Noname manuscript No.
(will be inserted by the editor)

Infinite-state Invariant Checking with IC3 and Predicate Abstraction

Alessandro Cimatti · Alberto Griggio · Sergio Mover · Stefano Tonetta

the date of receipt and acceptance should be inserted later

Abstract We address the problem of verifying invariant properties on infinite-state systems. We present a novel approach, IC3IA, for generalizing the IC3 invariant checking algorithm from finite-state to infinite-state transition systems, expressed over some background theories. The procedure is based on a tight integration of IC3 with Implicit Abstraction, a form of predicate abstraction that expresses abstract paths without computing explicitly the abstract system. In this scenario, IC3 operates only at the Boolean level of the abstract state space, discovering inductive clauses over the abstraction predicates. Theory reasoning is confined within the underlying SMT solver, and applied transparently when performing satisfiability checks. When the current abstraction allows for a spurious counterexample, it is refined by discovering and adding a sufficient set of new predicates. Importantly, this can be done in a completely incremental manner, without discarding the clauses found in the previous search.

The proposed approach has two key advantages. First, unlike previous SMT generalizations of IC3, it allows to handle a wide range of background theories without relying on ad-hoc extensions, such as quantifier elimination or theory-specific clause generalization procedures, which might not always be available and are often highly inefficient. Second, compared to a direct exploration of the concrete transition system, the use of abstraction gives a significant performance improvement, as our experiments demonstrate.

1 Introduction

IC3 [12] is an algorithm for the verification of invariant properties of transition systems. It builds an over-approximation of the reachable state space, using clauses

A. Cimatti · A. Griggio · S. Tonetta
Fondazione Bruno Kessler
{cimatti,griggio,tonettas}@fbk.eu

S. Mover
University Of Colorado Boulder
sergio.mover@colorado.edu

obtained by generalization while disproving candidate counterexamples. The generalization is based on the idea of keeping a sequence of over-approximations, each inductive relative to the previous one.

In the case of finite-state systems, the algorithm is implemented on top of Boolean SAT solvers, fully leveraging their features (e.g. incrementality). IC3 has demonstrated to be extremely effective, and it is a fundamental core in all the engines in hardware verification.

There have been several attempts to lift IC3 to the case of infinite-state systems, for its potential applications to software, RTL models, timed and hybrid systems (although the problem is in general undecidable). These approaches are set in the framework of Satisfiability Modulo Theory (SMT) [4] and hereafter are referred to as IC3 Modulo Theories [18,42,35,52]: the infinite-state transition system is symbolically described by means of SMT formulas, and an SMT solver plays the same role of the SAT solver in the discrete case. The key difference is the need in IC3 Modulo Theories for specific theory reasoning to deal with candidate counterexamples. This led to the development of various techniques, based on quantifier elimination or theory-specific clause generalization procedures. Unfortunately, such extensions are typically ad-hoc, and might not always be applicable in all theories of interest. Furthermore, being based on the fully detailed SMT representation of the transition systems, some of these solutions (e.g. based on quantifier elimination) can be highly inefficient.

We present a novel approach to IC3 Modulo Theories, which is able to deal with infinite-state systems by means of a tight integration with *predicate abstraction* (PA) [26], a standard abstraction technique that partitions the state space according to the equivalence relation induced by a set of predicates. In this work, we leverage *Implicit Abstraction* (IA) [48], which allows to express abstract transitions without computing explicitly the abstract system, and is fully incremental with respect to the addition of new predicates. In the resulting algorithm, called IC3IA, the search proceeds as if carried out in an abstract system induced by the set of current predicates $\mathbb{P}$. The key insight is to exploit IA to obtain an abstract version of the relative induction check, the central operation of IC3. We follow a Counter-Example Guided Abstraction-Refinement (CEGAR) approach: when an abstract counterexample is found, it is simulated in the concrete space and, if spurious, the current abstraction is refined by adding a set of predicates sufficient to rule it out.

