Check for updates

Infinite State Model Checking by Learning Transitive Relations

Florian Frohn(✉) and Jürgen Giesl(✉)

RWTH Aachen University, Aachen, Germany
{florian.frohn,giesl}@informatik.rwth-aachen.de

Abstract. We propose a new approach for proving safety of infinite state systems. It extends the analyzed system by *transitive relations* until its *diameter* D becomes finite, i.e., until constantly many steps suffice to cover all reachable states, irrespective of the initial state. Then we can prove safety by checking that no error state is reachable in D steps. To deduce transitive relations, we use *recurrence analysis*. While recurrence analyses can usually find conjunctive relations only, our approach also discovers disjunctive relations by combining recurrence analysis with *projections*. An empirical evaluation of the implementation of our approach in our tool LoAT shows that it is highly competitive with the state of the art.

1 Introduction

We consider relations defined by SMT formulas over two disjoint vectors of *pre-* and *post-variables* $\mathbf{x}$ and $\mathbf{x}'$. Such *relational formulas* can easily represent *transition systems* (TSs), linear *Constrained Horn Clauses* (CHCs), and *control-flow automata* (CFAs).[1] Thus, they subsume many popular intermediate representations used for verification of systems specified in more expressive languages.

In contrast to, e.g., source code, relational formulas are unstructured. However, source code may be unstructured, too (e.g., due to `gotos`), so being independent from the structure of the input makes our approach broadly applicable.

Example 1 (Running Example). Let $\tau := \tau_{\mathsf{inc}} \vee \tau_{\mathsf{dec}}$ with:

$$w \doteq 0 \wedge x' \doteq x + 1 \wedge y' \doteq y + 1 \qquad (\tau_{\mathsf{inc}})$$

$$w' \doteq w \wedge w \doteq 1 \wedge x' \doteq x - 1 \wedge y' \doteq y - 1 \qquad (\tau_{\mathsf{dec}})$$

We use "$\doteq$" for equality in relational formulas. The formula τ defines a relation $\to_\tau$ on $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ by relating the non-primed pre-variables with the primed post-variables. So for all $v_w, v_x, v_y, v'_w, v'_x, v'_y \in \mathbb{Z}$, we have $(v_w, v_x, v_y) \to_\tau (v'_w, v'_x, v'_y)$ iff $[w/v_w, x/v_x, y/v_y, w'/v'_w, x'/v'_x, y'/v'_y]$ is a model of τ. Let the set of *error states* be given by $\psi_{\mathsf{err}} := w \doteq 1 \wedge x \leq 0 \wedge y > 0$.

[1] To this end, it suffices to introduce one additional variable that represents the control-flow location (for TSs and CFAs) or the predicate (for linear CHCs).

© The Author(s) 2025
C. Barrett and U. Waldmann (Eds.): CADE 2025, LNAI 15943, pp. 468–489, 2025.
https://doi.org/10.1007/978-3-031-99984-0_25

With the *initial states* $\psi_{\mathsf{init}} := x \doteq 0 \wedge y \doteq 0$ this example[2] is challenging for existing model checkers: Neither the default configuration of Z3/Spacer [29, 36], nor Golem's [4] implementation of Spacer, LAWI [35], IMC [34], TPA [8], PDKIND [27], or predicate abstraction [25] can prove its safety. In contrast, all of these techniques can prove safety with the more general initial states $\psi_{\mathsf{init}} := x \doteq y$. As all of them are based on *interpolation*, the reason might be that the inductive invariant $x \doteq y$ is now a subterm of ψ_{init}, so it is likely to occur in interpolants. However, this explanation is insufficient, as all techniques fail again for $\psi_{\mathsf{init}} := x \doteq y \wedge y \doteq 0$.

This illustrates a well-known issue of interpolation-based verification techniques: They are highly sensitive to minor changes of the input or the underlying interpolating SMT solver (e.g., [31, p. 102]). So while they can often solve difficult problems quickly, they sometimes fail for easy examples like the one above.

In another line of research, *recurrence analysis* has been used for software verification [14,28]. Here, the idea is to extract recurrence equations from loops and solve them to summarize the effect of arbitrarily many iterations. While recurrence analysis often yields very precise summaries for loops without branching, these summaries are conjunctive. However, for loops with branching, disjunctive summaries are often important to distinguish the branches.

In this work, we embed recurrence analysis into *bounded model checking* (BMC) [3], resulting in a robust, competitive model checking algorithm. To find disjunctive summaries, we exploit the structure of the relational formula to partition the state space on the fly via *model based projection* [29] (which approximates quantifier elimination), and a variation of recurrence analysis called *transitive projection*.

Our approach is inspired by ABMC [19], which combines BMC with *acceleration* [5,15,30]. ABMC uses *blocking clauses* to speed up the search for counterexamples, but they turned out to be of little use for this purpose. Instead, they enable ABMC to prove safety of certain challenging benchmarks. This motivated the development of our novel dedicated algorithm for proving safety via BMC and blocking clauses. See Sect. 5 for a detailed comparison with ABMC.

Overview. We start with an informal explanation of our approach. Given a relational formula τ, one can prove safety with BMC by unrolling its transition relation $\rightarrow_\tau$ D times, where D is the *diameter* [3]. So D is the "longest shortest path" from an initial to some other state, or more formally:

$$D := \sup_{\mathbf{v}' \text{ is reachable from an initial state}} \left(\min\{i \in \mathbb{N} \mid \mathbf{v} \text{ is an initial state}, \mathbf{v} \rightarrow_\tau^i \mathbf{v}'\}\right)$$

So every reachable state can be reached in $\leq D$ steps. Hence, if BMC finds no counterexample in D steps, the system is safe. Our new algorithm *Transitive Relation Learning* (TRL) exploits this observation by iteratively constructing an SMT problem such that its unsatisfiability implies that the diameter has been

[2] extra-small-lia/bouncy_symmetry from the CHC competition.

reached (see the end of this section for more details). For infinite state systems, D is rarely finite: Consider the relational formula (1) with the initial state $x \doteq 0$.

$$x' \doteq x + 1 \qquad (1) \qquad\qquad n > 0 \wedge x' \doteq x + n \qquad (2)$$

Then $k \in \mathbb{N}$ steps are needed to reach a state with $x \doteq k$. So D is infinite, i.e., the diameter cannot be used to prove safety directly.

The core idea of TRL is to "enlarge" τ to decrease its diameter (even though TRL never computes the diameter explicitly). To this end, our approach "adds" transitive relations to τ, which will be called *learned relations* in the sequel. For (1), TRL would learn the relational formula (2). Then the diameter of (1) $\vee$ (2) is 1, as a state with $x \doteq k$ can be reached in 1 step by setting n to k. Transitive relations are particularly suitable for decreasing the diameter, as they allow us to ignore runs where the same learned relation is used twice in a row. TRL exploits this by adding assertions to the corresponding SMT problem that prevent consecutive uses of the same learned relation.

This raises the questions *when* and *how* new relations should be learned. Regarding the "when", our approach unrolls the transition relation, just like BMC. Thereby, it looks for *loops* and learns a relation when a new loop is encountered. However, as we analyze unstructured systems, the definition of a "loop" is not obvious. Details will be discussed in Sect. 3.2.

Regarding the "how", TRL ensures that we have a *model* for the loop, i.e., a valuation σ_{loop} that corresponds to an evaluation of the loop body. Then given a loop τ_{loop} and a model σ_{loop}, apart from transitivity we only require two more properties for a learned relation $\rightarrow_\pi$. First, the evaluation that corresponds to σ_{loop} must also be possible with $\rightarrow_\pi$. This ensures that TRL makes "progress", i.e., that we do not unroll the same loop with the same model again. Second, the following set must be finite:

$$\{\pi \mid \sigma_{\mathsf{loop}} \text{ is a model of } \tau_{\mathsf{loop}}, \rightarrow_\pi \text{ is learned from } \tau_{\mathsf{loop}} \text{ and } \sigma_{\mathsf{loop}}\}$$

So TRL only learns finitely many relations from a given loop τ_{loop}. While TRL may diverge (Remark 23), this "usually" ensures termination in practice.

Apart from these restrictions, we have lots of freedom when computing learned relations, as "enlarging" τ (i.e., adding disjuncts to τ) is *always* sound for proving safety. The *transitive projection* that we use to learn relations (see Sect. 4) heavily exploits this freedom. It modifies the recurrence analysis from [14] by replacing expensive operations – convex hulls and polyhedral projections – by a cheap variation of *model based projection* [29]. While convex hulls and polyhedral projections are over-approximations (for integer arithmetic), model based projections under-approximate. This is surprising at first, but the justification for using under-approximations is that, as mentioned above, "enlarging" τ is always sound.

Without our modifications, recurrence analysis over-approximates, so we "mix" over- and under-approximations. Thus, learned relations are *not* under-approximations, so TRL cannot prove unsafety and returns unknown if it fails to prove unreachability of the error states.

Learned relations may reduce the diameter, but computing the diameter is difficult. Instead, TRL adds *blocking clauses* to the SMT encoding that force the SMT solver to prefer learned relations over loops. Then unsatisfiability implies that the diameter has been reached, so that safe can be returned.

For our example (1), once (2) has been learned, it is preferred over (1). As (2) must not be used twice in a row, the corresponding SMT problem becomes unsatisfiable after adding a blocking clause that blocks (1) for the 1^{st} step, and one that blocks (1) for the 2^{nd} step. Since we check for reachability of error states after every step, this implies safety.

Outline. After introducing preliminaries in Sect. 2, we present our new algorithm TRL in Sect. 3. As TRL builds upon *transitive projections*, we show how to implement such a projection for linear integer arithmetic in Sect. 4. Finally, in Sect. 5 we discuss related work and evaluate our approach empirically.

