Accelerated Bounded Model Checking*

Florian Frohn and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract. *Bounded Model Checking* (BMC) is a powerful technique for proving reachability of error states, i.e., unsafety. However, finding *deep counterexamples* that require a large bound is challenging for BMC. On the other hand, *acceleration techniques* compute "shortcuts" that "compress" many execution steps into a single one. In this paper, we tightly integrate acceleration techniques into SMT-based bounded model checking. By adding suitable "shortcuts" to the SMT-problem on the fly, our approach can quickly detect deep counterexamples, even when only using small bounds. Moreover, using so-called *blocking clauses*, our approach can prove safety of examples where BMC diverges. An empirical comparison with other state-of-the-art techniques shows that our approach is highly competitive for proving unsafety, and orthogonal to existing techniques for proving safety.

1 Introduction

Bounded Model Checking (BMC) is a powerful technique for disproving safety properties of software. However, as it uses breadth-first search to find counterexamples, the search space grows exponentially w.r.t. the *bound*, i.e., the limit on the length of potential counterexamples. Thus, finding *deep counterexamples* that require large bounds is challenging for BMC. On the other hand, *acceleration techniques* can compute a first-order formula that characterizes the transitive closure of the transition relation induced by a loop. Intuitively, such a formula corresponds to a "shortcut" that "compresses" many execution steps into a single one. In this paper, we consider relations defined by quantifier-free first-order formulas over some background theory like non-linear integer arithmetic and two disjoint vectors of variables $\vec{x}$ and $\vec{x}'$, called the *pre-* and *post-variables*. Such *transition formulas* can easily represent, e.g., *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 programs written in more expressive languages.[2]

* funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2)

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

[2] Essentially, the same formalism is, e.g., used in the category "Linear Real Arithmetic - Transition Systems" of the annual CHC-competition [8], where linear CHCs with just a single uninterpreted predicate are considered, so all information about the control flow is encoded in the constraints.

Example 1. Consider the transition formula $\tau := \tau_{x<100} \vee \tau_{x=100}$ where

$$\tau_{x<100} := x < 100 \wedge x' = x + 1 \wedge y' = y \quad \text{and}$$
$$\tau_{x=100} := x = 100 \wedge x' = 0 \wedge y' = y + 1.$$

```
while (x <= 100) {
  while (x < 100) x++;
  x = 0, y++;
}
```

Listing 1: Implementation of τ

It defines a relation $\to_\tau$ on $\mathbb{Z} \times \mathbb{Z}$ by relating the pre-variables x and y with the post-variables x' and y'. So for all $c_x, c_y, c'_x, c'_y \in \mathbb{Z}$ we have $(c_x, c_y) \to_\tau (c'_x, c'_y)$ iff $[x/c_x, y/c_y, x'/c'_x, y'/c'_y]$ is a model of τ. In other words, we have $(c_x, c_y) \to^*_\tau (c'_x, c'_y)$ if and only if a state with $x = c'_x \wedge y = c'_y$ is reachable from a state with $x = c_x \wedge y = c_y$ in Listing 1. Assume that we want to prove that an *error state* which satisfies $\psi_{\mathsf{err}} := y \geq 100$ is reachable from an *initial state* which satisfies $\psi_{\mathsf{init}} := x \leq 0 \wedge y \leq 0$. To do so, BMC has to unroll τ 10100 times.

Our new technique called *Accelerated* BMC (ABMC) accelerates τ, resulting in the "shortcut"

$$n > 0 \wedge x + n \leq 100 \wedge x' = x + n \wedge y' = y, \qquad (\tau_{\mathsf{i}}^+)$$

meaning that we have $(c_x, x_y) \to^+_\tau (c'_x, c'_y)$ if $\tau_{\mathsf{i}}^+[x/c_x, y/c_y, x'/c'_x, y'/c'_y]$ is satisfiable. So τ_{i}^+ can simulate arbitrarily many steps with τ in a single step, as long as x does not exceed 100. Here, acceleration was applied to $\tau_{x<100}$, i.e., the projection of τ to the case $x < 100$, which corresponds to the inner loop of Listing 1. We also call such projections *transitions*. Later, ABMC also accelerates the outer loop (consisting of $\tau_{x=100}$, $\tau_{x<100}$, and τ_{i}^+), resulting in

$$n > 0 \wedge x = 100 \wedge 1 < x' \leq 100 \wedge y' = y + n. \qquad (\tau_{\mathsf{o}}^+)$$

For technical reasons, our algorithm accelerates $[\tau_{x=100}, \tau_{x<100}, \tau_{\mathsf{i}}^+]$ instead of just $[\tau_{x=100}, \tau_{\mathsf{i}}^+]$, so that τ_{o}^+ requires $1 < x'$, i.e., it only covers cases where $\tau_{x<100}$ is applied at least twice after $\tau_{x=100}$. Details will be clarified in Sect. 3.2, see in particular Fig. 1. Using these shortcuts, ABMC can prove unsafety with bound 7.

While our main goal is to improve BMC's capability to find deep counterexamples, the following straightforward observations can be used to *block* certain parts of the transition relation in ABMC:

1. After accelerating a sequence of transitions, the resulting accelerated transition should be preferred over that sequence of transitions.
2. If an accelerated transition has been used, then the corresponding sequence of transitions should not be used immediately afterwards.

Both observations are justified by the fact that an accelerated transition describes the transitive closure of the transition relation induced by the corresponding sequence of transitions. Due to its ability to block parts of the transition relation, ABMC is able to prove safety in cases where BMC would unroll the transition relation indefinitely.

Comparison with ADCL The idea of using acceleration to detect deep counterexamples is not new. In particular, we recently introduced *Acceleration Driven Clause Learning* [16, 17], a calculus that uses depth-first search and acceleration to find counterexamples. So one major difference between ABMC and ADCL is that ABMC performs breadth-first search, whereas ADCL performs depth-first search. Thus, ADCL requires a mechanism for backtracking to avoid getting stuck. To this end, it relies on a notion of *redundancy*, which is difficult to automate. Thus, in practice, approximations are used [17, Sect. 4]. However, even with a complete redundancy check, ADCL might get stuck in a safe part of the search space [17, Thm. 18]. ABMC does not suffer from such deficits.

Like ADCL, ABMC also tries to avoid redundant work (see Sections 3.3 and 4). However, doing so is crucial for ADCL due to its depth-first strategy, whereas it is a mere optimization for ABMC.

On the other hand, ADCL has successfully been adapted for proving non-termination [16], and it is unclear whether a corresponding adaption of ABMC would be competitive. Furthermore, ADCL applies acceleration in a very systematic way, whereas ABMC decides whether to apply acceleration or not based on the model that is found by the underlying SMT solver. Therefore, ADCL is advantageous for examples with deeply nested loops, where ABMC may require many steps until the SMT solver yields models that allow for accelerating the nested loops one after the other. Thus, both techniques are orthogonal. See Sect. 6 for an experimental comparison of ADCL with our novel ABMC technique.

Outline After introducing preliminaries in Sect. 2, we show how to use acceleration in order to improve the BMC algorithm to ABMC in Sect. 3. Sect. 4 refines ABMC by integrating blocking clauses. In Sect. 5, we discuss related work, and in Sect. 6, we evaluate our implementation of ABMC in our tool LoAT.

2 Preliminaries

We assume familiarity with basics from many-sorted first-order logic. Without loss of generality, we assume that all formulas are in negation normal form (NNF). $\mathcal{V}$ is a countably infinite set of variables and $\mathcal{A}$ is a first-order theory over a k-sorted signature Σ with carrier $\mathcal{C} = (\mathcal{C}_1, \dots, \mathcal{C}_k)$. 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. We let $\top$ and $\bot$ stand for "true" and "false", respectively.

