Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR)

Johannes Birgmeier[1⋆], Aaron R. Bradley[2⋆⋆], and Georg Weissenbacher[1⋆]

[1] Vienna University of Technology
[2] Mentor Graphics

Abstract. Typical CEGAR-based verification methods refine the abstract domain based on full counterexample traces. The finite state model checking algorithm IC3 introduced the concept of discovering, generalizing from, and thereby eliminating individual state counterexamples to induction (CTIs). This focus on individual states suggests a simpler abstraction-refinement scheme in which refinements are performed relative to single steps of the transition relation, thus reducing the expense of refinement and eliminating the need for full traces. Interestingly, this change in refinement focus leads to a natural spectrum of refinement options, including when to refine and which type of concrete single-step query to refine relative to. Experiments validate that CTI-focused abstraction refinement, or CTIGAR, is competitive with existing CEGAR-based tools.

1 Introduction

IC3 [10, 9] constructs an inductive proof of an invariance property by reacting to individual states. These states, called counterexamples to induction (CTIs), arise as counterexample models to one-step consecution queries: a CTI is not yet known to be unreachable and has at least one successor that either is or can lead to an error state. In focusing on states and single steps of the transition relation, IC3 differs from the k-induction [23] and interpolation [35, 36] extensions of BMC [7], which fundamentally rely on unrolling the transition relation. IC3's practical value is now widely appreciated.

This paper suggests a similar refocusing from sequences to single steps of the transition relation when performing predicate abstraction-refinement-based analysis of infinite state systems. The new method is referred to as CTIGAR,

⋆ Supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grants PROSEED, ICT12-059, and VRG11-005.

⋆⋆ This material is based upon work supported in part by the National Science Foundation under grants No. 0952617 and No. 1219067 and by the Semiconductor Research Corporation under contract GRC 1859. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

for counterexample to induction-guided abstraction-refinement, to contrast with CEGAR's focus on counterexample traces [18, 19].

Injecting predicate abstraction into IC3 is straightforward: an abstract CTI is a conjunction of the predicates that are satisfied by the corresponding concrete CTI. This straightforward and inexpensive abstraction contrasts with previous adaptations of IC3 to infinite state analysis that put excessive effort into computing non-trivial underapproximations of preimages [16, 31]. Because IC3's inductive generalization procedure typically expands the (abstract) CTI cube well beyond a preimage, there is little point in making such an effort.[3]

In CEGAR, failure to concretize an abstract counterexample trace of arbitrary length is the trigger for domain refinement. In CTIGAR, there are two triggering situations for domain refinement, both over single-step queries: for lifting or for consecution. This focus on single-step queries rather than traces contrasts with a recent attempt at combining CEGAR with IC3 [17].

Lifting [15] a full state to a partial assignment is an important generalization mechanism in state-of-the-art IC3 implementations. The partial assignment describes similar states that also step into the same target as the original full state. With a concrete CTI, the one-step lifting query must succeed [15]; however, with an abstract CTI, it can fail. A failure is one possible point for refinement.

Consecution relative to frame F_i, which over-approximates the set of states reachable in at most i steps, for CTI s checks whether any s-state is reachable from an F_i-state other than an s-state. It can happen that an abstract CTI $\hat{s}$ fails consecution while its corresponding concrete CTI s passes consecution. This situation is another possible point for refinement.

In both scenarios, one can eagerly address the failure or lazily ignore it and continue. Lazy operation allows the introduction of spurious transitions into the partially constructed traces. The corresponding CTIs are marked as having allowed such transitions and can be revisited later if necessary. Morever, in both cases, addressing a failure requires only looking at a one-step concrete query, not an arbitrarily long unwinding of the transition relation. When the underlying theory admits interpolation, an interpolant derived from the concrete query enriches the domain sufficiently so that the refined abstract CTI passes its query.

Overall, then, the characteristics of CTIGAR are as follows:

1. *straightforward abstraction*: an abstract CTI is derived from a concrete CTI by evaluating the available predicates over the (possibly partial) assignment of the concrete CTI;
2. *intermediate refinement triggers*: refinement is suggested either when lifting an abstract CTI fails or when consecution against an abstract CTI fails but against the corresponding concrete CTI succeeds;

[3] Abstract CTIs constitute underapproximate preimages whenever the abstract domain is sufficiently precise (Section 3.1), which can be enforced. Our experiments in Section 4, however, show that abstract CTIs that are not underapproximate preimages can be eliminated without costly refinement in many cases. In fact, the best experimental configurations do not enforce preimages.

3. *lazy or eager modes*: a suggested refinement can be addressed immediately (eager), ignored in hopes that the overall analysis succeeds or until the discovery of a counterexample trace (lazy), or delayed until an intermediate trigger, such as encountering a threshold number of suggested refinements;
4. *simple refinement*: refinement considers one step of the transition relation rather than an arbitrarily long unwinding;
5. *explicit concrete states*: the concrete CTIs that are derived from SMT models can be useful for some types of predicate synthesis; see Section 4.1.

CTIGAR otherwise operates identically to finite state IC3, except that an SMT solver is used in place of a SAT solver, and atoms are predicates.

This paper is organized as follows. In Section 2, basic concepts and IC3 are recalled. Section 3 presents CTIGAR; Section 4 evaluates CTIGAR empirically. Finally, Section 5 discusses CTIGAR in a broader context.

2 Preliminaries

2.1 Formulas and Transition Relations

The term *formula* refers to either a propositional logic formula or a formula in first-order logic.

Propositional Formulas. A propositional formula is defined as usual over a set X of propositional atoms, the logical constants $\top$ and $\bot$ (denoting true and false, respectively), and the standard logical connectives $\wedge$, $\vee$, $\rightarrow$, and $\neg$ (denoting conjunction, disjunction, implication, and negation, respectively). A *literal* is an *atom* $x \in X$ or its negation $\neg x$. A *clause* C is a set of literals interpreted as a disjunction. A *cube* is the negation of a clause.