2 Preliminaries

We assume familiarity with basics of first-order logic [13]. $\mathcal{V}$ is a countably infinite set of variables and $\mathcal{A}$ is a first-order theory with signature Σ and carrier $\mathcal{C}$. For each entity e, $\mathcal{V}(e)$ is the set of variables that occur in e. $\mathsf{QF}(\Sigma)$ denotes the set of all quantifier-free first-order formulas over Σ, and $\mathsf{QF}_\wedge(\Sigma)$ only contains conjunctions of Σ-literals. $\top$ and $\bot$ stand for "true" and "false", respectively.

Given $\psi \in \mathsf{QF}(\Sigma)$ with $\mathcal{V}(\psi) = \mathbf{y}$, we say that ψ is $\mathcal{A}$-*valid* (written $\models_\mathcal{A} \psi$) if every model of $\mathcal{A}$ satisfies the universal closure $\forall \mathbf{y}.\ \psi$ of ψ. A partial function $\sigma : \mathcal{V} \rightharpoonup \mathcal{C}$ is called a *valuation*. If $\mathcal{V}(\psi) \subseteq \mathrm{dom}(\sigma)$ and $\models_\mathcal{A} \sigma(\psi)$, then σ is an $\mathcal{A}$-*model* of ψ (written $\sigma \models_\mathcal{A} \psi$). Here, $\sigma(\psi)$ results from ψ by instantiating all variables according to σ. If ψ has an $\mathcal{A}$-model, then ψ is $\mathcal{A}$-*satisfiable*. If $\sigma(\psi)$ is $\mathcal{A}$-satisfiable (but not necessarily $\mathcal{V}(\psi) \subseteq \mathrm{dom}(\sigma)$), then we say that ψ is $\mathcal{A}$-*consistent* with σ. We write $\psi \models_\mathcal{A} \psi'$ for $\models_\mathcal{A} \psi \implies \psi'$, and $\psi \equiv_\mathcal{A} \psi'$ means $\models_\mathcal{A} \psi \iff \psi'$. In the sequel, we omit the subscript $\mathcal{A}$, and we just say "valid", "model", "satisfiable", and "consistent". We assume that $\mathcal{A}$ is complete (i.e., $\models \psi$ or $\models \neg\psi$ holds for every closed formula over Σ) and that $\mathcal{A}$ has an effective quantifier elimination procedure (i.e., quantifier elimination is computable).

We write $\mathbf{x}$ for sequences and x_i is the i^{th} element of $\mathbf{x}$, where x_1 denotes the first element. We use "::" for concatenation of sequences, where we identify sequences of length 1 with their elements, so, e.g., $x :: \mathbf{x} = [x] :: \mathbf{x}$.

Let $d \in \mathbb{N}$ be fixed, and let $\mathbf{x}, \mathbf{x}' \in \mathcal{V}^d$ be disjoint vectors of pairwise different variables, called the *pre-* and *post-variables*. All other variables are *extra variables*. Each $\tau \in \mathsf{QF}(\Sigma)$ induces a *transition relation* $\rightarrow_\tau$ on *states*, i.e., elements of $\mathcal{C}^d$, where $\mathbf{v} \rightarrow_\tau \mathbf{v}'$ iff $\tau[\mathbf{x}/\mathbf{v}, \mathbf{x}'/\mathbf{v}']$ is satisfiable. Here, $[\mathbf{x}/\mathbf{v}, \mathbf{x}'/\mathbf{v}']$ maps $x_i^{(\prime)}$ to $v_i^{(\prime)}$.

We call $\tau \in \mathsf{QF}(\Sigma)$ a *relational formula* if we are interested in τ's induced transition relation. *Transitions* are conjunctive relational formulas without extra variables (i.e., conjunctions of literals over pre- and post-variables). We sometimes identify τ with $\rightarrow_\tau$, so we may call τ a relation.

Algorithm 1: TRL – Input: a safety problem $\mathcal{T} = (\psi_{\mathsf{init}}, \tau, \psi_{\mathsf{err}})$

```
1  b ← 0;   π ← [τ];   BLOCKED ← ∅
2  add(μ_1(ψ_init))                                          // encode the initial states
3  while ⊤                                                   // main loop
4      b++;   push();   add(μ_b(ψ_err))                      // encode the error states
5      if check_sat() do return unknown else pop()           // check their reachability
6      push()                                                // add backtracking point
7      if b > 1 do add(x_id^(b) ≐ 1 ∨ x_id^(b) ≠̇ x_id^(b−1))   // encode transitivity
8      add(μ_b(⋁_{n=1}^{|π|}(π_n ∧ x_id ≐ n)))               // encode →_τ and learned relations
9      add(⋀_{(b,π)∈BLOCKED} π)                              // add blocking clauses for this b
10     if ¬check_sat() do return safe                        // check if the search space is exhausted
11     σ ← model();   τ ← trace_b(σ, π)                      // build trace from current model
12     if [τ_s, ..., τ_{s+ℓ−1}] is a loop do                 // search loop
13         σ_loop ← [x/σ(μ_{s,ℓ}(x)) | x ∈ x ∪ x']           // build the valuation for the loop
14         if no π ∈ tail(π) is consistent with σ_loop do    // redundancy check
15             τ_loop ← μ_{s,ℓ}^{-1}(⋀_{i=s}^{s+ℓ−1} μ_i(τ_i))   // build the loop
16             π ← π :: tp(τ_loop, σ ∘ μ_{s,ℓ})               // learn relation
17         let π ∈ tail(π) and σ̄ ⊇ σ_loop s.t. σ̄ ⊨ π      // pick suitable learned relation
18         BLOCKED.add(s + ℓ − 1, blocking_clause(s, ℓ, π, σ̄))   // block the loop
19         while b > s { pop(); b-- }                        // backtrack to the start of the loop
```

A τ-*run* is a sequence $\mathbf{v}_1 \to_\tau \ldots \to_\tau \mathbf{v}_k$. A *safety problem* $\mathcal{T}$ is a triple $(\psi_{\mathsf{init}}, \tau, \psi_{\mathsf{err}}) \in \mathsf{QF}(\Sigma) \times \mathsf{QF}(\Sigma) \times \mathsf{QF}(\Sigma)$ where $\mathcal{V}(\psi_{\mathsf{init}}) \cup \mathcal{V}(\psi_{\mathsf{err}}) \subseteq \mathbf{x}$. It is *unsafe* if there are $\mathbf{v}, \mathbf{v}' \in \mathcal{C}^d$ such that $[\mathbf{x}/\mathbf{v}] \models \psi_{\mathsf{init}}$, $\mathbf{v} \to_\tau^* \mathbf{v}'$, and $[\mathbf{x}/\mathbf{v}'] \models \psi_{\mathsf{err}}$.

Throughout the paper, we use c, d, e, k, ℓ, s for integer constants (where d always denotes the size of $\mathbf{x}$, and s and ℓ always denote the start and length of a loop); $\mathbf{v}$ for states; w, x, y for variables; τ, π for relational formulas; σ, θ for valuations; and μ for variable renamings.

3 Transitive Relation Learning

In this section, we present our novel model checking algorithm *Transitive Relation Learning* (TRL) in detail, see Algorithm 1. Here, for all $i, j \in \mathbb{N}_+ = \mathbb{N} \setminus \{0\}$ we define $\mu_{i,j}(x') := \overset{(i+j)}{x}$ if $x' \in \mathbf{x}'$ and $\mu_{i,j}(x) := \overset{(i)}{x}$, otherwise. So in particular, we have $\mu_{i,j}(\mathbf{x}) = \overset{(i)}{\mathbf{x}}$ and $\mu_{i,j}(\mathbf{x}') = \overset{(i+j)}{\mathbf{x}}$, where we assume that $\overset{(1)}{\mathbf{x}}, \overset{(2)}{\mathbf{x}}, \ldots \in \mathcal{V}^d$ are disjoint vectors of pairwise different fresh variables. Intuitively, the variables $\overset{(i)}{\mathbf{x}}$ represent the i^{th} state in a run, and applying $\mu_{i,j}$ to a relational formula yields a formula that relates the i^{th} and the $(i+j)^{th}$ state of a run. For convenience, we define $\mu_i := \mu_{i,1}$ for all $i \in \mathbb{N}_+$, i.e., $\mu_i(\mathbf{x}) = \overset{(i)}{\mathbf{x}}$ and $\mu_i(\mathbf{x}') = \overset{(i+1)}{\mathbf{x}}$. As in SMT-based BMC, TRL uses an incremental SMT solver to unroll the transition relation step by step (Line 8), but in contrast to BMC, TRL infers *learned relations* on the fly (Line 16). The *input formula* τ as well as all learned relations

are stored in $\boldsymbol{\pi}$. Before each unrolling, we set a backtracking point with the command push and add a suitably variable-renamed version of the description of the error states to the SMT problem (i.e., to the state of the underlying SMT solver) in Line 4. Then the command check_sat checks for reachability of error states, and the command pop removes all formulas from the SMT problem that have been added since the last invocation of push (Line 5), i.e., it removes the encoding of the error states (unless the check succeeds, so that TRL fails). For each unrolling, suitably variable renamed variants of $\boldsymbol{\pi}$'s elements are added to the underlying SMT problem with the command add in Line 8. If no error state is reachable after $b-1$ steps, but the transition relation cannot be unrolled b times (i.e., the SMT problem that corresponds to the b-fold unrolling is unsatisfiable), then the diameter of the analyzed system (including learned relations) has been reached, and hence safety has been proven (Line 10).

The remainder of this section is structured as follows: First, Sect. 3.1 introduces *conjunctive variable projections* that are used to compute the *trace* (Line 11 of Algorithm 1). Next, Sect. 3.2 defines *loops* and discusses how to find *non-redundant loops* that are suitable for learning new relations (Lines 12 to 15). Then, Sect. 3.3 introduces *transitive projections* that are used to learn relations (Lines 7 and 16). Finally, Sect. 3.4 presents *blocking clauses*, which ensure that learned relations are preferred over other (sequences of) transitions (Lines 9 and 17 to 19).

