Lazy Abstraction and SAT-Based Reachability in Hardware Model Checking

Yakir Vizel*, Orna Grumberg*, Sharon Shoham†
*Computer Science Department, The Technion, Haifa, Israel
†School of Computer Science, Academic College of Tel Aviv-Yaffo

Abstract—In this work we present a novel lazy abstraction-refinement technique for hardware model checking, integrated with the SAT-based algorithm IC3.

In contrast to most SAT-based model checking algorithms, IC3 avoids unrolling of the transition relation. Instead, it applies local checks, while computing over-approximated sets of reachable states. We find IC3 most suitable for lazy abstraction, since each one of its local checks requires different information from the checked model.

Similarly to IC3, our algorithm obtains a series of over-approximated sets of states. However, when constructing the series, different abstractions are used for different sets.

If an abstract counterexample is obtained, we either find a corresponding concrete one, or apply refinement to eliminate all counterexamples of the same length. Refinement makes the abstractions more precise as needed, and where needed. After refinement, the computation resumes from the same step where it was interrupted. The result is an incremental abstraction-refinement algorithm where the abstraction is lazy.

We implemented our algorithm, called L-IC3, and compared it with the original IC3 on large industrial hardware designs. We obtained significant speedups of up to two orders of magnitude.

I. INTRODUCTION

In this work we introduce a novel lazy abstraction-refinement technique for hardware model checking, integrated with the SAT-based algorithm IC3 [3].

Model checking [5] 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 [1, 3, 5, 6, 13] 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 [3] avoids such unrolling. To verify a safety property, IC3 gradually builds a series of sets of states \( F_0, \ldots, F_i, \ldots \), 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 [10], [13], 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 [11], 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 \( U_i \), 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 sequence \( \Omega \) 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 Ω 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.

A. Related Work

[6] and [2] suggest optimizations and extensions to IC3, but they do not combine it with a lazy abstraction-refinement mechanism ([6] suggests the use of abstraction for IC3 but without implementation details or results). In [14], [9], [7], [3], [8], SAT-based refinement is introduced. However, they use an unrolling of the model while we use local checks a-la IC3. Similarly to [14], [4], we also exploit an unSAT-core for refinement. However, we never unroll the model, while [14] does. Further, [14] is not incremental since after refinement it resumes its (abstract) model checking from time frame 0.

IC3 [3] 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 overapproximation-strengthening mechanism.

II. Preliminaries

Definition 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 $V' = \{v' | v \in V\}$ 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 $S = \{0, 1\}^V$, 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 $v'$.

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 $\eta$.

Definition 2. An over-approximated reachability sequence (OARS) with respect to a model $M$ and a property $AGp$, denoted $\Omega(M, p)$, is a sequence $(F_0, \ldots, F_k)$ of propositional formulas over $V$ such that the following holds:

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

The set of states represented by $F_i$ over-approximates the states 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 $\Omega$.

Definition 3. Let $\Omega$ be an OARS. 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 OARS, $\eta$ is an invariant up to $j$ iff it is inductive up to level $j - 1$, and in addition $F_0 \Rightarrow \eta$ (initialization).

A. SAT-based Reachability via IC3

IC3 [3] 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 OARS of length $k$ is extended to an OARS of length $k + 1$ 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 [3].
IC3 starts by checking if \( \text{INIT} \land \neg p \) or \( \text{INIT} \land TR \land \neg p' \) is satisfiable, in which case a counterexample of length zero or one is found and the algorithm terminates. If both are unsatisfiable, \( F_0 \) is initialized to \( \text{INIT} \) and \( F_1 \) is initialized to \( \neg p \). \( (F_0, F_1) \) is an OARS (it satisfies the conditions in Def. [3]).

IC3 extends and updates \( \Omega \), while strengthening the \( F_i \)'s. The 0th iteration starts from an OARS \( (F_0, \ldots, F_k) \). 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 if \( F_k \land TR \land \neg p' \) 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 \( \text{INIT} \), 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 OARS 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 \( s \) (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 \( F_i \land TR \land s' \). During this process, the \( F_i \)'s are strengthened by removing bad states that are not reachable if a state in \( F_0 = \text{INIT} \) is reached during the backwards traversal, then a counterexample is obtained.