The IC3IA approach has several advantages. First, unlike previous SMT generalizations of IC3, IC3IA allows to handle a wide range of background theories without relying on ad-hoc extensions, such as quantifier elimination or theory-specific clause generalization procedures. The only requirement is the availability of an effective technique for abstraction refinement, for which various solutions exist for many important theories (e.g. interpolation [33], unsat core extraction [29], or weakest precondition [38]). Second, the analysis of the infinite-state transition system is now carried out in the abstract space, which is often as effective as an exact analysis, but also much faster. Third, the approach is completely incremental, without having to discard or reconstruct clauses found in the previous iterations. Finally, the integration of SMT into IC3 is very natural, in the sense that it preserves the IC3 structure and works in each iteration on a well-defined transition system (namely, the minimal predicate abstraction given by the current set of

predicates). This allows for a simple implementation of the extension (especially if compared to the complexity of IC3).

We experimentally evaluated IC3IA on a set of benchmarks from heterogeneous sources [6,31,42], with very positive results. First, our implementation of IC3IA is significantly more expressive than the SMT-based IC3 of [18], being able to handle not only the theory of Linear Rational Arithmetic (LRA) like [18], but also those of Linear Integer Arithmetic (LIA) and fixed-size bit-vectors (BV). Second, in terms of performance IC3IA is uniformly superior to a wide range of alternative techniques and tools, including state-of-the-art implementations of the bit-level IC3 algorithm ([25,47,10]), other approaches for IC3 Modulo Theories ([18,35, 42]), and techniques based on k-induction and invariant discovery ([31,40]). Third, the results show how IC3IA is more efficient also with respect to CTIGAR [8], a more recent integration of IC3 and abstraction, similar in principle to IC3IA but with several important differences (which we point out in this paper).

A remarkable property of IC3IA is that it can deal with a large number of predicates: in several benchmarks, *hundreds of predicates* were introduced during the search. Considering that an explicit computation of the abstract transition relation (e.g. based on All-SMT [44]) often becomes impractical with a few dozen predicates, we conclude that Implicit Abstraction is fundamental to scalability, allowing for efficient reasoning in a fine-grained abstract space.

The rest of the paper is structured as follows. In Section 2 we present some background on IC3 and Implicit Abstraction. In Section 3 we describe IC3IA and prove its formal properties. In Section 4 we discuss the related work. In Section 5 we experimentally evaluate our method. In Section 6 we draw some conclusions and present directions for future work.

2 Background

2.1 Notation

Our setting is first order logic. We use the standard notions of theory, satisfiability, validity, and logical consequence. We also use notions from Satisfiability Modulo Theories (SMT) [4]. We denote generic theories as $\mathcal{T}$. We write $\varphi \models_{\mathcal{T}} \psi$ to denote that the formula ψ is a logical consequence of φ in the theory $\mathcal{T}$; when clear from context, we omit $\mathcal{T}$ and simply write $\varphi \models \psi$.

We denote formulas with $\phi, \varphi, \psi, I, T, P$, variables with x, y, and sets of variables with X, Y, $\overline{X}$, $X', X_{\mathbb{P}}$. Unless otherwise specified, we work on quantifier-free formulas. We refer to 0-arity predicates as Boolean variables, and to 0-arity uninterpreted functions as (theory) variables. A *literal* is an atom or its negation. A *clause* is a disjunction of literals, whereas a *cube* is a conjunction of literals. If s is a cube $l_1 \wedge \ldots \wedge l_n$, with $\neg s$ we denote the clause $\neg l_1 \vee \ldots \vee \neg l_n$, and vice versa. A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses, and in disjunctive normal form (DNF) if it is a disjunction of cubes. With a little abuse of notation, we denote formulas in CNF $c_1 \wedge \ldots \wedge c_n$ as sets of clauses $\{c_1, \ldots, c_n\}$, and vice versa. If $X_1, \ldots, X_n$ are sets of variables and φ is a formula, we write $\varphi(X_1, \ldots, X_n)$ to indicate that all the variables occurring in φ are elements of $\bigcup_{i=1}^{n} X_i$.