3.1 Conjunctive Variable Projections and Traces

To decide when to learn a new relation, TRL inspects the *trace* (Lines 11 and 12). The trace is a sequence of transitions induced by the formulas that have been added to the SMT problem while unrolling the transition relation, and by the current model (Line 11). To compute them, we use *conjunctive variable projections*, which are like *model based projections* [29], but always yield conjunctions.

Definition 2 (Conjunctive Variable Projection). *A function*

$$\mathsf{cvp} : \mathsf{QF}(\Sigma) \times (\mathcal{V} \rightharpoonup \mathcal{C}) \times 2^{\mathcal{V}} \to \mathsf{QF}(\Sigma)$$

is called a conjunctive variable projection *if*

1. $\sigma \models \mathsf{cvp}(\tau, \sigma, X)$
2. $\mathsf{cvp}(\tau, \sigma, X) \models \tau$
3. $\{\mathsf{cvp}(\tau, \theta, X) \mid \theta \models \tau\}$ *is finite*
4. $\mathcal{V}(\mathsf{cvp}(\tau, \sigma, X)) \subseteq X \cap \mathcal{V}(\tau)$
5. $\mathsf{cvp}(\tau, \sigma, X) \in \mathsf{QF}_\wedge(\Sigma)$

for all $\tau \in \mathsf{QF}(\Sigma)$, $X \subseteq \mathcal{V}$, *and* $\sigma \models \tau$. *We abbreviate* $\mathsf{cvp}(\tau, \sigma, \mathbf{x} \cup \mathbf{x}')$ *by* $\mathsf{cvp}(\tau, \sigma)$.

So like model based projection, cvp under-approximates quantifier elimination by projecting to the variables X (by (2) and (4)). To do so, it implicitly performs a finite case analysis (by (3)), which is driven by the model σ (by (1)). In contrast to model based projections, cvp always yields conjunctions (by (5)). Note that by (1), $\mathsf{cvp}(\tau, \sigma, X)$ may only contain variables from $\mathrm{dom}(\sigma)$.

Remark 3 (cvp *and* mbp*).* Conjunctive variable projections are obtained by combining a model based projection mbp (which satisfies Definition 2 (1–4)) with *syntactic implicant projection* sip [18]. To see how to compute mbp for linear integer arithmetic, recall that Cooper's method for quantifier elimination [11] essentially maps $\exists \mathbf{y}.\ \tau$ to a disjunction of formulas of the form $\tau[\mathbf{y}/\mathbf{t}]$, where the variables $\mathbf{y}$ do not occur in the terms $\mathbf{t}$. Instead, mbp just computes one of these disjuncts, which is satisfied by the provided model. For $\mathsf{sip}(\tau, \sigma)$, one computes the conjunction of all literals of τ's negation normal form that are satisfied by σ. Then $\mathsf{cvp}(\tau, \sigma) := \mathsf{sip}(\mathsf{mbp}(\tau, \sigma), \sigma)$.

Remark 4 (cvp *and Quantifier Elimination).* Definition 2 (1–4) imply

$$\exists \mathbf{y}.\ \tau \ \equiv \bigvee_{\sigma \models \tau} \mathsf{cvp}(\tau, \sigma) \qquad \text{where } \mathbf{y} \text{ are } \tau\text{'s extra variables.}$$

So cvp yields a quantifier elimination procedure qe which maps $\exists \mathbf{y}.\ \tau$ to *res*:

$$\mathit{res} \leftarrow \bot; \qquad \textbf{while } \tau \textit{ has a model } \sigma \ \{\mathit{res} \leftarrow \mathit{res} \vee \mathsf{cvp}(\tau, \sigma); \qquad \tau \leftarrow \tau \wedge \neg \mathsf{cvp}(\tau, \sigma)\}$$

But for a single model σ, $\mathsf{cvp}(\tau, \sigma)$ under-approximates quantifier elimination.

So $\mathsf{cvp}(\tau, \sigma)$ just computes one disjunct of $\mathsf{qe}(\exists \mathbf{y}.\ \tau)$ which is satisfied by σ. However, like model based projection, cvp can be implemented efficiently for many theories with effective, but very expensive quantifier elimination procedures.

Example 5 (cvp*).* Consider the following formula $\tilde{\tau}$:

$$\begin{array}{c}((w \doteq 0 \wedge \overset{(2)}{x} \doteq x{+}1 \wedge \overset{(2)}{y} \doteq y{+}1) \vee (\overset{(2)}{w} \doteq w \wedge w \doteq 1 \wedge \overset{(2)}{x} \doteq x{-}1 \wedge \overset{(2)}{y} \doteq y{-}1)) \wedge \\ ((\overset{(2)}{w} \doteq 0 \wedge x' \doteq \overset{(2)}{x}{+}1 \wedge y' \doteq \overset{(2)}{y}{+}1) \vee (w' \doteq \overset{(2)}{w} \wedge \overset{(2)}{w} \doteq 1 \wedge x' \doteq \overset{(2)}{x}{-}1 \wedge y' \doteq \overset{(2)}{y}{-}1))\end{array}$$

It encodes two steps with Example 1, where $\overset{(2)}{\mathbf{x}} = [\overset{(2)}{w}, \overset{(2)}{x}, \overset{(2)}{y}]$ represents the values after one step. In Line 11, Algorithm 1 might find a run like $\sigma(\overset{(1)}{\mathbf{x}}) \to_\tau \sigma(\overset{(2)}{\mathbf{x}}) \to_\tau \sigma(\overset{(3)}{\mathbf{x}})$ for

$$\sigma := [\overset{(1)}{w}/\overset{(1)}{x}/\overset{(1)}{y}/0,\ \ \overset{(2)}{w}/\overset{(2)}{x}/\overset{(2)}{y}/1,\ \ \overset{(3)}{w}/1,\ \overset{(3)}{x}/\overset{(3)}{y}/0].$$

Here, $[w/x/y/c, \ldots]$ abbreviates $[w/c, x/c, y/c, \ldots]$. Then the variable renaming $\mu_{1,2}$ allows us to instantiate the pre- and post-variables by the first and last state, resulting in the following model of $\tilde{\tau}$:

$$\tilde{\sigma} := \sigma \circ \mu_{1,2} = \sigma \cup [w/x/y/0,\ \ w'/1, x'/y'/0] \quad \text{where } (\sigma \circ \mu_{1,2})(x) = \sigma(\mu_{1,2}(x))$$

To get rid of $\overset{(2)}{w}, \overset{(2)}{x}, \overset{(2)}{y}$, one could compute $\mathsf{qe}(\exists \overset{(2)}{w}, \overset{(2)}{x}, \overset{(2)}{y}.\ \tilde{\tau})$, resulting in:

$$\begin{array}{rl} w \doteq 0 \wedge x' \doteq x + 2 \wedge y' \doteq y + 2 & (\mathsf{inc}) \\ \vee\ w \doteq 0 \wedge w' \doteq 1 \wedge x' \doteq x \wedge y' \doteq y & (\mathsf{eq}) \\ \vee\ w \doteq 1 \wedge w' \doteq 1 \wedge x' \doteq x - 2 \wedge y' \doteq y - 2. & (\mathsf{dec}) \end{array}$$

Instead, we may have $\mathsf{cvp}(\tilde{\tau}, \tilde{\sigma}) = (\mathsf{eq})$, as $\tilde{\sigma} \models (\mathsf{eq})$.

Intuitively, a relational formula τ describes how states can change, so it is composed of many different cases. These cases may be given explicitly (by disjunctions) or implicitly (by extra variables, which express non-determinism). Given a model σ of τ that describes a *concrete* change of state, cvp computes a description of the corresponding case. Computing *all* cases amounts to eliminating all extra variables and converting the result to DNF, which is impractical.

When unrolling the transition relation in Line 8 of Algorithm 1, we identify each relational formula π_n with its index n in the sequence $\boldsymbol{\pi}$. To this end, we use a fresh variable x_{id}, and our SMT encoding forces $\overset{(i)}{x_{\mathsf{id}}}$ to be the identifier of the relation that is used for the i^{th} step. Similarly to [19], the *trace* is the sequence of transitions that results from applying cvp to the unrolling of the transition relation that is constructed by Algorithm 1 in Line 8. So a trace is a sequence of transitions that can be applied subsequently, starting in an initial state.

Definition 6 (Trace). *Let $\boldsymbol{\pi}$ be a sequence of relational formulas, let*

$$\sigma \models \bigwedge_{i=1}^{b} \mu_i \left(\bigvee_{n=1}^{|\boldsymbol{\pi}|} (\pi_n \wedge x_{\mathsf{id}} \doteq n) \right) \qquad \text{where } b \in \mathbb{N}_+, \tag{3}$$

and let $\mathsf{id}(i) := \sigma(\overset{(i)}{x_{\mathsf{id}}})$. Then the trace induced by σ *is*

$$\mathsf{trace}_b(\sigma, \boldsymbol{\pi}) := [\mathsf{cvp}(\pi_{\mathsf{id}(i)}, \sigma \circ \mu_i)]_{i=1}^{b}.$$

Recall that μ_i renames $\mathbf{x}$ and $\mathbf{x}'$ into $\overset{(i)}{\mathbf{x}}$ and $\overset{(i+1)}{\mathbf{x}}$, and $\mathsf{id}(i) = \sigma(\overset{(i)}{x_{\mathsf{id}}})$ is the index of the relation from $\boldsymbol{\pi}$ that is used for the i^{th} step. So each model σ of (3) corresponds to a run $\sigma(\mu_1(\mathbf{x})) \to_{\pi_{\mathsf{id}(1)}} \ldots \to_{\pi_{\mathsf{id}(b-1)}} \sigma(\mu_b(\mathbf{x})) \to_{\pi_{\mathsf{id}(b)}} \sigma(\mu_b(\mathbf{x}'))$, and the trace induced by σ contains the transitions that were used in this run.

