Integrating Loop Acceleration into Bounded Model Checking[⋆]

Florian Frohn and Jürgen Giesl

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

Abstract. *Bounded Model Checking* (BMC) is a powerful technique for proving 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" on the fly, our approach can quickly detect deep counterexamples. 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, e.g., software or hardware systems. 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 systems specified in more expressive languages.

In contrast to, e.g., source code, transition formulas are completely unstructured. However, source code may be unstructured, too (e.g., due to `goto`s), i.e.,

⋆ 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 the predicate (for linear CHCs).

one cannot rely on the input being well structured. So the fact that our approach is independent from the structure of the input makes it broadly applicable.

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, i.e., iff there is a step from a state with $x = c_x \wedge y = c_y$ to a state with $x = c'_x \wedge y = c'_y$ in Listing 1. To prove that an *error state* satisfying $\psi_{\mathsf{err}} := y \geq 100$ is reachable from an *initial state* which satisfies $\psi_{\mathsf{init}} := x \leq 0 \wedge y \leq 0$, BMC has to unroll τ 10100 times.

Our new technique *Accelerated* BMC (ABMC) uses the following *acceleration*

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

of $\tau_{x<100}$: As we have $(c_x, c_y) \to^+_{\tau_{x<100}} (c'_x, c'_y)$ iff $\tau_{\mathsf{i}}^+[x/c_x, y/c_y, x'/c'_x, y'/c'_y]$ is satisfiable, τ_{i}^+ is a "shortcut" for many $\to_{\tau_{x<100}}$-steps.

To compute such a shortcut τ_{i}^+ from the formula $\tau_{x<100}$, we use existing acceleration techniques [18]. In the example above, n serves as loop counter. Then the literal $x' = x + 1$ of $\tau_{x<100}$ gives rise to the recurrence equations $x^{(0)} = x$ and $x^{(n)} = x^{(n-1)} + 1$, which yield the closed form $x^{(n)} = x + n$, resulting in the literal $x' = x + n$ of τ_{i}^+. Thus, the literal $x + n \leq 100$ of τ_{i}^+ is equivalent to $x^{(n-1)} < 100$. As x is monotonically increasing (i.e., $\tau_{x<100}$ implies $x < x'$), $x^{(n-1)} < 100$ implies $x^{(n-2)} < 100$, $x^{(n-3)} < 100$, ..., $x^{(0)} < 100$, i.e., the loop $\tau_{x<100}$ can indeed be executed n times.

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 exploit that an accelerated transition describes the transitive closure of the 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.

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. To increase ABMCs capabilities for proving safety, 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 [15]. 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 $\mathcal{C} = (\mathcal{C}_1, \ldots, \mathcal{C}_k)$. 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 $\mathsf{QF}_\land(\Sigma)$ 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 $\mathsf{QF}_\land(\Sigma)$) 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 $\circledcirc(\tau, \tau') := \tau[\vec{x}'/\vec{x}''] \land \tau'[\vec{x}/\vec{x}'']$ 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 $\to_{\vec{\tau}}$.

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     $\mathsf{push}()$;   $\mathsf{add}(\mu_b(\psi_{\mathsf{err}}))$
4     if $\mathsf{check\_sat}()$ do return $\mathsf{unsafe}$ else $\mathsf{pop}()$;   $\mathsf{add}(\mu_b(\tau))$
5     if $\neg\mathsf{check\_sat}()$ do return $\mathsf{safe}$ else $b \leftarrow b + 1$
```

Acceleration techniques compute the transitive closure of relations. In the following definition, we only consider conjunctive transition formulas, since many existing acceleration techniques do not support disjunctions [8], or approximate in the presence of disjunctions [18]. So the restriction to conjunctive formulas ensures that our approach works with arbitrary existing 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. 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 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 SMT problem. If this results in an unsatisfiable problem in Line 5, then the whole search space has been exhausted, i.e., then $\mathcal{T}$ is safe.

Otherwise, we enter the next iteration.

Example 3 (BMC). For the first 100 iterations of Alg. 1 on Ex. 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, etc. 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 is 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 [22]). *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:*

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

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 implicants of the formulas τ_i that were 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:

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

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:

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

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     $\mathsf{push}()$; $\mathsf{add}(\mu_b(\psi_{\mathsf{err}}))$
5     if $\mathsf{check\_sat}()$ do return unsafe else pop()
6     $\vec{\tau} \leftarrow \mathsf{trace}_b(\sigma)$;   $V \leftarrow V \cup \vec{\tau}$;   $E \leftarrow E \cup \{(\tau_1, \tau_2) \mid [\tau_1, \tau_2]$ is an infix of $\vec{\tau}\}$
7     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     else $\mathsf{add}(\mu_b(\tau))$
9     if $\neg\mathsf{check\_sat}()$ do return safe else $\sigma \leftarrow \mathsf{get\_model}()$;   $b \leftarrow b + 1$
```