Given $\psi \in \mathsf{QF}(\Sigma)$ with $\mathcal{V}(\psi) = \vec{y}$, we say that ψ is $\mathcal{A}$-*valid* (written $\models_\mathcal{A} \psi$) if every model of $\mathcal{A}$ satisfies the universal closure $\forall \vec{y}.\, \psi$ of ψ. Moreover, $\sigma : \mathcal{V}(\psi) \to \mathcal{C}$ is an $\mathcal{A}$-*model* of ψ (written $\sigma \models_\mathcal{A} \psi$) if $\models_\mathcal{A} \sigma(\psi)$, where $\sigma(\psi)$ results from ψ by instantiating all variables according to σ. If ψ has an $\mathcal{A}$-model, then ψ is $\mathcal{A}$-*satisfiable*. 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", and "satisfiable". We assume that $\mathcal{A}$ is complete, i.e., we either have $\models \psi$ or $\models \neg\psi$ for every closed formula over Σ.

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

Let $d \in \mathbb{N}$ be fixed, and let $\vec{x}, \vec{x}' \in \mathcal{V}^d$ be disjoint vectors of pairwise different variables, called the *pre-* and *post-variables*. Each $\tau \in \mathsf{QF}(\Sigma)$ induces a *transition relation* $\to_\tau$ on $\mathcal{C}^d$ where $\vec{s} \to_\tau \vec{t}$ iff $\tau[\vec{x}/\vec{s}, \vec{x}'/\vec{t}]$ is satisfiable. Here, $[\vec{x}/\vec{s}, \vec{x}'/\vec{t}]$ denotes the substitution θ with $\theta(x_i) = s_i$ and $\theta(x_i') = t_i$ for all $1 \leq i \leq d$. We refer to elements of $\mathsf{QF}(\Sigma)$ as *transition formulas* whenever we are interested in their induced transition relation. Moreover, we also refer to *conjunctive* transition formulas (i.e., elements of $\mathsf{QF}_\wedge(\Sigma)$) as *transitions*. 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 $\mathcal{V}(\psi_{\mathsf{init}}) \cup \mathcal{V}(\psi_{\mathsf{err}}) \subseteq \vec{x}$. It is *unsafe* if there are $\vec{s}, \vec{t} \in \mathcal{C}^d$ such that $[\vec{x}/\vec{s}] \models \psi_{\mathsf{init}}$, $\vec{s} \to_\tau^* \vec{t}$, and $[\vec{x}/\vec{t}] \models \psi_{\mathsf{err}}$.

The *composition* of τ and τ' is $\circledcirc(\tau, \tau') := \tau[\vec{x}'/\vec{x}''] \wedge \tau'[\vec{x}/\vec{x}'']$ where $\vec{x}'' \in \mathcal{V}^d$ is fresh. Here, we assume $\mathcal{V}(\tau) \cap \mathcal{V}(\tau') \subseteq \vec{x} \cup \vec{x}'$ (which can be ensured by renaming other variables correspondingly). So $\to_{\circledcirc(\tau,\tau')} = \to_\tau \circ \to_{\tau'}$ (where $\circ$ denotes relational composition). For finite sequences of transition formulas we define $\circledcirc([]) := (\vec{x} = \vec{x}')$ (i.e., $\to_{\circledcirc([])}$ is the identity relation) and $\circledcirc(\tau :: \vec{\tau}) := \circledcirc(\tau, \circledcirc(\vec{\tau}))$. We abbreviate $\to_{\circledcirc(\vec{\tau})}$ by $\to_{\vec{\tau}}$.

Acceleration techniques compute the transitive closure of relations. In the following definition, we only consider relations defined by conjunctive formulas, since many existing acceleration techniques do not support disjunctions [6], or have to resort to approximations in the presence of disjunctions [12]. So the restriction to conjunctive formulas ensures that our approach works with arbitrary acceleration techniques.

Definition 2 (Acceleration). *An* acceleration technique *is a function* $\mathsf{accel} : \mathsf{QF}_\wedge(\Sigma) \to \mathsf{QF}_\wedge(\Sigma')$ *such that* $\to_{\mathsf{accel}(\tau)} \subseteq \to_\tau^+$*, where* Σ' *is the signature of a first-order theory* $\mathcal{A}'$*.*

We abbreviate $\mathsf{accel}(\circledcirc(\vec{\tau}))$ by $\mathsf{accel}(\vec{\tau})$. So as we aim at finding counterexamples, we allow under-approximating acceleration techniques, i.e., we do not require $\to_{\mathsf{accel}(\tau)} = \to_\tau^+$. Def. 2 allows $\mathcal{A}' \neq \mathcal{A}$, as most theories are not "closed under acceleration". For example, accelerating the following Presburger formula on the left may yield the non-linear formula on the right:

$$x' = x + y \wedge y' = y \qquad\qquad n > 0 \wedge x' = x + n \cdot y \wedge y' = y.$$

3 From BMC to ABMC

In this section, we introduce accelerated bounded model checking. To this end, we first recapitulate bounded model checking in Sect. 3.1. Then we present ABMC in Sect. 3.2. To implement ABMC efficiently, heuristics to decide when to perform acceleration are needed. Thus, we present such a heuristic in Sect. 3.3.

3.1 Bounded Model Checking

Alg. 1 shows how to implement BMC on top of an incremental SMT solver. In Line 1, the description of the initial states is added to the SMT problem.

Algorithm 1: BMC

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

1 $b \leftarrow 0$; $\mathsf{add}(\mu_b(\psi_{\mathsf{init}}))$
2 **while** $\top$ **do**
3 $\quad \mathsf{push}()$; $\mathsf{add}(\mu_b(\psi_{\mathsf{err}}))$
4 $\quad$ **if** $\mathsf{check\_sat}()$ **do return** unsafe **else** $\mathsf{pop}()$; $\mathsf{add}(\mu_b(\tau))$
5 $\quad$ **if** $\neg\mathsf{check\_sat}()$ **do return** safe **else** $b \leftarrow b + 1$

Here and in the following, for all $i \in \mathbb{N}$ we define $\mu_i(x) := x^{(i)}$ if $x \in \mathcal{V} \setminus \vec{x}'$, and $\mu_i(x') = x^{(i+1)}$ if $x' \in \vec{x}'$. So in particular, we have $\mu_i(\vec{x}) = \vec{x}^{(i)}$ and $\mu_i(\vec{x}') = \vec{x}^{(i+1)}$, where we assume that $\vec{x}^{(0)}, \vec{x}^{(1)}, \ldots \in \mathcal{V}^d$ are disjoint vectors of pairwise different variables. In the loop, we set a backtracking point with the "$\mathsf{push}()$" command and add a suitably variable-renamed version of the description of the error states to the SMT problem in Line 3. Then we check for satisfiability to see if an error state is reachable with the current bound in Line 4. If this is not the case, the description of the error states is removed from the SMT problem with the "$\mathsf{pop}()$" command that deletes all formulas from the SMT problem that have been added since the last backtracking point. Then a variable-renamed version of the transition formula τ is added to the solver. If doing so results in an unsatisfiable SMT problem in Line 5, then the whole search space has been exhausted, i.e., the problem is safe. Otherwise, we enter the next iteration.

Example 3 (BMC). For the first 100 iterations of Alg. 1, all models found in Line 5 satisfy the 1^{st} disjunct $\mu_b(\tau_{x<100})$ of $\mu_b(\tau)$. Then we may have $x^{(100)} = 100$, so that the 2^{nd} disjunct $\mu_b(\tau_{x=100})$ of $\mu_b(\tau)$ applies once and we get $y^{(101)} = y^{(100)}+1$. After another 100 iterations, the 2^{nd} disjunct $\mu_b(\tau_{x=100})$ may apply again, and so on. After 100 applications of the 2^{nd} disjunct (and thus a total of 10100 steps), there is a model with $y^{(10100)} = 100$, so that unsafety can be proven.

3.2 Accelerated Bounded Model Checking

To incorporate acceleration into BMC, we have to bridge the gap between (disjunctive) transition formulas and acceleration techniques, which require conjunctive transition formulas. To this end, we use *syntactic implicants*.

