SAT-based Model Checking: Interpolation, IC3, and Beyond

Orna GRUMBERG\textsuperscript{a}, Sharon SHOHAM\textsuperscript{b} and Yakir VIZEL\textsuperscript{a}

\textsuperscript{a} Computer Science Department, Technion, Haifa, Israel
\textsuperscript{b} Computer Science Department, Tel-Aviv-Jafa

Abstract. Model checking [18] is an automatic approach to formally verifying that a given system satisfies a given specification. The system to be verified is modelled as a finite state machine and the specification is described using temporal logic. Model checking algorithms are typically based on an exploration of the model state space while searching for violations of the specification. In spite of its great success in verifying hardware and software systems, the applicability of model checking is impeded by its high space and time requirements. This is referred to as the state explosion problem.

The introduction of SAT-based model checking algorithms [7, 9, 39, 45, 48] significantly increases the size of the systems that can be model checked. In its early days SAT-based model checking was used mostly for bug hunting. The introduction of interpolation enabled an efficient complete algorithm, referred to as Interpolation-based model checking (ITP) [39]. ITP uses interpolation to extract an over-approximation of a set of reachable states from a proof of unsatisfiability generated by a SAT-solver. The set of reachable states computed by the reachability analysis is used by ITP to check if a system $M$ satisfies a safety property $\text{AG}p$.

In [9] an alternative SAT-based algorithm, called IC3, is introduced. Similarly to ITP, IC3 also computes over-approximations of sets of reachable states. However, ITP unrolls the model in order to obtain more precise approximations. In many cases, this is a bottleneck of the approach. IC3, on the other hand, improves the precision of the approximations by performing many local checks that do not require unrolling.

In this paper, we survey several approaches to enhancing SAT-based model checking. One approach, detailed in [48], uses interpolation sequence [32, 40] rather than interpolation in order to obtain a more precise over-approximation of the set of reachable states.

The other approach, described in [50], integrates lazy abstraction with IC3 in order to achieve scalability. Lazy abstraction [29, 40], originally developed for software model checking, is a specific type of abstraction that allows hiding different model details at different steps of the verification. We find the IC3 algorithm most suitable for lazy abstraction since its state traversal is performed by means of local reachability checks, each involving only two consecutive sets. A different abstraction can therefore be applied in each of the local checks.

Keywords. Model Checking, SAT-based Model Checking, Interpolation, Interpolation Sequence, Bounded Model Checking (BMC), IC3, Unbounded Model Checking, Abstraction, Lazy abstraction
1. Introduction

Computerized systems dominate almost every aspect of our lives and their correct behavior is essential. Model checking [18] is an automated verification technique for checking whether a given system satisfies a desired property. Unlike testing or simulation based verification, model checking tools are exhaustive in the sense that they traverse all behaviors of the system, and either confirm that the system behaves correctly or present a counterexample.

Model checking has been successfully applied to verifying hardware and software systems. However, with the rapid increase in size and complexity of computerized systems, there is a constant need for a similar increase in verification capabilities.

In this paper we will survey several model checking techniques which can improve model checking applicability and scalability.

One of the main limitations of model checking is the state explosion problem which arises due to the huge state space of the checked systems. The size of the model induces high memory and time requirements that may make model checking infeasible. Much research efforts have been invested along the years in trying to solve this problem.

The first significant solution was the introduction of BDDs [10] into model checking. BDD-based Symbolic Model Checking (SMC) [11] enabled model checking of real-life hardware designs with a few hundreds of state elements. However, current design blocks with well-defined functionality typically have thousands of state elements and more. To handle designs of that scale, SAT-based Bounded Model Checking (BMC) [7] has been developed. Its main drawback, however, is its orientation towards "bug-hunting" rather than full verification.

Ever since the introduction of SAT-based model checking in the context of BMC, several works extend the usage of SAT-based approaches to full verification [6,9,39,41,44]. It important to note that the introduction of algorithms such as in [7,9,39,45,48] significantly increased the size of the verified systems. Still, the search for improved, more scalable methods is neverending.

Abstraction-refinement is another well known methodology for tackling the state-explosion problem. Abstraction hides model details that are not relevant for the checked property. The resulting abstract model is then smaller, and therefore easier to handle by model checking algorithms. Lazy abstraction [29,40], developed for software model checking, is a specific type of abstraction that allows hiding different model details at different steps of the verification.

In this paper we present two SAT-based approaches to full verification which combine BMC with interpolation [20] or interpolation-sequence [32,40]. The proposed methods compute an over-approximated set of the system’s reachable states while checking that the specification is not violated. In addition to that, we present a lazy abstraction-refinement framework for hardware. We use the visible variables abstraction [34], which is particularly suitable for hardware systems. However, we use it in a lazy manner in the sense that different sets of visible variables are used in different iterations of the state-space traversal. In contrast to the first two methods, the underlying model checking algorithm used in the lazy abstraction framework is not based on interpolation, but on the successful IC3/PDR algorithm [7,9].
While the interpolation based algorithms we present are based on an unrolling of the model’s transition relation in order to traverse its state space, the recently introduced IC3 algorithm [9] avoids such unrolling. To verify a safety property, IC3 gradually builds a series of sets of states \( F_0, ..., F_i, ... \), where \( F_i \) over-approximates the set of states reachable within \( i \) steps from the initial states. The computation moves back and forth along the \( F_i \)'s and strengthens them by eliminating unreachable states. This is done via local reachability checks between consecutive sets \( F_i \) and \( F_{i+1} \). IC3 either reaches a fixpoint, in which case all reachable states satisfy the desired property, or returns a counterexample.

In order to compare the methods experimentally we implemented them within Intel’s verification tool. All experiments were conducted on models from Intel’s Sandy Bridge micro-architecture. The checked properties are real specifications, used to verify those designs. The experiments compare various parameters of the two methods. In all our experiments, when a fixpoint could be reached only at a high bound, the interpolation-sequence based (ISB) algorithm performed better than the interpolation based (IB) algorithm. The IB algorithm, on the other hand, performed better when a fixpoint could be reached at a low bound. Falsified properties always favored the ISB algorithm.

In this work we introduce a novel lazy abstraction-refinement technique for hardware model checking, integrated with the SAT-based algorithm IC3 [9].

Model checking [18] is an automatic procedure that determines whether a given system satisfies a specification. In spite of its great success in verifying hardware and software systems, the applicability of model checking is impeded by its high space and time requirements.

The introduction of SAT-based model checking algorithms [7,9,39,45,48] significantly increased the size of the verified systems. Still, the search for improved, more scalable methods is neverending.

Most SAT-based model checking algorithms are based on an unrolling of the model’s transition relation in order to traverse its state space. In contrast, the recently introduced IC3 algorithm [9] avoids such unrolling. To verify a safety property, IC3 gradually builds a series of sets of states \( F_0, ..., F_i, ... \), where \( F_i \) over-approximates the set of states reachable within \( i \) steps from the initial states. The computation moves back and forth along the \( F_i \)'s and strengthens them by eliminating unreachable states. This is done via local reachability checks between consecutive sets \( F_i \) and \( F_{i+1} \). IC3 either reaches a fixpoint, in which case all reachable states satisfy the desired property, or returns a counterexample.

Abstraction-refinement is a well known methodology for tackling the state-explosion problem. Abstraction hides model details that are not relevant for the checked property. The resulting abstract model is then smaller. Lazy abstraction [29,40], developed for software model checking, is a specific type of abstraction that allows hiding different model details at different steps of the verification.

In this work we develop, for the first time, a lazy abstraction-refinement framework for hardware. We use the visible variables abstraction [34], which is particularly suitable for hardware. However, we use it in a lazy manner in the sense that different sets of visible variables are used in different iterations of the state-space traversal.
We find the IC3 algorithm most suitable for lazy abstraction since its state traversal is performed by means of local reachability checks, each involving only two consecutive sets. Thus, at each check a different set of variables is relevant.

Our model checking algorithm, called L-IC3, thus integrates a lazy abstraction-refinement mechanism into IC3. Similarly to IC3, L-IC3 computes a series of over-approximating sets \( F_i \), each referring to a certain time frame. However, L-IC3 considers abstractions of the model during this computation. When constructing \( F_{i+1} \), we determine a set of variables \( U_i \), needed for its construction, and abstract both states and transitions accordingly. The variables in \( U_i \) are referred to as “visible”, while the others are invisible and treated as inputs.

The key ingredients of L-IC3 are therefore a series \( \Omega \) of over-approximating sets of states \( F_i \) and an abstraction sequence \( \bar{U} \) of sets of variables \( U_i \).

L-IC3 works in stages. Each stage consists of an abstract model checking step, followed by a refinement step. At a given stage, the abstract model checking extends both \( \Omega \) and \( \bar{U} \) and checks if they include a potential abstract counterexample. If not, the sequences are further extended. If a potential abstract counterexample is found, the algorithm strengthens the sets \( F_i \) by eliminating abstract states that might be a part of an abstract counterexample.

We use a nonstandard notion of abstract counterexample, based on both \( \Omega \) and \( \bar{U} \). It consists of a sequence of abstract states connected by abstract transitions, satisfying: (i) each transition is based on a different abstraction \( U_i \), and (ii) each abstract state intersects the set \( F_i \) at the corresponding time frame. Our notion of counterexample reflects the incorporation of lazy abstraction into the mechanism of computing \( \Omega \).

If an abstract counterexample is found, meaning that no strengthening is possible anymore based on the abstractions, the refinement step is invoked. Refinement applies just one iteration of a concrete variation of IC3, on the \( \Omega \) computed by the abstract model checking. By doing so, it either finds a concrete counterexample or strengthens the \( F_i \)’s so that all concrete counterexamples of length \( k \) are eliminated. In the latter case, the \( U_i \)’s are also refined by adding more visible variables to each of them, as needed and where needed. Once refinement is finished we move to the next L-IC3 stage and the abstract model checking is re-invoked, continuing the computation from iteration \( k+1 \), with the refined sequences. This makes L-IC3 incremental.

L-IC3 terminates with either a fixpoint, in which case we conclude that the system satisfies the property, or with a concrete counterexample.

In summary, the main contribution of our work is a novel lazy abstraction-refinement technique for hardware. To the best of our knowledge this is the first time lazy abstraction is considered in the context of hardware. Our abstract model checking and refinement are SAT-based. Both avoid unrolling of the transition relation. Since our framework is based on a subtle combination of the abstract and concrete models, we provide theoretical arguments to its correctness.

In order to evaluate our new algorithm we compared it with IC3 on a set of large industrial designs and properties. We obtained speedups of up to two orders of magnitude. Our experiments demonstrate that our lazy abstraction indeed uses different sets of variables in different time frames. Moreover, only a small portion of the design’s variables are used.
1.1. Related Work

[21] and [8] suggest optimizations and extensions to IC3, but they do not combine it with a lazy abstraction-refinement mechanism ([21] suggests the use of abstraction for IC3 but without implementation details nor results). In [17, 25, 26, 30, 41], SAT-based refinement is introduced. However, they use an unrolling of the model while we use local checks a-la IC3. Similarly to [17, 41], we also exploit an unSAT-core for refinement. However, we never unroll the model, while [41] does. Further, [41] is not incremental since after refinement it resumes its (abstract) model checking from time frame 0.

IC3 [9] is sometimes also viewed as an abstraction-refinement algorithm, since it refers to over-approximated sets $F_i$ and the strengthening of these sets resembles refinement. However, the underlying model used by IC3 is concrete, and only the concrete transition relation is considered. We, on the other hand, alternate between abstract transition relations (in the abstract model checking step) and the concrete transition relation (in the refinement step). Our algorithm thus adds a layer of abstraction-refinement on top of this over-approximation-strengthening mechanism.

2. Model Checking

Temporal logic model checking [18] is an automatic approach to formally verifying that a given system satisfies a given specification. The system is often modelled by a finite state transition model and the specification is written in a temporal logic. Determining whether a model satisfies a given specification is often based on an exploration of the model’s state space in a search for violations of the specification.

Definition 2.1. A finite state transition model is a tuple $M = (V, U, INIT, TR)$ where $V$ is a set of variables, $U \subseteq V$ is a set of state variables, $V \setminus U$ is a set of input variables, $INIT(V)$ is a propositional formula over $V$ describing the initial states and $TR(V, V')$ is a propositional formula over $V$ and the next-state variables $V' = \{v' \mid v \in V\}$ describing the transition relation.