**Definition 4.** Satisfiability checks of the form \( F_i \land TR \land \eta \) (where \( Vars(\eta) \subseteq V \lor V' \)) are called \( i \)-reachability checks.

**B. Abstraction**

Throughout the paper we consider the “visible variables” abstraction [11]. Let \( M_c = (V, U, INIT, TR) \) 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 \( TR_i \mid_{V_i} = \bigwedge_{v \in U_i} (v' = f_v(V, V')) \) is an abstract transition relation, and \( V_i = \{ v \in V \mid v \in Vars(TR_i) \lor v' \in Vars(TR_i) \} \subseteq V \). 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 OARS \( \Omega(M_c, p) = (F_0, \ldots, F_k) \) 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 5.** 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 \( M_c \).

**C. Lazy Abstraction**

As mentioned above, lazy abstraction [10] allows to use different details of the model at different iterations of the state-space traversal. We adapt the notion of lazy abstraction to abstraction based on visible variables [11], and allow different variables to be visible at different time frames.

**Definition 6.** An abstraction sequence w.r.t. a model \( M_c \) is a sequence \( \hat{U} = (U_0, \ldots, U_k) \) where \( U_i \subseteq U \) for \( 0 \leq i \leq k \), is a set of visible state-variables. \( \hat{U} \) is monotonic if \( U_i \subseteq U_{i+1} \) for each \( 0 \leq i < k \).

An abstraction sequence \( \hat{U} \) 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. [11-B]. If \( \hat{U} \) is monotonic, the induced sequence of abstract models is also monotonic in the sense that \( M_0 \preceq \cdots \preceq M_k \preceq M_c \).

**Definition 7.** Let \( \hat{U} = (U_0, \ldots, U_k) \) be a monotonic abstraction sequence and \( \Omega(M_c, p) = (F_0, \ldots, F_k) \) an OARS. A sequence \( s_1, \ldots, s_j \) 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 (ii) for each \( i \leq l \leq \min\{j, k\} \), \( s_j \cap F_l \neq \emptyset \).

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 \( F_i \).

**Definition 8.** 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 \( i \).

**III. 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 \( \hat{U} \) that induces different abstractions at different time frames as well as an OARS \( \Omega \).

L-IC3 starts with an initialization step and then works in stages (Fig. 1). 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 \( \langle F_0 = INIT, F_1 = p \rangle \), it initializes \( \bar{U} \) to \( \langle U_0 = \text{Vars}(p) \rangle \). Clearly, after initialization, \( \Omega \) is an OARS.

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 extends and updates the OARS \( \Omega \) until either a fixpoint is reached, or an abstract counterexample is found (line 6). In the latter case, the counterexample \( \bar{U} \) either finds an abstract counterexample (line 22), or reaches a fixpoint (line 7). If the refinement finds a concrete counterexample of length \( k \), \( \bar{U} \) is then performed (line 8). If the refinement finds a concrete counterexample, \( \Omega \) resumes from iteration \( k + 1 \) (line 14).

**Iterations** In iteration \( k \geq 1 \), the OARS \( \langle F_0, \ldots, F_k \rangle \) and the abstraction sequence \( \langle U_0, \ldots, U_{k-1} \rangle \) are extended by 1 and updated as follows (see Fig. 2).

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 \( F_0, \ldots, F_k \) are strengthened iteratively until \( \langle F_0, \ldots, F_{k+1} \rangle \) becomes an OARS, or an abstract counterexample is found.

Below we describe items 2 and 3 in more detail.

**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.

**Iterative Strengthening of \( \Omega \)**: A-IC3 obtains an OARS of length \( k + 1 \) by strengthening the \( F_i \)’s s.t. no abstract counterexample of length \( k+1 \) exists w.r.t. the OARS \( \langle F_0, \ldots, F_k \rangle \). This is a sufficient condition to ensure that \( \Omega \) is an OARS. For this purpose, A-IC3 finds 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. 2). Technically, such states are described by abstract proof obligations (similarly to the notion of proof obligations used in IC3).