Definition 4 (Syntactic Implicant Projection [17]). *Let $\tau \in \mathsf{QF}(\Sigma)$ be in NNF and assume $\sigma \models \tau$. We define the* syntactic implicants $\mathsf{sip}(\tau)$ *of τ as follows:*

$$\mathsf{sip}(\tau, \sigma) := \bigwedge \{\lambda \mid \lambda \text{ is a literal of } \tau,\ \sigma \models \lambda\} \qquad \mathsf{sip}(\tau) := \{\mathsf{sip}(\tau, \sigma) \mid \sigma \models \tau\}$$

Since τ is in NNF, $\mathsf{sip}(\tau, \sigma)$ implies τ, and it is easy to see that $\tau \equiv \bigvee \mathsf{sip}(\tau)$. Whenever the call to the SMT solver in Line 5 of Alg. 1 yields sat, the resulting model gives rise to a sequence of syntactic implicants, called the *trace*. To define the trace formally, note that when we integrate acceleration into BMC, we may not only add τ to the SMT formula as in Line 4, but also *learned transitions* that result from acceleration. Thus, the following definition allows for changing the transition formula. In the sequel, $\circ$ also denotes composition of substitutions, i.e., $\theta' \circ \theta := [x/\theta'(\theta(x)) \mid x \in \mathrm{dom}(\theta') \cup \mathrm{dom}(\theta)]$.

Definition 5 (Trace). *Let $[\tau_i]_{i=0}^{b-1}$ be a sequence of transition formulas and let σ be a model of $\bigwedge_{i=0}^{b-1} \mu_i(\tau_i)$. Then the* trace induced by σ *is*

$$\mathsf{trace}_b(\sigma, [\tau_i]_{i=0}^{b-1}) := [\mathsf{sip}(\tau_i, \sigma \circ \mu_i)]_{i=0}^{b-1}.$$

We write $\mathsf{trace}_b(\sigma)$ instead of $\mathsf{trace}_b(\sigma, [\tau_i]_{i=0}^{b-1})$ if $[\tau_i]_{i=0}^{b-1}$ is clear from the context.

So each model σ of $\bigwedge_{i=0}^{b-1} \mu_i(\tau_i)$ corresponds to a sequence of steps with the relations $\to_{\tau_0}, \to_{\tau_1}, \ldots, \to_{\tau_{b-1}}$, and the trace induced by σ contains the syntactic implicant of each τ_i that was used in this sequence.

Example 6 (Trace). Reconsider Ex. 3. After two iterations of the loop of Alg. 1, the SMT problem consists of the following formulas:

$$x^{(0)} \leq 0 \wedge y^{(0)} \leq 0 \qquad (\psi_{\mathsf{init}})$$
$$(x^{(0)} < 100 \wedge x^{(1)} = x^{(0)} + 1 \wedge y^{(1)} = y^{(0)}) \vee (x^{(0)} = 100 \wedge x^{(1)} = 0 \wedge y^{(1)} = y^{(0)} + 1) \quad (\tau)$$
$$(x^{(1)} < 100 \wedge x^{(2)} = x^{(1)} + 1 \wedge y^{(2)} = y^{(1)}) \vee (x^{(1)} = 100 \wedge x^{(2)} = 0 \wedge y^{(2)} = y^{(1)} + 1) \quad (\tau)$$

With $\sigma = [x^{(i)}/i, y^{(i)}/0 \mid 0 \leq i \leq 2]$, we get $\mathsf{trace}_2(\sigma) = [\tau_{x<100}, \tau_{x<100}]$, as:

$$\mathsf{sip}(\tau, \sigma \circ \mu_0) = \mathsf{sip}(\tau, [x/0, y/0, x'/1, y'/0]) = \tau_{x<100}$$
$$\mathsf{sip}(\tau, \sigma \circ \mu_1) = \mathsf{sip}(\tau, [x/1, y/0, x'/2, y'/0]) = \tau_{x<100}$$

To detect situations where applying acceleration techniques pays off, we need to distinguish traces that contain loops from non-looping ones. Since transition formulas are unstructured, the usual techniques for detecting loops (based on, e.g., program syntax or control flow graphs) do not apply in our setting. Instead, we rely on the *dependency graph* of the transition formula.

Definition 7 (Dependency Graph). *Let τ be a transition formula. Its* dependency graph $\mathcal{DG} = (V, E)$ *is a directed graph whose vertices $V := \mathsf{sip}(\tau)$ are τ's syntactic implicants, and $\tau_1 \to \tau_2 \in E$ if $\circledcirc(\tau_1, \tau_2)$ is satisfiable. We say that $\vec{\tau} \in \mathsf{sip}(\tau)^c$ is* $\mathcal{DG}$-cyclic *if $c > 0$ and $(\tau_1 \to \tau_2), \ldots, (\tau_{c-1} \to \tau_c), (\tau_c \to \tau_1) \in E$.*

So intuitively, the syntactic implicants correspond to the different cases of $\to_\tau$, and τ's dependency graph corresponds to the control flow graph of $\to_\tau$.

Example 8 (Dependency Graph). For Ex. 1, the dependency graph is:

However, as the size of $\mathsf{sip}(\tau)$ is worst-case exponential in the number of disjunctions in τ, we do not compute τ's dependency graph eagerly. Instead, ABMC maintains an under-approximation, i.e., a subgraph $\mathcal{G}$ of the dependency graph, which is extended whenever two transitions that are not yet connected by an edge occur consecutively on the trace. As soon as a $\mathcal{G}$-cyclic suffix $\vec{\tau}^{\circlearrowleft}$ is detected on the trace, we may accelerate it. Therefore, the trace may also contain the learned transition $\mathsf{accel}(\vec{\tau}^{\circlearrowleft})$ in subsequent iterations. Hence, to detect cyclic suffixes that contain learned transitions, they have to be represented in $\mathcal{G}$ as well. Thus, $\mathcal{G}$ is

Algorithm 2: ABMC

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

1 $b \leftarrow 0$; $V \leftarrow \varnothing$; $E \leftarrow \varnothing$; $\mathsf{add}(\mu_b(\psi_{\mathsf{init}}))$
2 **if** $\neg\mathsf{check\_sat}()$ **do return** safe **else** $\sigma \leftarrow \mathsf{get\_model}()$
3 **while** $\top$ **do**
4 $\quad \mathsf{push}()$; $\mathsf{add}(\mu_b(\psi_{\mathsf{err}}))$
5 $\quad$ **if** $\mathsf{check\_sat}()$ **do return** unsafe **else** $\mathsf{pop}()$
6 $\quad \vec{\tau} \leftarrow \mathsf{trace}_b(\sigma)$; $V \leftarrow V \cup \vec{\tau}$; $E \leftarrow E \cup \{(\tau_1, \tau_2) \mid [\tau_1, \tau_2] \text{ is an infix of } \vec{\tau}\}$
7 $\quad$ **if** $\vec{\tau} = \vec{\pi} :: \vec{\pi}^{\circlearrowleft} \wedge \vec{\pi}^{\circlearrowleft}$ *is cyclic* $\wedge\, \mathsf{should\_accel}(\vec{\pi}^{\circlearrowleft})$ **do** $\mathsf{add}(\mu_b(\tau \vee \mathsf{accel}(\vec{\pi}^{\circlearrowleft})))$
8 $\quad$ **else** $\mathsf{add}(\mu_b(\tau))$
9 $\quad$ **if** $\neg\mathsf{check\_sat}()$ **do return** safe **else** $\sigma \leftarrow \mathsf{get\_model}()$; $b \leftarrow b + 1$

in fact a subgraph of the dependency graph of $\tau \vee \bigvee \mathcal{L}$, where $\mathcal{L}$ is the set of all transitions that have been learned so far.

This gives rise to the ABMC algorithm, which is shown in Alg. 2. Here, we just write "cyclic" instead of (V, E)-cyclic. The difference to Alg. 1 can be seen in Lines 6 and 7. In Line 6, the trace is constructed from the current model. Then, the approximation of the dependency graph is refined such that it contains vertices for all elements of the trace, and edges for all transitions that occur consecutively on the trace. In Line 7, the trace may get accelerated if it has a cyclic suffix, provided that the call to $\mathsf{should\_accel}$ (which will be discussed in more detail in Sect. 3.3) returns $\top$. In this way, in the next iteration the SMT solver can choose a model that satisfies $\mathsf{accel}(\vec{\pi}^{\circlearrowleft})$ and thus simulates several instead of just one $\to_\tau$-step. Note, however, that we do *not* update τ with $\tau \vee \mathsf{accel}(\vec{\pi}^{\circlearrowleft})$. So in every iteration, at most one learned transition is added to the SMT problem. In this way, we avoid blowing up τ unnecessarily.