First-Order Logic. The logical connectives from propositional logic carry over into first-order logic. First-order terms are constructed as usual over a set of variables V, functions, and constant symbols. An atom in first-order logic is a predicate symbol applied to a tuple of terms.

Semantics and Satisfiability. A *model* of a formula consists of a non-empty domain and an interpretation that assigns a denotation to the predicate, function, and constant symbols. A formula is *satisfiable* if there is some model under which it is true, and unsatisfiable otherwise. A formula F implies another formula G, denoted $F \Rightarrow G$, if every model of F is a model of G. Given a conjunction, an *unsatisfiable core* is a subset of the conjuncts that is unsatisfiable.

Theories. A first-order *theory* is defined by a *signature*, which is a fixed set of function and predicate symbols, and a set of *axioms* restricting the models under consideration to those that satisfy the axioms. Symbols that do not occur in the axioms are called uninterpreted and interpreted otherwise. *Quantifier free linear arithmetic (QFLIA/QFLRA)* is the theory for the first order language over the functions $+$, $-$, the predicates $<$ and $=$, and the constants $0, 1, \ldots$ interpreted over either the integers $\mathbb{Z}$ or the rational numbers $\mathbb{Q}$.

Transition Systems. Let X be a fixed set of uninterpreted symbols representing the state variables or registers (in either propositional or first-order logic). A state s is an interpretation mapping X to elements of the domain. The symbolic representation of the state s is a cube that is true under s but false in all other states. Depending on the context, s may denote a state or its symbolic counterpart. A formula represents the set of states in which it evaluates to true. $I(X)$ and $P(X)$ are used to represent the initial and the safe states of a transition system, respectively. Given X, let X' be a corresponding set of primed symbols, and let A' be the formula obtained by replacing the symbols X in a formula A with the corresponding symbols in X'. Z is a set of symbols used to encode primary inputs (which may be introduced to "determinize" a non-deterministic choice). A transition relation $T : (X \cup Z) \times X'$ associates states s to their successor states t' under an input assignment z.

A formula S (representing a set of states) satisfies *consecution* if $S \wedge T \Rightarrow S'$. S satisfies consecution *relative to* a formula G if $G \wedge S \wedge T \Rightarrow S'$. A formula S satisfies *initiation* if $I \Rightarrow S$, i.e., if the corresponding set of states contains all initial states.

2.2 IC3 for Finite State Transition Systems

IC3 maintains a growing sequence of *frames* $F_0(X), \ldots, F_k(X)$ satisfying the four invariants to the right. Each frame F_i over-approximates the states reachable from I in i or fewer steps (due to invariants 1, 2, and 4).

$$I \Leftrightarrow F_0 \quad (1)$$
$$\forall 0 \leq i < k \,.\, F_i \Rightarrow F_{i+1} \quad (2)$$
$$\forall 0 \leq i \leq k \,.\, F_i \Rightarrow P \quad (3)$$
$$\forall 0 \leq i < k \,.\, F_i \wedge T \Rightarrow F'_{i+1} \quad (4)$$

IC3 aims at finding either a counterexample to safety or an inductive invariant F_i such that $F_i \Leftrightarrow F_{i+1}$ for some level $0 \leq i < k$. Until this goal is reached, the algorithm alternates between two phases:

- If no bad state is reachable from the *frontier* F_k (i.e., $F_k \wedge T \Rightarrow P$), then k is increased, and the new frontier is initialized to P. Furthermore, consecution is checked for each clause in each frame, and passing clauses are pushed forward. Otherwise, IC3 adds a $\neg P$-predecessor s as *proof obligation* $\langle s, k-1 \rangle$.
- IC3 processes a queue of proof obligations $\langle s, i \rangle$, attempting to prove that the state s that is backwards reachable from $\neg P$ is unreachable from F_i. This attempt succeeds if IC3 finds a clause $c \subseteq \neg s$ satisfying consecution relative to F_i (i.e., $F_i \wedge c \wedge T \Rightarrow c'$), in which case the frames $F_1, \ldots, F_{i+1}$ are strengthened by adding c.[4] Otherwise, the failed consecution query reveals a predecessor t of s. If $i = 0$ and $t \wedge I$ is satisfiable, then t provides the initial state of a counterexample. Otherwise, a new proof obligation $\langle t, i-1 \rangle$ is added.

For a more detailed introduction to IC3, the reader is referred to [10, 11]. Proof obligations are the focus of the extension of IC3 to infinite state transition systems, presented in the next section.

[4] To initiate forward propagation and in anticipation that s will be rediscovered at a higher level, $\langle s, i+1 \rangle$ is added as a proof obligation unless $i = k$.

3 CTIGAR

CTIGAR is the natural abstraction-refinement extension of IC3 to infinite state systems. Not only does much of the algorithmic flow remain the same, but the extra abstraction-refinement machinery follows IC3 in spirit: it performs one-step incremental refinements in response to CTIs. This section presents CTIGAR within the known framework of IC3, first expanding IC3 concepts as necessary, then presenting CTIGAR's handling of extended proof obligations, and finally discussing when and how domain refinement is accomplished.

3.1 CTIGAR Extensions of IC3 Concepts

Concrete counterexample to induction (CTI). Central to IC3 is the evaluation of many consecution queries. Each has the form $F_i \wedge \neg s \wedge T \Rightarrow \neg s'$ and tests for the inductiveness of formula $\neg s$ relative to frame F_i. When the query is not valid, the counterexample reveals a predecessor, t, of s. CTI t explains why s is not inductive relative to F_i: t can reach s, and it is not known to be unreachable within i steps. A CTI can be expressed in any theory as a conjunction of equations between state variables and values. In CTIGAR, t is called a *concrete CTI* to distinguish it from an *abstract CTI*, introduced next.