We assume that $TR(V, V') = \bigwedge_{v \in U} (v' = f_v(V, V'))$ where $f_v(V, V')$ is a propositional formula that assigns the next value to $v \in U$ based on current and next-state variables. Note that for an input variable $v \in V \setminus U$, $f_v$ is not defined. From this point on $M$ is a finite state transition model.

The set of boolean variables of $M$ induces a set of states $S = \{0, 1\}^{\mid V\mid}$, where each state $s \in S$ is given by a valuation of the variables in $V$. A formula over $V$ (resp. $V, V'$) represents the set of states (resp. pairs of states) obtained by its satisfying assignments. With abuse of notation we will refer to a formula $\eta$ over $V$ as a set of states and therefore use the notion $s \in \eta$ for states represented by $\eta$. The formula $\eta[V \leftarrow V']$, or $\eta'$ in short, is identical to $\eta$ except that each variable $v \in V$ is replaced with $v'$. In the general case $V^i$ is used to denote the
variables in $V$ after $i$ time units (thus, $V^0 \equiv V$). Let $\eta$ be a formula over $V^i$, the
formula $\eta[V^i \leftarrow V^j]$ is identical to $\eta$ except that for each variable $v \in V$, $v^j$ is
replaced with $v^i$.

For a formula $\eta$ over $V \cup V'$ we use $\text{Vars}(\eta) \subseteq V \cup V'$ to denote all (current
or next state) variables appearing in $\eta$.

A path in $M$ is a sequence of states $\pi = (s_0, s_1, \ldots)$ such that for all $i \geq 0,
(s_i, s_{i+1}) \in \text{TR}$. The length of a path is denoted by $|\pi|$. If $\pi$ is infinite
then $|\pi| = \infty$. If $\pi = (s_0, s_1, \ldots, s_n)$ then $|\pi| = n$. A path is an initial path when
$s_0 \in \text{INIT}$. We sometimes refer to a prefix of a path as a path as well.

A formula in Linear Temporal Logic (LTL) \cite{18,43} is of the form $A f$ where $f$
is a path formula. A model $M$ satisfies an LTL property $A f$ if all infinite initial
paths in $M$ satisfy $f$. If there exists an infinite initial path not satisfying $f$,
this path is defined to be a counterexample.

In this paper we consider a subset of LTL formulas of the form $A G p$, where
$p$ is a propositional formula. As mentioned before, this does not restrict the
cherality of the suggested methods since model checking of liveness properties can
be reduced to handling safety properties \cite{4}. Further, model checking of safety
properties can be reduced to handling properties of the form $A G p$ \cite{33}.

The model checking problem is the problem of determining whether a given
model satisfies a given property. Let $M$ be a model, $\text{Reach}$ be the set of reachable
states in $M$, and $f = A G p$ be a property. If for every $s \in \text{Reach}$, $s \models p$ then the
property holds in $M$. On the other hand, if there exists a state $s \in \text{Reach}$ such
that $s \models \neg p$ then there exists an initial path $\pi = s_0, s_1, \ldots, s_n$ such that $s_n = s$.
The path $\pi$ is a counterexample for the property $f$.

Model checking has been successfully applied in hardware verification, and is
emerging as an industrial standard tool for hardware design. A partial list of tools
for hardware verification includes SMV \cite{37} and NuSMV \cite{15}, FormalCheck \cite{27},
RuleBase \cite{2}, and Forecast \cite{23}. In recent years, several tools for model checking
of software have been developed and applied to non-trivial examples. A partial
list consists of SPIN \cite{31}, Bandera \cite{19}, Java PathFinder \cite{28}, SLAM, Bebop, and
Zing \cite{1}, Blast \cite{3}, Magic \cite{13}, and CBMC \cite{16}. An extensive overview of model
checking algorithms can be found in \cite{18}.

The main technical challenge in model checking is the state explosion problem
which occurs if the system is a composition of several components or if the system
variables range over large domains.

An explicit state model checker is a program
which performs model checking directly on a Kripke structure. SPIN \cite{31} is an
example of a successful tool of that kind. Large models are often handled im-
licitly, based on a symbolic representation of the Kripke structure by means of
Boolean functions or propositional formulas. Two widely used such approaches
are the BDD-based model checking \cite{12,37} and the SAT-based bounded model
checking \cite{5}, described in the following sections.
3. Bounded Model Checking

***** BMC cannot compute a RS explicitly, but rather does that implicitly - explain this, this is why it’s not complete, only sound *****.

Many problems, including some versions of model checking, can naturally be translated into the satisfiability problem of the propositional calculus. The satisfiability problem is known to be NP-complete. Nevertheless, modern SAT-solvers, developed in recent years, can handle formulas with several thousands of variables within a few seconds. SAT-solvers such as Grasp [35], Prover [46], Chaff [42], MiniSAT [22], and many others, are based on sophisticated learning techniques and data structures that accelerate the search for a satisfying assignment, if exists. ***** There are major difference in the actual strength of these SAT solvers. DPLL vs. CDCL are theoretically different. CDCL is equivalent to the Resolution Proof System, while DPLL represents a weaker proof system. *****

A SAT solver is a complete decision procedure that given a propositional formula, determines whether the formula is satisfiable or unsatisfiable. Most SAT solvers assume a formula in Conjunctive Normal Form (CNF), consisting of a conjunction of a set of clauses, each of which is a disjunction of propositional variables or their negation. A CNF formula is satisfiable if there exists a satisfying assignment for which every clause in the set is evaluated to \( \top \). If the clause set is satisfiable then the SAT solver returns a satisfying assignment for it. If it is not satisfiable (unsatisfiable), meaning, it has no satisfying assignment, then modern SAT solvers produce a proof of unsatisfiability [41,51]. The proof of unsatisfiability has many useful applications. We will introduce one of them in the next section.

Below we describe a simple way to exploit satisfiability for bounded model checking of properties of the form \( \text{AG} p \), where \( p \) is a Boolean formula.

**Formula 1.** \( \varphi^k_M(f) = \text{INIT}(V^0) \land \text{TR}(V^0, V^1) \land \text{TR}(V^1, V^2) \land \ldots \land \text{TR}(V^{k-1}, V^k) \land (\neg p(V^k)) \)

\( \varphi^k_M(f) \) is then passed to a SAT solver which searches for a satisfying assignment. If there exists a satisfying assignment for \( \varphi^k_M(f) \) then the property is violated, since there exists a path of \( M \) of length \( k \) violating the property. In order to conclude that there is no counterexample of length \( k \) or less, BMC iterates all lengths from 0 up to the given bound \( k \). At each iteration a SAT procedure is invoked.

When \( M \) and \( f \) are obvious from the context we omit them from the formula \( \varphi^k_M(f) \) denoting it as \( \varphi^k \). The BMC algorithm is described in Figure 1.

The main drawback of this approach is its incompleteness. It can only guarantee that there is no counterexample of size smaller or equal to \( k \). It cannot guarantee that there is no counterexample of size greater than \( k \).
1: function BMC($M$, $f$, $k$)
2: $i := 0$
3: while $i \leq k$ do
4: build $\varphi^i_M(f)$
5: result = $SAT(\varphi^i_M(f))$
6: if result = true then
7: return cex // returning the counterexample
8: else
9: $i = i + 1$
10: end if
11: end while
12: return No cex for bound $k$
13: end function

Figure 1. Bounded model checking

Thus, this method is mainly suitable for refutation. Verification is obtained only if the bound $k$ exceeds the length of the longest path among all shortest paths from an initial state to some state in $M$. In practice, it is hard to compute this bound and even when known, it is often too large to handle. Several methods for full verification with SAT have been suggested, such as induction [44], ALL-SAT [14, 24, 38], interpolation [36, 39, 49], and Property Directed Reachability (PDR/IC3) [7, 7, 9]. In the rest of the paper we will focus on SAT-based verification with interpolation and PDR.

4. Interpolation

In this section we introduce two notions, interpolation [20] and interpolation-sequence [32] that, when combined with BMC can provide full program verification.

Throughout the paper we denote the value false as $\bot$ and the value true as $\top$. For a formula $X$, $L(X)$ is the set of variables appearing in $X$. For a set of formulas $\{X_1, \ldots, X_n\}$ we will use $L(X_1, \ldots, X_n)$ to denote the variables appearing in $X_1, \ldots, X_n$. That is, $L(X_1, \ldots, X_n) = L(X_1) \cup \ldots \cup L(X_n)$.

**Definition 4.1.** Let $(A, B)$ be a pair of formulas such that $A \land B \equiv \bot$. The interpolant for $(A, B)$ is a formula $I$ such that:

- $A \Rightarrow I$.
- $I \land B \equiv \bot$.
- $L(I) \subseteq L(A) \cap L(B)$.

The interpolant can be viewed as the part of $A$ that is sufficient to contradict $B$. As mentioned above, modern SAT solvers produce a proof of unsatisfiability if the checked formula is unsatisfiable. An interpolant can be extracted from a proof of unsatisfiability [39], where different proofs yield different interpolants.

A similar notion can be defined when we have a sequence of formulas whose conjunction is unsatisfiable.
**Definition 4.2.** Let $\Gamma = \{A_1, A_2, \ldots, A_n\}$ be a set of formulas such that $\bigwedge \Gamma \equiv \bot$. That is $\bigwedge \Gamma = A_1 \wedge \ldots \wedge A_n$ is unsatisfiable. An **interpolation-sequence** for $\Gamma$ is a set $\{I_0, I_1, \ldots, I_n\}$ such that:

1. $I_0 \equiv \top$ and $I_n \equiv \bot$
2. For every $0 \leq j < n$ it holds that $I_j \wedge A_{j+1} \Rightarrow I_{j+1}$
3. For every $0 < j < n$ it holds that $L(I_j) \subseteq L(A_1, \ldots, A_j) \cap L(A_{j+1}, \ldots, A_n)$

Computing an interpolation-sequence for a sequence of formulas is done in the following way: for each $I_i$, $0 < i < n$, the sequence of formulas is partitioned in a different way such that $I_i$ is the interpolant for the formulas $A(i) = \bigwedge_{j=1}^{i} A_j$ and $B(i) = \bigwedge_{j=i+1}^{n} A_j$. In fact, all interpolants $I_i$ in the sequence can be computed efficiently at once, by a single traversal of a given proof of unsatisfiability [49].

**Theorem 4.3.** Let $\Gamma = \{A_1, A_2, \ldots, A_n\}$ be a set of formulas such that $\bigwedge \Gamma \equiv \bot$ and let $\Pi$ be a proof of unsatisfiability for $\bigwedge \Gamma$. For every $1 \leq i < n$ let us define $A(i) = A_1 \wedge \ldots \wedge A_i$ and $B(i) = A_{i+1} \wedge \ldots \wedge A_n$. Let $I_i$ be the interpolant for the pair $(A(i), B(i))$ extracted using $\Pi$ then the set $\{\top, I_1, I_2, \ldots, I_{n-1}, \bot\}$ is an interpolant sequence for $\Gamma$.

5. Exploiting Interpolation-Sequence in Model Checking

In this section we present a SAT-based algorithm for full verification (sometimes also called *unbounded model checking* (UMC)), which combines BMC and interpolation-sequence [49]. BMC is used to search for counterexamples while the interpolation-sequence is used to produce over-approximated sets of reachable states and to check for termination.

Interpolation-sequence has been introduced and used in [32] and [40]. In [32] it is used for computing an abstract model based on predicate abstraction for software model checking. In [40] interpolation-sequence is used for software model checking and lazy abstraction and is applied to individual execution paths in the control flow graph. The method presented in this section exploits interpolation-sequence in a different manner. In particular, it is applied to the whole model for imitating SMC.

From this point and on, we will use $M$ to denote the transition system and $f = AGp$ for a propositional formula $p$, as the property to be verified.

In order to better understand the algorithm and the motivation behind it, we first review some basic concepts of symbolic model checking (SMC).

5.1. Revisiting Symbolic Reachability Analysis

SMC performs **forward reachability analysis** by computing sets of reachable states $S_j$ where $j$ is the number of transitions needed to reach a state in $S_j$ when starting from the initial states. Further, for every $j \geq 1$, $S_j(V) \wedge TR(V, V') \equiv S_{j+1}(V')$. Once $S_j$ is computed, if it contains states violating $p$, a counterexample of length $i$
is found and returned. Otherwise, if \( S_j \subseteq \bigcup_{i=1}^{j-1} S_i \) then a fixpoint has been reached, meaning that all reachable states have been found already. If no reachable state violates the property then the algorithm concludes that \( M \models f \).