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 $\to_\tau$. The dependency graph for Ex. 1 is on the side.

$$\begin{tikzcd} \boxed{\tau_{x<100}} \arrow[loop left] \arrow[r, bend left] & \boxed{\tau_{x=100}} \arrow[l, bend left] \end{tikzcd}$$

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 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 consecutive elements of the trace. In Line 7, a cyclic suffix of the trace may get accelerated, provided that the call to $\mathsf{should\_accel}$ (which will be discussed in 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. Note that we only accelerate "real" cycles $\vec{\pi}^{\circlearrowleft}$ where $\circledcirc(\vec{\pi}^{\circlearrowleft})$ is satisfiable, since $\vec{\pi}^{\circlearrowleft}$ is a suffix of the trace, whose satisfiability is witnessed by σ.

As we rely on syntactic implicants and dependency graphs to detect cycles, ABMC is decoupled from the specific encoding of the input. So for example, transition formulas may be represented in CNF, DNF, or any other structure.

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 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, and the formulas that give rise to the 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 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 $[\tau_{x<100}, \tau_{x<100}, \tau_{\mathsf{i}}^{+}, \tau_{x=100}, \tau_{x<100}, \tau_{\mathsf{i}}^{+}]$ 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 8 (Properties of ABMC). *ABMC is*
Sound: *If* ABMC($\mathcal{T}$) *returns* (un)safe, *then $\mathcal{T}$ is (un)safe.*
Refutationally Complete: *If $\mathcal{T}$ is unsafe, then* ABMC($\mathcal{T}$) *returns* unsafe.
Non-Terminating: *If $\mathcal{T}$ is safe, then* ABMC($\mathcal{T}$) *may not terminate.*

3.3 Fine Tuning Acceleration

We now discuss 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 $\to_{\mathsf{accel}(\vec{\tau})} \subset \to_{\vec{\tau}}^{+}$. This is uncritical for correctness, as using acceleration in Alg. 2 is *always* sound.

First, acceleration should be applied to cyclic suffixes consisting of a single *original* (i.e., non-learned) transition. 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 1. should_accel$([\pi]) = \top$ *iff* $\pi \in \mathsf{sip}(\tau)$.

ABMC($\mathcal{T}$)