Abstract CTI. As in standard predicate abstraction, the abstraction domain is a set of first-order atoms X over state variables V. An *abstract CTI* $\hat{s} = \alpha(s)$ corresponding to a given concrete CTI s is an over-approximation of s that is expressed as a Boolean combination of the predicates of the domain.

For a concrete state s that assigns values to every state variable in V, $\alpha(s)$ is a cube obtained by evaluating the atoms X over s, and it is the most precise abstraction. Expressing the most precise abstraction of a partial assignment requires, in general, a disjunction of cubes (obtained by an AllSAT query [34]). However, a "best effort" cube abstract CTI can be derived more simply by including only first-order literals that are equivalent to $\top$ under the partial assignment. The latter abstraction method is used in this work.[5]

For example, consider concrete CTI $s : x = 1 \wedge y = -1 \wedge z = 0$ and abstract domain $\{x < y,\ x < z\}$. The corresponding abstract CTI is $\hat{s} : \neg(x < y) \wedge \neg(x < z)$. If w were also a state variable, making s partial, and the domain were to include the predicate $y < w$, then the abstract CTI would remain the same: $y < w$ is equivalent to neither $\top$ nor $\bot$ under the partial assignment s.

Lifted CTI. A failed consecution query $F_i \wedge \neg t \wedge T \Rightarrow \neg t'$ reveals a concrete CTI s as well as an assignment z to the primary inputs. "Lifting" the full assignment s to a partial one is an important generalization mechanism in state-of-the-art IC3 implementations. In the original paper on IC3, static lifting was accomplished by considering the k-step cone of influence [10]; a dynamic approach

[5] Both methods were implemented, and the "best effort" cube-based one was found to be both simpler to implement and faster: experiments show that AllSAT-derived (DNF) abstract CTIs fare no better than "best effort" (cube) abstract CTIs.

(a) Lifting failure

(b) Consecution failure

Fig. 1: Single-step abstraction failures

based on ternary simulation was then proposed [24]; and a SAT-based approach was described in [15]. The SAT-based approach extends to the theory setting in a straightforward manner and is thus appropriate for CTIGAR.

The lifting query takes the form $s \wedge z \wedge T \wedge \neg t'$, which asks whether an s-state has a successor other than t under input assignment z. Since this is not the case by construction, the query yields an unsatisfiable core that typically reveals a significantly reduced partial assignment that can replace s.

Assuming that T is total, lifting the concrete CTI always succeeds. However, in CTIGAR it is the *abstract CTI* rather than the concrete CTI that is important. In the corresponding query $\hat{s} \wedge z \wedge T \wedge \neg t'$, the abstract state $\hat{s}$ replaces s. If this query is unsatisfiable, both $\hat{s}$ and the lifted abstract state revealed by the unsatisfiable core constitute an underapproximate preimage of the successor CTI. The query, however, may be satisfiable, since $\hat{s}$ over-approximates s and may therefore include states that transition to $\neg t$-states under input assignment z in addition to those—s at minimum—that transition to t-states. Failed lifting may eventually result in a *spurious* CTI, as discussed in Section 3.2 below.

3.2 The CTIGAR Flow

IC3 with CTIGAR is, as in propositional IC3, centered around the handling of proof obligations in lowest-frame-first order. Recall from Sections 2.2 and 3.1 that two types of queries are performed in relation to proof obligation $\langle s, i \rangle$:

① A lifting query $u \wedge z \wedge T \Rightarrow t'$ is performed to eliminate non-essential symbols from the original predecessor cube u to obtain cube s.

② A consecution query $F_i \wedge \neg s \wedge T \Rightarrow \neg s'$ tests if $\neg s$ is inductive relative to F_i.
- If it succeeds, the argument is generalized to produce a stronger clause $c \subseteq \neg s$ that is inductive relative to F_i.
- If it fails, the assignment to unprimed state variables provides a CTI v, which is lifted to cube $t \subseteq v$ and enqueued as proof obligation $\langle t, i-1 \rangle$.

Abstraction failures. The presence of abstract states complicates the situation in the sense that the following *abstraction failures* may arise:

① *Lifting Abstraction Failure* (LAF). The formula $\hat{s} \wedge z \wedge T \wedge \neg \hat{t}'$ is satisfiable, i.e., $\hat{s}$ contains at least one concrete state that has a successor outside of $\hat{t}$ under the inputs z, as indicated in Figure 1(a).

② *Consecution Abstraction Failure* (CAF). The formula $F_i \wedge \neg \hat{s} \wedge T \wedge \hat{s}'$ is satisfiable when $F_i \wedge \neg s \wedge T \wedge s'$ is not. In this setting, F_i contains at least one concrete state t outside $\hat{s}$ which has successor(s) in $\hat{s}$ that are not s, as illustrated in Figure 1(b). The transition from t to $\hat{s}$ is *spurious*: $\neg s$ is strong enough to be relatively inductive, while $\neg \hat{s}$ is not.

Proof Obligations. In CTIGAR, the components of a proof obligation reflect the possibility of abstraction failures. A proof obligation $\langle \hat{s},\ [s,]\ i,\ n \rangle$ comprises:

- an abstract CTI $\hat{s}$ (reduced from $\alpha(s)$ if abstract lifting succeeded);
- an *optional* concrete CTI s present if an LAF occurred;
- the frame index i, as in propositional IC3;
- the number n of spurious transitions encountered along the trace.