5.2. Interpolation-Sequence Based Model Checking (ISB)

The method presented in this section demonstrates how over-approximated sets, similar to \( S_i \) in their characteristics, can be extracted from BMC, based on interpolation-sequences.

As we have seen, BMC alone is only sound and not complete. In order to be able and determine if \( M \models f \), current SAT-based model checking algorithms are based on a computation that over-approximates the reachable states of \( M \). We use the notion of Reachability Sequence:

**Definition 5.1.** An reachability sequence (RS) with respect to a model \( M \) and a property \( AGp \), denoted \( \Omega(M,p) \), is a sequence \( \langle F_0, \ldots, F_k \rangle \) of propositional formulas over \( V \) such that the following holds:

- \( F_0 = INIT \)
- \( F_i \land TR \Rightarrow F_{i+1} \) for \( 0 \leq i < k \)
- \( F_i \Rightarrow p \) for \( 0 \leq i \leq k \)

A reachability sequence \( \Omega \) is said to be monotonic when \( F_i \Rightarrow F_{i+1} \) for \( 0 \leq i < k \).

The set of states represented by \( F_i \) over-approximates the states reachable from \( INIT \) in exactly \( i \) steps. When \( \Omega \) is monotonic \( F_i \) represents all the states that are reachable from \( init \) in at most \( i \) steps. We refer to \( i \) as time frame (or frame) \( i \). When \( M \) and \( p \) are clear from the context we omit them and write \( \Omega \).

Informally, we will use the notion of fixpoint when we can conclude that all reachable states in the model have been visited\(^1\). Using a RS enables us to determine whether a fixpoint has been reached or not.

We can now show how we use BMC and interpolation-sequence to compute a RS. Note that, the interpolation-sequence exists for a bound \( N \) only when there is no counterexample of length \( N \). In case a counterexample exists, BMC returns a counterexample and the interpolation-sequence is not needed.

**Definition 5.2.** A BMC-partitioning for \( \varphi^N \) is the set \( \Gamma = \{ A_1, A_2, \ldots, A_{N+1} \} \) of formulas such that \( A_i = INIT(V^0) \land TR(V^0, V^i) \), for every \( 2 \leq i \leq N \) \( A_i = TR(V^{i-1}, V^i) \) and \( A_{N+1} = \neg p(V^N) \). Note that \( \varphi^N = \bigwedge_{i=1}^{N+1} A_i \) (= \( \bigwedge \Gamma \)).

For a bound \( N \), consider a BMC formula \( \varphi^N \) and its BMC-partitioning \( \Gamma \). In case \( \varphi^N \) is unsatisfiable, its interpolation-sequence is denoted by \( \bar{I}^N = (I_0^N, I_1^N, \ldots, I_{N+1}^N) \). Note that the BMC-partitioning for \( \varphi^N \) contains \( N + 1 \) el-
\[\text{\(^1\)Since we compute over-approximated sets of reachable states, the computed sets are not monotonic. Therefore, we cannot define a monotonic function \( g \) for which the existence of a fixpoint is guaranteed.}\]
there exists an interpolation-sequence of the form \( \bar{\text{I}} \) of \( N + 2 \) elements and therefore the interpolation-sequence contains \( N + 2 \) elements where the first element and the last one are always \( \top \) and \( \bot \), respectively.

Next, we intuitively explain our method. We start with \( N = 1 \). Consider the formula \( \varphi^1 \) and its BMC-partitioning: \( \{ A_1, A_2 \} \). In case \( \varphi^1 \) is unsatisfiable, there exists an interpolation-sequence of the form \( \bar{I}^1 = (I^1_0, I^1_1, I^1_2 = \bot) \). By Def. 4.2, \( S_1 \subseteq I^1_1 \), where \( S_1 \) is the set of states reachable from the initial states in one transition. This is because \( \top \land A_1 \Rightarrow I^1_1 \) where \( A_1 = \text{INIT}(V^0) \land TR(V^0, V^1) \). Also, \( I^1_1 \land \neg p(V^1) \) is unsatisfiable, since \( I^1_1 \land A_2 \Rightarrow \bot \), where \( A_2 = \neg p(V^1) \). Therefore, \( I^1_1 \models p \).

In the next BMC iteration, for \( N = 2 \), consider \( \varphi^2 \) and its BMC-partitioning \( \{ A_1, A_2, A_3 \} \). In case \( \varphi^2 \) is unsatisfiable, we get \( \bar{I}^2 = (\top, I^2_1, I^2_2, \bot) \). Here too, \( S_1 \subseteq I^2_1 \) and the states reachable from it in one transition are a subset of \( I^2_1 \land A_2 \Rightarrow I^2_2 \). Also, \( S_2 \subseteq I^2_1 \) and \( I^2_2 \models p \). Let us define the sets \( F_1 = I^2_1 \land I^2_1 \) and \( F_2 = I^2_2 \). These sets have the following properties, \( S_1 \subseteq F_1 \), \( S_2 \subseteq F_2 \), \( F_1 \models p \) and \( F_2 \models p \). Moreover, \( F_1[V^1 \leftarrow V] \land TR(V, V') \Rightarrow F_2[V^2 \leftarrow V'] \).

In the general case if \( \varphi^N \) is unsatisfiable then for every \( 1 \leq j \leq N \), \( S_j \subseteq I^N_j \).

If we now define \( F_j = \bigwedge_{k=j}^N I^k_j \) then for every \( 1 \leq j \leq N \) we get:

1. \( F_j \models p \) since \( I^k_j \models p \).
2. \( F_j \land TR(V, V') \Rightarrow F_{j+1} \) since \( I^k_j \land TR(V^j, V^{j+1}) \Rightarrow I^k_{j+1} \) for every \( 1 \leq k \leq N \).
3. \( S_j \subseteq F_{j-1} \) since \( S_j \subseteq I^k_{j-1} \) for every \( 1 \leq k \leq N \).

As a result, the sequence \( \langle F_0 = \text{INIT}, F_1, F_2, \ldots, F_N \rangle \) is a RS and can be used to determine if \( M \models f \). Intuitively, the sets \( I_j \) are similar to the sets \( S_j \) computed by SMC except that they are over-approximations of \( S_j \). Therefore, these sets can be used to imitate the forward reachability analysis of the model’s state-space by means of an over-approximation. This is done in the following manner: BMC runs as usual with one extension. After checking bound \( N \), if a counterexample is found, the algorithm terminates. Otherwise, the interpolation-sequence \( \bar{I}^N \) is extracted and the sets \( F_j \) for \( 1 \leq j \leq N \) are updated. If \( F_j \Rightarrow \bigvee_{i=1}^{j-1} F_i \) for some \( 1 \leq j \leq N \), then we conclude that a fixpoint has been reached and all reachable states have been visited. Thus, \( M \models f \). If no fixpoint is found, the bound \( N \) is increased and the computation is repeated for \( N + 1 \).

Next, we explain why the ISB algorithm uses \( F_j = \bigwedge_{k=j}^N I^k_j \) rather than \( I^N_j \) in its \( N \)th iteration. Informally, the following facts are needed in order to guarantee the correctness of the algorithm. For every \( 1 \leq j \leq N \) we need:

1. \( F_j \) should satisfy \( p \).
2. \( F_j(V) \land TR(V, V') \Rightarrow F_{j+1}(V') \) for \( j \neq N \).
3. \( S_j \subseteq F_j \).

This means that the algorithm cannot be implemented using the extracted interpolation sequence \( \bar{I}^N \) alone. This is because \( I^N \) does not satisfy condition (1): while \( I^N_j \models p \), \( I^N_j \) for \( j \neq N \), does not necessarily satisfy \( p \). This can be
remedied by conjoining each $I_j^N$ with $I_j$. However, now condition (2) no longer holds. Taking $F_j = \bigwedge_{k=j}^N I_k^j$ results in a sequence with all three properties. By that, the sequence follows the properties of Def. 5.1.

The algorithms for updating the RS and checking for a fixpoint are described in Figure 2 and Figure 3, respectively. The complete model checking algorithm using the method described above is given in Figure 4.

It is important to note that a call to UpdateReachability changes all elements of the RS $\Omega$. Therefore, the function FixpointReached cannot count on inclusion checks done in previous iterations and needs to search for a fixpoint at every point in $\Omega$. Moreover, it is not sufficient to check for inclusion of only the last element $I_N$ of $\Omega$. Indeed, if for any $j \leq N$, $F_j \Rightarrow \bigvee_{i=1}^{j-1} F_i$ then all reachable states have been found already. However, the implication $F_N \Rightarrow \bigvee_{i=1}^{N-1} F_i$ might not hold due to additional unreachable states in $I_N$. This is because for all $1 \leq j < N$, $F_{j+1}$ is an over-approximation of the states reachable from $F_j$ and not the exact image (That is, $F_j \land TR(V,V') \Rightarrow F_{j+1}[V \leftarrow V']$ rather than $F_j \land TR(V,V') \equiv F_{j+1}[V \leftarrow V']$).
5.3. Correctness of the ISB algorithm

The following lemmas and definition formalize the above explanation and prove the correctness of the algorithm.

**Lemma 5.3.** If $M$ does not have a counterexample of length $N$, then $S_j \subseteq I_j^N[V^j \leftarrow V]$ for every $1 \leq j \leq N$ and $I_j^N \models p$.

**Proof.** $M$ does not have a counterexample of length $N$. Therefore, the formula $\varphi^N$ is unsatisfiable. Let $I_j^N$ be the interpolation-sequence for the BMC-partitioning of $\varphi^N$. By Definitions 4.2 and 5.2, for $j = 1$, $\top \land INIT(V^0) \land TR(V^0, V^1) \Rightarrow I_1^N$. For each $2 \leq j \leq N$, $I_j^N \land TR(V^j, V^{j+1}) \Rightarrow I_{j+1}^N$. Hence, $S_j \subseteq I_j^N$. Def. 4.2 also state that $I_j^N \land \neg p(V^N) \Rightarrow \bot$ and therefore $I_j^N \models p$. □

**Lemma 5.4.** If $M$ does not have a counterexample of length $N$ or less, then $S_j \subseteq F_j$ and $F_j \models p$ for every $1 \leq j \leq N$.

**Proof.** For every $j \leq k \leq N$ by Lemma 5.3 $S_j \subseteq I_j^k$ and $I_j^k \models p$. Since $F_j$ is the conjunction of $I_j^k$ for every $j \leq k \leq N$, $S_j \subseteq F_j$ and $F_j \models p$. □

**Lemma 5.5.** Let $\Omega = (F_0, F_1, F_2, \ldots, F_N)$ be a sequence s.t. $F_j = \bigwedge_{k=j}^N I_j^k[V^j \leftarrow V]$. For every $1 \leq j < N$, $F_j \land TR(V, V') \Rightarrow F_{j+1}[V \leftarrow V']$.

**Proof.** Def. 4.2 implies that for every $j \leq k \leq N$, $I_{j-1}^k \land TR(V^{j-1}, V^j) \Rightarrow I_j^k$. We get $F_j \land TR(V, V') \Rightarrow F_{j+1}[V \leftarrow V']$. □

**Theorem 5.6.** Assume there is no path of length $N$ or less violating $f$ in $M$. If there exist $1 < j \leq N$ such that $F_j \models \bigvee_{i=1}^{j-1} F_i$, then $M \models f$.

**Proof.** By assumption, there is no path in $M$ of length $N$ or less that violates $f$. We now show that given $F_j \models \bigvee_{i=1}^{j-1} F_i$ we can conclude that there is no path of any length violating $f$. Let $R = \bigvee_{i=1}^{j-1} F_i$. By assumption, $F_j \Rightarrow R$ and by Lemma 5.5, for every $1 \leq i < j$, $F_i \land TR(V^i, V^{i+1}) \Rightarrow I_{i+1}$. Thus, $R(V) \land TR(V, V') \Rightarrow R(V')$ (1). Moreover, for every $1 \leq i \leq j$ the formula $F_i \land \neg p$ is unsatisfiable (since $F_i \models p$ by Lemma 5.4). Hence, $R \land \neg p$ is unsatisfiable (2).