Fig. 1 shows a run of Alg. 2 on Ex. 1, where the formulas that are added to the SMT problem are highlighted in gray, and $x^{(i)} \mapsto c$ abbreviates $\sigma(x^{(i)}) = c$. For simplicity, we assume that $\mathsf{should\_accel}$ always returns $\top$, and the model σ is only extended in each step, i.e., $\sigma(x^{(i)})$ and $\sigma(y^{(i)})$ remain unchanged for all $0 \leq i < b$. In general, the SMT solver can choose different values for $\sigma(x^{(i)})$ and $\sigma(y^{(i)})$ in every iteration. On the right, we show the current bound b, as well as the formulas that give rise to the highlighted formulas on the left when renaming their variables suitably with μ_b. Initially, the approximation $\mathcal{G} = (V, E)$ of the dependency graph is empty. When $b = 2$, the trace is $[\tau_{x<100}, \tau_{x<100}]$, and the corresponding edge is added to $\mathcal{G}$. Thus, the trace has the cyclic suffix $\tau_{x<100}$ and we accelerate it, resulting in τ_{i}^+, which is added to the SMT problem. Then we obtain the trace $[\tau_{x<100}, \tau_{x<100}, \tau_{\mathsf{i}}^+]$, and the edge $\tau_{x<100} \to \tau_{\mathsf{i}}^+$ is added to $\mathcal{G}$. Note that Alg. 2 does not enforce the use of τ_{i}^+, so τ might still be unrolled thousands of times instead, depending on the models found by the SMT solver. We will address this issue in Sect. 4.

Next, $\tau_{x=100}$ already applies with $b = 4$ (whereas it only applied with $b = 100$ in Ex. 3). So the trace is $[\tau_{x<100}, \tau_{x<100}, \tau_{\mathsf{i}}^+, \tau_{x=100}]$, and the edge $\tau_{\mathsf{i}}^+ \to \tau_{x=100}$ is added to $\mathcal{G}$. Then we obtain the trace $[\tau_{x<100}, \tau_{x<100}, \tau_{\mathsf{i}}^+, \tau_{x=100}, \tau_{x<100}]$, and add $\tau_{x=100} \to \tau_{x<100}$ to $\mathcal{G}$. Since the suffix $\tau_{x<100}$ is again cyclic, we accelerate it and add τ_{i}^+ to the SMT problem. After one more step, the trace

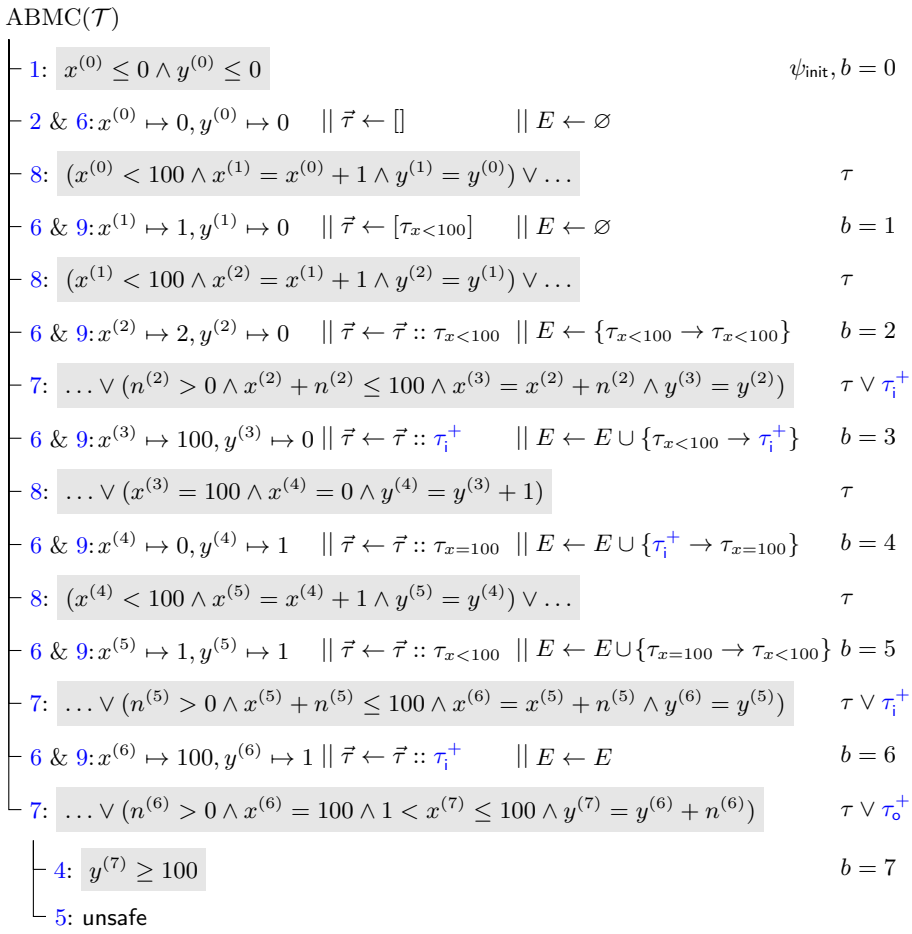


Fig. 1: Running ABMC on Ex. 1

$[\tau_{x<100}, \tau_{x<100}, \tau_{\mathsf{i}}^{+}, \tau_{x=100}, \tau_{x<100}, \tau_{\mathsf{i}}^{+}]$ has the cyclic suffix $[\tau_{x=100}, \tau_{x<100}, \tau_{\mathsf{i}}^{+}]$. Accelerating it yields τ_{o}^{+}, which is added to the SMT problem. Afterwards, unsafety can be proven with $b = 7$.

Since using acceleration is just a heuristic to speed up BMC, all basic properties of BMC immediately carry over to ABMC.

Theorem 9 (Properties of ABMC). *ABMC is*
Refutationally Complete: *If $\mathcal{T}$ is unsafe, then* ABMC($\mathcal{T}$) *returns* unsafe.
Sound: *If* ABMC($\mathcal{T}$) *returns* (un)safe, *then $\mathcal{T}$ is (un)safe.*
Non-Terminating: *If $\mathcal{T}$ is safe, then* ABMC($\mathcal{T}$) *may not terminate.*

3.3 Fine Tuning Acceleration

We now turn our attention to should_accel, our heuristic for applying acceleration. To explain the intuition of our heuristic, we assume that acceleration does not

approximate and thus $\to_{\mathsf{accel}(\vec{\tau})} = \to_{\vec{\tau}}^+$, but in our implementation, we also use it if $\to_{\mathsf{accel}(\vec{\tau})} \subset \to_{\vec{\tau}}^+$. Doing so is uncritical for correctness, as using acceleration is *always* sound. First, acceleration should be applied to cyclic suffixes consisting of a single *original* (i.e., non-learned) transition.

Requirement 1. $\mathsf{should\_accel}([\pi]) = \top$ *if* $\pi \in \mathsf{sip}(\tau)$.

However, applying acceleration to a single learned transition is pointless, as

$$\to_{\mathsf{accel}(\mathsf{accel}(\tau))} = \to_{\mathsf{accel}(\tau)}^+ = (\to_\tau^+)^+ = \to_\tau^+ = \to_{\mathsf{accel}(\tau)}.$$

Requirement 2. $\mathsf{should\_accel}([\pi]) = \bot$ *if* $\pi \notin \mathsf{sip}(\tau)$.