A *trace* is a sequence of proof obligations in which the last element is a CTI to the property P; and for each two consecutive elements, the CTI of the first element stems from a failed consecution query of the second. In this context, t_a denotes a CTI derived from the unprimed state variables V of a failed *abstract* consecution query $F_i \wedge \neg \hat{s} \wedge T \wedge \hat{s}'$, and t_c a CTI derived from a failed *concrete* consecution query $F_i \wedge \neg s \wedge T \wedge s'$.

The concrete CTI s is not included if abstract lifting succeeds because the lifted abstract CTI $\hat{s}$ describes only states that transition into the successor. In other words, the lifted abstract cube $\hat{s}$ is *as good as s in a concrete counterexample trace* when abstract lifting succeeds.

Because of the possibility of abstraction failures when lifting (LAF) or testing consecution (CAF), the operations of lifting to construct a new proof obligation and of handling a proof obligation are tightly coupled:

① *Lifting in CTIGAR.* Let s be either the concrete CTI s_a, derived via a failed abstract consecution query, or s_c, derived via a failed concrete consecution query. The cube t ($\hat{t}$, respectively), represents the successor of s, and z describes the primary input assignment from the failed query. The new proof obligation is constructed as follows:

1. Construct the abstract CTI $\hat{s} = \alpha(s)$;
2. Perform abstract lifting via the query $\begin{cases} \hat{s} \wedge z \wedge T \Rightarrow t' \text{ if } s = s_c \\ \hat{s} \wedge z \wedge T \Rightarrow \hat{t}' \text{ if } s = s_a \end{cases}$:
   (a) if lifting succeeds, let $\hat{s}_\ell$ be the lifted abstract CTI and enqueue new obligation $\langle \hat{s}_\ell, i-1, m \rangle$, where $m = n+1$ if s is the result of a CAF (see ②) and therefore spurious, and $m = n$ otherwise;
   (b) if lifting fails, enqueue the new obligation $\langle \hat{s}, s, i-1, m \rangle$, where the presence of s indicates an LAF and the value of m is as above.

Table 1: Overview of Lifting and Consecution in CTIGAR

<table>
<tr><td colspan="2">① Lifting</td><td colspan="3">② Consecution</td></tr>
<tr><td colspan="2">$\hat{s} \wedge z \wedge T \Rightarrow t' \;/\; \hat{s} \wedge z \wedge T \Rightarrow \hat{t'}$</td><td colspan="3">$F_i \wedge \neg\hat{s} \wedge T \Rightarrow \neg\hat{s}'$</td></tr>
<tr><td>succeeds:
Proof obligation
$\langle \hat{s}_\ell, i-1, m\rangle$,
where $m = n+1$ in case of CAF
and $m = n$ otherwise</td><td>fails (LAF):
Proof obligation
$\langle \hat{s}, s, i-1, m\rangle$,</td><td>succeeds:
generalize and
add $c \subseteq \neg\hat{s}$
to $F_1, \ldots, F_{i+1}$</td><td colspan="2">fails for $\langle \hat{s}, i, n\rangle$:
Extract and consider CTI t_a
fails for $\langle \hat{s}, s, i, n\rangle$:
query $F_i \wedge \neg s \wedge T \Rightarrow \neg s'$</td></tr>
<tr><td colspan="2"></td><td></td><td>succeeds:
Extract CTI t_c</td><td>fails:
CTI t_a (CAF)</td></tr>
</table>

Case 2b indicates the occurrence of an LAF, which will be discussed in Section 3.3. Analogously to propositional IC3, CTIGAR uses consecution queries to discharge proof obligations.

② *Consecution in CTIGAR.* Let $\langle \hat{s},\ [s,]\ i,\ n\rangle$ be an extended proof obligation. A failure of consecution when $i = 0$ indicates a counterexample trace. This situation is addressed in Section 3.3. Consecution is checked as follows:

Abstract consecution is checked via the query $F_i \wedge \neg\hat{s} \wedge T \Rightarrow \neg\hat{s}'$;

1. if consecution succeeds, an SMT solver is used to generalize $\hat{s}$ in standard IC3 fashion ([28, 12]), resulting in a clause $c \subseteq \neg\hat{s}$ that is inductive relative to F_i.
2. if consecution fails, the CTI t_a is extracted;
   (a) if concrete CTI s is present, then *concrete consecution* is checked via the query $F_i \wedge \neg s \wedge T \Rightarrow \neg s'$;
      i. if concrete consecution succeeds, then t_a triggers a new proof obligation (see ①)—this situation constitutes a CAF;
      ii. if concrete consecution fails, CTI t_c is extracted, and t_c triggers a new proof obligation (see ①).
   (b) if s is absent, then t_a is not spurious, and t_a triggers a new proof obligation (see ①).

The CAF in step 2(a)i is addressed Section 3.3. Table 1 summarizes the scenarios that can arise in CTIGAR.

The following section addresses abstraction lifting (LAF) and consecution (CAF) failures and counterexample traces.

3.3 Refinement

During lifting and the handling of proof obligations in Section 3.2, abstraction failures of type LAF or CAF may occur. This section presents a range of refinement strategies to address these failures. CTIGAR can react to LAFs and CAFs *eagerly* (immediately when they occur), *lazily*, or on a spectrum in between. In the latter two cases, refinement is postponed until a possible counterexample

trace is discovered (which cannot be ignored), or until the number of spurious transitions exceeds a threshold.

Refinement can take many forms depending on the abstract domain. In the context of predicate abstraction, Craig interpolation [21, 36] (popularized by [29]) is widely used to obtain refinement predicates. An interpolant for a pair of formulas (A, B), where $A \Rightarrow B$ is valid, is a formula J whose uninterpreted symbols occur in both A and B, such that $A \Rightarrow J$ and $J \Rightarrow B$. Interpolants always exist in first-order logic, and efficient interpolating decision procedures are available for a wide range of theories (e.g., [13, 22]).