We can show by induction that all reachable states are in $R^* = R \lor INIT$. The base case handles initial states. This holds trivially by the definition of $R^*$. Now let us assume it holds for all states reachable in $k$ steps. It should be proved for states reachable in $k+1$ steps. Let $s_{k+1}$ be a state reachable in $k+1$ steps from an initial state. Let $\pi = s_0, s_1, \ldots, s_k, s_{k+1}$ be an initial path to $s_{k+1}$. By the induction hypothesis $s_k \in R^*$. From (1) we know that $R[V \leftarrow V^k] \land TR(V^k, V^{k+1}) \Rightarrow R[V \leftarrow V^{k+1}]$. Therefore, $s_{k+1} \in R^*$.

By assumption, $INIT \models p$ since there is no path of length $N$ or less violating $f$. By that and (2), $R^* \models p$. Thus, the set of reachable states satisfy $p$ which implies that $M \models f$. 

1: function ISB($M, f$)
2:   $k := 0$
3:   result = BMC($M, f, 0$)
4:   if (result == cex) then
5:     return cex
6: end if
7: $\Omega = \langle \text{INIT} \rangle$ // Reachability sequence
8: while (true) do
9:   $k = k + 1$
10:  result = BMC($M, f, k$)
11:  if (result == cex) then
12:     return cex
13: end if
14:  $I^k = (\top, I^k_1, \ldots, I^k_k, \bot)$
15:  UpdateReachable($\Omega, I^k$)
16:  if (FixpointReached($\Omega$) == true) then
17:     return true
18: end if
19: end while
20: end function

Figure 4. The ISB Algorithm

Lemma 5.7. Suppose $M \models f$ then there exists a bound $N$ such that $\Omega = \langle F_0, F_1, I_2, \ldots, I_N \rangle$ and there exists an index $0 < j \leq N$ such that $F_j \Rightarrow \bigvee_{i=0}^{j-1} F_i$.

Proof. The set of states $S$ is finite. Let us define $N = j = |S| + 1$. $M \models f$ hence for every $0 \leq k \leq N$, $\varphi^k$ is unsatisfiable. Thus, the interpolation-sequence $I^k$ exists for every $0 \leq k \leq N$ and by that the reachability sequence $\Omega = \langle F_0, F_1, F_2, \ldots, F_N \rangle$ exists. Since $|S| < \infty$ we get $F_j \Rightarrow \bigvee_{i=0}^{j-1} F_i$.

Theorem 5.8. There exists a path $\pi$ of length $N$ such that $\pi$ violates $f$ if and only if ISB terminates and returns cex.

Proof. Assume that the minimal violating path is of length $N$. For $N - 1$ there is no path in $M$ violating $f$. By Theorem 5.6 we get that for every $j$ such that $1 \leq j < N$, $F_j \Rightarrow \bigvee_{i=0}^{j-1} F_i$ does not hold. Therefore, the algorithm cannot terminate by returning true in the first $N - 1$ iterations. When the algorithm reaches the $N$-th iteration, $\text{BMC}(M, f, N)$ will return a counterexample and the algorithm terminates. The other direction is immediate.

Theorem 5.9. For every model $M$ and a property $f = AGp$ there exists $N$ such that ISB terminates.
function CheckReachable(M, f, k)  
  R = M.INIT // Initialize R - initial states of M  
  if !(BMC(M, f, 1, k) == cex) then  
    return cex  
  end if  
  M' = M  
  repeat  
    A = J(V^0) \land TR(V^0, V^1)  
    B = TR(V^1, V^2) \land \ldots \land TR(V^{k-1}, V^k) \land ( \bigvee_{j=1}^k \neg p(V^j))  
    J = SAT.getInterpolant(A, B)  
    if J \subseteq R then  
      return fixpoint  
    end if  
    R = R \cup J  
    M'.INIT = J  
  until !(BMC(M', f, 1, k) == cex)  
  return abort  
end function

Figure 5. Computing reachable states using interpolation and BMC with a specific bound k

Proof. If M |= f it follows by Lemma 5.7 that the algorithm terminates and returns true. If there is a path in M that violates f, it follows by Theorem 5.8 that the algorithm terminates and returns cex. □

6. Interpolation Based Model Checking (IB)

***** Should add a description of how the interpolants are pare of a reachability sequence *****

In [39], interpolation has been suggested for the first time in order to obtain a SAT-based model checking algorithm for full verification.

The algorithm combines BMC and Craig’s Interpolation [20]. Similarly to the ISB algorithm presented in the previous section, the interpolant is used to compute an over-approximation of the set of reachable states. However, the computation is done differently. As before, the algorithm concludes that the property holds when a fixpoint is reached during the computation of the reachable states and none of the computed state violates the property.

The following definition is useful in explaining the interpolation based algorithm. Recall that the verified property is of the form f = AGp.

Definition 6.1. For a set of states T, T is a S_j-approximation w.r.t N, where 1 \leq j \leq N, if the following two conditions hold: S_j \subseteq T and there is no path of length (N - j) or less violating p, starting from a state s \in T. We write S_j \preceq_N T to denote that T is a S_j-approximation w.r.t N.

The formula \varphi^k is used in BMC to represent a counterexample of length exactly k. This formula can be modified to represent a counterexample of length
For $1 \leq l \leq k$. We denote this formula by $\varphi^{1,k}$ and write $BMC(M, f, 1, k)$ when
BMC runs on $\varphi^{1,k}$.

**Formula 2.** $\varphi^{1,k} = INIT(V^0) \land TR(V^0, V^1) \land TR(V^1, V^2) \land \ldots \land TR(V^{k-1}, V^k) \land$

$(\bigvee_{j=1}^{k} \neg p(V^j))$

Consider the following partitioning for $\varphi^{1,k}$:

- $A = INIT(V^0) \land TR(V^0, V^1)$
- $B = \bigwedge_{i=1}^{k-1} TR(V^i, V^{i+1}) \land (\bigvee_{j=1}^{k} \neg p(V^j))$.

Clearly $\varphi^{1,k} \equiv A \land B$. Assume that $\varphi^{1,k}$ is unsatisfiable. By the interpolation theorem [20], there exists an interpolant $J_i^k$ which, by Def. 4.1, has the following properties:

- $J_i^k$ is defined over the variables of $L(A) \cap L(B)$, namely, $V^1$.
- $A \Rightarrow J_i^k$. Hence, $S_i \subseteq J_i^k$.
- $J_i^k(V_1) \land B$ is unsatisfiable. This means that there is no path of length $k - 1$ or less, starting from $J_i^k$, which violates $p$.

By the above we get that $S_i \preceq_{k} J_i^k$. We can now proceed by replacing the initial states of $M$ with the computed interpolant $J_i^k$. BMC is reinvoked with the same bound $k$ and with the modified model $M' = (S, J_i^k[V^1 \leftarrow V], TR, L)$ in which the initial states are $J_i^k$. A new interpolant $J_i^{k+1}$ is then extracted. $J_i^{k+1}$ satisfies $S_{i+1} \preceq_{k+1} J_i^k$. It is important to notice that $J_i^k$ now satisfies $S_i \preceq_{k+1} J_i^k$ since the BMC run on $M'$ did not find a counterexample of length $k$ starting from a state in $J_i^k$. In the general case we replace $INIT$ with $J_i^k$ and get $J_i^{k+1}$.

Figure 5 presents, for a given bound $k$, the computation of an over-approximated set of reachable states. Note that after $L$ iterations of the main loop in CheckReachable we get $L$ interpolants and for every $1 \leq i \leq L$, $S_i \preceq_{k+L} J_i^k$. All computed states are collected in $R$. If at any iteration, the interpolant $J_i$ is contained in $R$, then all reachable states have been found with no violation of $f$. CheckReachable then returns “fixpoint”.

On the other hand, if a counterexample is found on a modified model, then CheckReachable$(M, f, k)$ is aborted and CheckReachable$(M, f, k+1)$ is initiated. Recall that the counterexample has been obtained on an over-approximated set of states and therefore might not represent a real counterexample in the original model. In case a real counterexample exists, it will be found during a BMC run on the original model $M$ for a larger bound.

In [47], an optimization for CheckReachable is suggested. If the current bound is $k$ and at the $L$-th iteration a counterexample is found, then CheckReachable is reinvoked with bound $k + L$ (rather than $k + 1$). This is possible since $M$ is known not to have a counterexample of length $k + L - 1$. The usefulness of this heuristic highly depends on the type of property that is checked. On the one hand, if the property is false, this heuristic indeed results in a better performance. On the other hand, for true properties, this approach may hurt performance since a fixpoint could have been found at a lower bound than $k + L$ (e.g. $k + 1$).
7. Comparing Interpolation-Sequence Based MC to Interpolation Based MC

In the previous sections we presented two model checking algorithms which combine BMC and interpolation: the Interpolation-Sequence Based (ISB) [49] and the Interpolation Based (IB) [39]. In this section we analyze the differences between the algorithms. In the next section we compare them experimentally.

Both methods compute an over-approximation of the set of reachable states. However, their state traversal is different. As a result, none is better than the other in all cases. In specific cases, though, one may converge faster.

Several technical details distinguish between ISB and IB. First, the formulas from which the interpolants are extracted are different. For a given bound $N$, ISB uses the formula $\varphi_N$ while IB uses $\varphi^1_N$.

Second, the approximated sets are computed in different manners. ISB computes the sets $F_j$ incrementally and refines them after each iteration of BMC, as part of the BMC loop. IB, on the other hand, recomputes the interpolants whenever the bound is incremented (that is, whenever CheckReachable is called with a greater bound).

Third, ISB can be viewed as an addition to the BMC loop. At each application of BMC (with a different bound), the addition includes the extraction of an interpolation-sequence and the check if a fixpoint has been reached. Indeed, after $N$ iterations of the BMC loop in ISB, there are $N$ over-approximated sets of states, $F_1, \ldots, F_N$, satisfying, for each $1 \leq j \leq N$, $S_j \preceq N F_j$.

On the other hand, IB consists of two nested loops. The outer loop increments the bounds while the inner loop computes over-approximated sets of reachable states. If the outer loop is at some bound $N > 1$ and the inner loop performs $L$ iterations then there are $L$ sets of states $J^N_1, \ldots, J^N_L$, each satisfying $S_i \preceq N+L J^N_i$ $(1 \leq i \leq L)$. Table 1 summarizes the above differences.

In summary, IB can compute, at a given bound $N$, as many sets as needed as long as no counterexample is found (not necessarily a real counterexample). On the other hand, for bound $N$, ISB can only compute $N$ sets. However, it does not need recurrent BMC calls for each bound (only one is needed). Thus, we can conclude that in cases IB can compute all the needed sets at a low bound it performs better than ISB. However, for examples where the needed sets can only

<table>
<thead>
<tr>
<th>SMC</th>
<th>ISB</th>
<th>IB</th>
</tr>
</thead>
<tbody>
<tr>
<td>${S_1, \ldots, S_N}$</td>
<td>$(F_1, F_2, \ldots, F_N)$</td>
<td>$(J^1_N, J^2_N, \ldots, J^N_N)$</td>
</tr>
<tr>
<td>$S_i \preceq N F_i$</td>
<td>$S_i \preceq N J^1_i$</td>
<td>$N$ iterations at bound 1, if possible</td>
</tr>
<tr>
<td>After checking bounds 1 to $N$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(S_1, \ldots, S_{N+L})$</td>
<td>$(F_1, \ldots, F_{L}, \ldots, F_{N+L})$</td>
<td>$(J^N_N, J^{N+1}_N, \ldots, J^N_L)$</td>
</tr>
<tr>
<td>$S_i \preceq N+L F_i$</td>
<td>$S_i \preceq N+L J^N_i, (1 \leq i \leq L)$</td>
<td>$L$ iterations at bound $N$, if possible</td>
</tr>
<tr>
<td>After checking bounds 1 to $N+L$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. The correlation between the interpolants computed by ISB and IB to the sets computed by SMC.
be computed using higher bounds, ISB has an advantage. This fact is reflected in the experimental results.

As mentioned before, when a counterexample exists the over-approximated sets of reachable states are not needed. If a property is violated then there exists a minimal bound \( N \) for which a violating path of length \( N \) exists. Both algorithms have to reach this bound in order to find the counterexample. Here, ISB has a clear advantage over IB. This is because after each BMC run on the original model, IB executes at least one additional BMC run on a modified model. Thus, IB invokes at least two BMC runs for each bound from 1 to \( N - 1 \). Clearly, the second BMC run is more demanding than the inclusion check performed by ISB. In all our experiments, this kind of properties always favored ISB.