Given a set of variables X, a signature Σ, a domain M, an interpretation function $\mathcal{I}$ of the symbols in Σ on the domain M, an assignment σ to the variables in X on the domain M, and a σ-formula $\phi(X)$ with free variables in X, the satisfaction relation $\langle M, \mathcal{I}\rangle \models \phi$ is defined in the usual way. Given $\mu = \langle M, \mathcal{I}, \sigma\rangle$ and ϕ, we denote by $\mu(\phi)$ the truth value given by μ to the formula ϕ.

When Σ is implicit and clear from the context, we call $\langle M, \mathcal{I}, \sigma\rangle$ an interpretation of X. Given two disjoint sets of variables X and Y, an interpretation $\mu_1 = \langle M, \mathcal{I}, \sigma_1\rangle$ of X and an interpretation $\mu_2 = \langle M, \mathcal{I}, \sigma_2\rangle$ of Y with the same structure $\langle M, \mathcal{I}\rangle$, then $\mu_1 \cup \mu_2$ denotes the interpretation $\langle M, \mathcal{I}, \sigma\rangle$ of $X \cup Y$ such that $\sigma(x) = \sigma_1(x)$ for every $x \in X$ and $\sigma(y) = \sigma_2(y)$ for every $y \in Y$. Given an interpretation $\mu_1 = \langle M, \mathcal{I}, \sigma_1\rangle$ of X, we call the *projection* of μ_1 over $Y \subseteq X$ the interpretation $\mu_2 = \langle M, \mathcal{I}, \sigma_2\rangle$ such that $\sigma_2(y) = \sigma_1(y)$ for all $y \in Y$. Given an interpretation $\mu = \langle M, \mathcal{I}, \sigma\rangle$ of X, we denote by $\mu_Y[y := c_y]$ the interpretation $\langle M, \mathcal{I}, \sigma_Y\rangle$ of Y such that $\sigma_Y(y) = c_y$ for every $y \in Y$. With abuse of notation, given an interpretation $\mu = \langle M, \mathcal{I}, \sigma\rangle$ of X and $x \in X$, we use $\mu(x)$ to denote $\sigma(x)$. We also use μ to denote the cube representing it.

For each variable x, we assume that there exists a corresponding variable x', called the *primed version* of x. If X is a set of variables, X' is the set obtained by replacing each element x with its primed version ($X' = \{x' \mid x \in X\}$). φ' is the formula obtained by replacing each occurrence variable in φ with the corresponding primed. x^n is the variable obtained by adding n primes to x, while X^n is the corresponding set ($X^n = \{x^n \mid x \in X\}$). $\overline{X}$ is a copy of the variables X, i.e., is the set variables obtained from X by replacing each $x \in X$ with $\overline{x}$ ($\overline{X} = \{\overline{x} \mid x \in X\}$). Given a set of predicates $\mathbb{P}$, $X_{\mathbb{P}}$ is a set of variables containing one variable x_p for every predicate $p \in \mathbb{P}$ ($X_{\mathbb{P}} = \{x_p \mid p \in \mathbb{P}\}$).

2.2 Transition Systems

In the following, the signature σ and the theory $\mathcal{T}$ are implicitly given. A *transition system (TS)* S is a tuple $\langle X, I, T\rangle$ where X is a set of (state) variables, $I(X)$ is a formula representing the initial states, and $T(X, X')$ is a formula representing the transitions. A *state* s of S is an interpretation of the state variables X. A (finite) *path* of S is a finite sequence $\pi \doteq s_0, s_1, \ldots, s_k$ of states, with the same domain and interpretation of symbols in the signature Σ, such that $s_0 \models I$ and for all i, $0 \leq i < k$, $s_i, s'_{i+1} \models T$. We say that a state s is reachable in S iff there exists a path of S ending in s.

Example 1 $S = \langle \{c, d\}, c = 0 \wedge d = 0, c' = c + d \wedge d' = d + 1\rangle$ is a transition system with integer variables $\{c, d\}$, where d is increased by one in every transition, while c is incremented by the current value of d.