① *Lifting refinement.* Recall from Section 3.2 (Figure 1(a)) that an LAF arises when the domain is too weak for abstract lifting. An LAF occurs when $s \wedge z \wedge T \Rightarrow t'$ holds, where t is the successor of s and z is the assignment to the inputs, while $\hat{s} \wedge z \wedge T \Rightarrow t'$ fails. Refinement ensures that the lifting query will succeed for the newly computed abstraction $\hat{s}$. When interpolation is possible, one can extract from the valid query $s \wedge z \wedge T \Rightarrow t'$ an interpolant R:

$$s \Rightarrow R \quad \text{and} \quad R \Rightarrow (z \wedge T \rightarrow t') \; .$$

The conjuncts of the formula R are added as first-order atoms to the abstract domain. Since $s \Rightarrow R$, the new precise abstraction of s is $\hat{s} \wedge R$, where $\hat{s}$ is the old abstraction of s. Furthermore, because $R \Rightarrow (z \wedge T \rightarrow t')$, the new abstract lifting query $(\hat{s} \wedge R) \wedge z \wedge T \Rightarrow t'$ is valid. Abstract lifting succeeds in the refined domain, thus eliminating this particular LAF.

② *Consecution refinement.* Recall from Section 3.2 that a CAF introduces a spurious transition (Figure 1(b)). In other words, the abstract domain is too weak for $\neg\hat{s}$ to be relatively inductive even though $\neg s$ is. A CAF occurs when $F_i \wedge \neg s \wedge T \Rightarrow \neg s'$ holds but $F_i \wedge \neg\hat{s} \wedge T \Rightarrow \neg\hat{s}'$ fails. Refinement ensures that the abstract consecution query will succeed for the newly computed abstraction $\hat{s}$. When interpolation is possible, one can extract from the valid query $F_i \wedge \neg s \wedge T \Rightarrow \neg s'$ an interpolant R:

$$F_i \wedge \neg s \wedge T \Rightarrow R' \quad \text{and} \quad R' \Rightarrow \neg s' \; .$$

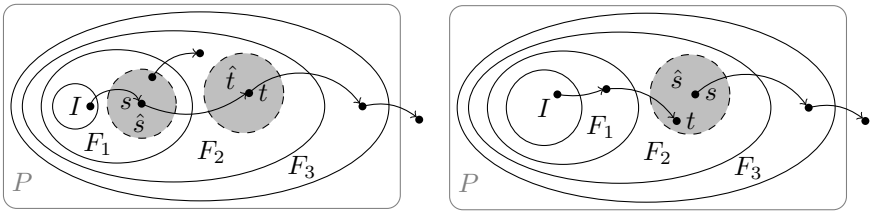
The formula $\neg R$ is added to the abstract domain. Since $R' \Rightarrow \neg s'$, $s \Rightarrow \neg R$, so that the new cube abstraction of s is $\hat{s} \wedge \neg R$, where $\hat{s}$ is the old abstraction of s. Furthermore, because $s \Rightarrow \hat{s} \wedge \neg R$,

$$F_i \wedge (\neg\hat{s} \vee R) \wedge T \;\Rightarrow\; F_i \wedge \neg s \wedge T \;\Rightarrow\; R'$$

so that the new abstract consecution query

$$F_i \wedge (\neg\hat{s} \vee R) \wedge T \Rightarrow (\neg\hat{s}' \vee R')$$

is valid. Under the refined domain, abstract consecution thus succeeds, eliminating this particular CAF.



(a) Lifting failure (b) Consecution failure

Fig. 2: Lazy refinement of abstraction failures

Eager and Lazy Refinement. In the flow described in Section 3.2, CAFs merely trigger an incrementation of the *spurious transition count (STC)*. When a potential new obligation's STC reaches some threshold controlling the degree of laziness, a *consecution refinement* is triggered. The STC indicates the number of spurious transitions on the trace rooted at that obligation. In this setting, a refinement can be triggered for four reasons:

- A counterexample trace is discovered, but the trace has at least one CAF anywhere (Figure 2(b)), triggering a *consecution refinement*.[6]
- A CTI s is disjoint from the initial states I, its abstraction $\hat{s}$ is not, and abstract lifting fails (a LAF). This situation triggers a *lifting refinement*.
- An obligation's STC reaches a threshold, triggering either a *consecution refinement* or a *lifting refinement*.[7]
- The trace rooted at an obligation has reached a threshold number of LAFs (Figure 2(a)), triggering a *lifting refinement*.[8]

Any (even multiple) CAF or LAF points can be analyzed during refinement. Addressing any one blocks the current arrangement of the obligation queue.

4 Implementation and Experimental Evaluation

4.1 Implementation

The experimental evaluation in this section is performed using a prototype of CTIGAR based on the IC3 reference implementation [8]. It uses linear integer arithmetic as the background theory and a combination of MathSAT 5 [2] and Z3 [22] as SMT solvers. The implementation includes a simple ANTLR 4 [39] parser that does not perform any optimizations at all on the resulting control flow graphs.

[6] Otherwise, the trace is a witness to the failure of the property.
[7] CAFs only occur for obligations for which LAFs occurred, so both are useful.
[8] If lifting refinements are triggered eagerly, CAFs never occur.

Abstract Domain. The abstract domain is initialized with the the atom I (encoding the initial program location), and inequalities of the form $x < y$ for all pairs of program variables x, y; additionally, the initial domain is enriched according to the equations discovered by a Karr analysis [32, 38] of the whole program.

