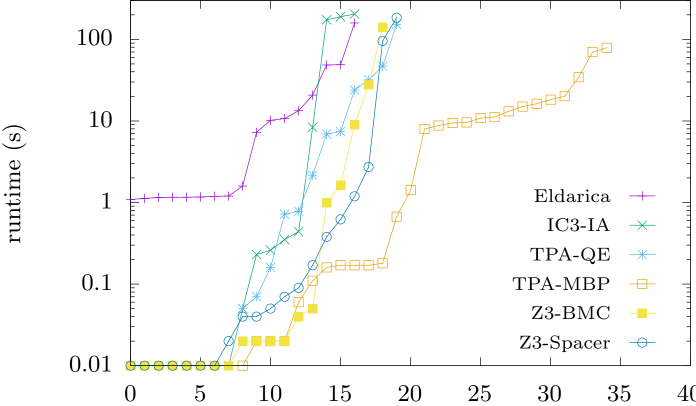

Fig. 3: Results on 54 multi-phase unsafe benchmarks

of up to a thousand steps (1001 steps in benchmark 17 solved by Z3-Spacer). TPA-QE matched this performance (1001 steps in benchmark 27), but TPA-MBP managed to solve benchmarks with a counterexample of more than *ten thousand* steps (17650 in benchmark 42). Thus, our technique significantly improves upon state-of-the-art with respect to the length of the counterexample it can detect.

Finally, we successfully tested our implementation on the safe version of the 54 multi-phase benchmarks and on the general set of 498 benchmarks from CHC-COMP'21, the category of transition systems over linear real arithmetic. TPA-MBP managed to prove 10 of the multi-phase benchmarks safe. Z3-Spacer, IC3-IA and Eldarica proved safe 9, 20 and 26 of these benchmarks, respectively. On the CHC-COMP LRA-TS benchmark set, TPA-MBP was able to solve 70 unsafe benchmarks (from 90+ known unsafe benchmarks in the set) and 67 safe benchmarks.

6 Related work

Loop acceleration [5,12,22] is a related approach for loop analysis that enables both proving safety and detection of deep bugs. It transforms the loop to a single quantifier-free formula representing all possible executions of the loop. While offering significant improvement for a limited types of integer loops, it is not applicable for code with control-flow divergence and/or data structures. Acceleration has also been combined with interpolation-based model-checking [13,24]. In contrast, our technique does not accelerate paths but builds over-approximations of bounded number of iterations. It is not restricted to any specific type of loops, and it works over any theory supporting interpolation and quantifier elimination.

Another technique for fast detection of deep counterexamples for C programs was proposed in [32]. Given a path through a loop, it computes a new path that

under-approximates an arbitrary number of iterations of the original path. In contrast to loop acceleration, this technique only under-approximates the loop behaviour, but it can handle conditionals and richer background theories. Our approach targets the same goal but it is *over-approximating*, which allows for detecting (transition) invariants and proving safety. Their prototype aims at C programs only (and does not seem to be maintained anymore). Our implementation works on transition systems in the form of constrained Horn clauses (CHC) and thus is agnostic to the programming language.

Abstracting transition relation using interpolation has been employed in [27]. They use interpolation to compute and refine abstract version of the transition relation. However, they abstract only a single step of the transition relation. Instead, we use interpolation to compute relations that over-approximate multiple (and increasingly larger number of) steps of the transition relation.

Transition invariants [40] have been successfully employed for proving liveness properties, especially termination [33,41]. Our technique can discover transition invariants and use them to prove safety. However, in this paper we focused on finding counterexamples and the directed search for invariants is left for future work.

Our technique can find a possible application in automating test-case generation. A given program can be automatically annotated with assertions representing the reachability of all the branches. Having the goal to detect a set of input values for maximizing the test coverage [46], our technique would be called repeatedly to find many counterexamples for a subset of assertions (including deep ones) and prove the unreachability of the remaining ones.

7 Conclusion and Future Work

This paper introduces a novel model-checking algorithm for safety properties of transition systems with a focus on finding deep counterexamples. The idea is based on maintaining a sequence of transition formulas, called the *transition power abstraction* (TPA) sequence, where each element over-approximates a sequence of transition steps *twice as long as* its predecessor. The sequence is used in answering bounded reachability queries, which in turn results in new information that further refines the sequence. We proved the correctness of this algorithm and showed that it eventually finds a counterexample if one exists, assuming the background theory admits interpolation and quantifier elimination. For performance reasons, our implementation applies quantifier elimination lazily using model-based projection that lets the approach to outperform state-of-the-art on a class of problems with multi-phase loops. The experiments confirmed that it is able to detect counterexamples of much greater depth than existing tools within the same time constraints.

As future work, we plan to investigate possible improvements of the algorithm and tailor it for finding transition invariants. This would contribute to its ability to prove programs safety and enable the modular reasoning to support arbitrary systems of constrained Horn clauses.