**Definition 9.** An abstract proof obligation, or an obligation in short, is a pair \((s_a, n)\) consisting of a level \( n \leq k \) and an abstract state \( s_a \) s.t. (1) \( s_a \) is a “bad state” that reaches \( \neg p \) along some abstract path, (2) \( \neg s_a \) is an invariant up until \( n \), (3) \( s_a \cap F_{n+1} \neq \emptyset \), and (4) \( F_n \) reaches \( s_a \) in one step of \( TR_n \).

Thus \( n + 1 \) is the minimal level intersecting \( s_a \), and \( n \) is the minimal level reaching \( s_a \) in one abstract step. Note that it is

![Fig. 1: L-IC3](image1)

![Fig. 2: A-IC3](image2)
possible that \( F_n \) cannot reach \( s_a \) along the concrete transitions. A-IC3 maintains two sets of obligations - may and must.

**Definition 10.** An obligation \((s_a, n)\) is a must obligation w.r.t. iteration \( k \) if \( s_a \) 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 \( s_a \) can reach \( \neg p \) via an abstract path from level \( n + 1 \) to level \( k + 1 \), then \((s_a, n)\) is a must obligation: unless \( s_a \) is blocked from \( F_{n+1} \) (by removing from \( F_{n} \) all states that reach \( s_a \) in one step), an abstract counterexample of length \( k + 1 \) would exist. The same violation may also be reached from \( s_a \) 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 \( s_a \). Therefore, we may also want to block \( s_a \) 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 \( s_a \) 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 \( TR_j \). 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. 

The may obligations are not 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 if a 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 \( F_n \land TR_n \land s_i' \), defining new proof obligations based on them, and handling these obligations (by the same procedure). If \( F_n \) is successfully strengthened s.t. \( F_n \land TR_n \land s_i' \) becomes unsatisfiable, then \( \neg s_a \) becomes an invariant up to \( n + 1 \).

**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 abstract model \( M_n \). Namely, \( \neg s_a \) is strengthened to some sub-clause\(^4\) of \( c \) s.t. \( F_0 \Rightarrow c \) and \( F_n \land c \land TR_n \Rightarrow \neg c \), i.e. \( c \) is inductive up to \( n \) w.r.t. \( M_n \) and hence, by Lemma\(^5\) also w.r.t. \( M_i \). 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) OARS\(^6\).

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

**B. 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. Both an updated OARS \( \Omega'' = \langle F_0', \ldots, F_{k+1}' \rangle \) and a refined monotonic abstraction sequence \( U'' = \langle U_0', \ldots, U_k' \rangle \) are returned.

The Refine procedure is described in Fig. 3. Refine first invokes 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 OARS) obtained from the abstract model checking. If a concrete counterexample is found the algorithm terminates (lines \( 29-32 \)). Otherwise, no concrete counterexample of length \( k + 1 \) exists. Moreover, the updated (strengthened) sets \( F_0', \ldots, F_{k+1}' \) comprise an OARS. It remains to refine the abstraction sequence \( U \) in order to eliminate all abstract counterexamples of length \( k + 1 \) as well. Thus, RefineAbstraction is invoked (line \( 33 \)).

**RefineABSTRACTION**

A-IC3 found an abstract counterexample since it failed to strengthen the \( F_i \)’s. Meaning, the relevant \( i \)-reachability checks \( F_i \land TR_i \land t_a' \) could not be made unsatisfiable when using \( TR_i \). C-STRENGTHEN, on the other hand, succeeds to do so. Namely, for each \( i \)-satisfiability check \( F_i \land TR_i \land t_a' \) of A-IC3 that was satisfiable, C-STRENGTHEN manages to make the corresponding check \( F_i' \land TR \land t' \) for each \( t \leq t_a' \) unsatisfiable, either by strengthening \( F_i' \) or simply since it considers \( TR \). Moreover, once \( F_i' \land TR \land t' \) becomes unsatisfiable, C-STRENGTHEN derives from it a clause \( c \Rightarrow \neg t \) s.t. \( F_i' \land c \land TR \Rightarrow c' \) holds. C-STRENGTHEN strengthens \( \Omega' \) 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 \( i + 1 \), if \( F_i' \land c \land TR_i' \Rightarrow \neg c' \) (with \( TR_i' \) instead of \( TR \)) also holds. Meaning, \( c \) is inductive up to \( i \) w.r.t. \( M_i \).