Refinement predicates of the form $\sum_i b_i x_i = c$ are replaced by $\sum_i b_i x_i \leq c$ and $\sum_i b_i x_i \geq c$, and conjunctions are split into their arguments. Interpolants used for refinement are usually conjunctions in practice. Otherwise, the entire interpolant can be treated as a new predicate; additionally, atoms can be extracted and added as predicates as well.

Refinement State Mining. Orthogonal to interpolation-based refinement, refinement state mining (RSM) is a predicate discovery scheme deriving linear equalities from CTIs. The concrete cubes encountered in lifting and consecution queries are partitioned into sets S_l according to their program location l (represented by dedicated *program counter* variable `pc`). If the size of an S_l exceeds a threshold, a solver is deployed to discover a linear equality $\sum_k b_k x_k = c$ (where all b_k and c are coefficients, and x_k are program variables in S_l) covering as many states in S_l as possible while minimizing the number of coefficients that are zero. If the query succeeds, the covered states are removed from S_l and the resulting predicate is added to the abstract domain.

Similar to the DAIKON tool [25], the discovered predicates are not necessarily invariants or guaranteed to eliminate spurious CTIs. Alternatively, invariant finding algorithms such as the one described in [40] could be used.

4.2 Benchmarking

The prototype CTIGAR implementation was run on a collection of 110 linear integer arithmetic benchmarks from various sources: The InvGen benchmark suite as found in [27] , the Dagger benchmarks suite as found in [26], and the benchmark suite as found in [1]. Duplicates were only run once. Some benchmarks were omitted from this collection. The benchmarks `crawl_cbomb.c`, `fragtest.c`, `linpack.c`, `SpamAssassin-loop*.c` and `p*-*.c` contain pointers or other C constructs that the prototype does not handle. The benchmarks `half.c`, `heapsort*.c`, and `id_trans.c` contain truncating integer divisions, which the prototype does not handle. The benchmarks `puzzle1.c`, `sort_instrumented.c`, and `test.c` do not contain assert statements. The benchmarks `spin*.c` rely on functions that provide mutex functionality, which the prototype does not handle. All benchmarks are safe.

4.3 Evaluation Configurations

CTIGAR was run in multiple configurations. All configurations that use lazy refinement permit at most three spurious transitions in a single trace to the error. We chose 3 based on a manual analysis: three spurious transitions seem sufficient for lazy refinement while avoiding long irrelevant trace postfixes.

① **Configurations using lifting refinement:**
   (a) **LLE**: Eager refinement, triggered by a LAF.
   (b) **LLL**: Lazy refinement, triggered by a LAF.
   (c) **LCE**: Eager refinement, triggered by a CAF.
   (d) **LCL**: Lazy refinement, triggered by a CAF.

② **Configurations using consecution refinement**:
   (a) **CCE**: Eager refinement. Refinement is triggered by every CAF, regardless of whether the abstract state is lifted or not.
   (b) **CCL**: Lazy refinement. Refinement is triggered as above.
   (c) **CAE**: Eager refinement. Refinement is triggered by a CAF only if the abstract state is unlifted.
   (d) **CAL**: Lazy refinement. Refinement is triggered as above.

These versions of the prototype implementation of CTIGAR were compared against CPAChecker [6], the winner of the second software verification competition. The last column in Table 2 refers to the performance of CPAChecker in its competition configuration:
`config/sv-comp13--combinations-predicate.properties`.

The measurements were performed on AMD Opteron(TM) 6272 CPUs at 2100 MHz. No memory threshold was set. The timeout set for the benchmarks was 1200 seconds, wall time. However, if CTIGAR or CPAChecker did not run into the timeout, the run time is reported in the operating systems's user mode used for the benchmark, which is more accurate than the wall time.

4.4 Discussion of Runtime Results

All configurations solved substantially more benchmarks than CPAChecker.[9] CPAChecker was typically faster on benchmarks that were solved by both the prototype CTIGAR implementation and CPAChecker. However, there were 16-20 benchmarks in each configuration that were solved faster by our prototype.

The consecution refinement strategies proved be be somewhat more successful and faster than the lifting refinement strategies. In general, lazy refinement strategies seem to be slightly more successful than eager refinement strategies.

Deploying the interpolation procedure presented in [1] increased the computational overhead of interpolation while not providing measurable improvement of the abstract domain.

[9] CPAChecker returned UNSAFE on `MADWiFi-encode_ie_ok.c`, but it assigns non-integer values to some integer variables in its error path assignment. A manual inspection of the benchmark reveals that it is in fact safe; nonetheless, the benchmark is counted as solved by CPAChecker. In addition to the 64 solved benchmarks, CPAChecker returned with the message `Analysis incomplete: no errors found, but not everything could be checked.` on 16 benchmarks.

Table 2: Runtime results for CTIGAR and CPAChecker. All times are in seconds.

Lifting refinement	**LLE**	**LLL**	**LCE**	**LCL**	**CPAChecker**
Number of solved benchmarks	87	86	83	87	64
Cumulative time	7061.68	7547.85	8516.06	7745.06	1170.85
# Solved — unsolved by CPA	29	31	30	34	
Cumulative time	1134.49	2702.17	5425.5	5113.44	
# Solved — faster than CPA	16	16	18	20	
Cumulative time (CTIGAR)	12.38	17.35	14.68	29.24	
Cumulative time (CPA)	53.34	848.99	59.14	860.31	
Consecution refinement	**CCE**	**CCL**	**CAE**	**CAL**	**CPAChecker**
Number of solved benchmarks	86	91	91	92	64
Cumulative time	5414.57	8150.29	6154.33	5880.74	1170.85
# Solved — unsolved by CPA	31	36	34	36	
Cumulative time	2010.26	5149.72	2033.03	2247.59	
# Solved — faster than CPA	19	20	20	20	
Cumulative time (CTIGAR)	18.03	34.34	18.92	21.06	
Cumulative time (CPA)	62.05	863.98	65.34	863.98	

Figure 3 to the right presents a comparison of the number of predicates in the abstraction domain vs. the runtime for all terminating instances across all configurations in a log-log-plot. It shows that performance only degrades polynomially with the number of predicates in the domain rather than exponentially.