8. SAT-based Reachability via IC3

***** In this section we describe bla-bla-bla *****

**Definition 8.1.** Let \( \Omega \) be an RS. A formula \( \eta \) is inductive up to \( j \), if \( F_j \land \eta \land TR \Rightarrow \eta' \).

\( \eta \) is an invariant up to level \( j \) if \( F_i \Rightarrow \eta \) holds for each \( i \leq j \).

Note that if \( \eta \) is inductive up to \( j \) then \( F_i \land \eta \land TR \Rightarrow \eta' \) holds for each \( i \leq j \). Due to the properties of an RS, \( \eta \) is an invariant up to level \( j - 1 \), and in addition \( F_0 \Rightarrow \eta \) (initialization).

IC3 [9] is a SAT-based model checking algorithm that, given a model \( M \) and a property \( AGp \), computes increasingly long sequences \( \Omega(M, p) \). The algorithm works iteratively, where at iteration \( k \), the RS of length \( k + 1 \) is extended to an RS of length \( k + 2 \) by initializing the set \( F_{k+1} \) and possibly updating previous sets (with index \( i \leq k + 1 \)). The computation continues until either a counterexample is found or a fixpoint is reached (i.e. \( F_{i+1} \Rightarrow F_i \) for some \( i \)).

One of the main features of IC3 is the fact that no unrolling of the transition relation is needed. We give a brief overview of how it operates. More details are given along the paper as needed. For the exact details we refer the reader to [9].

IC3 extends and updates \( \Omega \), while strengthening the \( F_i \)'s. The \( k \)th iteration starts from an RS \( \langle F_0, \ldots, F_k \rangle \). Then \( F_{k+1} \) is initialized to \( p \). Clearly, \( F_k \Rightarrow F_{k+1} \) and \( F_{k+1} \Rightarrow p \) hold. Therefore, the purpose of strengthening is to ensure that \( F_k \land TR \Rightarrow F_{k+1}' \). This is done by checking that \( F_k \land TR \land \neg p' \) is unsatisfiable. If this formula is satisfiable then a state \( s \in F_k \) is retrieved from the satisfying assignment. \( s \) is a bad state since it reaches \( \neg p \) (and by that violates \( F_k \land TR \Rightarrow F_{k+1}' \)). At this point, either \( s \) is reachable from \( INIT \), in which case a counterexample exists, or \( s \) is unreachable and needs to be removed from \( F_k \). In order to determine if \( s \) is reachable, IC3 checks the formula: \( F_{k-1} \land TR \land s' \). If this formula is unsatisfiable, then \( s \) can be removed from \( F_k \) (since the property \( F_{k-1} \land TR \Rightarrow F_k' \) of an RS holds without it as well), and the same process is repeated for other states in \( F_k \) that can reach \( \neg p \) (if any). However, if \( F_{k-1} \land TR \land s' \) is satisfiable, a
predecessor \( t \in F_{k-1} \) of \( s \) is extracted and handled similarly to \( s \) in order to determine if \( t \) (which is also a bad state) is reachable from \( \text{INIT} \) or not. IC3 therefore moves back and forth along the \( F_i \)'s, while retrieving bad states \( b \) and checking their reachability from \( \text{INIT} \) via local reachability checks of the form \( F_i \land TR \land b' \). During this process, the \( F_i \)'s are strengthened by removing bad states that are not reachable\(^2\). If a state in \( F_0 = \text{INIT} \) is reached during the backwards traversal, then a counterexample is obtained.

**Definition 8.2.** Satisfiability checks of the form \( F_i \land TR \land \eta \) (where \( \text{Vars}(\eta) \subseteq V \cup V' \)) are called \( i \)-reachability checks.

### 9. Abstraction

Throughout the paper we consider the “visible variables” abstraction [34]. Let \( M_c = (V,U,\text{INIT},TR) \) be a model and let \( U_i \subseteq U \) be a set of state-variables. We refer to \( U_i \) as the set of “visible variables”.

Given \( U_i \), we consider an abstract model \( M_i = (V_i,U_i,TR_i) \) of \( M_c \) where \( TR_i = \bigwedge_{v \in U_i} (v' = f_v(V,V')) \) is an abstract transition relation, and \( V_i = \{ v \in V \mid v \in \text{Vars}(TR_i) \lor v' \in \text{Vars}(TR_i) \} \subseteq V \). Note that the behavior of invisible state variables (in \( U \setminus U_i \)) is nondeterministic.

We do not introduce an abstraction of \( \text{INIT} \) as part of \( M_i \) since we always consider the concrete set of initial states. \( M_i \) is an abstraction of \( M_c \), denoted \( M_c \preceq M_i \), in the sense that both its set of states and its transition relation are abstractions of the concrete ones. \( M_i \) induces a set of abstract states \( S_i \) which includes all valuations to \( V_i \). Specifically, each concrete state \( s \in S \) is abstracted by the abstract state \( s_i \in S_i \) that agrees with \( s \) on the assignment to the joint variables in \( V_i \). In this case we write \( s \preceq s_i \). We sometimes refer to \( s_i \) as the set of concrete states it abstracts: \( \{ s \in S \mid s \preceq s_i \} \).

In addition, \( TR \) is abstracted by \( TR_i \) in the sense that \( TR \Rightarrow TR_i \). Formally, the relation \( \{(s,s_i) \mid s \preceq s_i \} \) is a simulation relation from \( M_c \) to \( M_i \).

Given an RS \( \Omega(M_c,p) = (F_0,\ldots,F_k) \) and an abstract model \( M_i \), we say that a formula \( \eta \) is inductive up to level \( j \) w.r.t. \( M_i \), if \( F_j \land \eta \land TR_i \Rightarrow \eta' \).

**Lemma 9.1.** Any formula inductive up to \( j \) w.r.t. \( M_i \) is also inductive up to \( j \) w.r.t. \( M_c \).

The lemma holds since \( TR \Rightarrow TR_i \). When we do not explicitly mention a model, we refer to inductiveness w.r.t. \( M_c \). The notion of an invariant always refers to \( M_c \).

#### 9.1. Lazy Abstraction

As mentioned above, lazy abstraction [29] allows to use different details of the model at different iterations of the state-space traversal. We adapt the notion of

\(^2\)In fact, in order to remove a bad state \( b \) from \( F_i \), IC3 finds a clause \( c \) that is an invariant up to \( i \) and implies \( \neg b \), and adds \( c \) to \( F_i \) as a conjunct.
1: function L-IC3(p)
2:   \( \Omega = \langle \text{INIT}, p \rangle; \bar{U} = \langle \text{Vars}(p) \rangle \)
3:   if \( \text{Init-IC3}(\Omega, \bar{U}, p) == \text{cex} \) then
4:     return cex
5:   end if
6:   while \( \text{A-IC3}(\Omega, \bar{U}) == \text{abs-cex} \) do
7:     if \( \text{Refine}(\Omega, \bar{U}) == \text{cex} \) then
8:       return cex
9:     end if
10:   end while
11:   return fixpoint
12: end function

Figure 6. L-IC3

lazy abstraction to abstraction based on visible variables [34], and allow different
variables to be visible at different time frames.

Definition 9.2. An abstraction sequence w.r.t. a model \( M_c \) is a sequence \( \bar{U} = \langle U_0, \ldots, U_k \rangle \) where \( U_i \subseteq U \) for \( 0 \leq i \leq k \), is a set of visible state-variables. \( \bar{U} \) is monotonic if \( U_i \subseteq U_{i+1} \) for each \( 0 \leq i < k \).

An abstraction sequence \( \bar{U} \) represents different levels of abstraction of \( M_c \). It induces a sequence of abstract models \( \langle M_0, \ldots, M_k \rangle \) where \( M_i \) is defined as in Sec. 9. If \( \bar{U} \) is monotonic, the induced sequence of abstract models is also monotonic in the sense that \( M_0 \succeq \ldots \succeq M_k \succeq M_c \).

Definition 9.3. Let \( \bar{U} = \langle U_0, \ldots, U_k \rangle \) be a monotonic abstraction sequence and \( \Omega(M_c, p) = \langle F_0, \ldots, F_k \rangle \) an RS. A sequence \( s_0, \ldots, s_j \) of abstract states where \( 0 \leq i < j \leq k + 1 \) is an abstract path from \( i \) to \( j \) if (i) for each \( i \leq l < j \), \( (s_l, s_{l+1}) \models TR_i \), and\( ^3 \) (ii) for each \( i \leq l \leq \min\{j, k\} \), \( s_l \cap F_i \neq \emptyset \).

An abstract path \( s_0, \ldots, s_j \) from 0 to \( j \) is an abstract counterexample of length \( j \) if \( s_j \cap \neg p \neq \emptyset \).

Note that the definition above is not standard. It refers to different transition relations at different steps. Also, it requires the abstract states to be part of the corresponding \( F_i \) in the sense that \( s_i \cap F_i \neq \emptyset \). Unlike with concrete states, it is possible that \( s_i \cap F_i \neq \emptyset \) but \( s_i \not\subseteq F_i \). As a result we do not write \( s_i \in F_i \).

Definition 9.4. An abstraction sequence \( \langle U_0^r, \ldots, U_k^r \rangle \) is a refinement of an abstraction sequence \( \langle U_0, \ldots, U_k \rangle \) if \( U_i \subseteq U_i^r \) for each \( i \).

10. Lazy Abstraction and IC3

In this section we describe our proposed algorithm for lazy abstraction, called L-IC3. The key ingredients of L-IC3 are an abstraction sequence \( \bar{U} \) that induces different abstractions at different time frames as well as an RS \( \Omega \).

---

\(^3\)Requirement (ii) dismisses paths that are known to be spurious based on \( \Omega \). min\{\( j, k \)\} is used for the case where \( j = k + 1 \), in which nonempty intersection is required only up to \( k \).
L-IC3 starts with an initialization step and then works in stages (Fig. 6). Its initialization (lines 2-5) is similar to the initialization of IC3 with one exception. If no counterexample of length 0 or 1 exists, then in addition to initializing $\Omega$ to $\langle F_0 = \text{INIT}, F_1 = p \rangle$, it initializes $\bar{U}$ to $\langle U_0 = \text{Vars}(p) \rangle$. Clearly, after initialization, $\Omega$ is an RS.

Each L-IC3 stage (lines 6-10) consists of an abstract model checking step and a refinement step, both performed by variations of IC3. $\bar{U}$ and $\Omega$ are updated in both steps.

The abstract model checking A-IC3 gradually extends and updates the RS $\Omega$ until either a fixpoint is reached, or an abstract counterexample is found (line 6). In the former case, the property is proved and L-IC3 terminates (line 11). In the latter case, the counterexample is abstract since it is computed w.r.t. the abstract transitions. However, it is also restricted by $\Omega$ (see Def. 9.3). A refinement is then performed (line 7). If the refinement finds a concrete counterexample then it terminates. Otherwise it refines $\bar{U}$ and updates $\Omega$ into an RS (of the same length). A new L-IC3 stage (line 6) of abstraction-refinement then begins, invoking A-IC3 with the updated $\Omega$ and the refined $\bar{U}$.

Altogether, an invocation of L-IC3 results in either a fixpoint (in which case the property is proved) or a concrete counterexample.

**Iterations of L-IC3** The stages of L-IC3 should not be confused with the iterations of IC3 as a stage may extend $\Omega$ by more than one set. Similarly to IC3, we define an iteration of L-IC3 to include the effort involved in the extension of $\Omega$ by one set. In iteration $k$, $\Omega$ is extended from $\langle F_0, \ldots, F_k \rangle$ to $\langle F_0, \ldots, F_k, F_{k+1} \rangle$. If no abstract counterexample is found, the iteration is performed in full by A-IC3. In fact, several iterations can be performed by a single invocation of A-IC3 (within a single stage of L-IC3), extending $\Omega$ by several sets (as long as no abstract counterexample is found). When an abstract counterexample is found, the corresponding iteration that starts at A-IC3 continues at the refinement step.

10.1. Abstract Model Checking via A-IC3

The abstract model checking algorithm, A-IC3 (Fig. 7), either finds an abstract counterexample (line 22), or reaches a fixpoint (line 26) by computing an RS $\Omega$.