Example 7 (Trace). Consider the extension of σ from Example 5 with $[\overset{(1)}{x_{\mathsf{id}}}/1,\ \overset{(2)}{x_{\mathsf{id}}}/1]$:

$$\sigma := [\overset{(1)}{w}/\overset{(1)}{x}/\overset{(1)}{y}/0, \overset{(1)}{x_{\mathsf{id}}}/1, \quad \overset{(2)}{w}/\overset{(2)}{x}/\overset{(2)}{y}/\overset{(2)}{x_{\mathsf{id}}}/1, \quad \overset{(3)}{w}/1, \overset{(3)}{x}/\overset{(3)}{y}/0]$$

Thus, $\mathsf{id}(1) = \sigma(\overset{(1)}{x_{\mathsf{id}}}) = 1$, $\mathsf{id}(2) = \sigma(\overset{(2)}{x_{\mathsf{id}}}) = 1$, and $\pi_{\mathsf{id}(1)} = \pi_{\mathsf{id}(2)} = \pi_1 = \tau$. Then

$$\begin{aligned}&\mathsf{trace}_2(\sigma, [\tau, \tau]) = [\mathsf{cvp}(\tau, \sigma \circ \mu_1), \mathsf{cvp}(\tau, \sigma \circ \mu_2)] \\ &= [\mathsf{cvp}(\tau, [w/x/y/0,\ w'/x'/y'/1]), \mathsf{cvp}(\tau, [w/x/y/1,\ w'/1, x'/y'/0])] = [\tau_{\mathsf{inc}}, \tau_{\mathsf{dec}}].\end{aligned}$$

3.2 Loops

As $\boldsymbol{\pi}$ only gives rise to finitely many transitions, the trace is bound to contain *loops*, eventually (unless Algorithm 1 terminates beforehand).

Definition 8 (Loop). *A sequence of transitions $\tau_1, \ldots, \tau_k$ is called a* loop *if there are $\mathbf{v}_0, \ldots, \mathbf{v}_{k+1} \in \mathcal{C}^d$ such that $\mathbf{v}_0 \to_{\tau_1} \ldots \to_{\tau_k} \mathbf{v}_k \to_{\tau_1} \mathbf{v}_{k+1}$.*

Intuitively, these loops are the reason why BMC may diverge. To prevent divergence, TRL learns a new relation when a loop is detected (Line 12).

Remark 9 (Finding Loops). Loops can be detected by SMT solving. A cheaper way is to look for duplicates on the trace, but then loops are found "later", as a trace $[\ldots, \pi, \pi, \ldots]$ is needed to detect a loop π, but one occurrence of π is insufficient. As a trade-off between precision and efficiency, our implementation uses an approximation based on *dependency graphs* [19]. More precisely, our implementation maintains a graph $\mathcal{G}$ whose nodes are transitions, and it adds an edge between two transitions if they occur subsequently on the trace at some point. Then it considers a subsequence $\tau_i, \ldots, \tau_j$ of the trace to be a potential loop if $\mathcal{G}$ contains an edge from τ_j to τ_i.

Remark 10 (Disregarding "Learned" Loops). One should disregard "loops" consisting of a single *learned transition*, i.e., a transition that results from applying cvp to some $\pi \in \mathsf{tail}(\boldsymbol{\pi})$, where $\mathsf{tail}(\tau :: \boldsymbol{\pi}') := \boldsymbol{\pi}'$. Here, $\mathsf{tail}(\boldsymbol{\pi})$ contains all learned relations, as the first element of $\boldsymbol{\pi}$ is the input formula τ. The reason is that our goal is to deduce transitive relations, but learned relations are already transitive. In the sequel, we assume that the check in Line 12 fails for such loops.

If there are several choices for s and ℓ in Line 12, then our implementation only considers loops of minimal length and, among those, it minimizes s.

Example 11 (Detecting Loops). Consider the following model for τ from Example 1.

$$\sigma := [\overset{(1)}{w}/\overset{(1)}{x}/\overset{(1)}{y}/0, \overset{(1)}{x_{\mathsf{id}}}/1, \quad \overset{(2)}{w}/0, \overset{(2)}{x}/\overset{(2)}{y}/1]$$

Then $\mathsf{trace}_1(\sigma, [\tau]) = [\tau_{\mathsf{inc}}]$. As τ_{inc} is a loop,[3] this causes TRL to learn a relation like the following one (see Sect. 3.3 for details).

$$w \doteq 0 \wedge x' > x \wedge x' - x \doteq y' - y \qquad (\tau_{\mathsf{inc}}^{+})$$

TRL only learns relations from loops that are *non-redundant* w.r.t. all relations that have been learned before [18].

Definition 12 (Redundancy). *If* $\rightarrow_{\tau} \subseteq \rightarrow_{\tau'}$*, then* τ *is* redundant *w.r.t.* τ'*.*

Example 13 The relation τ_{inc} is redundant w.r.t. τ_{inc}^{+}, but τ_{dec} is not.

Line 14 uses a sufficient criterion for non-redundancy: If all learned relations are falsified by the values before and after the loop, then $\tau_s, \ldots, \tau_{s+\ell-1}$ cannot be simulated by a previously learned relation, so it is non-redundant and we learn a new relation. The values before and after the loop are obtained from the

[3] Depending on the technique that is used to detect loops, an actual implementation might require one more unrolling of $\rightarrow_{\tau}$ to obtain the trace $[\tau_{\mathsf{inc}}, \tau_{\mathsf{inc}}]$ in order to detect the loop τ_{inc}, see Remark 9.

current model σ by setting $\mathbf{x}$ to $\sigma(\overset{(s)}{\mathbf{x}})$ and $\mathbf{x}'$ to $\sigma(\overset{(s+\ell)}{\mathbf{x}})$, i.e., we use $\sigma \circ \mu_{s,\ell}$ in Line 13.

To learn a new relation, we first compute the relation

$$\tau_{\mathsf{loop}} := \mu_{s,\ell}^{-1}(\varphi_{\mathsf{loop}}) \qquad \text{where} \qquad \varphi_{\mathsf{loop}} := \bigwedge_{i=s}^{s+\ell-1} \mu_i(\tau_i) \tag{4}$$

of the loop in Line 15, where $\mu_{s,\ell}^{-1}$ is the inverse of $\mu_{s,\ell}$. So in Example 11, we have $\sigma \circ \mu_{1,1} \supseteq [w/x/y/0,\, w'/0, x'/y'/1]$ and $\tau_{\mathsf{loop}} := \mu_{1,1}^{-1}(\mu_1(\tau_{\mathsf{inc}})) = \tau_{\mathsf{inc}}$ as $s = \ell = 1$. So $\sigma \circ \mu_{s,\ell}$ indeed corresponds to one evaluation of the loop, as $\sigma \circ \mu_{s,\ell} \models \tau_{\mathsf{loop}}$.

To see that τ_{loop} is also the desired relation in general, note that φ_{loop} is the conjunction of the transitions that constitute the loop, where all variables are renamed as in Line 8 of Algorithm 1, i.e., in such a way that the post-variables of the i^{th} step are equal to the pre-variables of the $(i+1)^{th}$ step. So we have $\sigma \models \varphi_{\mathsf{loop}}$ and thus $\sigma \circ \mu_{s,\ell} \models \tau_{\mathsf{loop}}$. Hence, we can use τ_{loop} and $\sigma \circ \mu_{s,\ell}$ to learn a new relation via so-called *transitive projections* in Line 16.

3.3 Transitive Projections

We now define *transitive projections* that approximate transitive closures of loops. As explained in Sect. 1, we do not restrict ourselves to under- or over-approximations, but we allow "mixtures" of both. Analogously to cvp, transitive projections perform a finite case analysis that is driven by the provided model σ.

Definition 14 (Transitive Projection). *A function*

$$\mathsf{tp} : \mathsf{QF}(\Sigma) \times (\mathcal{V} \rightharpoonup \mathcal{C}) \to \mathsf{QF}(\Sigma)$$

is called a transitive projection *if the following holds for all transitions* $\tau \in \mathsf{QF}(\Sigma)$ *and all* $\sigma \models \tau$*:*

1. $\mathsf{tp}(\tau, \sigma)$ *is consistent with* σ
2. $\{\mathsf{tp}(\tau, \theta) \mid \theta \models \tau\}$ *is finite*
3. $\rightarrow_{\mathsf{tp}(\tau,\sigma)}$ *is transitive*

Clearly, the specifics of tp depend on the underlying theory. Our implementation of tp for quantifier-free linear integer arithmetic is explained in Sect. 4.

*Example 15 (*tp*).* For Example 1, $\tau_{\mathsf{ti}} := x' - x \doteq y' - y$ over-approximates the transitive closure $\rightarrow_\tau^+$. Such over-approximations are also called *transition invariants* [38]. With τ_{ti}, one can prove safety for any ψ_{init} with $\psi_{\mathsf{init}} \models x \doteq y$, as then $\psi_{\mathsf{init}} \wedge \tau_{\mathsf{ti}} \models x' \doteq y'$, which shows that no error state with $w \doteq 1 \wedge x \leq 0 \wedge y > 0$ is reachable.

By using cvp, TRL instead considers τ_{inc} and τ_{dec} separately and learns

$$\mathsf{tp}(\tau_{\mathsf{inc}}, \sigma_{\mathsf{inc}}) := w \doteq 0 \wedge x' > x \wedge x' - x \doteq y' - y \tag{τ_{inc}^+}$$

$$\mathsf{tp}(\tau_{\mathsf{dec}}, \sigma_{\mathsf{dec}}) := w' \doteq w \wedge w \doteq 1 \wedge x' < x \wedge x' - x \doteq y' - y \tag{τ_{dec}^+}$$

if $\sigma_{\mathsf{inc}} \models \tau_{\mathsf{inc}}$ and $\sigma_{\mathsf{dec}} \models \tau_{\mathsf{dec}}$. In this way, Algorithm 1 can learn disjunctive relations like $\tau^+_{\mathsf{inc}} \vee \tau^+_{\mathsf{dec}}$, even if tp only yields conjunctive relational formulas (which is true for our current implementation of tp – see Sect. 4 – but not enforced by Definition 14).