1: $x^{(0)} \leq 0 \land y^{(0)} \leq 0$			$\psi_{\mathsf{init}}, b = 0$
2 & 6: $x^{(0)} \mapsto 0, y^{(0)} \mapsto 0$	$\mid\mid \vec{\tau} \leftarrow []$	$\mid\mid E \leftarrow \varnothing$	
8: $(x^{(0)} < 100 \land x^{(1)} = x^{(0)} + 1 \land y^{(1)} = y^{(0)}) \lor \ldots$			τ
6 & 9: $x^{(1)} \mapsto 1, y^{(1)} \mapsto 0$	$\mid\mid \vec{\tau} \leftarrow [\tau_{x<100}]$	$\mid\mid E \leftarrow \varnothing$	$b = 1$
8: $(x^{(1)} < 100 \land x^{(2)} = x^{(1)} + 1 \land y^{(2)} = y^{(1)}) \lor \ldots$			τ
6 & 9: $x^{(2)} \mapsto 2, y^{(2)} \mapsto 0$	$\mid\mid \vec{\tau} \leftarrow \vec{\tau} :: \tau_{x<100}$	$\mid\mid E \leftarrow \{\tau_{x<100} \to \tau_{x<100}\}$	$b = 2$
7: $\ldots \lor (n^{(2)} > 0 \land x^{(2)} + n^{(2)} \leq 100 \land x^{(3)} = x^{(2)} + n^{(2)} \land y^{(3)} = y^{(2)})$			$\tau \lor \tau_{\mathsf{i}}^{+}$
6 & 9: $x^{(3)} \mapsto 100, y^{(3)} \mapsto 0$	$\mid\mid \vec{\tau} \leftarrow \vec{\tau} :: \tau_{\mathsf{i}}^{+}$	$\mid\mid E \leftarrow E \cup \{\tau_{x<100} \to \tau_{\mathsf{i}}^{+}\}$	$b = 3$
8: $\ldots \lor (x^{(3)} = 100 \land x^{(4)} = 0 \land y^{(4)} = y^{(3)} + 1)$			τ
6 & 9: $x^{(4)} \mapsto 0, y^{(4)} \mapsto 1$	$\mid\mid \vec{\tau} \leftarrow \vec{\tau} :: \tau_{x=100}$	$\mid\mid E \leftarrow E \cup \{\tau_{\mathsf{i}}^{+} \to \tau_{x=100}\}$	$b = 4$
8: $(x^{(4)} < 100 \land x^{(5)} = x^{(4)} + 1 \land y^{(5)} = y^{(4)}) \lor \ldots$			τ
6 & 9: $x^{(5)} \mapsto 1, y^{(5)} \mapsto 1$	$\mid\mid \vec{\tau} \leftarrow \vec{\tau} :: \tau_{x<100}$	$\mid\mid E \leftarrow E \cup \{\tau_{x=100} \to \tau_{x<100}\}$	$b = 5$
7: $\ldots \lor (n^{(5)} > 0 \land x^{(5)} + n^{(5)} \leq 100 \land x^{(6)} = x^{(5)} + n^{(5)} \land y^{(6)} = y^{(5)})$			$\tau \lor \tau_{\mathsf{i}}^{+}$
6 & 9: $x^{(6)} \mapsto 100, y^{(6)} \mapsto 1$	$\mid\mid \vec{\tau} \leftarrow \vec{\tau} :: \tau_{\mathsf{i}}^{+}$	$\mid\mid E \leftarrow E$	$b = 6$
7: $\ldots \lor (n^{(6)} > 0 \land x^{(6)} = 100 \land 1 < x^{(7)} \leq 100 \land y^{(7)} = y^{(6)} + n^{(6)})$			$\tau \lor \tau_{\mathsf{o}}^{+}$
4: $y^{(7)} \geq 100$			$b = 7$
5: unsafe			

Fig. 1: Running ABMC on Ex. 1

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

$$\to_{\mathsf{accel}(\vec{\pi}::\mathsf{accel}(\vec{\pi}))} = \to^{+}_{\vec{\pi}::\mathsf{accel}(\vec{\pi})} = (\to_{\vec{\pi}} \circ \to^{+}_{\vec{\pi}})^{+} = \to_{\vec{\pi}} \circ \to^{+}_{\vec{\pi}} = \to_{\vec{\pi}::\mathsf{accel}(\vec{\pi})},$$

and thus accelerating $\vec{\pi} :: \mathsf{accel}(\vec{\pi})$ is pointless, too. More generally, we want to prevent acceleration of sequences $\vec{\pi}_2 :: \mathsf{accel}(\vec{\pi}) :: \vec{\pi}_1$ where $\vec{\pi} = \vec{\pi}_1 :: \vec{\pi}_2$ as

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

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 in general, the cyclic suffix of such a trace consists of a cycle $\vec{\pi}$ and $\mathsf{accel}(\vec{\pi})$, but it does not necessarily start with either of them. To take this into account, we rely on the notion of *conjugates*.

Definition 9 (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 2. $\mathsf{should\_accel}(\vec{\pi}') = \bot$ *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 3. $\mathsf{should\_accel}(\vec{\pi}') = \top$ *if* $\vec{\pi}' \not\equiv_\circ \vec{\pi} :: \mathsf{accel}(\vec{\pi})$ *for all* $\vec{\pi}$*.*

Requirements 1 to 3 give rise to a complete specification for $\mathsf{should\_accel}$: If the cyclic suffix is a singleton, the decision is made based on Req. 1, and otherwise the decision is made based on Requirements 2 and 3. 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 4. $\mathsf{should\_accel}(\vec{\pi}) = \bot$ *if* $\vec{\pi}$ *contains a square.*

Thus, $\mathsf{should\_accel}(\vec{\pi}')$ yields $\top$ iff the following holds:

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

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 [22], 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 only sound for proving unsafety, but not for proving safety.

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.