**Using different abstractions** The computation of $\Omega$ is done using a variation of IC3 which considers a sequence of abstract models, induced by a monotonic abstraction sequence $\bar{U} = \langle U_0, \ldots, U_k \rangle$. Both abstract transition relations and abstract states are used. Even though abstract models are used, the obtained RS satisfies the requirements of Def. 5.1, which refer to the concrete transition relation $TR$.

To emphasize this, we sometimes refer to the sequence as a concrete RS.

Recall that IC3 performs $i$-reachability checks of the form $F_i \land TR \land \eta$. A-IC3 also performs these checks (within Strengthen, line 20), but instead of using the concrete $TR$ it uses the abstract $TR_i$. This means that when traversing the model’s state space, A-IC3 uses different abstract transition relations at different time frames. Further, when $F_i \land TR_i \land \eta$ is satisfiable, A-IC3 retrieves an abstract state $s_\alpha \in M_i$ from the satisfying assignment. This abstract state is either used to strengthen $\Omega$, or it is part of an abstract counterexample.
13: function A-IC3(Ω, U)
14:  \( k = |Ω| - 1 \)
15:  while Ω.fixpoint() == false do
16:  \( U_k = U_{k-1} \)
17:  \( \bar{U}.add(U_k) \)
18:  \( F_{k+1} = p \)
19:  \( \Omega.add(F_{k+1}) \)
20:  result = STRENGTHEN(Ω, \( \bar{U}, k \))
21:  if result == abs-cex then
22:    return abs-cex
23:  end if
24:  \( k = k + 1 \)
25:  end while
26:  return fixpoint
27: end function

Figure 7. A-IC3

Incrementality A-IC3 is an iterative algorithm. Each iteration of L-IC3 consists of an iteration of A-IC3 (possibly followed by refinement). We therefore use the same number for the L-IC3 iteration and the A-IC3 iteration. If A-IC3 finds a counterexample at iteration \( k \) it returns. After refinement (line 7) A-IC3 is re-invoked with an updated Ω that is an RS of the same length. The computation of Ω resumes from iteration \( k + 1 \) (line 14).

Iterations In iteration \( k \geq 1 \), the RS \( \langle F_0, \ldots, F_k \rangle \) and the abstraction sequence \( \langle U_0, \ldots, U_{k-1} \rangle \) are extended by 1 and updated as follows (see Fig. 7).

1. Check if a fixpoint is reached. If not:
2. \( U_k \) is initialized to \( U_{k-1} \) and added to \( \bar{U} \).
3. \( F_{k+1} \) is initialized to \( p \) and added to \( \Omega \).
4. The sets \( F_0, \ldots, F_{k+1} \) are strengthened iteratively until \( \langle F_0, \ldots, F_{k+1} \rangle \) becomes an RS, or an abstract counterexample is found.

Note that if no counterexample is found, then an iteration of A-IC3 and an iteration of L-IC3 coincide. However, if an abstract counterexample is found, then the corresponding iteration of L-IC3 includes the iteration of A-IC3 as well as the following refinement step.

Below we describe items 2 and 4 in more detail.

(2) Extending \( \bar{U} \): \( U_k \) is initialized to \( U_{k-1} \) (line 16). This is aimed at immediately eliminating from \( TR_k \) spurious transitions that lead from states in \( F_{k-1} \subseteq F_k \) to \( \neg p \) and were already removed from \( TR_{k-1} \). Note that this initialization does not imply that the \( U_i \) sets will always be equal, since refinement might change them in different ways.

(4) Iterative Strengthening of \( \Omega \): At the beginning of the iteration, \( \langle F_0, \ldots, F_k \rangle \) is a concrete RS. However, the addition of \( F_{k+1} = p \) might cause the implication \( F_k \land TR \Rightarrow F_{k+1} \) not to hold. When considering the abstract models and transition relations (as does A-IC3) this means that \( F_k \land TR_k \Rightarrow F_{k+1} \) does not hold, i.e., there exists a bad abstract state at \( F_k \) that reaches \( \neg F_{k+1} = \neg p \). To obtain an abstract counterexample is found w.r.t. \( \Omega = \langle F_0, \ldots, F_{k+1} \rangle \) produced in iteration \( k \), where \( |Ω| = k + 2 \). When A-IC3 is re-invoked, \( k \) is set to \( |Ω| - 1 = k + 1 \).
implication, this state needs to be eliminated from $F_k$. However, its removal can only be done if it does not damage the implication $F_{k-1} \land TR \Rightarrow F_k$. Note that this implication does not necessarily hold w.r.t. to the abstract $TR_{k-1}$ even at the beginning of the iteration. Still, its concrete version holds and to obtain an RS one needs to make sure that it keeps holding. Since A-IC3 only considers the abstract models, the way to go about it is by making sure that there is no abstract predecessor of the bad state in $F_{k-1}$. If there is one, then it also needs to be eliminated to regain implication, and so on. Each of these states is also a bad abstract state that reaches $\neg p$ along an abstract path in $F_0, \ldots, F_k$. Therefore, A-IC3 obtains an RS of length $k+1$ by strengthening the $F_i$’s to exclude bad abstract states that reach $\neg p$ along an abstract path in $F_0, \ldots, F_k$. A sequence of such bad states of length $k+1$ is an abstract counterexample of length $k+1$. In this sense, A-IC3 can also be viewed as trying to eliminate all (suffixes) of abstract counterexamples of length $k+1$ w.r.t. $\langle F_0, \ldots, F_k \rangle$. From this point of view, A-IC3 identifies abstract states that might be a part of an abstract counterexample at a certain time frame, and attempts to block them by learning corresponding invariants. Recall that the abstract counterexamples we consider are restricted not only by the abstract transition relations, but also by the $F_i$ sets (Def. 9.2).

Technically, bad abstract states are described by abstract proof obligations (similarly to the notion of proof obligations used in IC3).

**Definition 10.1.** An abstract proof obligation, or an obligation in short, is a pair $(s, n)$ consisting of a level $n \leq k$ and an abstract state $s$ from which abstract model? over what variables? s.t. (1) $s$ is a “bad state” that reaches $\neg p$ along some abstract path in which models? (2) $\neg s$ is an invariant up until $n$, (3) $s \cap F_{n+1} \neq \emptyset$, and (4) $F_n$ reaches $s$ in one step of $TR_n$.

Thus $n+1$ is the minimal level intersecting $s$, and $F_n$ reaches $s$ in one abstract step. Note that it is possible that $F_n$ cannot reach $s$ along the concrete transitions. A-IC3 maintains two sets of obligations - may and must.

**Definition 10.2.** An obligation $(s, n)$ is a must obligation w.r.t. iteration $k$ if $s$ must be shown unreachable from $F_n$ in one step w.r.t. $TR_n$, in order to ensure that no abstract counterexample of length $k+1$ exists. All other obligations are may obligations w.r.t. $k$.

If $s$ can reach $\neg p$ via an abstract path from level $n+1$ to level $k+1$, then $(s, n)$ is a must obligation: unless $s$ is blocked from $F_{n+1}$ (by removing from $F_n$ all states that reach $s$ in one step), an abstract counterexample of length $k+1$ would exist. The same violation may also be reached from $s$ in later levels $F_j$, $n+1 < j \leq k+1$, in which case it will be a suffix of a longer abstract counterexample with a longer prefix up to $s$. Therefore, we may also want to block $s$ in $F_j$, $n+1 < j \leq k+1$. However, since different abstract transition relations are considered at each level, it is also possible that the same path leading from $s$ to $\neg p$ is not valid from level $j > n+1$ since, for example, $U_j \supset U_{n+1}$ and hence the first transition along the path does not satisfy $TR_j$. In this case,
a longer counterexample is not a valid abstract path since its suffix is not valid.
The attempt to block a state \( s_a \) that is known to reach a violation from level \( n + 1 \) in levels greater than \( n + 1 \) creates may obligations\(^{5} \).

The may obligations are not \textit{required} to be blocked, but blocking them can prevent A-IC3 from encountering the same obligations/states in future iterations. On the other hand, if we report an abstract counterexample based on a may obligation, it is possible that no real abstract counterexample exists, resulting in an unnecessary refinement step which can damage the efficiency of the algorithm. We therefore greedily try to handle may obligations and strengthen \( \Omega \) accordingly, but refrain from reporting abstract counterexamples based on them. Note that in the latter case, if the may obligation is in fact a must w.r.t. some greater \( k \), then it will reappear as a must obligation in the following iterations.

In order to handle an obligation \((s_a, n)\) and show \( s_a \) to be unreachable from \( F_n \) in one step, A-IC3 attempts to strengthen \( F_n \) by extracting predecessors \( t_a \) of \( s_a \) that satisfy \( F_n \land TR_n \land s'_a \), defining new proof obligations based on them, and handling these obligations (by the same procedure). If \( F_n \) is successfully strengthened s.t. \( F_n \land TR_n \land s'_a \) becomes unsatisfiable, then \( \neg s_a \) becomes an invariant up to \( n + 1 \). \( s_a \) is blocked by strengthening \( F_0, \ldots, F_{n+1} \) accordingly.

\footnote{IC3 does not make a distinction between may and must obligations and handles them all the same since in the concrete case, a longer counterexample is always a valid path (its suffix reaching a violation is always valid).}

\footnote{A state \( s_a \) is represented by a conjunction of literals, which makes its negation \( \neg s_a \) a clause (i.e., a disjunction of literals). A sub-clause of \( \neg s_a \) consists of a subset of its literals.}

\footnote{\( c \) is not necessarily inductive w.r.t. \( M_i \) where \( i < n \) (in case \( U_i \subseteq U_n \)).}

\footnote{Adding Invariants If \( \neg s_a \) is an invariant up to \( n + 1 \), then a stronger invariant that blocks \( s_a \) up to \( F_{n+1} \) is learned based on the \textit{abstract model} \( M_n \). Namely, \( \neg s_a \) is strengthened to some sub-clause\(^{6} \) \( c \) s.t. \( F_0 \Rightarrow c \) and \( F_n \land c \land TR_n \Rightarrow c' \), i.e. \( c \) is inductive up to \( n \) w.r.t. \( M_n \) and hence, by Lemma 9.1, also w.r.t. \( M_k \). Consequently, \( c \) is also an invariant up to \( n + 1 \), but it is a stronger invariant than \( \neg s_a \) (since \( c \Rightarrow \neg s_a \)). The clause \( c \) is added as a conjunct to \( F_0, \ldots, F_{n+1} \) while maintaining the properties of a (concrete) RS\(^{7} \).}

Key procedures used by A-IC3 are described in Sec. 10.2.

10.2. Detailed Description of Strengthening

We now describe the procedures used by A-IC3 in detail.

\textbf{Strengthen (Fig. 8)}

\begin{quote}
**Strengthen** starts by checking \( F_k \land TR_k \land \neg p' \) (line 29). If it is unsatisfiable, then \( F_k \land TR \land \neg p' \) is unsatisfiable as well (since \( TR \Rightarrow TR_k \)). Thus \( \Omega \) is already an RS and no further strengthening is needed.

Assume \( F_k \land TR_k \land \neg p' \) is satisfiable. An abstract state \( s_a \in M_k \) that reaches \( \neg p \) in one abstract step is extracted from the satisfying assignment, meaning \( s_a \cap F_k \neq \emptyset \). All concrete states in \( s_a \cap F_k \) can reach \( \neg p \) via \( TR_k \) and therefore,
\end{quote}
28: function Strengthen($\Omega, C, k$)
29:     while $F_k \land TR_k \land \neg \psi' == SAT$ do
30:         obligations = $\emptyset$
31:         retrieve abstract predecessor $s_k$
32:         if BlockState($\Omega, s_k, k, \text{must}$) == abs-cex then
33:             return abs-cex
34:     end if
35:     while obligations $\neq \emptyset$ do
36:         $(s_a, n), \text{handleMay} = \text{ChooseNext}($obligations$)
37:         if $F_n \land TR_n \land s'_a == SAT$ then
38:             retrieve abstract predecessor $t_a$
39:             if BlockState($\Omega, t_a, n, k, \text{must}$) == abs-cex then
40:                 if handleMay then
41:                     obligations.clearAllMust()
42:                 else
43:                     return abs-cex
44:                 end if
45:             end if
46:         else
47:             obligations.removeMust($s_a, n$)
48:             BlockState($\Omega, s_a, n + 2, k, \text{may}$)
49:         end if
50:     end while
51:     end while
52:     PropagateClauses($\Omega$)
53:     return done
54: end function

Figure 8. Iterative strengthening of A-IC3

if the property is to be proven, $s_a$ must be blocked in $F_k$. Otherwise, an abstract counterexample exists.