**Lemma 11.** Let \( c \) be a clause learned by C-STRENGTHEN at level \( i + 1 \). If \( F_i' \land c \land TR_i' \Rightarrow F_{i+1}' \land c \land TR_i' \Rightarrow c' \).

Based on the previous lemma, in order to ensure \( F_i' \land c \land TR_i' \Rightarrow c' \), 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.

\(^4\)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).

\(^5\)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.

\(^6\)\( c \) is not necessarily inductive w.r.t. \( M_i \) where \( i < n \) (in case \( U_i \subset U_n \)).
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 \( NS(\text{unsatCore}) = \{ v \in V \mid v' \in \text{Vars}(\text{unsatCore}) \} \), are added to \( U_i \).

**Lemma 12.** Let \( F_i \land TR_i \land \eta_i \) be an unsatisfiable formula and let UnSatCore be its unsat core. Let \( U'_i \supseteq NS(\text{UnSatCore}) \). Then \( F_i \land TR_i \land \eta_i \) 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. make the transition relation \( TR_i \) more precise, then the corresponding formulas remain unsatisfiable.

**C. Correctness Arguments**

The OARS obtained by L-IC3 is concrete. Specifically, it does not necessarily satisfy \( F_i \land TR_i \Rightarrow F_{i+1} \). 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. \( TR_i \). 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 \( F_i \land TR_i \Rightarrow F_{i+1} \)). Now consider an abstract proof obligation \((s,a,n)\). If we assume to the contrary that the predecessor \( a \) intersects some \( F_i \) (for \( i < n \)) then we can still deduce that the transition \((a,s) \models TR_n\) also exists at a lower frame, i.e. \((a,s) \models TR_i\) for \( i < n \). This is since \( TR_n \Rightarrow TR_i \) (recall that the same does not necessarily hold for \( i > n \)). However, if \( a \otimes F_i \neq \emptyset \), we cannot immediately deduce that \( s \otimes F_{i+1} \neq \emptyset \) since \( F_i \land TR_i \Rightarrow F_{i+1} \) might not hold. It turns out that this property does hold (see Lemma 15), but more complicated arguments are needed, based on the following:

**Lemma 13.** Let \( \Omega = \langle F_0, \ldots, F_k \rangle \) and \( \bar{U} = \langle U_0, \ldots, U_k \rangle \) be the sequences obtained at the end of a refinement step or at the end of an iteration of A-IC3 in the case that no counterexample was found. Then

1) \( \Omega \) is an OARS.
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 \( \Omega \).

**Theorem 14.** L-IC3 either terminates with a fixpoint, in which case the property holds, or with a concrete counterexample.

**D. Detailed Description of Strengthening**

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

**STRENGTHEN (Fig. 4).** STRENGTHEN starts by checking \( F_n \land TR_n \land \neg p' \) (line 37). If it is unsatisfiable, then \( F_n \land TR_n \land \neg p' \) is also unsatisfiable as well (since \( TR \Rightarrow TR_n \)). Thus \( \Omega \) is already an OARS and no further strengthening is needed.