In contrast to conjunctive variable projections, $\mathsf{tp}(\tau, \sigma)$ may contain extra variables that do not occur in τ (which will be exploited in Sect. 4). Hence, instead of $\sigma \models \mathsf{tp}(\tau, \sigma)$ we require consistency with σ, i.e., $\sigma(\mathsf{tp}(\tau, \sigma))$ must be satisfiable.

Remark 16 (Properties of tp*).* Due to Definition 14 (1), our definition of tp implies

$$\tau \models \exists \mathbf{y}. \bigvee_{\sigma \models \tau} \mathsf{tp}(\tau, \sigma), \quad \text{and thus,} \quad \to_\tau \subseteq \bigcup_{\sigma \models \tau} \to_{\mathsf{tp}(\tau,\sigma)},$$

where $\mathbf{y}$ are the extra variables of $\bigvee_{\sigma \models \tau} \mathsf{tp}(\tau, \sigma)$. However, Definition 14 does *not* ensure $\to^+_\tau \subseteq \bigcup_{\sigma \models \tau} \to_{\mathsf{tp}(\tau,\sigma)}$. So there is no guarantee that tp covers $\to^+_\tau$ entirely, i.e., tp cannot be used to compute transition invariants, in general. Definition 14 does not ensure $\to^+_\tau \supseteq \bigcup_{\sigma \models \tau} \to_{\mathsf{tp}(\tau,\sigma)}$ either, as $\mathsf{tp}(\tau, \sigma)$ does not imply $\sigma(\mathbf{x}) \to^+_\tau \sigma(\mathbf{x}')$.

Example 17. To see that tp computes no over- or under-approximations, let

$$\tau := x' \doteq x + 1 \wedge y' \doteq y + x.$$

Then for all $\sigma \models \tau$, we might have:

$$\mathsf{tp}(\tau, \sigma) = \begin{cases} x \geq 0 \wedge x' > x \wedge y' \geq y, & \text{if } \sigma(x) \geq 0 \\ x < 0 \wedge x' > x \wedge y' < y, & \text{if } \sigma(x) < 0 \end{cases}$$

However, $(x \geq 0 \wedge x' > x \wedge y' \geq y) \vee (x < 0 \wedge x' > x \wedge y' < y)$ is not an over-approximation of $\to^+_\tau$ (i.e., $\to^+_\tau \not\subseteq \bigcup_{\sigma \models \tau} \to_{\mathsf{tp}(\tau,\sigma)}$), as we have, e.g.,

$$(-1, 0) \to_\tau (0, -1) \to_\tau (1, -1) \to_\tau (2, 0), \qquad \text{but} \qquad (-1, 0) \not\to_{\mathsf{tp}(\tau,\sigma)} (2, 0)$$

for all $\sigma \models \tau$. Moreover, we also have $\to^+_\tau \not\supseteq \bigcup_{\sigma \models \tau} \to_{\mathsf{tp}(\tau,\sigma)}$, since

$$(-1, 0) \to_{\mathsf{tp}(\tau,\sigma)} (10, -20), \qquad \text{but} \qquad (-1, 0) \not\to^+_\tau (10, -20)$$

if $\sigma(x) < 0$. In contrast to $\mathsf{tp}(\tau, \sigma)$, linear over-approximations for $\to^+_\tau$ like $x' > x$ cannot distinguish whether y increases or decreases.

As TRL proves safety via *blocking clauses* (Sect. 3.4) that only block steps that are covered by learned relations, the fact that tp does not yield over-approximations does not affect soundness. However, it may cause divergence (Remark 23).

Recall that our SMT encoding forces $\overset{(i)}{x}_{\mathsf{id}}$ to be the identifier of the relation that is used for the i^{th} step (Line 8). To exploit transitivity of tp, we add the constraint $\overset{(b)}{x}_{\mathsf{id}} \doteq 1 \vee \overset{(b)}{x}_{\mathsf{id}} \neq \overset{(b-1)}{x}_{\mathsf{id}}$ in Line 7, so that learned relations (with index > 1) are not used several times in a row, since this is unnecessary for transitive relations.

3.4 Blocking Clauses

In Line 17, we are guaranteed to find a learned relation π which is consistent with σ_{loop}: If our sufficient criterion for non-redundancy in Line 14 failed, then the existence of π is guaranteed. Otherwise, we learned a new relation π in Line 16 which is consistent with $\sigma_{\mathsf{loop}} \subseteq \sigma \circ \mu_{s,\ell}$ by definition of tp. Thus, we can use π and a model $\overline{\sigma} \supseteq \sigma_{\mathsf{loop}}$ of π to record a *blocking clause* in Line 18.

Definition 18 (Blocking Clauses). *Consider a relational formula π, and let $\overline{\sigma}$ be a model of π. We define:*

$$\mathsf{blocking_clause}(s,\ell,\pi,\overline{\sigma}) := \begin{cases} \mu_{s,\ell}(\neg\mathsf{cvp}(\pi,\overline{\sigma})) \vee \overset{(s)}{x}_{\mathsf{id}} > 1, & \text{if } \ell = 1 \\ \mu_{s,\ell}(\neg\mathsf{cvp}(\pi,\overline{\sigma})), & \text{if } \ell > 1 \end{cases}$$

Here, s and ℓ are natural numbers such that $[\tau_i]_{i=s}^{s+\ell-1}$ is a (possibly) redundant loop on the trace. Blocking clauses exclude models that correspond to runs

$$\mathbf{v}_1 \rightarrow_{\tau_1} \ldots \rightarrow_{\tau_{s-1}} \mathbf{v}_s \rightarrow_{\tau_s} \ldots \rightarrow_{\tau_{s+\ell-1}} \mathbf{v}_{s+\ell} \tag{5}$$

where $\mathbf{v}_s \rightarrow_\pi \mathbf{v}_{s+\ell}$. Intuitively, if $\ell = 1$ then $\mathsf{blocking_clause}(s,\ell,\pi,\overline{\sigma})$ states that one may still evaluate $\mathbf{v}_s$ to $\mathbf{v}_{s+\ell}$, but one has to use a learned transition. If $\ell > 1$, then $\mathsf{blocking_clause}(s,\ell,\pi,\overline{\sigma})$ states that one may still evaluate $\mathbf{v}_s$ to $\mathbf{v}_{s+\ell}$, but not in ℓ steps. More precisely, blocking clauses take into account that

$$\mathbf{v}_1 \rightarrow_{\tau_1} \ldots \rightarrow_{\tau_{s+\ell-2}} \mathbf{v}_{s+\ell-1} \tag{6} \quad \text{and} \quad \mathbf{v}_1 \rightarrow_{\tau_1} \ldots \rightarrow_{\tau_{s-1}} \mathbf{v}_s \rightarrow_\pi \mathbf{v}_{s+\ell} \tag{7}$$

must not be blocked to ensure that $\mathbf{v}_2, \ldots, \mathbf{v}_{s+\ell}$ remain reachable. For the former, note that blocking clauses affect the suffix $\mathbf{v}_s \rightarrow_{\tau_s} \ldots \rightarrow_{\tau_{s+\ell-1}} \mathbf{v}_{s+\ell}$ of (5) (as they contain $\mu_{s,\ell}(\neg\mathsf{cvp}(\pi,\overline{\sigma}))$), but not (6), so $\mathbf{v}_2, \ldots, \mathbf{v}_{s+\ell-1}$ remain reachable.

Regarding (7), note that (5) corresponds to one unrolling of the loop (without using the newly learned relation π). In contrast, (7) simulates one unrolling of the loop using the new relation π. To see that our blocking clauses prevent the sequence (5), but not the sequence (7), first consider the case $\ell > 1$. Then (7) is not affected by the blocking clause, as it requires less than $s + \ell$ steps. If $\ell = 1$, then the loop that needs to be blocked is a single *original transition* (i.e., a transition that results from applying cvp to τ) due to Remark 10. So $\overset{(s)}{x}_{\mathsf{id}} > 1$ is falsified by (5), as τ_s is an original transition, i.e., using it for the s^{th} step implies $\overset{(s)}{x}_{\mathsf{id}} \doteq 1$. However, $\overset{(s)}{x}_{\mathsf{id}} > 1$ is satisfied by (7), as π is a learned transition, so using it implies $\overset{(s)}{x}_{\mathsf{id}} > 1$.

Remark 19 (Extra Variables and Negation). In Definition 18, cvp is used to project π according to the model $\overline{\sigma}$. In this way, negation has the intended effect, i.e.,

$$[\mathbf{x}/\mathbf{v}, \mathbf{x}'/\mathbf{v}'] \models \neg\mathsf{cvp}(\ldots) \qquad \text{iff} \qquad \mathbf{v} \not\rightarrow_{\mathsf{cvp}(\ldots)} \mathbf{v}',$$

as $\mathsf{cvp}(\ldots)$ has no extra variables. To see why this is important here, consider the relation $\tau := n > 0 \wedge x' \doteq x + n$, where n is an extra variable. Then $0 \to_\tau 1$, but

$$\neg\tau[x/0, x'/1] = (n \leq 0 \vee x' \neq x + n)[x/0, x'/1] = n \leq 0 \vee 1 \neq n$$

is satisfiable, so $\neg\tau$ is not a suitable characterization of $\not\to_\tau$. The reason is that n is implicitly existentially quantified in τ. So to characterize $\not\to_\tau$, we have to negate $\exists n.\ \tau$ instead of τ, resulting in $\forall n.\ n \leq 0 \vee x' \neq x + n$. Then, as desired,

$$(\forall n.\ n \leq 0 \vee x' \neq x + n)[x/0, x'/1] = \forall n.\ n \leq 0 \vee 1 \neq n$$

is invalid. To avoid quantifiers, we eliminate extra variables via cvp instead.