In order to block $s_a$ in $F_k$, Strengthen calls BlockState on the bad state $s_a$ at level $k$ (line 32). BlockState either finds a counterexample or initializes the set(s) of obligations to reflect the need to block $s_a$ (and possibly adds invariants to the $F_i$’s).

Strengthen then handles the proof obligations one at a time. ChooseNext (line 36) first considers obligations from the must set only. Obligations are chosen in increasing order of their time frames. If the must set becomes empty, then as long as the may set is not empty, one may obligation with a minimal time frame is moved from the may set to the must set. Strengthen then continues, with the exception that counterexamples are no longer reported.

Given a proof obligation $(s_a, n)$:

- If $F_n$ can indeed reach $s_a$ in one (abstract) step, i.e., $F_n \land TR_n \land s'_a$ is satisfiable, then a predecessor $t_a$ of $s_a$ s.t. $t_a \cap F_n \neq \emptyset$ is extracted from the satisfying assignment (line 38). By Lemma 10.3, $t_a \cap F_i = \emptyset$ for all $i < n$. Thus $\neg t_a$ is an invariant up to $n - 1$. Next, the state $t_a$ needs to be blocked (eliminated) from level $l = n$ (line 39).
- When $F_n \land TR_n \land s'_a$ becomes unsatisfiable, the proof obligation $(s_a, n)$ is removed (line 47) since $s_a$ can no longer be reached from level $n$. In fact, $\neg s_a$ is now an invariant up to level $n + 1$. In order not to encounter $s_a$ in later iterations, we speculatively attempt to block (eliminate) $s_a$ from level
Lemma 10.3. Let \((s_a, n)\) be a proof obligation, and let \(t_a\) be an abstract state such that \((t_a, s_a) \models TR_n\). Then \(t_a \cap F_i = \emptyset\) for every \(i \leq n - 1\).

Proof. Let \((s_a, n)\) be a proof obligation. In particular, \(s_a \cap F_{n+1} \neq \emptyset\). Suppose further that \((t_a, s_a) \models TR_n\). We show that \(t_a \cap F_i = \emptyset\) for every \(i \leq n - 1\). Since \(F_i \subseteq F_{n-1}\) for every \(i \leq n - 1\), it suffices to show that \(t_a \cap F_{n-1} = \emptyset\).

At the beginning of the \(n - 1\)'th L-IC3-iteration (which added \(F_n\)), it was the case that \(s_a \cap F_n \neq \emptyset\). This is because \(F_n\) was initialized to \(p\), and clearly \(s_a \cap p \neq \emptyset\) (since \(F_{n+1} \subseteq p\) and \(s_a \cap F_{n+1} \neq \emptyset\)). On the other hand, at the current L-IC3-iteration \(s_a \cap F_n = \emptyset\), since for a proof obligation \((s_a, n)\), \(\neg s_a\) is an invariant up to \(n\).

This means that there is a set of clauses \(C\) that were added to \(F_n\) such that \(p \cap \bigcap C \cap s_a = \emptyset\) and for every \(c \in C\), \(s_a \nsubseteq c\) or equivalently \(s_a \cap \neg c \neq \emptyset\) (other clauses are not considered as they do not contribute to blocking \(s_a\) anyway). Every clause \(c\) added to \(F_n\) is inductive at some frame \(\geq n - 1\) (see Lemma ??). Therefore, for every \(c \in C\) there is a frame \(i_c \geq n - 1\) such that \(F_{i_c} \land c \land TR_{i_c} \Rightarrow c'\).

Furthermore, since \(s_a \cap F_{n+1} \neq \emptyset\), at least one of these clauses was not added to \(F_{n+1}\) which ensures that there is some \(c \in C\) such that \(i_c \leq n - 1\), i.e. \(i_c = n - 1\). We denote such a clause by \(c_0\). We have that \(F_{n-1} \land c_0 \land TR_{n-1} \Rightarrow c_0'\), or equivalently \(F_{n-1} \land c_0 \land TR_{n-1} \land \neg c_0' = UNSAT\).

Now assume to the contrary that \(t_a \cap F_{n-1} \neq \emptyset\). To reach a contradiction we first note that since \((t_a, s_a) \models TR_n\), it is also the case that \((t_a, s_a) \models TR_{n-1}\). Therefore by our assumption we have that \((t_a, s_a) \models F_{n-1} \land TR_{n-1} \equiv F_{n-1} \land c_0 \land TR_{n-1}\). The equivalence is since \(c_0\) is a clause in \(F_{n-1}\), together with the property that \(s_a \cap \neg c_0 \neq \emptyset\) (since \(c_0 \in C\) and by the choice of \(C\)), we have that \((t_a, s_a)\) is a satisfying assignment for \(F_{n-1} \land c_0 \land TR_{n-1} \land \neg c_0'\), in contradiction to the property that it is \(UNSAT\). 

As explained above, a counterexample found by BlockState is reported by Strengthen iff may obligations are not yet handled (lines 33 and 43).

Remark 1. Note that ignoring a counterexample reported by BlockState when it failed to block a may obligation \((s_a, n)\) does not compromise the correctness of the algorithm, since an RS up to level \(k+1\) is still obtained. Moreover, if \(s_a\) does reach a violation from level \(n+1\), which means that the same obligation is in fact required for the property to hold, then it will reappear as a must obligation in the following iterations. In fact, even if the abstract counterexample is a real abstract counterexample, it might be worth while to defer handling it. This is because it is possible that in later iterations, where the abstraction becomes more precise, it will cease to exist, whereas a pre-mature invocation of refinement, which traverses the concrete state space restricted by the \(F_i\)'s, might be costly.

BlockState (Fig. 9)

BlockState\((\Omega, t_a, l, k, type)\) is used for blocking a “bad state” \(t_a\) from level \(l\) up to \(k+1\), where \(\neg t_a\) is already known to be an invariant up to \(l-1\).
function BlockState(Ω, t_a, l, k, type)
    if l > k + 1 then
        min = k + 1
    else
        min = FindNonInductive(Ω, ¬t_a, l - 1, k)
        if min == 0 then
            return abs-cex
        end if
        if min <= k then
            if type == must && min == l - 1 then
                obligations.addMust(t_a, min)
            else
                obligations.addMay(t_a, min)
            end if
        end if
        AddInvariant(Ω, ¬t_a, min)
        return done
    end function

Figure 9. BlockState procedure of A-IC3

Note that if l > k (line 57) then t_a is already blocked up to k + 1. Thus
¬t_a is added as an invariant up to k + 1 (line 71). Otherwise, BlockState looks
for a level such that ¬t_a is invariant up to it.

Specifically, BlockState looks for the minimal level min between l − 1 and
k s.t. F_{min} \land TR_{min} \land t'_a is satisfiable (line 59) (meaning that t_a can be reached
in one step from min). The important property is that ¬t_a is an invariant up to
min: If min = l − 1, this holds since ¬t_a is already known to be an invariant up to
level l − 1 (this is also why the search for min starts at l − 1). If min > l − 1, then
the fact that F_{min-1} \land TR_{min-1} \land t'_a is unsatisfiable implies that ¬t_a is inductive
at min − 1 w.r.t. M_{min-1}, and hence, by Lemma 9.1 also w.r.t. M_c. Thus, it is
an invariant up to min.

If min = 0, then the “bad state” t_a is reachable from INIT in one step of
TR_0. Thus, an abstract counterexample is reported (line 61). If min = k + 1 then
no corresponding level was found up to k, i.e., ¬t_a is an invariant up to k + 1
and no new proof obligation is added. However, if min ≤ k is found then the pair
(t, min) is added as a new proof obligation (lines 64-68). Either way, ¬t_a is added
as an invariant up to min by calling AddInvariant (line 71). AddInvariant
learns an invariant that strengthens ¬t_a and adds it to F_0, ..., F_{min}.

Classifying obligations as may/must is performed in lines 64-68 of Block-
State. Note that only obligations of the form (t_a, l − 1) are must obligations.
The initial obligations generated by the call BlockState(Ω, s_a, k, k, type) in line
32 of Strengthen whose level is exactly k − 1 become must obligations. Later
on, only obligations of the form (t_a, n − 1) generated by the call to Block-
State(Ω, t_a, n, k, type) in line 39 of Strengthen when handling a must obligation
(s_a, n), where t_a is a predecessor of s_a, are considered must obligations. The rest
are may obligations.
AddInvariant

If for some state $t_a$, and some level $min \leq k + 1$, the formula $\neg t_a$ is an invariant up to level $min$, then AddInvariant (called from BlockState line 71) is used to add a strengthening of $\neg t_a$ to all $F_i$’s s.t. $j \leq min$. More precisely, $\neg t_a$ is strengthened to some subclause $\neg (i.e., a disjunction of literals). A subclause of $\neg$ is inductive w.r.t. $\neg M_{\text{min-1}}$ and hence, by Lemma 9.1, also w.r.t. $M_i$. Consequently, $c$ is also an invariant up to $min$, but it is a stronger invariant than $\neg t_a$ (since $c \Rightarrow \neg t_a$). The clause $c$ is added as a conjunct to $F_0, \ldots, F_{\text{min}}$ while maintaining the properties of a (concrete) RS$^9$. AddInvariant always finds a clause to add, since $\neg t_a$ itself satisfies the requirements.

***** formalize and prove that the sequence remains an RS. Let $\langle F_0, \ldots, F_k \rangle$ be an RS. Let $n \leq k$ and let $c$ be a clause that is an invariant up to $n + 1$ w.r.t. $M_i$. Then $\langle F'_0, \ldots, F'_k \rangle$ where $F'_i = F_i \land c$ for $i \leq n + 1$ and $F'_i = F_i$ otherwise is also an RS. *****

PropagateClauses

Similarly to IC3, if the main loop in Strengthen terminates, added clauses are propagated forward by PropagateClauses (line 52). Specifically, if $F_i \land c \land TR_i \land c' \land TR_{i + 1}$ is unsatisfiable then the clause $c$ from $F_i$ can safely be added to $F_{i + 1}$ while maintaining the properties of an RS. ***** add theorem/lemma? ***** This is done in order to get to a fixpoint.

10.3. Monotonicity of the Abstraction Sequence

Monotonicity of the abstraction sequence ensures that when A-IC3 attempts to block a state $t_a$ that reaches a violation at level $n$, then $\neg t_a$ is necessarily an invariant up until $n - 1$ (see Lemma 10.3). Recall that if some state $t_a$ reaches a violation from step $n$ along the abstract transitions, it is not guaranteed that the same violation can be reached from $t_a$ at level $i > n$. However, the fact that for each $i < n$, $U_i \subseteq U_a$, and as a result $TR_n \Rightarrow TR_i$, ensures that the same violation can be reached from $t_a$ at any level $i < n$. This ensures that $\neg t_a$ is an invariant up until $n - 1$, otherwise, the violation would have been found in previous iterations.

The same property does not hold if a non-monotonic abstraction sequence is used, which means that in this case deducing that $\neg t_a$ is an invariant up to $n - 1$ when attempting to block a state $t_a$ at level $n$ is simply incorrect.

Another motivation for the monotonicity of the abstraction sequence is the following. Recall that $F_i \subseteq F_{i + 1}$ for each $i$. This means that any state $t_a \in F_i$, and in particular states that reach a violation along some abstract path, will be encountered again in $F_{i + 1}$. As a result, the same information needed to show that $t_a \in F_i$ cannot reach a violation from level $i$ is likely to be needed to show that $t_a$ cannot reach a violation from $F_{i + 1}$ as well. Restricting the discussion

$^8$A state $t_a$ is represented by a conjunction of literals, which makes its negation $\neg t_a$ a clause (i.e., a disjunction of literals). A subclause of $\neg t_a$ consists of a subset of its literals.