Assume \( F_i \land TR_i \land \neg p' \) 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 \( s_a \otimes F_k \neq \emptyset \). All concrete states in \( s_a \otimes F_k \) can reach \( \neg p \) via \( TR_k \) and therefore, 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 40). 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. CHOOSE_NEXT (line 43) 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. \( t_a \otimes F_n \neq \emptyset \) is extracted from the satisfying assignment (line 46). By Lemma 15 \( t_a \otimes F_i = \emptyset \) for all \( i < n \). Thus \( -t_a \) is an invariant up to \( n-1 \). Next, the state \( t_a \) needs to be blocked (eliminated) from level \( l = n \) (line 47).
- When \( F_n \land TR_n \land s'_a \) becomes unsatisfiable, the proof obligation \((s_a,n)\) is removed (line 55) since \( s_a \) can no longer be reached from level \( n \). In fact, \( -s_a \) 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 \( l = n + 2 \), while using the may parameter (line 56). A counterexample found by BLOCKSTATE is reported iff may obligations are not yet handled (lines 44 and 51).
Lemma 15. 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 \(i \leq n - 1\).

**BlockState** (Fig. 5): **BlockState**\((\Omega, t_a, l, k, \text{type})\) 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 \(l - 1\).

Note that if \(l > k + 1\) (line 65) then \(t_a\) is already blocked up to \(k + 1\). Thus \(\neg t_a\) is added as an invariant up to \(k + 1\) (line 79). Otherwise, **BlockState** looks for a level such that \(\neg t_a\) is invariant up to it.

Specifically, **BlockState** looks for the minimal level \(\text{min}\) between \(l - 1\) and \(k\) s.t. \(F_{\text{min}} \land TR_{\text{min}} \land t_a\) is satisfiable (line 67). The important property is that \(\neg t_a\) is an invariant up to \(\text{min}\): If \(\text{min} = l - 1\), this holds since \(\neg t_a\) is already known to be an invariant up to level \(l - 1\) (this is also why the search for \(\text{min}\) starts at \(l - 1\)). If \(\text{min} > l - 1\), then the fact that \(F_{\text{min}} \land TR_{\text{min}} \land t_a\) is unsatisfiable implies that \(\neg t_a\) is inductive at \(\text{min} - 1\) w.r.t. \(M_{\text{min}} - 1\), and hence, by Lemma 5 also w.r.t. \(M_c\). Thus, this is an invariant up to \(\text{min}\).

If \(\text{min} = 0\), then the “bad state” \(t_a\) is reachable from \(\text{INIT}\) in one step of \(TR_0\). Thus, an abstract counterexample is reported (line 69). If \(\text{min} = k + 1\) then no corresponding level was found up to \(k\), i.e., \(\neg t_a\) is an invariant up to \(k + 1\) and no new proof obligation is added. However, if \(\text{min} \leq k\) is found then the pair \((t, \text{min})\) is added as a new proof obligation (lines 72 to 76). Either way, \(\neg t_a\) is added as an invariant up to \(\text{min}\) by calling **AddInvariant** (line 79). **AddInvariant** learns an invariant that strengthens \(\neg t_a\) and adds it to \(F_0, \ldots, F_{\text{min}}\).

Classifying obligations as may/must is performed in lines 72 to 76 of **BlockState**. Note that only obligations of the form \((t_a, l - 1)\) are must obligations.

**PropagateClauses**: Similarly to IC3, if the main loop in **Strengthen** terminates, added clauses are propagated forward by **PropagateClauses** (line 60). Specifically, if \(F_i \land c \land TR_i \land \neg c'\) is unsatisfiable then the clause \(c\) from \(F_i\) can safely be added to \(F_{i+1}\) while maintaining the properties of an OARS. This is done in order to get to a fixpoint.

### IV. Experimental Results

For the implementation of the two algorithms we collaborated with **Jasper Design Automation**. We used Jasper’s formal verification platform in order to implement both the original IC3 and our L-IC3 algorithm. In both implementations we used optimizations from **[6]** (such as ternary simulation). Implementing these algorithms using Jasper’s platform allowed us to develop and experiment with various real-life industrial designs and properties from various major semiconductor companies. All designs contain thousands of state variables in the cone of influence of the properties.

The timeout was set to 3600 seconds and experiments were conducted on systems with Intel Xeon X5660 running at 2.8GHz and 24GB of main memory.