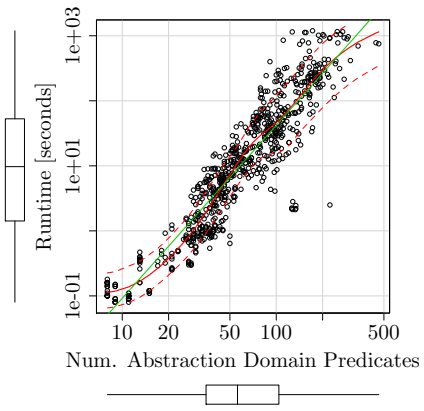
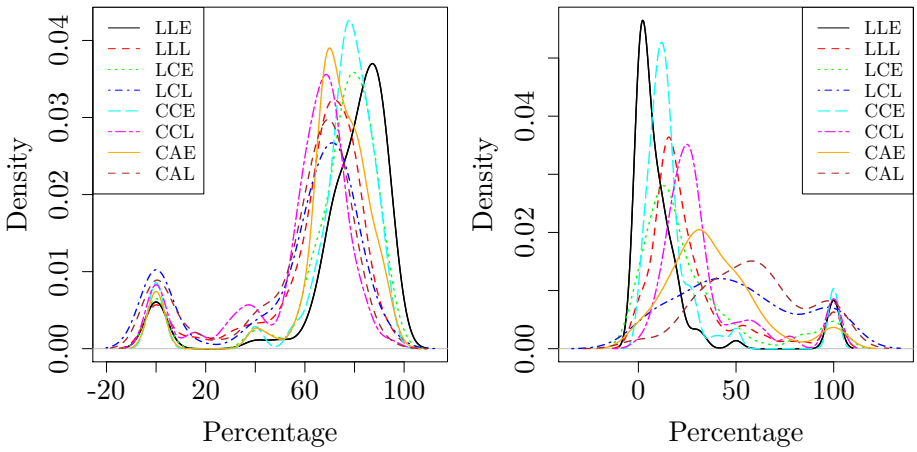


Fig. 3: Number of abstraction domain predicates vs. runtime for terminating instances.

Figure 4(a) depicts the percentage of successful abstract lifting calls across different configurations (both in consecution and lifting refinement). Abstract lifting succeeds in around 60% to 80% of all cases, providing CTIs that are underapproximate preimages.

As evident from Figure 4(b), the best configurations (notably CAL) do not immediately address lifting failures but instead lazily proceed with abstract CTIs that do not underapproximate preimages. Strictly using underapproximate preimages is, apparently, not essential. This observation contrasts with previous approaches [16, 31]. The experiments also show that a large portion of abstract CTIs are not underapproximate preimages yet are successfully generalized and eliminated, avoiding the cost of computing non-trivial underapproximate preimages.



(a) Percentage of successful abstract liftings of all abstract lifting tries. (The implementation lifts abstract states repeatedly after refinement.)

(b) Percentage of abstract states that are still unlifted after having been successfully used for strengthening up to the frontier level.

Fig. 4

5 Related Work

Since the inception of the original IC3 [10, 9] numerous attempts have been made to lift the approach to richer logics and infinite domains. Welp and Kühlmann [42] propose interval simulation as a means of generalizing proof obligations in the domain of bit-vectors. Refinement is not required in this setting, as intervals approximate values in the finite concrete domain conservatively. The same holds for region abstraction applied in the context of timed systems [33].

A more general approach applicable to infinite state transition systems and a wider set of theories is to replace the SAT engine underlying IC3 with an SMT solver. In an attempt to avoid a diverging sequence of proof obligations in the infinite concrete domain, Cimatti and Griggio [16] suggest a non-trivial under-approximation of the pre-image (an effort countermanded by the subsequent generalization step). To avert the overhead of the pre-image computation, the algorithm in [16] relies on the Lazy Abstraction with Interpolants (LAwI) refinement scheme [37] as long as the resulting interpolants can be converted into clausal form efficiently, effectively using IC3 as a fallback only.

An inherent drawback of the path-wise unwinding deployed in [16] is that the generalized clauses are not relatively inductive. A recent follow-up publication [17] therefore uses a monolithic transition relation (previously dismissed as inefficient in [16]), replacing the pre-image computation with (implicit) predicate abstraction. Unlike in CTIGAR, refinement is triggered by an abstract coun-

terexample trace and based on an unwinding of the transition relation. Hoder and Bjørner [31] uses Horn clauses to represent recursive predicate transformers. Proof obligations are generalized using a specialized interpolation procedure for linear arithmetic. Effectively, this amounts to an eager refinement step potentially introducing new literals that are linear combinations of the atoms in the consecution query. Vizel et al. [41] implement lazy abstraction for finite state systems by projecting the frames to a sequence of variable sets (of monotonically increasing size), which are refined if a spurious counterexample trace is found.

The following discussion represents an attempt to put CTIGAR into a broader context. Unlike CTIGAR, conventional predicate abstraction tools [4, 20] construct an explicit abstract transition relation. Most of these tools, however, use Cartesian abstraction rather than computing the most precise abstraction [5] and refine spurious abstract transitions using a focus operation [3]. SatAbs [20] in particular prioritizes transition refinement (triggered by a spurious abstract counterexample trace) over refining the abstract domain, resulting in a succession of relatively simple single-step SAT queries. In contrast, CTIGAR, following IC3, strengthens frames (rather than the abstract transition relation) using single-step consecution queries triggered by single states, and only refines the domain in case of abstraction failures. CTIGAR as well as [17] deploy implicit predicate abstraction. Similarly, lazy abstraction [30, 37] does not maintain an explicit abstract transition relation, but uses traces and sequence interpolation to refine the safely reachable states. The fact that CTIGAR derives interpolants from single transition steps instead may have advantages beyond the resulting simplicity of the SMT queries: Cabodi's work suggests that—at least in the propositional case—sequential interpolation is inferior to standard interpolation [14].