Algorithm 3: $\mathsf{ABMC_b}$ – 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      $\mathsf{push}();\quad \mathsf{add}(\mu_b(\psi_{\mathsf{err}}))$
5      if $\mathsf{check\_sat}()$ do return unsafe else $\mathsf{pop}()$
6      $\vec{\tau} \leftarrow \mathsf{trace}_b(\sigma);\quad V \leftarrow V \cup \vec{\tau};\quad E \leftarrow E \cup \{(\tau_1, \tau_2) \mid [\tau_1, \tau_2]$ is an infix of $\vec{\tau}\}$
7      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          if $\exists \tau_c.\ (\vec{\pi}^{\circlearrowleft}, \tau_c) \in \mathsf{cache}$ do
9              $\tau_\ell \leftarrow \tau_c$                    // the result of accelerating $\vec{\pi}^{\circlearrowleft}$ was cached
10         else
11             $\mathsf{id} \leftarrow \mathsf{id} + 1;\ \ \tau_\ell \leftarrow \mathsf{accel}(\vec{\pi}^{\circlearrowleft}) \wedge \ell = \mathsf{id}$ // generate new ID and accelerate
12             $\mathsf{cache} \leftarrow \mathsf{cache} \cup \{(\vec{\pi}^{\circlearrowleft}, \tau_\ell)\}$                // update cache
13         $\beta_1 \leftarrow \neg\left(\bigwedge_{i=0}^{|\vec{\pi}^{\circlearrowleft}|-1} \mu_{b+i}(\pi_i^{\circlearrowleft})\right)$          // neither unroll $\vec{\pi}^{\circlearrowleft}$ right now...
14         $\beta_2 \leftarrow \ell^{(b)} \neq \mathsf{id} \vee \neg\left(\bigwedge_{i=0}^{|\vec{\pi}^{\circlearrowleft}|-1} \mu_{b+i+1}(\pi_i^{\circlearrowleft})\right)$     // ...nor after using the
15         $\mathsf{add}(\mu_b(\tau \vee \tau_\ell) \wedge \beta_1 \wedge \beta_2)$                 // accelerated transition
16     else $\mathsf{add}(\mu_b(\tau))$
17     if $\neg\mathsf{check\_sat}()$ do return safe else $\sigma \leftarrow \mathsf{get\_model}();\quad b \leftarrow b + 1$
```

Example 10 (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

$$\neg\mu_2(\tau_{x<100}) \tag{β_1}$$

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 $\to_{\vec{\pi}'} \subseteq \to_{\vec{\tau}}$ 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:

$$\ell^{(2)} \neq 1 \vee \neg\mu_3(\tau_{x<100}) \tag{β_2}$$

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 11). Moreover, as explained above, $\ell = 0$ is conjoined to τ (Line 1), and $\ell = \mathsf{id}$ is conjoined to each learned transition (Line 11).

In Lines 13 and 14, the blocking clauses β_1 and β_2 which correspond to the superfluous traces $\vec{\pi}$ and $\vec{\pi}'$ above are created, and they are added to the SMT problem in Line 15. Here, $\pi_i^{\circlearrowleft}$ denotes the i^{th} transition in the sequence $\vec{\pi}^{\circlearrowleft}$.

Importantly, Alg. 3 caches (Line 12) and reuses (Line 9) 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, as shown in the following example.

Example 11 (Caching). Let τ have the dependency graph given below. 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

$$\begin{tikzcd} \boxed{\tau_1} \arrow[r] & \boxed{\tau_2} \arrow[loop above] \arrow[r, bend left] & \boxed{\tau_3} \arrow[l, bend left] \arrow[ll, bend right=40] \end{tikzcd}$$

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

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 12 (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 $\ell^{(2)} \neq 1$, but 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, due to the blocking clauses, $\mathsf{ABMC_b}$ can prove safety with the bound $b = 3$.

Like ABMC, $\mathsf{ABMC_b}$ preserves BMC's main properties (see [23] for a proof).

Theorem 13. $\mathsf{ABMC_b}$ *is sound and refutationally complete, but non-terminating.*

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 [5,6] 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., [13,17,25,27,29,36]). 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 *Acceleration Driven Clause Learning* [21, 22], 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 [22, Sect. 4]. However, even with a complete redundancy check, ADCL might get stuck in a safe part of the search space [22, 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 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. Furthermore, ADCL has successfully been adapted for proving non-termination [21], and it is unclear whether a corresponding adaption of ABMC would be competitive. Thus, both techniques are orthogonal. See Sect. 6 for an experimental comparison of ADCL with ABMC.

Other acceleration-based approaches [4, 9, 19] 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 [30], 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 [31], the approach from [30] 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, [31] 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 [31] in our setting.