In Line 18, a pair consisting of $s + \ell - 1$ and the blocking clause is added to BLOCKED. The first component means that the blocking clause has to be added to the SMT encoding when the transition relation is unrolled for the $(s+\ell-1)^{th}$ time, i.e., when $b = s+\ell-1$. So blocking clauses are added to the SMT encoding "on demand" (in Line 9) to block loops that have been found on the trace at some point. Afterwards, TRL backtracks to the last step before the loop in Line 19.

Remark 20 (Adding Blocking Clauses). To see why blocking clauses must only be added to the SMT encoding in the $(s+\ell-1)^{th}$ unrolling, assume $\pi \equiv \top$. Then, e.g., $\mathsf{blocking_clause}(1, 2, \pi, \overline{\sigma}) \equiv \bot$. This means that unrolling the transition relation twice is superfluous, as every state can be reached in a single step with π, so the diameter is 1. But after learning π when $b = 2$ and backtracking to $b = 0$, adding such a blocking clause too early (e.g., before the first unrolling of the transition relation) would *immediately* result in an unsatisfiable SMT problem.

Example 21 (Blocking Redundant Loops). Consider the model

$$\sigma := [\overset{(1)}{w}/\overset{(1)}{x}/\overset{(1)}{y}/0, \overset{(1)}{x_{\mathsf{id}}}/2, \quad \overset{(2)}{w}/0, \overset{(2)}{x}/\overset{(2)}{y}/2, \overset{(2)}{x_{\mathsf{id}}}/1, \quad \overset{(3)}{x}/\overset{(3)}{y}/3]$$

and assume that TRL has already learned the relation τ^+_{inc} (i.e., $\boldsymbol{\pi} = [\tau, \tau^+_{\mathsf{inc}}]$). Moreover, assume that the trace is $[\tau^+_{\mathsf{inc}}, \tau_{\mathsf{inc}}]$, so that TRL detects the loop τ_{inc}. To check for non-redundancy, we instantiate the pre- and post-variables in τ^+_{inc} according to σ, taking the renaming μ_2 into account (note that here $s = 2$, $\ell = 1$, and $\mu_{s,\ell} = \mu_{2,1} = \mu_2$):

$$\sigma(\mu_2(\tau^+_{\mathsf{inc}})) = \tau^+_{\mathsf{inc}}[w/0, x/y/2, x'/y'/3] \equiv \top.$$

So our sufficient criterion for non-redundancy fails, as τ_{inc} is indeed redundant w.r.t. τ^+_{inc}. Thus, TRL records that the following blocking clause has to be added

for the second unrolling (i.e., when $b = s + \ell - 1 = 2$).

$$\begin{aligned}
&\mu_{s,\ell}(\neg \mathsf{cvp}(\tau^+_{\mathsf{inc}}, \overline{\sigma})) \vee \overset{(s)}{x}_{\mathsf{id}} > 1 \;=\; \mu_{s,\ell}(\neg \tau^+_{\mathsf{inc}}) \vee \overset{(s)}{x}_{\mathsf{id}} > 1 \;\text{(as } \tau^+_{\mathsf{inc}} \text{ is a transition)} \\
&= \mu_2(\neg(w \doteq 0 \wedge x' > x \wedge x' - x \doteq y' - y)) \vee \overset{(2)}{x}_{\mathsf{id}} > 1 \\
&\equiv (w \not\doteq 0 \vee x' \leq x \vee x' - x \not\doteq y' - y)[w/\overset{(2)}{w}, x/\overset{(2)}{x}, y/\overset{(2)}{y}, x'/\overset{(3)}{x}, y'/\overset{(3)}{y}] \vee \overset{(2)}{x}_{\mathsf{id}} > 1 \\
&= \overset{(2)}{w} \not\doteq 0 \vee \overset{(3)}{x} \leq \overset{(2)}{x} \vee \overset{(3)}{x} - \overset{(2)}{x} \not\doteq \overset{(3)}{y} - \overset{(2)}{y} \vee \overset{(2)}{x}_{\mathsf{id}} > 1
\end{aligned}$$

As this blocking clause is falsified by σ, it prevents TRL from finding the same model again after backtracking in Line 19, so that TRL makes progress.

The following theorem states that our approach is sound.

Theorem 22. *If TRL($\mathcal{T}$) returns* safe*, then $\mathcal{T}$ is safe.*

Proof (Sketch). The key idea of the proof, which works by induction over the length of runs, is to show that blocking clauses do not prevent us from reaching all reachable states. See [20] for the full proof.

Remark 23 (Termination). In general, Algorithm 1 does not terminate, since tp decomposes the relation into finitely many cases and approximates their transitive closures independently, but $\rightarrow^+_{\tau_{\mathsf{loop}}} \subseteq \bigcup_{\sigma \models \tau_{\mathsf{loop}}} \rightarrow_{\mathsf{tp}(\tau_{\mathsf{loop}}, \sigma)}$ is not guaranteed (Remark 16). To see why this may prevent termination, consider a loop τ_{loop} and assume that there are reachable states $\mathbf{v}, \mathbf{v}'$ with $\mathbf{v} \rightarrow^+_{\tau_{\mathsf{loop}}} \mathbf{v}'$, but $\mathbf{v} \not\rightarrow_{\mathsf{tp}(\tau_{\mathsf{loop}}, \sigma)} \mathbf{v}'$ for all models σ of τ_{loop}. Then TRL may find a model that corresponds to a run from $\mathbf{v}$ to $\mathbf{v}'$. Unless $\mathbf{v}$ can be evaluated to $\mathbf{v}'$ with another learned transition $\pi \notin \{\mathsf{tp}(\tau_{\mathsf{loop}}, \sigma) \mid \sigma \models \tau_{\mathsf{loop}}\}$ by coincidence, this loop cannot be blocked and TRL learns a new relation. Thus, TRL may keep learning new relations as long as there are loops whose transitive closure is not yet covered by learned relations.

As the elements of $\{\mathsf{tp}(\tau, \sigma) \mid \sigma \models \tau\}$ are independent of each other, a more "global" view may help to enforce convergence. We leave that to future work.

Example 24 (Example 1 Finished). After learning τ^+_{dec} and τ^+_{inc}, the underlying SMT problem becomes unsatisfiable when $b = 3$ after adding appropriate blocking clauses, so that τ^+_{dec} and τ^+_{inc} are preferred over τ_{dec} and τ_{inc}. The reason is that τ^+_{dec} and τ^+_{inc} must not be used twice in a row due to Line 7 of Algorithm 1, and τ^+_{inc} cannot be used after τ^+_{dec}, as it requires $w \doteq 0$, but τ^+_{dec} sets w to 1. Thus, Algorithm 1 returns safe. See [20] for a detailed run of Algorithm 1 on Example 1.

4 Implementing tp for Linear Integer Arithmetic

We now explain how to compute transitive projections for quantifier-free linear integer arithmetic via recurrence analysis. As in SMT-LIB [2], in our setting linear integer arithmetic also features (in)divisibility predicates of the form $e|t$

(or $e \not| t$) where $e \in \mathbb{N}_+$ and t is an integer-valued term. Then we have $\sigma \models e|t$ iff $\sigma(t)$ is a multiple of e, and $\sigma \models e \not| t$, otherwise.

The technique that we use is inspired by the recurrence analysis from [14]. However, there are some important differences. The approach from [14] computes convex hulls to over-approximate disjunctions by conjunctions, and it relies on polyhedral projections. In our setting, we always have a suitable model at hand, so that we can use cvp instead. Hence, our recurrence analysis can be implemented more efficiently.[4] Additionally, our recurrence analysis can handle divisibility predicates, which are not covered in [14].

On the other hand, [14] yields an over-approximation of the transitive closure of the given relation, whereas our approach performs an implicit case analysis (via cvp) and only yields an over-approximation of the transitive closure of one out of finitely many cases.

Moreover, the recurrence analysis from [14] also discovers non-linear relations, and then uses linearization techniques to eliminate them. For simplicity, our recurrence analysis only derives linear relations so far. However, just like [14], we could also derive non-linear relations and linearize them afterwards. Apart from these differences, our technique is analogous to [14].

In the sequel, let τ and $\sigma \models \tau$ be fixed. Our implementation of $\mathsf{tp}(\tau, \sigma)$ first searches for *recurrent literals*, i.e., literals of the form[5]

$$t \bowtie 0 \text{ or } e|t \quad \text{where} \quad t = \sum_{x \in \mathbf{x}} c_x \cdot (x' - x) + c, \ \bowtie \in \{\leq, \geq, <, >, \doteq\}, \text{ and } c_x, c \in \mathbb{Z}.$$

Hence, these literals provide information about the change of values of variables. To find such literals, we introduce a fresh variable x_δ for each $x \in \mathbf{x}$, and we conjoin $x_\delta \doteq x' - x$ to τ, i.e., we compute

$$\tau_{\wedge\delta} := \tau \wedge \bigwedge_{x \in \mathbf{x}} x_\delta \doteq x' - x.$$

So the value of x_δ corresponds to the change of x when applying τ. Next, we use cvp to eliminate all variables but $\{x_\delta \mid x \in \mathbf{x}\}$ from $\tau_{\wedge\delta}$, resulting in τ_δ. More precisely, we have

$$\tau_\delta := \mathsf{cvp}(\tau_{\wedge\delta}, \quad \sigma \uplus [x_\delta / \sigma(x' - x) \mid x \in \mathbf{x}], \quad \{x_\delta \mid x \in \mathbf{x}\}).$$

[4] The *double description method*, which is popular for computing polyhedral projections and convex hulls, and other state-of-the-art approaches have exponential complexity [7,39]. See [23] for an easily accessible discussion of the complexity of the double description method. In contrast, combining the model based projection from [29] with syntactic implicant projection [18] yields a polynomial time algorithm for cvp.