6 Conclusion

The impact of using abstract CTIs on lifting and consecution queries is inevitable: abstraction introduces spurious transitions. Focusing on that impact within the IC3 algorithm, rather than outside of it, naturally leads to a CTI-guided, rather than a counterexample trace-guided, abstraction-refinement scheme—CTIGAR rather than CEGAR. The potential benefits of CTIGAR over CEGAR are obvious: faster and more focused refinement triggers, explicit states for state-mining-based predicate synthesis, and one-step interpolation queries for interpolation-based refinement. More broadly, CTIGAR continues the trend started by IC3 of focusing on individual states and single-step queries instead of traces and multi-step queries (BMC and its derivatives).

The prototype implementation of CTIGAR performs competitively against a state-of-the-art CEGAR-based tool in terms of number of solved benchmarks, confirming its potential. Results vary but are robust across parameter settings: lazy vs. eager, lifting- vs. consecution-based refinement. It is expected that further experience with CTIGAR will reveal implementation techniques that close the performance gap between our CTIGAR prototype implementation and well-tuned checkers like CPAChecker.

References

1. Aws Albarghouthi and Kenneth L. McMillan. Beautiful Interpolants. In *Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 313–329. Springer, 2013.
2. Alessandro Cimatti and Alberto Griggio and Bastiaan Schaafsma and Roberto Sebastiani. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7795 of *Lecture Notes in Computer Science*. Springer, 2013.
3. Thomas Ball, Byron Cook, Satyaki Das, and Sriram K. Rajamani. Refining Approximations in Software Predicate Abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*. Springer, 2004.
4. Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.
5. Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *Software Tools for Technology Transfer (STTT)*, 5(1):49–58, 2003.
6. Dirk Beyer and M. Erkan Keremoglu. CPAChecker: a Tool for Configurable Software Verification. In *Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.
7. Armin Biere. Bounded Model Checking. In *Handbook of Satisfiability*, pages 457–481. IOS Press, 2009.
8. Aaron R. Bradley. IC3 reference implementation. https://github.com/arbrad/IC3ref/.
9. Aaron R. Bradley. k-Step Relative Inductive Generalization. *The Computing Research Repository*, abs/1003.3649, 2010.
10. Aaron R. Bradley. SAT-Based Model Checking Without Unrolling. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.
11. Aaron R. Bradley. Understanding IC3. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2012.
12. Aaron R. Bradley and Zohar Manna. Checking Safety by Inductive Generalization of Counterexamples to Induction. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 173–180. IEEE, 2007.
13. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 299–303. Springer, 2008.
14. Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. Interpolation sequences revisited. In *Design Automation and Test in Europe (DATE)*, pages 316–322. IEEE, 2011.
15. Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental Formal Verification of Hardware. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 135–143. IEEE, 2011.
16. Alessandro Cimatti and Alberto Griggio. Software Model Checking via IC3. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 277–293. Springer, 2012.

17. Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 Modulo Theories via Implicit Predicate Abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, 2014. To appear.
18. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, 2000.
19. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5), September 2003.
20. Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176. Springer, 2004.
21. William Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.
22. Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
23. Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software Verification Using k-Induction. In *Static Analysis Symposium (SAS)*, volume 6887 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2011.
24. Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient Implementation of Property Directed Reachability. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134. IEEE, 2011.
25. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1-3):35–45, December 2007.
26. Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Dagger Benchmarks Suite. http://www.cfdvs.iitb.ac.in/ bhargav/dagger.php.
27. Ashutosh Gupta and Andrey Rybalchenko. InvGen Benchmarks Suite. http://pub.ist.ac.at/ agupta/invgen/.
28. Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Better Generalization in IC3. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2013.
29. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from Proofs. In *Principles of Programming Languages (POPL)*, pages 232–244. ACM, 2004.
30. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy Abstraction. In *Principles of Programming Languages (POPL)*, pages 58–70. ACM, 2002.
31. Kryštof Hoder and Nikolaj Bjørner. Generalized Property Directed Reachability. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.
32. Michael Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
33. Roland Kindermann, Tommi A. Junttila, and Ilkka Niemelä. SMT-Based Induction Methods for Timed Systems. In *Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 7595 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2012.

34. Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT Techniques for Fast Predicate Abstraction. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 424–437. Springer, 2006.
35. Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.
36. Kenneth L. McMillan. An Interpolating Theorem Prover. *Theoretical Computer Science*, 345(1):101–121, 2005.
37. Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.
38. Markus Müller-Olm and Helmut Seidl. A Note on Karr's algorithm. In *Automata, Languages and Programming (ICALP)*, volume 3142 of *Lecture Notes in Computer Science*, pages 1016–1028. Springer, 2004.
39. Terence Parr. ANTLR4. http://www.antlr.org/.
40. Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A Data Driven Approach for Algebraic Loop Invariants. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, Proceedings of the European Symposium on Programming, pages 574–592. Springer, 2013.
41. Yakir Vizel, Orna Grumberg, and Sharon Shoham. Lazy Abstraction and SAT-Based Reachability in Hardware Model Checking. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 173–181. IEEE, 2012.
42. Tobias Welp and Andreas Kuehlmann. QF_BV Model Checking with Property Directed Reachability. In *Design Automation and Test in Europe (DATE)*, pages 791–796. EDA Consortium, 2013.