Another related approach is described in [26], 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, [26] 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) [7] 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, which integrates acceleration techniques into bounded model checking. By enabling BMC to find deep counterexamples, it targets a major limitation of BMC. However, whether ABMC makes use of transitions that result from acceleration depends on the models found by the underlying SMT solver. Hence, we introduced *blocking clauses* to enforce the use of accelerated transitions, which also enable ABMC to prove safety in cases where BMC fails.

We implemented ABMC in our tool LoAT [20]. It uses the SMT solvers Z3 [33] and Yices [14]. Currently, our implementation is restricted to integer arithmetic. It uses the acceleration technique from [18] 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 (i.e., here we have $\to_{\mathsf{accel}(\tau)} = \to_{\tau}^{+}$). 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)[2] from the CHC competition '23 [11], which contain problems from numerous applications like verification of C, Rust, Java, and higher-order programs, and regression verification of LLVM programs, see [12] for details. By using CHCs as input format, our approach can be used by any CHC-based tool like Korn [16] and SeaHorn [24] for C and C++ programs, JayHorn for Java

[2] The restriction of our approach to linear clauses (with at most one negative literal) is "inherited" from BMC. In contrast, our approach also supports non-linear arithmetic, but we are not aware of corresponding benchmark collections.

programs [28], HornDroid for Android [10], RustHorn for Rust programs [32], and SmartACE [35] and SolCMC [3] for Solidity.

We compared several configurations of LoAT with the techniques of 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$_b$), and ADCL (LoAT ADCL).

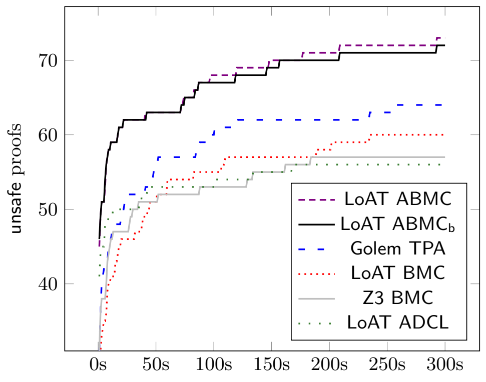
Z3 [**33**] We used Z3 4.13.0, where we evaluated its implementations of the Spacer algorithm (Spacer [29]) and BMC (Z3 BMC).

Golem [**7**] We used Golem 0.5.0, where we evaluated its implementations of *transition power abstraction* (Golem TPA [7]) and BMC (Golem BMC).

Eldarica [**27**] We used Eldarica 2.1.0. We tested all five configurations that are used in parallel in its portfolio mode (`-portfolio`), and included the two that found the most counterexamples: CEGAR with acceleration as preprocessing (Eldarica CEGAR, `eld -splitClauses:1 -abstract:off -stac`) and symbolic execution (Eldarica SYM, `eld -splitClauses:1 -sym`).

Note that all configurations except Spacer and Eldarica CEGAR are specifically designed for finding counterexamples. We did not include further techniques for proving safety in our evaluation, as our focus is on *dis*proving safety. We ran our experiments on StarExec [34] 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$_b$	72	0	75	11
Golem TPA	64	0	83	5
LoAT BMC	60	0	36	0
Z3 BMC	57	–	21	–
LoAT ADCL	56	1	0	–
Golem BMC	55	–	20	–
Spacer	51	4	151	53
Eldarica CEGAR	46	1	107	13
Eldarica SYM	46	1	68	15



The results can be seen in the table above. The columns with **!** show the number of unique proofs, i.e., 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. For instance, LoAT's, Z3's, and Golem's implementations of the BMC algorithm work well on the same class of examples, so that none of them finds unique proofs if all of them are taken into account. Thus, for **!** we disregarded LoAT ABMC, Z3 BMC, and Golem BMC.

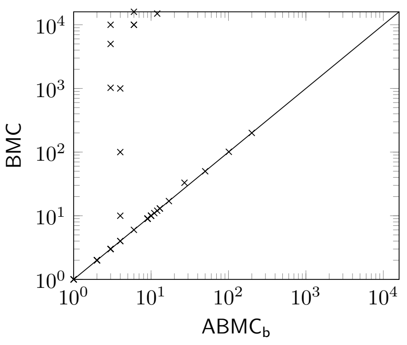
The table shows that our implementation of ABMC is very powerful for proving unsafety. In particular, it shows a significant improvement over LoAT BMC, which is implemented very similarly, but does not make use of acceleration.

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 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 produces smaller formulas, so that applying substitutions is cheaper.

Regarding safe examples, the table shows that our implementation of ABMC is not competitive with state-of-the-art techniques.[3] 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 plot on the previous page shows how many unsafety proofs were found within 300 s, where we only include the six best configurations for readability. It shows that ABMC is highly competitive on unsafe instances, not only in terms of solved examples, but also in terms of runtime. The plot on the right compares the length of the counterexamples found by LoAT $\mathsf{ABMC_b}$ and BMC to show the impact of acceleration. Here, only examples where both techniques disprove safety are considered, and the counterexamples found by $\mathsf{ABMC_b}$ may contain accelerated transitions. There are no points below the diagonal, i.e., the counterexamples found by $\mathsf{ABMC_b}$ are at most as long as those found by BMC. The points above the diagonal indicate that the counterexamples found by $\mathsf{ABMC_b}$ are sometimes shorter by orders of magnitude (note that the axes are log-scaled).