Next, for every cyclic sequence $\vec{\pi}$, we have

$$\begin{aligned}\to_{\mathsf{accel}(\vec{\pi}::\mathsf{accel}(\vec{\pi}))} = \to_{\vec{\pi}::\mathsf{accel}(\vec{\pi})}^+ = (\to_{\vec{\pi}} \circ \to_{\mathsf{accel}(\vec{\pi})})^+ = (\to_{\vec{\pi}} \circ \to_{\vec{\pi}}^+)^+ \\ = \to_{\vec{\pi}} \circ \to_{\vec{\pi}}^+ = \to_{\vec{\pi}} \circ \to_{\mathsf{accel}(\vec{\pi})} = \to_{\vec{\pi}::\mathsf{accel}(\vec{\pi})},\end{aligned}$$

and thus accelerating $\vec{\pi} :: \mathsf{accel}(\vec{\pi})$ is pointless, too. Hence, we obtain:

Requirement 3. $\mathsf{should\_accel}(\vec{\pi} :: \mathsf{accel}(\vec{\pi})) = \bot$.

However, Req. 3 is too specific, as it, e.g., does not prevent acceleration of other sequences $\vec{\pi}_2 :: \mathsf{accel}(\vec{\pi}) :: \vec{\pi}_1$ where $\vec{\pi} = \vec{\pi}_1 :: \vec{\pi}_2$. For such sequences, we have

$$\to_{\vec{\pi}_2::\mathsf{accel}(\vec{\pi})::\vec{\pi}_1}^2 = \to_{\vec{\pi}_2::\mathsf{accel}(\vec{\pi})::\vec{\pi}::\mathsf{accel}(\vec{\pi})::\vec{\pi}_1} \subseteq \to_{\vec{\pi}_2::\mathsf{accel}(\vec{\pi})::\vec{\pi}_1}$$

and thus $\to_{\mathsf{accel}(\vec{\pi}_2::\mathsf{accel}(\vec{\pi})::\vec{\pi}_1)} = \to_{\vec{\pi}_2::\mathsf{accel}(\vec{\pi})::\vec{\pi}_1}^+ = \to_{\vec{\pi}_2::\mathsf{accel}(\vec{\pi})::\vec{\pi}_1}$, so accelerating them is pointless, too. Thus in general, the problem is that the cyclic suffix of the trace might consist of a cycle $\vec{\pi}$ and $\mathsf{accel}(\vec{\pi})$, but it might not necessarily start with either of them. Hence, we generalize Req. 3 using the notion of *conjugates*.

Definition 10 (Conjugate). *We say that two vectors* $\vec{v}, \vec{w}$ *are* conjugates *(denoted* $\vec{v} \equiv_\circ \vec{w}$*) if* $\vec{v} = \vec{v}_1 :: \vec{v}_2$ *and* $\vec{w} = \vec{v}_2 :: \vec{v}_1$.

So a conjugate of a cycle corresponds to the same cycle with another entry point.

Requirement 4. $\mathsf{should\_accel}(\vec{\pi}') = \bot$ *if* $\vec{\pi}' \equiv_\circ \vec{\pi} :: \mathsf{accel}(\vec{\pi})$ *for some* $\vec{\pi}$.

In general, however, we also want to accelerate cyclic suffixes that contain learned transitions to deal with nested loops, as in the last acceleration step of Fig. 1.

Requirement 5. $\mathsf{should\_accel}(\vec{\pi}') = \top$ *if* $\vec{\pi}' \not\equiv_\circ \vec{\pi} :: \mathsf{accel}(\vec{\pi})$ *for all* $\vec{\pi}$.

Req. 1, 2, 4, and 5 give rise to a complete specification for $\mathsf{should\_accel}$: If the cyclic suffix is a singleton, the decision is made based on Requirements 1 and 2, and otherwise the decision is made based on Requirements 4 and 5.

$$\mathsf{should\_accel}(\vec{\pi}') := (|\vec{\pi}'| = 1 \wedge \vec{\pi}' \in \mathsf{sip}(\tau)) \vee (|\vec{\pi}'| > 1 \wedge \forall \vec{\pi}.\ (\vec{\pi}' \not\equiv_\circ \vec{\pi} :: \mathsf{accel}(\vec{\pi})))$$

However, this specification misses one important case: Recall that the trace was $[\tau_{x<100}, \tau_{x<100}]$ before acceleration was applied for the first time in Fig. 1. While both $[\tau_{x<100}]$ and $[\tau_{x<100}, \tau_{x<100}]$ are cyclic, the latter should not be accelerated, since $\mathsf{accel}([\tau_{x<100}, \tau_{x<100}])$ is a special case of τ_i^+ that only represents an even number of steps with $\tau_{x<100}$. Here, the problem is that the cyclic suffix contains a *square*, i.e., two adjacent repetitions of the same non-empty sub-sequence.

Requirement 6. $\mathsf{should\_accel}(\vec{\pi}) = \bot$ *if* $\vec{\pi}$ *contains a square.*

Thus, we obtain the following specification for $\mathsf{should\_accel}$:

$$\mathsf{should\_accel}(\vec{\pi}') := |\vec{\pi}'| = 1 \wedge \vec{\pi}' \in \mathsf{sip}(\tau) \quad \vee$$
$$|\vec{\pi}'| > 1 \wedge \vec{\pi}' \text{ is square-free} \wedge \forall \vec{\pi}.\ (\vec{\pi}' \not\equiv_\circ \vec{\pi} :: \mathsf{accel}(\vec{\pi}))$$

All properties that are required to implement $\mathsf{should\_accel}$ can easily be checked automatically. To check $\vec{\pi}' \not\equiv_\circ \vec{\pi} :: \mathsf{accel}(\vec{\pi})$, our implementation maintains a map from learned transitions to the corresponding cycles that have been accelerated.

However, to implement Alg. 2, there is one more missing piece: As the choice of the cyclic suffix in Line 7 is non-deterministic, a heuristic for choosing it is required. In our implementation, we choose the *shortest* cyclic suffix such that $\mathsf{should\_accel}$ returns $\top$. The reason is that, as observed in [17], accelerating short cyclic suffixes before longer ones allows for learning more general transitions.

4 Guiding ABMC with Blocking Clauses

As mentioned in Sect. 3.2, Alg. 2 does not enforce the use of learned transitions. Thus, depending on the models found by the SMT solver, ABMC may behave just like BMC. We now improve ABMC by integrating *blocking clauses* that prevent it from unrolling loops instead of using learned transitions. Here, we again assume $\to_{\mathsf{accel}(\vec{\tau})} = \to_{\vec{\tau}}^+$, i.e., that acceleration does not approximate. Otherwise, blocking clauses are unsound (which is taken into account by our implementation).

Blocking clauses exploit the following straightforward observation: If the learned transition $\tau_\ell = \mathsf{accel}(\vec{\pi}^{\circlearrowleft})$ has been added to the SMT problem with bound b and an error state can be reached via a trace with prefix

$$\vec{\pi} = [\tau_0, \ldots, \tau_{b-1}] :: \vec{\pi}^{\circlearrowleft} \qquad \text{or} \qquad \vec{\pi}' = [\tau_0, \ldots, \tau_{b-1}, \tau_\ell] :: \vec{\pi}^{\circlearrowleft},$$

then an error state can also be reached via a trace with the prefix $[\tau_0, \ldots, \tau_{b-1}, \tau_\ell]$, which is not continued with $\vec{\pi}^{\circlearrowleft}$. Thus, we may remove traces of the form $\vec{\pi}$ and $\vec{\pi}'$ from the search space by modifying the SMT problem accordingly.

To do so, we assign a unique identifier to each learned transition, and we introduce a fresh integer-valued variable ℓ which is set to the corresponding identifier whenever a learned transition is used, and to 0, otherwise.