[5] W.l.o.g., we assume that literals are never negated, as we can negate the corresponding (in)equalities or divisibility predicates directly instead. Furthermore, in our implementation, we replace disequalities $s \neq t$ with $s > t \vee s < t$ and eliminate the resulting disjunction via cvp to obtain a transition without disequalities.

Finally, to obtain a formula where all literals are recurrent, we replace each x_δ by its definition, i.e., we compute

$$\tau_{\mathsf{rec}} := \tau_\delta[x_\delta / x' - x \mid x \in \mathbf{x}].$$

Example 25 (Finding Recurrent Literals). Consider the transition τ_{dec}. We first construct the formula

$$\tau_{\wedge\delta} := \tau_{\mathsf{dec}} \wedge w_\delta \doteq w' - w \wedge x_\delta \doteq x' - x \wedge y_\delta \doteq y' - y.$$

Then for any model $\sigma \models \tau_{\mathsf{dec}}$, we get[6]

$$\begin{aligned}\tau_\delta &:= \mathsf{cvp}(\tau_{\wedge\delta}, \quad \sigma \uplus [w_\delta/0,\ x_\delta/-1,\ y_\delta/-1], \quad \{w_\delta, x_\delta, y_\delta\}) \\ &= w_\delta \doteq 0 \wedge x_\delta \doteq -1 \wedge y_\delta \doteq -1.\end{aligned}$$

Next, replacing w_δ, x_δ, and y_δ with their definition results in

$$\begin{aligned}\tau_{\mathsf{rec}} &:= w' - w \doteq 0 \wedge x' - x \doteq -1 \wedge y' - y \doteq -1 \\ &\equiv w' - w \doteq 0 \wedge x' - x + 1 \doteq 0 \wedge y' - y + 1 \doteq 0.\end{aligned}$$

Then the construction of $\mathsf{tp}(\tau, \sigma)$ proceeds as follows:

- $\mathsf{tp}(\tau, \sigma)$ contains the literal $n > 0$, where $n \in \mathcal{V}$ is a fresh extra variable
- for each literal $\sum_{x \in \mathbf{x}} c_x \cdot (x' - x) + c \bowtie 0$ of τ_{rec}, $\mathsf{tp}(\tau, \sigma)$ contains the literal $\sum_{x \in \mathbf{x}} c_x \cdot (x' - x) + n \cdot c \bowtie 0$
- for each literal $e \mid \sum_{x \in \mathbf{x}} c_x \cdot (x' - x) + c$ of τ_{rec}, $\mathsf{tp}(\tau, \sigma)$ contains the literal $e \mid \sum_{x \in \mathbf{x}} c_x \cdot (x' - x) + n \cdot c$

Intuitively, the extra variable n can be thought of as a "loop counter", i.e., when n is instantiated with some constant k, then the literals above approximate the change of variables when $\rightarrow_\tau$ is applied k times.

Example 26 (Computing tp *(1)).* Continuing Example 25, $\mathsf{tp}(\tau_{\mathsf{dec}}, \sigma)$ contains the literals

$$\begin{aligned}&n > 0 \wedge w' - w + n \cdot 0 \doteq 0 \wedge x' - x + n \cdot 1 \doteq 0 \wedge y' - y + n \cdot 1 \doteq 0 \\ \equiv\ &n > 0 \wedge w' \doteq w \wedge x' \doteq x - n \wedge y' \doteq y - n\end{aligned}$$

for any model $\sigma \models \tau_{\mathsf{dec}}$. Note that in our example, this formula precisely characterizes the change of the variables after n iterations of $\rightarrow_{\tau_{\mathsf{dec}}}$. To simplify the formula above, we can propagate the equality $n = x - x'$, resulting in:

$$\begin{aligned}&x - x' > 0 \wedge w' \doteq w \wedge y' \doteq y - x + x' \\ \equiv\ &w' \doteq w \wedge x' < x \wedge x' - x \doteq y' - y \end{aligned} \tag{8}$$

[6] In the case of τ_{dec}, we obtain the same formula τ_δ for *every* model $\sigma \models \tau_{\mathsf{dec}}$, as variables can simply be eliminated by propagating equalities.

Compared to τ_{dec}^{+}, (8) lacks the literal $w \doteq 1$. To incorporate information about the pre- and post-variables (but not about their relation) we conjoin $\mathsf{cvp}(\tau, \sigma, \mathbf{x})$ and $\mathsf{cvp}(\tau, \sigma, \mathbf{x}')$ to $\mathsf{tp}(\tau, \sigma)$.

Example 27 (Computing tp *(2)).* We finish Example 26 by conjoining

$$\mathsf{cvp}(\tau_{\mathsf{dec}}, \sigma, \{w, x, y\}) = w \doteq 1 \qquad \text{and} \qquad \mathsf{cvp}(\tau_{\mathsf{dec}}, \sigma, \{w', x', y'\}) = w' \doteq 1$$

to (8), resulting in:

$$w' \doteq w \wedge x' < x \wedge x' - x \doteq y' - y \wedge w \doteq 1 \wedge w' \doteq 1 \quad \equiv \quad \tau_{\mathsf{dec}}^{+}$$

Example 28 (Divisibility). To see how tp can handle divisibility predicates, consider the transition

$$\tau := 2|x \wedge 3|x' - x + 1.$$

Then our approach identifies the recurrent literal $\tau_{\mathsf{rec}} = 3|x' - x + 1$, so that $\mathsf{tp}(\tau, \sigma)$ contains the literal $3|x' - x + n$. To see why we conjoin this literal to $\mathsf{tp}(\tau, \sigma)$, note that $3|x' - x + 1$ and $3|x' - x + n$ are equivalent to $x' - x + 1 \equiv_3 0$ and $x' - x + n \equiv_3 0$, respectively, where "$\equiv_3$" denotes congruence modulo 3. So $e \to_\tau^n e'$ implies $e' - e + n \equiv_3 0$, just like $e \to_\pi^n e'$ implies $e' - e + n = 0$ for $\pi := x' - x + 1 \doteq 0$. Moreover, we have $\mathsf{cvp}(\tau, \sigma, \mathbf{x}) = 2|x$, and thus

$$\mathsf{tp}(\tau, \sigma) = n > 0 \wedge 3|x' - x + n \wedge 2|x.$$

We refer to [20] for the straightforward proof of the following theorem:

Theorem 29 (Correctness of tp**).** *The function* tp *as defined above is a transitive projection.*

5 Related Work and Experiments

Related Work. For reasons of space, we only discuss the most important related work and refer to [20] for more details. Most state-of-the-art infinite state model checking algorithms use *interpolation* [8,26,27,29,31,34,35,42]. As discussed in Sect. 1, these techniques are very powerful, but also fragile. In contrast, robustness of TRL depends on the underlying transitive projection. Our implementation for linear integer arithmetic is a variation of the recurrence analysis from [14], which is very robust in our experience. In particular, our recurrence analysis does not require an SMT solver.

Currently, the most powerful model checking algorithm is *Spacer with global guidance* (GSpacer). In GSpacer, interpolation is optional, and thus it is more robust than Spacer (without global guidance) and other interpolation-based techniques. GSpacer is part of the IC3/PDR family of model checking algorithms [6], which differ fundamentally from BMC-based approaches like TRL: The latter unroll the transition relation, whereas GSpacer and other variants of IC3/PDR for infinite state systems prove global properties by combining local

reasoning about a single step with techniques like induction, interpolation, or global guidance.

Many techniques for program verification use *transition invariants* or related concepts like *loop summarization* or *acceleration* [1,5,14,18,19,32,33,37,41]. All of them rely on over- or under-approximations, whereas TRL can use both.

TRL has been inspired by ABMC [19], which is also restricted to under-approximations and thus weaker for proving safety. However, ABMC is currently one of the most powerful techniques for proving unsafety. Other important conceptual differences include TRL's use of: blocking clauses on demand; model based projection, which cannot be used by ABMC, as acceleration yields formulas in theories without quantifier elimination; backtracking, which is useful for proving safety as it keeps the SMT problem small, but it would slow down the discovery of counterexamples in ABMC.

Our tool LoAT also implements ADCL [18], which embeds acceleration into a depth-first search for counterexamples (whereas ABMC and TRL perform breadth-first search). Thus, TRL is conceptually closer to ABMC, and for proving unsafety, ABMC is superior to ADCL (see [19]). However, as witnessed by the results of the annual *Termination Competition* [24], ADCL is currently the most powerful technique for proving non-termination of transition systems (see [17]).

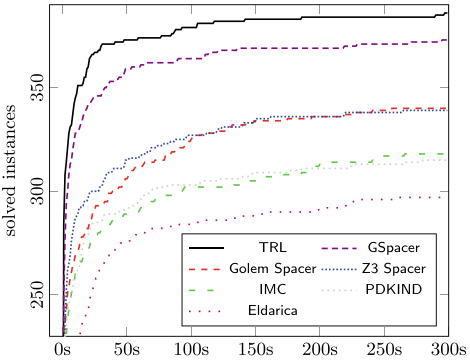
Experiments. We implemented TRL in our tool LoAT [16], which is available on Github [22] and uses the SMT solvers Yices [12] and Z3 [36]. We evaluated it on all linear CHCs from the CHC competitions '23 and '24 [9] (excluding duplicates), resulting in 626 problems from applications like verification of C, Rust, and Java.