$^9$Note that while $c$ is inductive w.r.t. $M_{\text{min-1}}$ up to $\text{min} - 1$, it is not necessarily inductive w.r.t. $M_i$ where $i < \text{min} - 1$ (in case $U_i \subset U_{\text{min-1}}$). Still, it is safely added to $F_{i+1}$ for $i < \text{min} - 1$ since it is an invariant w.r.t. $M_i$. 
74: \textbf{function} \texttt{Refine}(\Omega, \bar{U})
75:     result = \texttt{C-Strengthen}(\Omega)
76:     \textbf{if} result == cex \textbf{then}
77:         \textbf{return} cex
78:     \textbf{end if}
79:     \texttt{RefineAbstraction}(\Omega, \bar{U})
80:     \textbf{return} done
81: \textbf{end function}

Figure 10. Refine procedure of A-IC3

to monotonic abstraction sequences automatically ensures that if the abstract transition relations carry enough information to refute all violations starting at states from \(F_i\), then the same holds when considering the same states in \(F_{i+1}\). While it is possible that a different abstraction can be used to refute the existence of a violation from \(i + 1\), in most cases the effort of computing this abstraction (by invoking refinement multiple times) exceeds its potential benefit.

10.4. Refinement

If A-IC3 finds an abstract counterexample of length \(k + 1\), refinement is invoked by L-IC3 (line 7). Refinement either finds a concrete counterexample or eliminates all concrete spurious counterexamples of length \(k + 1\). In the latter case, refinement also refines \(U\) to ensure that no abstract counterexample of length \(k + 1\) exists. Is this true? seems like in the case that no concrete counterexample existed, no update will be done and there will still be an abstract counterexample after refinement. This doesn’t seem to damage correctness, but it makes part of the lemma incorrect. Either change the lemma or say that if no concrete counterexample exists, we update the abstraction of the last set to reflect that there is no transition to \(\neg p\). Both an updated RS \(\Omega^r = \langle F^r_0, \ldots, F^r_{k+1} \rangle\) and a refined monotonic abstraction sequence \(\bar{U}^r = \langle U^r_0, \ldots, U^r_{k+1} \rangle\) are returned.

The \texttt{Refine} procedure is described in Fig. 10. \texttt{Refine} first invokes \texttt{C-Strengthen}, the strengthening procedure of the concrete IC3, on the sequence \(\langle F_0, \ldots, F_{k+1} \rangle\) (whose prefix up to \(F_k\) is an RS) obtained from the abstract model checking. If a concrete counterexample is found the algorithm terminates (lines 75-78). Otherwise, no concrete counterexample of length \(k + 1\) exists. Moreover, the updated (strengthened) sets \(F^r_0, \ldots, F^r_{k+1}\) comprise an RS. It remains to refine the abstraction sequence \(\bar{U}\) in order to eliminate all abstract counterexamples of length \(k + 1\) as well. Thus, \texttt{RefineAbstraction} is invoked (line 79).

\texttt{RefineAbstraction}

A-IC3 found an abstract counterexample since it failed to strengthen the \(F_i\)’s. Meaning, the relevant \(i\)-reachability checks \(F_i \land TR \land t'\) could not be made unsatisfiable when using \(TR\). \texttt{C-Strengthen}, on the other hand, succeeds to do so. Namely, for each \(i\)-satisfiability check \(F_i \land TR \land t'\) of A-IC3 that was satisfiable, \texttt{C-Strengthen} manages to make the corresponding check \(F^r_i \land TR \land t'\) for each \(t \preceq t_a\) unsatisfiable, either by strengthening \(F^r_i\) or simply since it considers \(TR\). Moreover, once \(F^r_i \land TR \land t'\) becomes unsatisfiable, \texttt{C-Strengthen} derives from it a clause \(c \Rightarrow \neg t\) s.t. \(F^r_i \land c \land TR \Rightarrow c'\) holds. \texttt{C-Strengthen} strengthens \(\Omega^r\)
by adding $c$ (invariant) as a new clause in all sets up to $F_{i+1}'$. We consider it a
learned clause at level $i + 1$. The purpose of RefineAbstraction is to ensure
that for a learned clause $c$ at level $i + 1$, $F_i' \land c \land TR_i' \Rightarrow c'$ (with $TR_i'$ instead of
$TR$) also holds. Meaning, $c$ is inductive up to $i$ w.r.t. $M_i$.

**Lemma 10.4.** Let $c$ be a clause learned by C-STRENGTHEN at level $i + 1$. If $F_i' \land
TR_i' \Rightarrow F_{i+1}'$ then $F_i' \land c \land TR_i' \Rightarrow c'$.

Based on the previous lemma, in order to ensure $F_i' \land TR_i' \Rightarrow c'$, it suffices
to ensure unsatisfiability of $F_i' \land TR_i' \land \neg F_{i+1}'$ for every level $i + 1$ in which learned clauses exist.

***** refer to the case where there is no concrete counterexample to begin
with. I.e., $F_i' \land TR \land \neg p \Rightarrow UNSAT$. Check if it’s needed for correctness. What
is the harm in having an abstract counterexample if we ignore it? *****

To ensure unsatisfiability of a formula $F_i' \land TR_i' \land \neg F_{i+1}'$, we consider the
same formula over $TR$, which is clearly unsatisfiable. We derive from it an
unSAT-core. The next-state variables that appear in the unSAT-core, denoted
$NS(\text{unSatCore}) = \{ \nu \in V | \nu' \in \text{Vars(\text{unSatCore})} \}$, are added to $U_i$.

**Lemma 10.5.** Let $F_i' \land TR \land \eta'$ be an unsatisfiable formula and let UnSatCore be its
unsat core. Let $U_i' \supseteq NS(\text{UnSatCore})$. Then $F_i' \land TR_i' \land \eta'$ is unsatisfiable.

Finally, we propagate variables that were added to $U_i'$ forward in order to
obtain a monotonic abstraction sequence. Since we only add variables to $U_i'$, i.e.
made the transition relation $TR_i'$ more precise, then the corresponding formulas
remain unsatisfiable.

10.5. Correctness Arguments

The RS obtained by L-IC3 is concrete. Specifically, it does not necessarily satisfy
$F_i \land TR \Rightarrow F_{i+1}$. This results both from refinement that adds invariants learned
based on the concrete $TR$, and from A-IC3 that learns an invariant based on some
$TR_i$, but also adds it to $F_{i+1}$ for $j < i$ even if it is not inductive w.r.t. $TR_j$. This
complicates the correctness proof.

In particular, in IC3, when a proof obligation $(s, n)$ is handled, then for any
predecessor $t$ of $s, \neg t$ is an invariant up to $n - 1$, otherwise $s$ would belong to a
lower frame (since $F_i \land TR \Rightarrow F_{i+1}$). Now consider an abstract proof obligation
$(s_n, n)$. If we assume to the contrary that the predecessor $t_n$ intersects some $F_i$ (for
$i < n$) then we can still deduce that the transition $(t_n, s_n) \models TR_n$ also exists at
a lower frame, i.e. $(t_n, s_n) \models TR_n$ for $i < n$. This is since $TR_n \Rightarrow TR_i$ (recall that
the same does not necessarily hold for $i > n$). However, we cannot immediately
deduce that $s_n \land F_{i+1} \neq \emptyset$ since $F_i \land TR_i \Rightarrow F_{i+1}$ might not hold. It turns out
that this property does hold (see Lemma 10.3), but more complicated arguments
are needed, based on the following:

**Lemma 10.6.** Let $\Omega = \langle F_0, \ldots, F_k \rangle$ and $\bar{U} = \langle U_0, \ldots, U_k \rangle$ be the sequences ob-
tained at the end of the k’th iteration of L-IC3, i.e. either at the end of a refine-
ment step or at the end of an iteration of A-IC3 in the case that no counterex-
ample was found. Then
1. $\Omega$ is an RS.

2. For every clause $c$ that was added to some $F_i$ in $\Omega$ there exists some $j \geq i - 1$ s.t. $c$ is inductive up to $j$ w.r.t. $M_j$.

3. No abstract counterexample of length $k+1$ exists w.r.t. the prefix $\langle F_0, \ldots, F_k \rangle$ of $\Omega$.  

Proof. ***check terminology; iteration, stage, names of procedures***** The proof is inductive. Consider an iteration of L-IC3. It consists of an iteration of A-IC3 adding $F_{i+1}$, possibly followed by a refinement step.

We first show that for every clause $c$ that was added to some $F_i$ in $\Omega$ during the above, there exists some $j \geq i - 1$ s.t. $c$ is inductive up to $j$ w.r.t. $M_j$. An important property to note is that during the run of the algorithm, the $F_i$ sets, as well as the $TR_i$ transition relations are only strengthened. Therefore, if at some point during the algorithm a clause $c$ is inductive up to $j$ w.r.t. $M_j$ for some $j$, meaning that $F_j \land c \land TR_j \Rightarrow c'$ holds, then $c$ will remain inductive up to $j$ w.r.t. $M_j$ later on as well, since $F_j \land c \land TR_j \Rightarrow c'$ will keep holding with the strengthened sets. As a result, it suffices to show that every clause $c$ that was added to some $F_i$ in $\Omega$ is inductive up to some $j \leq i - 1$ w.r.t. $M_j$, at some point during the iteration.

If a clause $c$ is added by A-IC3, i.e., by calling $\text{AddInvariant}$ at some level $\min$, then $c$ is inductive at level $\min - 1$ when it is added. If a clause is added during refinement as a learned clause at level $i + 1$, then by the property of IC3, it is inductive up to $i$ w.r.t. $M_c$. Lemma 10.5 ensures that at the end of the refinement step it is also inductive up to $i$ w.r.t. (the refined) $M'_i$.

We now show that the obtained $\Omega$ is an RS. $F_0 = \text{INIT}$ holds due to the initialization. Similarly, $F_i \Rightarrow p$ holds due to the initialization of the $F_i$ sets to $p$, and due to the property that the sets are only strengthened later on. $F_i \Rightarrow F_{i+1}$ holds when $F_{i+1}$ is initialized (since it is initialized to $p$ and $F_i \Rightarrow p$), and continues to hold since any clause that is added to $F_{i+1}$ is also added to $F_i$. Finally, it remains to show that $F_i \land TR \Rightarrow F'_i$. We show this in two parts. First, we show that it holds at the end of the L-IC3 iteration (possibly including a refinement step) that added $F_{i+1}$ to the RS. Next, we show that later updates of $F_i$ and $F_{i+1}$ maintain this property.

To show that $F_i \land TR \Rightarrow F'_i$ holds at the end of the iteration that added $F_{i+1}$ to the RS, we recall that $F_{i+1}$ is initialized to $p$ and we note that the termination condition of an iteration of A-IC3 is that $F_i \land TR_i \land \neg p' = \text{UNSAT}$ (see $\text{Strengthen}$), meaning that $F_i \land TR_i \Rightarrow p' = F'_{i+1}$. Moreover, since $TR \Rightarrow$
The latter implies that \( F_i \land TR \Rightarrow p' = F'_{i+1} \). Similarly, the termination condition of the refinement step (if applicable) is that \( F_i \land TR \Rightarrow p' = F'_{i+1} \). To show that later updates of \( F_i \) and \( F_{i+1} \) maintain this property, we rely on the property that any clause added to \( F_{i+1} \) is inductive up to some \( j \geq i \) w.r.t. \( M_j \). This means, it is also inductive w.r.t. to \( M_c \). Therefore, \( F_i \land c \land TR \Rightarrow c' \). Since \( c \) is added both to \( F_i \) and to \( F_{i+1} \), the property \( F_i \land TR \Rightarrow F'_{i+1} \) is maintained. Note that adding clauses to \( F_i \) cannot damage the implication since it only strengthens the left hand side.

It remains to show that no abstract counterexample of length \( k + 1 \) exists w.r.t. the prefix \( \langle F_0, \ldots, F_k \rangle \) of \( \Omega \). Continue

In particular, this means that the clauses added to the last set of the sequence, \( F_{k+1} \), are inductive up to \( k \) w.r.t. \( M_k \), hence at the end of the \( k \)th iteration of L-IC3 adding \( F_{k+1} \) it holds that \( F_k \land TR_k \Rightarrow F_{k+1} \) (recall that the same does not necessarily hold for \( i < k \)).

Note that if a clause \( c \) is inductive up to \( j \) w.r.t. \( M_j \) at some point during the run of the algorithm, then it will remain inductive up to \( j \) w.r.t. \( M_j \) later on as well. This holds since \( F_j \) and \( TR_j \) are only strengthened during the run, therefore \( F_j \land c \land TR_j \Rightarrow c' \) keeps holding.

In the termination argument, need to argue that AddInvariant will not fail, since when it is called it is indeed called with an inductive invariant. This follows from Lemma ???. Make sure to include it somewhere. Need to think of proper organization.

Theorem 10.7. L-IC3 either terminates with a fixpoint, in which case the property holds, or with a concrete counterexample.

References