Example 11 (Blocking Clauses). Reconsider Fig. 1 and assume that we modify τ by conjoining $\ell = 0$, and τ_{i}^+ by conjoining $\ell = 1$. Thus, we now have

$$\tau_{x<100} \equiv x < 100 \wedge x' = x + 1 \wedge y' = y \wedge \ell = 0 \qquad \text{and}$$
$$\tau_{\mathsf{i}}^+ \equiv n > 0 \wedge x + n \leq 100 \wedge x' = x + n \wedge y' = y \wedge \ell = 1.$$

When $b = 2$, the trace is $[\tau_{x<100}, \tau_{x<100}]$, and in the next iteration, it may be extended to either $\vec{\pi} = [\tau_{x<100}, \tau_{x<100}, \tau_{x<100}]$ or $\vec{\tau} = [\tau_{x<100}, \tau_{x<100}, \tau_{\mathsf{i}}^+]$. However, as $\to_{\tau_{\mathsf{i}}^+} = \to_{\tau_{x<100}}^+$, we have $\to_{\vec{\pi}} \subseteq \to_{\vec{\tau}}$, so the entire search space can be covered without considering the trace $\vec{\pi}$. Thus, we add the blocking clause

$$\neg \mu_2(\tau_{x<100}) \qquad (\beta_1)$$

Algorithm 3: $\mathsf{ABMC}_{\mathsf{block}}$

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

1 $b \leftarrow 0$; $V \leftarrow \varnothing$; $E \leftarrow \varnothing$; $\mathsf{id} \leftarrow 0$; $\tau \leftarrow \tau \wedge \ell = 0$; $\mathsf{cache} \leftarrow \varnothing$; $\mathsf{add}(\mu_b(\psi_{\mathsf{init}}))$

2 **if** $\neg\mathsf{check\_sat}()$ **do return** safe **else** $\sigma \leftarrow \mathsf{get\_model}()$

3 **while** $\top$ **do**

4 $\quad \mathsf{push}()$; $\mathsf{add}(\mu_b(\psi_{\mathsf{err}}))$

5 $\quad$ **if** $\mathsf{check\_sat}()$ **do return** unsafe **else** $\mathsf{pop}()$

6 $\quad \vec{\tau} \leftarrow \mathsf{trace}_b(\sigma)$; $V \leftarrow V \cup \vec{\tau}$; $E \leftarrow E \cup \{(\tau_1, \tau_2) \mid [\tau_1, \tau_2]$ is an infix of $\vec{\tau}\}$

7 $\quad$ **if** $\vec{\tau} = \vec{\pi} :: \vec{\pi}^{\circlearrowleft} \wedge \vec{\pi}^{\circlearrowleft}$ *is* (V, E)*-cyclic* $\wedge\, \mathsf{should\_accel}(\vec{\pi}^{\circlearrowleft})$ **do**

8 $\quad\quad$ **if** $\exists \tau_c.\ (\vec{\pi}^{\circlearrowleft}, \tau_c) \in \mathsf{cache}$ **do** $\tau_\ell \leftarrow \tau_c$

9 $\quad\quad$ **else** $\mathsf{id} \leftarrow \mathsf{id} + 1$; $\tau_\ell \leftarrow \mathsf{accel}(\vec{\pi}^{\circlearrowleft}) \wedge \ell = \mathsf{id}$; $\mathsf{cache} \leftarrow \mathsf{cache} \cup \{(\vec{\pi}^{\circlearrowleft}, \tau_\ell)\}$

10 $\quad\quad \beta_1 \leftarrow \neg \left(\bigwedge_{i=0}^{|\vec{\pi}^{\circlearrowleft}|-1} \mu_{b+i}(\pi_i^{\circlearrowleft}) \right)$; $\beta_2 \leftarrow \ell^{(b)} \neq \mathsf{id} \vee \neg \left(\bigwedge_{i=0}^{|\vec{\pi}^{\circlearrowleft}|-1} \mu_{b+i+1}(\pi_i^{\circlearrowleft}) \right)$

11 $\quad\quad \mathsf{add}(\mu_b(\tau \vee \tau_\ell) \wedge \beta_1 \wedge \beta_2)$

12 $\quad$ **else** $\mathsf{add}(\mu_b(\tau))$

13 $\quad$ **if** $\neg\mathsf{check\_sat}()$ **do return** safe **else** $\sigma \leftarrow \mathsf{get\_model}()$; $b \leftarrow b + 1$

to the SMT problem to prevent ABMC from finding a model that gives rise to the trace $\vec{\pi}$. Note that we have $\mu_2(\tau_{\mathsf{i}}^+) \models \beta_1$, as $\tau_{x<100} \models \ell = 0$ and $\tau_{\mathsf{i}}^+ \models \ell \neq 0$. Thus, β_1 blocks $\tau_{x<100}$ for the third step, but τ_{i}^+ can still be used without restrictions. Therefore, adding β_1 to the SMT problem does not prevent us from covering the entire search space.

Similarly, we have $\rightarrow_{\vec{\pi}'} \subseteq \rightarrow_{\vec{\tau}}$ for $\vec{\pi}' = [\tau_{x<100}, \tau_{x<100}, \tau_{\mathsf{i}}^+, \tau_{x<100}]$. Thus, we also add the following blocking clause to the SMT problem:

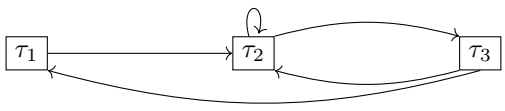
$$\ell^{(2)} \neq 1 \vee \neg\mu_3(\tau_{x<100}) \qquad (\beta_2)$$

ABMC with blocking clauses can be seen in Alg. 3. The counter id is used to obtain unique identifiers for learned transitions. Thus, it is initialized with 0 (Line 1) and incremented whenever a new transition is learned (Line 9). Moreover, as explained above, $\ell = 0$ is conjoined to τ (Line 1), and $\ell = \mathsf{id}$ is conjoined to each learned transition (Line 9).

In Line 10, the blocking clauses corresponding to the superfluous traces $\vec{\pi}$ and $\vec{\pi}'$ above are created, and they are added to the SMT problem in Line 11. Here, $\pi_i^{\circlearrowleft}$ denotes the i^{th} transition in the sequence $\vec{\pi}^{\circlearrowleft}$.

Importantly, Alg. 3 caches (Line 9) and reuses (Line 8) learned transitions. In this way, the learned transitions that are conjoined to the SMT problem have the same id if they stem from the same cycle, and thus the blocking clauses β_1 and β_2 can also block sequences $\vec{\pi}^{\circlearrowleft}$ that contain learned transitions.

Example 12 (Caching). Assume that τ has the following dependency graph:



As Alg. 3 conjoins $\ell = 0$ to τ, assume $\tau_i \models \ell = 0$ for all $i \in \{1, 2, 3\}$. Moreover, assume that accelerating τ_2 yields τ_2^+ with $\tau_2^+ \models \ell = 1$. If we obtain the trace $[\tau_1, \tau_2^+, \tau_3]$, it can be accelerated. Thus, Alg. 3 would add

$$\beta_1 \equiv \neg \left(\mu_3(\tau_1) \wedge \mu_4(\tau_2^+) \wedge \mu_5(\tau_3)\right)$$

to the SMT problem. If the next step yields the trace $[\tau_1, \tau_2^+, \tau_3, \tau_2]$, then τ_2 is accelerated again. Without caching, acceleration may yield a new transition $\tau_{2'}^+$ with $\tau_{2'}^+ \models \ell = 2$. As the SMT solver may choose a different model in every iteration, the trace may also change in every iteration. So after two more steps, we could get the trace $[\tau_1, \tau_2^+, \tau_3, \tau_1, \tau_{2'}^+, \tau_3]$. At this point, the "outer" loop consisting of τ_1, arbitrarily many repetitions of τ_2, and τ_3, has been unrolled a second time, which should have been prevented by β_1. The reason is that $\tau_2^+ \models \ell = 1$, whereas $\tau_{2'}^+ \models \ell = 2$, and thus $\tau_{2'}^+ \models \neg\tau_2^+$. With caching, we again obtain τ_2^+ when τ_2 is accelerated for the second time, such that this problem is avoided.

Remarkably, blocking clauses allow us to prove safety in cases where BMC fails.