We experimented with 122 real safety properties from different designs. Fig. 6 shows two different analyses comparing the runtime of L-IC3 and IC3. Runtime trends are shown in Fig. 6a. As can be seen, the overall trend is in favor of L-IC3. In Fig. 6b runtime for IC3 and L-IC3 is represented by the X-axis and Y-axis respectively. We can clearly see the advantage of using L-IC3 on the more complicated test cases. These test cases are represented by the dots that are below the diagonal by a big margin. On these examples, the improvement in runtime is up to two orders of magnitude. The cases where IC3 performs better are usually cases where L-IC3 spends most of the time in refinement. Also, for false properties (counterexample exists), the performance of L-IC3 is affected by the way we treat may and must obligations. Due to our special handling, L-IC3 may lose the ability to
TF stands for the time frames and of variables in refined along increasing time frames. Different frames contain IC3 is 75558 seconds while for L-IC3 it is 55424 seconds. but are solved by L-IC3; 5 properties cannot be solved by L-
the scatter at the middle is a bunch of comparable properties where both algorithms are on par.

<p>| TABLE I: Lazy abstraction. N stands for the name of the verified property. Vars stands for the number of state variables in the concrete model M_c. |</p>
<table>
<thead>
<tr>
<th>N</th>
<th>Vars</th>
<th>lazy</th>
<th>TF</th>
<th>2AV</th>
<th>3AV</th>
<th>4AV</th>
<th>5AV</th>
<th>6AV</th>
<th>7AV</th>
<th>8AV</th>
<th>9AV</th>
</tr>
</thead>
</table>

find a counterexample which is longer than the length of the computed Ω. In those cases, IC3 may perform better. Note that the scatter at the middle is a bunch of comparable properties where both algorithms are on par.

In the given timeout, 7 properties cannot be solved by IC3 but are solved by L-IC3; 5 properties cannot be solved by L-IC3 but are solved by IC3. There are also 5 properties that cannot be solved by either algorithm. The overall runtime for IC3 is 75558 seconds while for L-IC3 it is 55424 seconds.

The laziness of our abstraction-refinement algorithm is demonstrated in Table I. The table shows how the abstraction is refined along increasing time frames. Different frames contain different variables that are needed in order to prove or disprove the given property. This demonstrates the fact that L-IC3 indeed takes advantage of the lazy abstraction framework.

Table II presents runtime characteristics for L-IC3 and IC3. In particular, it shows the number of clauses and the number of variables in Ω when either a fixpoint or a counterexample is found. In many of the examples the number of clauses produced by L-IC3 for its Ω is significantly smaller than the number of clauses produced by IC3. Recall that each of the clauses is learned via several local reachability checks. The reduced number of clauses thus indicates that L-IC3 applies a smaller number of calls and therefore issues a smaller number of calls to the SAT solver. This can explain the speedups it obtains.

An additional reason for the speedups is the fact that the local reachability checks of L-IC3 are easier than those of IC3. This is because the abstract transition relations TR_i are much smaller (in number of variables) than TR (see Table I). Further, the sets F_i computed by L-IC3 are smaller than those computed by IC3 (see Table II).

Recall that in Section III-A we distinguish between must and may obligations. The results reported above are obtained while using this distinction and handling all the may obligations after the must obligations, as described there. We also tried other configurations. For example, we ran experiments that do not distinguish between must and may obligations. Our experiments show that distinguishing between the two yields a better overall performance.

In addition to the industrial experiments, we also ran experiments on the HWMCC’11 benchmark. We used the testcases with single properties. Most of the properties in this benchmark are fairly easy and can be solved in a matter of a few seconds both by IC3 and L-IC3. There are also a few cases where IC3 performs better or even reaches a result while L-IC3 does not. In these cases L-IC3 spends most of the time in refinement. On the other hand, there are several test cases that can only be solved by L-IC3 while IC3 reaches timeouts.

### V. ACKNOWLEDGMENTS

The authors would like to thank Håkan Hjort, Ziyad Hanna and Yael Meller for valuable comments. Jasper Design Automation is thanked for the help in conducting the experiments.
REFERENCES