References

1. Alt, L., Asadi, S., Chockler, H., Even Mendoza, K., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Hifrog: SMT-based function summarization for software verification. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 207–213. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
2. Alt, L., Hyvärinen, A.E.J., Sharygina, N.: LRA interpolants from no man's land. In: Strichman, O., Tzoref-Brill, R. (eds.) HVC 2017. LNCS, vol. 10629, pp. 195–210. Springer, Cham (2017)
3. Asadi, S., Blicha, M., Fedyukovich, G., Hyv\"arinen, A., Even-Mendoza, K., Sharygina, N., Chockler, H.: Function summarization modulo theories. In: Barthe, G., Sutcliffe, G., Veanes, M. (eds.) LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 57, pp. 56–75. EasyChair (2018)
4. Asadi, S., Blicha, M., Hyvärinen, A.E.J., Fedyukovich, G., Sharygina, N.: Incremental verification by SMT-based summary repair. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 77–82. IEEE (2020)
5. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: Acceleration from theory to practice. International Journal on Software Tools for Technology Transfer **10**(5), 401–424 (2008)
6. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at http://smtlib.cs.uiowa.edu
7. Barrett, C., de Moura, L., Ranise, S., Stump, A., Tinelli, C.: The SMT-LIB initiative and the rise of SMT. In: Barner, S., Harris, I., Kroening, D., Raz, O. (eds.) Hardware and Software: Verification and Testing. pp. 3–3. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
8. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. Journal of Automated Reasoning **60**(3), 299–335 (Mar 2018)
9. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Tools and Alg. for the Const. and Anal. of Systems (TACAS '99). LNCS, vol. 1579, pp. 193–207 (1999)
10. Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: Fehnker, A., McIver, A., Sutcliffe, G., Voronkov, A. (eds.) LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations. EPiC Series in Computing, vol. 35, pp. 15–27. EasyChair (2015)
11. Blicha, M., Hyvärinen, A.E.J., Kofroň, J., Sharygina, N.: Decomposing Farkas interpolants. In: Vojnar, T., Zhang, L. (eds.) Proc. TACAS 2019. LNCS, vol. 11427, pp. 3–20. Springer (2019)
12. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 227–242. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
13. Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating interpolation-based model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 428–442. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
14. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification. pp. 277–293. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

15. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 46–61. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
16. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. ACM Trans. Comput. Logic **12**(1), 7:1–7:54 (Nov 2010)
17. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018)
18. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. The Journal of Symbolic Logic **22**(3), 269–285 (1957)
19. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: VMCAI 2010. LNCS, vol. 5944, pp. 129–145. Springer (2010)
20. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: TACAS, Part I. LNCS, vol. 10805, pp. 251–269. Springer (2018)
21. Fedyukovich, G., Rümmer, P.: Competition report: CHC-COMP-21. In: Hojjat, H., Kafle, B. (eds.) Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021. EPTCS, vol. 344, pp. 91–108 (2021)
22. Frohn, F.: A calculus for modular loop acceleration. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 58–76. Springer International Publishing, Cham (2020)
23. Govind, H., Fedyukovich, G., Gurfinkel, A.: Word level property directed reachability. In: 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD). pp. 1–9 (2020)
24. Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: Chakraborty, S., Mukund, M. (eds.) Automated Technology for Verification and Analysis. pp. 187–202. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
25. Hojjat, H., Rümmer, P.: The ELDARICA Horn Solver. In: FMCAD. pp. 158–164. IEEE (2018)
26. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 547–553. Springer, Cham (2016)
27. Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S.K. (eds.) Computer Aided Verification. pp. 39–51. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
28. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Piskac, R., Talupur, M. (eds.) Proc. FMCAD 2016. pp. 85–92. IEEE (2016)
29. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using Horn clauses over integers and arrays. In: 2015 Formal Methods in Computer-Aided Design (FMCAD). pp. 89–96 (2015)
30. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods in System Design **48**(3), 175–205 (Jun 2016)
31. Krajíček, J.: Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. The Journal of Symbolic Logic **62**(2), 457–486 (1997)
32. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. Formal Methods in System Design **47**(1), 75–92 (2015)

33. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 89–103. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
34. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2013. pp. 1–13. Springer, Heidelberg (2003)
35. McMillan, K.L.: Applications of Craig interpolants in model checking. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
36. McMillan, K.L.: An interpolating theorem prover. Theoretical Computer Science **345**(1), 101–121 (2005)
37. McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification (CAV '06). LNCS, vol. 4144, pp. 123–136 (2006)
38. McMillan, K.L.: Lazy annotation revisited. In: Proc. CAV 2014. LNCS, vol. 8559, pp. 243–259. Springer (2014)
39. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. pp. 337–340. Springer, Heidelberg (2008)
40. Podelski, A., Rybalchenko, A.: Transition invariants. In: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004. pp. 32–41 (2004)
41. Podelski, A., Rybalchenko, A.: Transition invariants and transition predicate abstraction for program termination. In: Abdulla, P.A., Leino, K.R.M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 3–10. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
42. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic **62**(3), 981–998 (1997)
43. Rümmer, P.: Competition report: CHC-COMP-20. Electronic Proceedings in Theoretical Computer Science **320**, 197–219 (Aug 2020)
44. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 703–719. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
45. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Proc. FMCAD 2014. pp. 1–8. IEEE (2009)
46. Zlatkin, I., Fedyukovich, G.: Maximizing branch coverage with constrained horn clauses. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg (2022)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.