2.3 Invariant Verification Problems

Given a formula $P(X)$ and a transition system $S = \langle X, I, T\rangle$, the *invariant verification problem*, denoted with $S \models P$, is the problem to check if for all the finite paths $s_0, s_1, \ldots, s_k$ of S, for all i, $0 \leq i \leq k$, $s_i \models P$. Its dual formulation in terms of *reachability* of $\neg P$ is the problem to find a path $s_0, s_1, \ldots, s_k$ of S such that

```
bool IC3 (I, T, P):
1.  if not I ⊨ P: return FALSE # violation in the initial states
2.  F_0 := I  # first elem of trace is init formula
3.  k := 1, F_k := ⊤  # add a new frame to the trace
4.  while TRUE:
        # blocking phase
5.      while F_k ∧ ¬P ⊭ ⊥:
6.          extract a cube c from the model of F_k ∧ ¬P
7.          if not RECBLOCK(c, k):
                # s_0 bwd-reached at depth 0, build π ≐ s_0, ..., s_k
8.              return FALSE # counterexample found
        # propagation phase
9.      k := k + 1, F_k := ⊤
10.     for i := 1 to k − 1:
11.         for each clause c ∈ F_i:
12.             if RelInd(F_i, T, c) ⊨ ⊥:
13.                 add c to F_{i+1}
14.         if F_i = F_{i+1}:
15.             return TRUE # property proved with inductive invariant F_i

bool RECBLOCK(s, i):
1.  if i = 0: return FALSE # reached initial states
2.  while RelInd(F_{i−1}, T, ¬s) ⊭ ⊥:
3.      extract a cube c from the model of RelInd(F_{i−1}, T, ¬s)
        # c is a predecessor of s
4.      if not RECBLOCK(c, i − 1): return FALSE
5.  g = GENERALIZE(¬s, i)  # standard IC3 generalization [12, 25]
6.  for j := 1 to i: add g to F_j
7.  return TRUE
```

Fig. 1 High-level description of IC3.

$s_k \models \neg P$. P represents the "good" states, while $\neg P$ represents the "bad" states. $S \models P$ iff all the reachable states of S are good, i.e. no bad state is reachable.

Inductive invariants and relative inductive invariants are central notions to solve the invariant verification problem. F is an *inductive invariant* for S iff $I(X) \models F(X)$, and $F(X) \wedge T(X, X') \models F(X')$. A typical verification strategy is to look for an inductive invariant F such that $F \models P$ (thus, yielding that $S \models P$). F is *inductive relative* to the formula $\phi(X)$ iff $I(X) \models F(X)$, and $\phi(X) \wedge F(X) \wedge T(X, X') \models F(X')$. It is sometimes useful to first prove some lemma and then search for an invariant that is inductive relative to such lemma.

Note that we use the symbol $\models$ with three different denotations: if ϕ is a formula, $\phi \models \psi$ denotes that ψ is a logical consequence of ϕ; if μ is an interpretation, $\mu \models \psi$ denotes that μ is a model of ψ; if S is a transition system, $S \models \psi$ denotes that ψ is an invariant of S. The different usages of $\models$ will be clear from the context.

2.4 IC3 for finite- and infinite-state systems

IC3 [12] is an efficient SAT-based algorithm for the verification of finite-state systems, with Boolean state variables and propositional logic formulas. IC3 was

subsequently extended to the case of infinite-state systems in [18,35], leveraging the power of SMT. In the following, we present its main ideas, following the description of [18]. Additional details can be found in [12,51,18,35].

The IC3 algorithm tries to prove that $S \models P$ by finding a suitable inductive invariant $F(X)$ such that $F(X) \models P(X)$. In order to construct F, IC3 maintains a sequence of formulas (called *trace*) $F_0(X), \ldots, F_k(X)$ such that: (i) $F_0 = I$; (ii) $F_i \models F_{i+1}$; (iii) $F_i(X) \wedge T(X, X') \models F_{i+1}(X')$; (iv) for all $i < k$, $F_i \models P$. Therefore, each element of the trace F_{i+1}, called *frame*, is inductive relative to the previous one, F_i. IC3 strengthens the frames by finding new relative inductive clauses. A clause c is inductive relative to the frame F, i.e. $F \wedge c \wedge T \models c'$, iff the formula

$$RelInd(F, T, c) \doteq F \wedge c \wedge T \wedge \neg c' \tag{1}$$

is unsatisfiable, so that a check of relative inductiveness can be directly tackled by a SAT (or SMT) solver.