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, $\mathsf{ABMC_b}$ solved one instance less than ABMC (which can, however, also be solved by $\mathsf{ABMC_b}$ 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.

In future work, we plan to support other theories like reals, bitvectors, and arrays, and we will investigate an extension to non-linear CHCs. 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 [2]. An artifact containing LoAT which allows to replicate our experiments is available at [1].

[3] LoAT ABMC finds fewer safety proofs than LoAT BMC since acceleration sometimes yields transitions with non-linear arithmetic that make the SMT problem harder.

References

1. Artifact for "Integrating Loop Acceleration into Bounded Model Checking" (2024), https://zenodo.org/doi/10.5281/zenodo.11954015
2. Evaluation of "Integrating Loop Acceleration into Bounded Model Checking" (2024), https://loat-developers.github.io/abmc-eval/
3. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: SolCMC: Solidity compiler's model checker. In: CAV '22. pp. 325–338. LNCS 13371 (2022). https://doi.org/10.1007/978-3-031-13185-1_16
4. 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
5. 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
6. 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
7. 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
8. 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
9. 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
10. Calzavara, S., Grishchenko, I., Maffei, M.: HornDroid: Practical and sound static analysis of Android applications by SMT solving. In: EuroS&P '16. pp. 47–62. IEEE (2016). https://doi.org/10.1109/EuroSP.2016.16
11. CHC Competition, https://chc-comp.github.io
12. De Angelis, E., Govind V K, H.: CHC-COMP 2023: Competition report (2023), https://chc-comp.github.io/2023/CHC_COMP_2023_Competition_Report.pdf
13. 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
14. 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
15. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press (1972)
16. Ernst, G.: Loop verification with invariants and contracts. In: VMCAI '22. pp. 69–92. LNCS 13182 (2022). https://doi.org/10.1007/978-3-030-94583-1_4
17. 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
18. 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
19. 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
20. 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

21. Frohn, F., Giesl, J.: Proving non-termination by Acceleration Driven Clause Learning. In: CADE '23. pp. 220–233. LNCS 14132 (2023). https://doi.org/10.1007/978-3-031-38499-8_13
22. 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
23. Frohn, F., Giesl, J.: Integrating loop acceleration into bounded model checking. CoRR **abs/2401.09973** (2024). https://doi.org/10.48550/arXiv.2401.09973
24. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: CAV '15. pp. 343–361. LNCS 9206 (2015). https://doi.org/10.1007/978-3-319-21690-4_20
25. 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
26. 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
27. Hojjat, H., Rümmer, P.: The Eldarica Horn solver. In: FMCAD '18. pp. 1–7 (2018). https://doi.org/10.23919/FMCAD.2018.8603013
28. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: A framework for verifying Java programs. In: CAV '16. pp. 352–358. LNCS 9779 (2016). https://doi.org/10.1007/978-3-319-41528-4_19
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. 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
31. 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
32. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for Rust programs. ACM Trans. Program. Lang. Syst. **43**(4), 15:1–15:54 (2021). https://doi.org/10.1145/3462205
33. 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
34. 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
35. Wesley, S., Christakis, M., Navas, J.A., Trefler, R.J., Wüstholz, V., Gurfinkel, A.: Verifying Solidity smart contracts via communication abstraction in SmartACE. In: VMCAI '22. pp. 425–449. LNCS 13182 (2022). https://doi.org/10.1007/978-3-030-94583-1_21
36. 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