Example 13 (Proving Safety with Blocking Clauses). Consider the safety problem $(x \leq 0, \tau, x > 100)$ with $\tau \equiv x < 100 \wedge x' = x + 1$. Alg. 1 cannot prove its safety, as τ can be unrolled arbitrarily often (by choosing smaller and smaller initial values for x). With Alg. 3, we obtain the following SMT problem with $b = 3$.

$$\mu_0(x \leq 0) \qquad \text{(initial states)}$$
$$\mu_0(\tau \wedge \ell = 0) \qquad (\tau)$$
$$\mu_1(\tau \wedge \ell = 0) \qquad (\tau)$$
$$\neg\mu_2(\tau \wedge \ell = 0) \qquad (\beta_1)$$
$$\ell^{(2)} \neq 1 \vee \neg\mu_3(\tau \wedge \ell = 0) \qquad (\beta_2)$$
$$\mu_2((\tau \wedge \ell = 0) \vee (n > 0 \wedge x + n \leq 100 \wedge x' = x + n \wedge \ell = 1)) \qquad (\tau \vee \mathsf{accel}(\tau))$$
$$\mu_3(\tau \wedge \ell = 0) \qquad (\tau)$$

From the last formula and β_2, we get $\ell^{(2)} \neq 1$. On the other hand, the formula labeled with $(\tau \vee \mathsf{accel}(\tau))$ and β_1 imply $\mu_2(\ell = 1) \equiv \ell^{(2)} = 1$, resulting in a contradiction. Thus, $\mathsf{ABMC}_{\mathsf{block}}$ can prove safety with the bound $b = 3$.

Like ABMC, $\mathsf{ABMC}_{\mathsf{block}}$ preserves all important properties of BMC (see [14] for a proof).

Theorem 14. $\mathsf{ABMC}_{\mathsf{block}}$ *is*
Refutationally Complete: *If* $\mathcal{T}$ *is unsafe, then* $\mathsf{ABMC}_{\mathsf{block}}(\mathcal{T})$ *returns* unsafe.
Sound: *If* $\mathsf{ABMC}_{\mathsf{block}}(\mathcal{T})$ *returns* (un)safe, *then* $\mathcal{T}$ *is (un)safe.*
Non-Terminating: *If* $\mathcal{T}$ *is safe, then* $\mathsf{ABMC}_{\mathsf{block}}(\mathcal{T})$ *may not terminate.*

5 Related Work

There is a large body of literature on bounded model checking that is concerned with encoding temporal logic specifications into propositional logic, see [3, 4] as starting points. This line of work is clearly orthogonal to ours.

Moreover, numerous techniques focus on proving *safety* or *satisfiability* of transition systems or CHCs, respectively (e.g., [9, 11, 18, 20, 21, 26]). A comprehensive overview is beyond the scope of this paper. Instead, we focus on techniques that, like ABMC, aim to prove unsafety by finding long counterexamples.

The most closely related approach is ADCL, which has already been discussed in Sect. 1. Other acceleration-based approaches [2, 7, 15] can be seen as generalizations of the classical state elimination method for finite automata: Instead of transforming finite automata to regular expressions, they transform transition systems to formulas that represent the runs of the transition system. During this transformation, acceleration is the counterpart to the Kleene star in the state elimination method. Clearly, these approaches differ fundamentally from ours.

In [22], under-approximating acceleration techniques are used to enrich the control-flow graph of C programs. Then an external model checker is used to find counterexamples. In contrast, ABMC tightly integrates acceleration into BMC, and thus enables an interplay of both techniques: Acceleration changes the state of the bounded model checker by adding learned transitions to the SMT problem. Vice versa, the state of the bounded model checker triggers acceleration. Doing so is impossible if the bounded model checker is used as an external black box.

In [23], the approach from [22] is extended by a program transformation that, like our blocking clauses, rules out superfluous traces. For structured programs, program transformations are quite natural. However, as we analyze unstructured transition formulas, such a transformation would be very expensive in our setting. More precisely, [23] represents programs as CFAs. To transform them, the edges of the CFA are inspected. In our setting, the syntactic implicants correspond to these edges. An important goal of ABMC is to avoid computing them explicitly. Hence, it is unclear how to apply the approach from [23] in our setting.

Another related approach is described in [19], where acceleration is integrated into a CEGAR loop in two ways: (1) as preprocessing and (2) to generalize interpolants. In contrast to (1), we use acceleration "on the fly". In contrast to (2), we do not use abstractions, so our learned transitions can directly be used in counterexamples. Moreover, [19] only applies acceleration to conjunctive transition formulas, whereas we accelerate conjunctive variants of arbitrary transition formulas. So in our approach, acceleration techniques are applicable more often, which is particularly useful for finding long counterexamples.

Finally, *transition power abstraction* (TPA) [5] computes a sequence of over-approximations for transition systems where the n^{th} element captures 2^n instead of just n steps of the transition relation. So like ABMC, TPA can help to find long refutations quickly, but in contrast to ABMC, TPA relies on over-approximations.

6 Experiments and Conclusion

We presented ABMC, an adaption of bounded model checking that makes use of acceleration techniques. By enabling BMC to find deep counterexamples, it targets one of the main limitations of BMC. However, without further precautions, ABMC is not guaranteed to make use of the transitions that result from applying acceleration, since whether they are used or not depends on the models found by the underlying SMT solver. Hence, we introduced *blocking clauses* to enforce the use of accelerated transitions. In this way, they also enable ABMC to prove safety in cases where BMC fails to do so.

We implemented ABMC in our tool LoAT [13]. It uses the SMT solvers Z3 [24] and Yices [10]. Currently, our implementation is restricted to integer arithmetic. It uses the acceleration technique from [12] which, in our experience, is precise in most cases where the values of the variables after executing the loop can be expressed by polynomials of degree ≤ 2. If acceleration yields a non-polynomial formula, then this formula is discarded by our implementation, since Z3 and Yices only support polynomials. We evaluate our approach on the examples from the category LIA-Lin (linear CHCs with linear integer arithmetic) from the CHC competition '23 [8], which contain numerous problems from program verification.

We compared several configurations of LoAT with other leading CHC solvers. More precisely, we evaluated the following configurations:

LoAT We used LoAT's implementations of Alg. 1 (LoAT BMC), Alg. 2 (LoAT ABMC), Alg. 3 (LoAT ABMC$_{\text{block}}$), and ADCL (LoAT ADCL).

Z3 [24] We used Z3 4.12.2, where we evaluated its implementations of the Spacer algorithm (Spacer [21]) and BMC (Z3 BMC).

Golem [5] We used Golem 0.4.3, where we evaluated its implementations of *transition power abstraction* (Golem TPA [5]) and BMC (Golem BMC).

Eldarica [20] We used the default configuration of Eldarica 2.0.9.

We ran our experiments on StarExec [25] with a wallclock timeout of 300s, a cpu timeout of 1200s, and a memory limit of 128GB per example.

2023	unsafe		safe	
	✓	!	✓	!
LoAT ABMC	73	–	31	–
LoAT ABMC$_{\text{block}}$	72	0	75	12
Golem TPA	63	4	88	3
LoAT BMC	60	0	36	0
Z3 BMC	58	–	21	–
LoAT ADCL	56	1	0	–
Golem BMC	55	–	20	–
Spacer	52	5	156	51
Eldarica	29	0	121	17