A high-level description of IC3 is shown in Figure 1 as pseudo-code. The algorithm proceeds incrementally, by alternating two phases: a blocking phase, and a propagation phase. In the *blocking* phase (lines 5–8 of Figure 1), the trace is analyzed to prove that no intersection between F_k and $\neg P(X)$ is possible. During this phase, the trace is enriched with additional formulas, which can be seen as strengthening the approximation of the reachable state space. At the end of the blocking phase, either $F_k \models P$ is proved or a counterexample is generated.

The *propagation* phase (lines 9–15 of Figure 1) tries to extend the trace with a new formula F_{k+1}, moving forward the clauses from preceding F_i's. If, during this process, two consecutive frames become identical (i.e. $F_i = F_{i+1}$), then a fixpoint is reached, and IC3 terminates with F_i being an inductive invariant proving the property.

In the blocking phase IC3 maintains a set of pairs (s, i), where s is a set of states that can lead to a bad state, and $i > 0$ is a position in the current trace. New formulas (in the form of clauses) to be added to the current trace are derived by (recursively) proving that a cube s of a pair (s, i) is unreachable starting from the formula F_{i-1} (RecBlock procedure of Figure 1).[1] This is done by checking the satisfiability of the formula $RelInd(F_{i-1}, T, \neg s)$. If the formula is unsatisfiable, then $\neg s$ is *inductive relative to* F_{i-1}, and the bad state s can be *blocked* at i. This is done by (i) *generalizing* $\neg s$ to a *stronger clause* $\neg g$ that is still inductive relative to F_{i-1} (Generalize procedure), and (ii) adding $\neg g$ to F_i. Inductive generalization is a central step of IC3, that is crucial for the performance of the algorithm. Adding $\neg g$ to F_i blocks not only the bad cube s, but possibly also many others, thus allowing for a faster convergence of the algorithm. At a high level, the algorithm for performing inductive generalization works by dropping some literals from the input clause $\neg s$ and testing whether the result is still inductive relative to F_{i-1}, until a stopping criterion is reached (e.g. a fix-point or a resource bound). In the literature, several variants of this basic approach have been proposed. We refer the reader to [13,25,32,51,27] for more information.

If, instead, (1) is satisfiable, then the overapproximation F_{i-1} is not strong enough to show that s is unreachable. In this case, let p be a subset of the states

[1] The recursive procedure RecBlock is an oversimplification of the one actually used by IC3. In practice, RecBlock is implemented using a priority queue. This is however not important for our purposes, and we refer the reader to [12,25] for more information.

in $F_{i-1} \wedge \neg s$ such that all the states in p lead to a state in s' in one transition step. Then, IC3 continues by trying to show that p is not reachable in one step from F_{i-2} (that is, it tries to block the pair $(p, i-1)$). This procedure continues recursively, possibly generating other pairs to block at earlier points in the trace, until either IC3 generates a pair $(q, 0)$, meaning that the system does not satisfy the property, or the trace is eventually strengthened so that the original pair (s, i) can be blocked.

A key difference between the original Boolean IC3 and its SMT extensions in [18,35] is in the way sets of states to be blocked or generalized are constructed. In the blocking phase, when trying to block a pair (s, i), if the formula (1) is satisfiable, then a new pair $(p, i-1)$ has to be generated such that p is a cube in the *preimage of s wrt. T* (i.e., for every state in p there exists a transition to a state in s). In the propositional case, p can be obtained from the model μ of (1) generated by the SAT solver, by simply dropping the primed variables occurring in μ. A naïve lifting of this procedure to the first-order case could however be quite inefficient, since this would lead IC3 to exclude only a single point at a time in an infinite state space, resulting in a high chance of divergence in the blocking phase. The solution proposed in [18] is to compute p by existentially quantifying (1) and then applying an *under-approximated* existential elimination algorithm for linear rational arithmetic formulas. In the following, we refer to the algorithm proposed in [18] as IC3QE. In [35], instead, the problem is addressed by proposing a theory-aware extension of inductive generalization, to generalize $\neg s$ from a single point to a larger region before adding it to F_i, after having successfully blocked it. In addition to dropping literals from $\neg s$, like in the Boolean case, the algorithm also performs generalizations at the theory level, by weakening inequalities $(t \leq c)$ to $(t \leq c + c')$ with $c' > 0$, while ensuring that (1) still holds for the resulting clause. Also in this case, however, the procedure (based on Craig interpolation) is limited to linear rational arithmetic.