We compared 14 techniques of leading CHC solvers:

LoAT's implementations of TRL (LoAT TRL), ABMC [19], and *k-induction* [40].
Z3 4.13.3 [36] Z3's implementations of the Spacer [29] (Z3 Spacer) and GSpacer [31] (Z3 GSpacer) algorithms, and of BMC.
Golem 0.6.2 [4] Golem's implementations of *transition power abstraction* [8], *interpolation based model checking* [34] (Golem IMC), *lazy abstraction with interpolants* [35], *predicate abstraction/CEGAR* [10,25], *property directed k-induction* [27] (Golem PDKIND), Spacer (Golem Spacer), and BMC.
Eldarica 2.1 [26] Eldarica's implementation of predicate abstraction/CEGAR.

We ran our experiments on CLAIX-2023-HPC nodes of the RWTH University High Performance Computing Cluster with a memory limit of 10560 MiB ($\approx$ 11GB) and a timeout of 300 s per example. Here, we report on the results of the seven best configurations, and we refer to [20,21] for more details. Note that our implementation can also prove unsafety, by using the *acceleration technique* from [15] instead of transitive projections, see [20] for details.

	✓	safe		unsafe	
		✓	!	✓	!
LoAT TRL	386	267	24	119	22
Z3 GSpacer	373	287	18	86	4
Golem Spacer	340	231	–	109	–
Z3 Spacer	339	236	–	103	–
Golem IMC	318	215	2	103	2
Golem PDKIND	315	224	2	91	2
Eldarica	298	219	8	79	0



In the table above, the first column marked with ✓ contains the number of solved instances (i.e., all remaining instances could not be solved by the respective configuration), and the columns with **!** show the number of examples that could only be solved by the corresponding configuration. Such a comparison only makes sense if just one implementation of each algorithm is considered, so here we disregarded Z3 Spacer and Golem Spacer (as GSpacer is a variant of Spacer).

The table shows that TRL is highly competitive: Overall, it solves the most instances, and only GSpacer can prove safety more often. More importantly, TRL finds many unique proofs, i.e., it is orthogonal to existing model checking algorithms, and hence it improves the state of the art.

The plot on the right shows how many instances were solved within 300 s. It shows that TRL is also highly competitive in terms of runtime.

6 Conclusion

We presented *Transitive Relation Learning (TRL)*, a powerful model checking algorithm for infinite state systems. TRL adds transitive relations to the analyzed system until its *diameter* (the number of steps that is required to cover all reachable states) becomes finite, which facilitates a safety proof. As it does not search for invariants, TRL does not need interpolation, which is in contrast to most state-of-the-art techniques. Nevertheless, our evaluation shows that TRL is highly competitive. Moreover, not using interpolation allows us to avoid the well-known fragility of interpolation-based approaches.

In future work, we plan to support other theories like reals, bitvectors, and arrays, and we will investigate an extension to temporal verification.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: ATVA 2005, pp. 474–488. LNCS 3707 (2005). https://doi.org/10.1007/11562948_35

2. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB). www.SMT-LIB.org (2016)
3. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2
4. Blicha, M., Britikov, K., Sharygina, N.: The Golem Horn solver. In: CAV 2023, pp. 209–223. LNCS 13965 (2023). https://doi.org/10.1007/978-3-031-37703-7_10
5. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: CAV 2010, pp. 227–242. LNCS 6174 (2010). https://doi.org/10.1007/978-3-642-14295-6_23
6. Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI 2011, pp. 70–87. LNCS 6538 (2011). https://doi.org/10.1007/978-3-642-18275-4_7
7. Bremner, D.: Incremental convex hull algorithms are not output sensitive. Discret. Comput. Geom. **21**(1), 57–68 (1999). https://doi.org/10.1007/PL00009410
8. Britikov, K., Blicha, M., Sharygina, N., Fedyukovich, G.: Reachability analysis for multiloop programs using transition power abstraction. In: FM 2024, pp. 558–576. LNCS 14933 (2024). https://doi.org/10.1007/978-3-031-71162-6_29
9. CHC Competition, https://chc-comp.github.io
10. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV 2000, pp. 154–169. LNCS 1855 (2000). https://doi.org/10.1007/10722167_15
11. Cooper, D.C.: Theorem proving in arithmetic without multiplication. Mach. Intell. **7**, 91–99 (1972)
12. Dutertre, B.: Yices 2.2. In: CAV 2014, pp. 737–744. LNCS 8559 (2014). https://doi.org/10.1007/978-3-319-08867-9_49
13. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press (1972)
14. Farzan, A., Kincaid, Z.: Compositional recurrence analysis. In: FMCAD 2015, pp. 57–64 (2015). https://doi.org/10.1109/FMCAD.2015.7542253
15. Frohn, F.: A calculus for modular loop acceleration. In: TACAS 2020, pp. 58–76. LNCS 12078 (2020). https://doi.org/10.1007/978-3-030-45190-5_4
16. Frohn, F., Giesl, J.: Proving non-termination and lower runtime bounds with LoAT (system description). In: IJCAR 2022, pp. 712–722. LNCS 13385 (2022). https://doi.org/10.1007/978-3-031-10769-6_41
17. Frohn, F., Giesl, J.: Proving non-termination by acceleration driven clause learning. In: CADE 2023, pp. 220–233. LNCS 14132 (2023). https://doi.org/10.1007/978-3-031-38499-8_13
18. Frohn, F., Giesl, J.: ADCL: acceleration driven clause learning for constrained Horn clauses. In: SAS 2023, pp. 259–285. LNCS 14284 (2023). https://doi.org/10.1007/978-3-031-44245-2_13
19. Frohn, F., Giesl, J.: Integrating loop acceleration into bounded model checking. In: FM 2024, pp. 73–91. LNCS 14933 (2024). https://doi.org/10.1007/978-3-031-71162-6_4
20. Frohn, F., Giesl, J.: Infinite state model checking by learning transitive relations. CoRR **abs/2502.04761** (2025). https://doi.org/10.48550/arXiv.2502.04761
21. Frohn, F., Giesl, J.: Evaluation of "Infinite State Model Checking by Learning Transitive Relations" (2025), https://loat-developers.github.io/trl-eval/
22. Frohn, F., Giesl, J.: LoAT website, https://loat-developers.github.io/LoAT/
23. Genov, B.: The convex hull problem in practice: improving the running time of the double description method. Ph.D. thesis, Bremen University, Germany (2015), https://media.suub.uni-bremen.de/handle/elib/833

24. Giesl, J., Rubio, A., Sternagel, C., Waldmann, J., Yamada, A.: The termination and complexity competition. In: TACAS 2019, pp. 156–166. LNCS 11429 (2019). https://doi.org/10.1007/978-3-030-17502-3_10
25. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV 1997, pp. 72–83. LNCS 1254 (1997). https://doi.org/10.1007/3-540-63166-6_10
26. Hojjat, H., Rümmer, P.: The Eldarica Horn solver. In: FMCAD 2018, pp. 1–7 (2018). https://doi.org/10.23919/FMCAD.2018.8603013
27. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: FMCAD 2016, pp. 85–92 (2016). https://doi.org/10.1109/FMCAD.2016.7886665
28. Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.W.: Compositional recurrence analysis revisited. In: PLDI 2017, pp. 248–262 (2017). https://doi.org/10.1145/3062341.3062373
29. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods Syst. Des. **48**(3), 175–205 (2016). https://doi.org/10.1007/s10703-016-0249-4
30. Konečný, F.: PTIME computation of transitive closures of octagonal relations. In: TACAS 2016, pp. 645–661. LNCS 9636 (2016). https://doi.org/10.1007/978-3-662-49674-9_42
31. Krishnan, H.G.V., Chen, Y., Shoham, S., Gurfinkel, A.: Global guidance for local generalization in model checking. In: CAV 2020, pp. 101–125. LNCS 12225 (2020). https://doi.org/10.1007/978-3-030-53291-8_7
32. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using state and transition invariants. Formal Methods Syst. Des. **42**(3), 221–261 (2013). https://doi.org/10.1007/S10703-012-0176-Y
33. Kroening, D., Lewis, M., Weissenbacher, G.: Proving safety with trace automata and bounded model checking. In: FM 2015, pp. 325–341. LNCS 9109 (2015). https://doi.org/10.1007/978-3-319-19249-9_21
34. McMillan, K.L.: Interpolation and SAT-based model checking. In: CAV 2003, pp. 1–13. LNCS 2725 (2003). https://doi.org/10.1007/978-3-540-45069-6_1
35. McMillan, K.L.: Lazy abstraction with interpolants. In: CAV 2006, pp. 123–136. LNCS 4144 (2006). https://doi.org/10.1007/11817963_14
36. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008, pp. 337–340. LNCS 4963 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
37. Pimpalkhare, N., Kincaid, Z.: Semi-linear VASR for over-approximate semi-linear transition system reachability. In: RP 2024, pp. 154–166. LNCS 15050 (2024). https://doi.org/10.1007/978-3-031-72621-7_11
38. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS 2004, pp. 32–41 (2004). https://doi.org/10.1109/LICS.2004.1319598
39. Promies, V., Ábrahám, E.: A divide-and-conquer approach to variable elimination in linear real arithmetic. In: FM 2024, pp. 131–148. LNCS 14933 (2024). https://doi.org/10.1007/978-3-031-71162-6_7
40. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD 2000, pp. 108–125. LNCS 1954 (2000). https://doi.org/10.1007/3-540-40922-X_8
41. Silverman, J., Kincaid, Z.: Loop summarization with rational vector addition systems. In: CAV 2019, pp. 97–115. LNCS 11562 (2019). https://doi.org/10.1007/978-3-030-25543-5_7
42. Solanki, M., Chatterjee, P., Lal, A., Roy, S.: Accelerated bounded model checking using interpolation based summaries. In: TACAS 2024, pp. 155–174. LNCS 14571 (2024). https://doi.org/10.1007/978-3-031-57249-4_8

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.