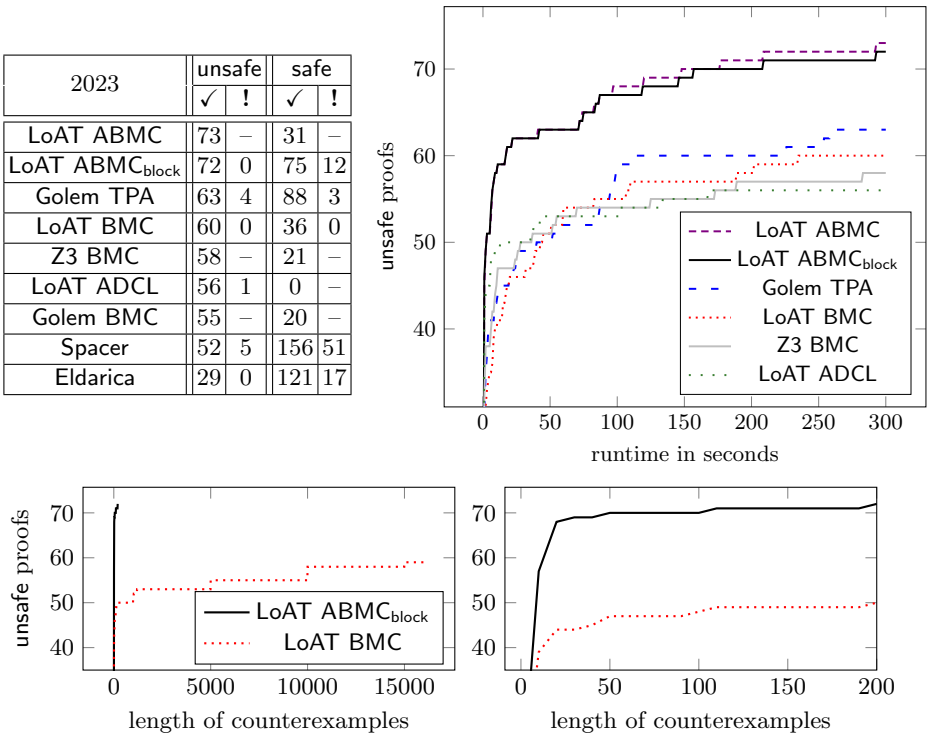


Fig. 2: Comparing ABMC with other techniques and tools for CHC solving

The results can be seen in Fig. 2. In the table, the columns marked with **!** show the number of unique proofs, i.e., the number of examples that could only be solved by the corresponding configuration. Clearly, such a comparison only makes sense if just one implementation of each algorithm is considered. Hence, here we disregarded LoAT ABMC, Z3 BMC, and Golem BMC.

The table shows that our implementation of ABMC is very powerful for proving unsafety. In particular, it shows a significant improvement over LoAT BMC, which is implemented very similarly, but does not make use of acceleration.

Note that all unsafe instances that can be solved by ABMC can also be solved by other configurations. This is not surprising, as LoAT ADCL is also based on acceleration techniques. Hence, ABMC combines the strengths of ADCL and BMC, and conversely, unsafe examples that can be solved with ABMC can usually also be solved by one of these techniques. So for unsafe instances, the main contribution of ABMC is to have *one* technique that performs well both on instances with shallow counterexamples (which can be solved by BMC) as well as instances with deep counterexamples only (which can often be solved by ADCL).

On the one instance that can only be solved by ADCL, our (A)BMC implementation spends most of the time with applying substitutions, which clearly shows potential for further optimizations. Due to ADCL's depth-first strategy, it tends to produce smaller formulas, so that applying substitutions is cheaper.

Regarding safe examples, the tables show that our implementation of ABMC is not competitive with state-of-the-art techniques. However, it finds several unique proofs. This is remarkable, as LoAT is not at all fine-tuned for proving safety. For example, we expect that LoAT's results on safe instances can easily be improved by integrating over-approximating acceleration techniques. While such a variant of ABMC could not prove unsafety, it would presumably be much more powerful for proving safety. We leave that to future work.

The upper right plot shows how many proofs of unsafety were found within a given runtime, where we only include the six best configurations for readability. They clearly show that ABMC is highly competitive on unsafe instances, not only in terms of solved examples, but also in terms of runtime. The plots on the bottom show how many counterexamples of a certain length have been found by LoAT BMC and $\mathsf{ABMC}_{\mathsf{block}}$ (the right one is truncated after 200 steps). They show the impact of acceleration: While LoAT BMC needs up to 16000 steps, the longest counterexamples found by LoAT $\mathsf{ABMC}_{\mathsf{block}}$ has length 200.

Our results also show that blocking clauses have no significant impact on ABMC's performance on unsafe instances, neither regarding the number of solved examples, nor regarding the runtime. In fact, $\mathsf{ABMC}_{\mathsf{block}}$ solved one instance less than ABMC (which can, however, also be solved by $\mathsf{ABMC}_{\mathsf{block}}$ with a larger timeout). On the other hand, blocking clauses are clearly useful for proving safety, where they even allow LoAT to find several unique proofs.

Our implementation is open-source and available on Github. For the sources, a pre-compiled binary, and more information on our evaluation, we refer to [1].

References

1. Evaluation of "Accelerated Bounded Model Checking" (2023), https://loat-developers.github.io/abmc-eval/
2. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: Acceleration from theory to practice. Int. J. Softw. Tools Technol. Transf. **10**(5), 401–424 (2008). https://doi.org/10.1007/s10009-008-0064-3
3. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2
4. Biere, A.: Bounded model checking. In: Handbook of Satisfiability - Second Edition, pp. 739–764. Frontiers in Artificial Intelligence and Applications 336, IOS Press (2021). https://doi.org/10.3233/FAIA201002
5. Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Transition power abstractions for deep counterexample detection. In: TACAS '22. pp. 524–542. LNCS 13243 (2022). https://doi.org/10.1007/978-3-030-99524-9_29
6. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: TACAS '09. pp. 337–351. LNCS 5505 (2009). https://doi.org/10.1007/978-3-642-00768-2_29
7. Bozga, M., Iosif, R., Konečný, F.: Relational analysis of integer programs. Tech. Rep. TR-2012-10, VERIMAG (2012), https://www-verimag.imag.fr/TR/TR-2012-10.pdf
8. CHC Competition, https://chc-comp.github.io
9. Dietsch, D., Heizmann, M., Hoenicke, J., Nutz, A., Podelski, A.: Ultimate TreeAutomizer (CHC-COMP tool description). In: HCVS/PERR@ETAPS '19. pp. 42–47. EPTCS 296 (2019). https://doi.org/10.4204/EPTCS.296.7
10. Dutertre, B.: Yices 2.2. In: CAV '14. pp. 737–744. LNCS 8559 (2014). https://doi.org/10.1007/978-3-319-08867-9_49
11. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained Horn clauses using syntax and data. In: FMCAD '18. pp. 1–9 (2018). https://doi.org/10.23919/FMCAD.2018.8603011
12. Frohn, F.: A calculus for modular loop acceleration. In: TACAS '20. pp. 58–76. LNCS 12078 (2020). https://doi.org/10.1007/978-3-030-45190-5_4
13. Frohn, F., Giesl, J.: Proving non-termination and lower runtime bounds with LoAT (system description). In: IJCAR '22. pp. 712–722. LNCS 13385 (2022). https://doi.org/10.1007/978-3-031-10769-6_41
14. Frohn, F., Giesl, J.: Accelerated bounded model checking. arXiv:2401.09973 [cs.LO] (2024). https://doi.org/10.48550/arXiv.2401.09973
15. Frohn, F., Naaf, M., Brockschmidt, M., Giesl, J.: Inferring lower runtime bounds for integer programs. ACM Trans. Program. Lang. Syst. **42**(3), 13:1–13:50 (2020). https://doi.org/10.1145/3410331
16. Frohn, F., Giesl, J.: Proving non-termination by Acceleration Driven Clause Learning. In: CADE '23. LNCS 14132 (2023). https://doi.org/10.1007/978-3-031-38499-8_13
17. Frohn, F., Giesl, J.: ADCL: Acceleration Driven Clause Learning for constrained Horn clauses. In: SAS '23. pp. 259–285. LNCS 14284 (2023). https://doi.org/10.1007/978-3-031-44245-2_13
18. Hoder, K., Bjørner, N.S.: Generalized property directed reachability. In: SAT '12. pp. 157–171. LNCS 7317 (2012). https://doi.org/10.1007/978-3-642-31612-8_13
19. Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: ATVA '12. pp. 187–202. LNCS 7561 (2012). https://doi.org/10.1007/978-3-642-33386-6_16

20. Hojjat, H., Rümmer, P.: The Eldarica Horn solver. In: FMCAD '18. pp. 1–7 (2018). https://doi.org/10.23919/FMCAD.2018.8603013
21. 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
22. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. Formal Methods Syst. Des. **47**(1), 75–92 (2015). https://doi.org/10.1007/s10703-015-0228-1
23. Kroening, D., Lewis, M., Weissenbacher, G.: Proving safety with trace automata and bounded model checking. In: FM '15. pp. 325–341. LNCS 9109 (2015). https://doi.org/10.1007/978-3-319-19249-9_21
24. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS '08. pp. 337–340. LNCS 4963 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
25. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: IJCAR '14. pp. 367–373. LNCS 8562 (2014). https://doi.org/10.1007/978-3-319-08587-6_28
26. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: PLDI '18. pp. 707–721 (2018). https://doi.org/10.1145/3192366.3192416