2.5 Abstraction

2.5.1 Predicate abstraction

Abstraction [24] is a very powerful approach to verification. It is used to reduce the search space while preserving the satisfaction of the property of interest. If $\widehat{S}$ is an abstraction of S and a state is reachable in S, then also its abstract version is reachable in $\widehat{S}$. Thus, in order to prove that a set of states is not reachable in S, it is sufficient to prove that its abstract version is not reachable in $\widehat{S}$.

In Predicate Abstraction [26], the abstract state-space is induced by a set of predicates $\mathbb{P}$. Each predicate $p \in \mathbb{P}$ is a formula over the variables X. Intuitively, these characterize relevant facts of the system. For every $p \in \mathbb{P}$ we introduce a new Boolean variable x_p, referred to as the predicate name or abstract variable. We write $X_{\mathbb{P}}$ to denote the set of predicate names $\{x_p\}_{p \in \mathbb{P}}$. The abstraction relation $H_{\mathbb{P}}$ is then defined as

$$H_{\mathbb{P}}(X, X_{\mathbb{P}}) \doteq \bigwedge_{p \in \mathbb{P}} x_p \leftrightarrow p(X)$$

Given a formula $\phi(X)$, the predicate abstraction of ϕ with respect to $\mathbb{P}$, denoted $\widehat{\phi}_{\mathbb{P}}$, is obtained by adding the abstraction relation to it and then existentially

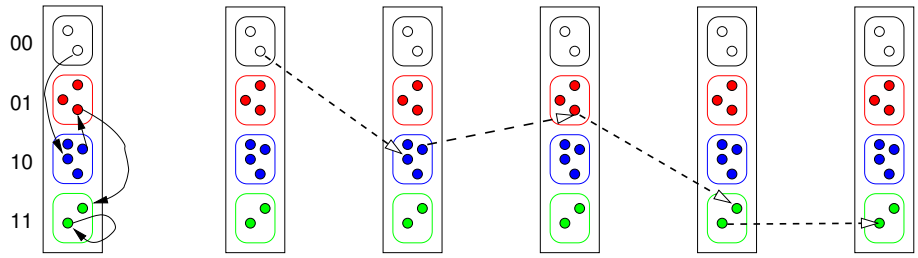


Fig. 2 (Left) Concrete and abstract system: Each circle represents a concrete state and each solid arrow is a concrete transition. Each rectangle with rounded corners represents an abstract state (different colors represent different abstract states). **(Right)** Abstract path: each rectangle represents a copy of the state space, and a dashed arrow represents an abstract transition between two abstract states (notice that, to make the example clearer, we kept the concrete states, and the abstract transitions correspond also to an existing concrete transition in the system).

quantifying the variables X, i.e.,

$$\widehat{\phi}_{\mathbb{P}}(X_{\mathbb{P}}) \doteq \exists X.(\phi(X) \wedge H_{\mathbb{P}}(X, X_{\mathbb{P}}))$$

and similarly for a (transition) formula over X and X'

$$\widehat{\phi}_{\mathbb{P}}(X_{\mathbb{P}}, X'_{\mathbb{P}}) \doteq \exists X, X'.(\phi(X, X') \wedge H_{\mathbb{P}}(X, X_{\mathbb{P}}) \wedge H_{\mathbb{P}}(X', X'_{\mathbb{P}}))$$

Given an interpretation μ of X, the corresponding abstract interpretation of $X_{\mathbb{P}}$ is denoted by $\widehat{\mu}$ and defined as the interpretation $\mu_{X_{\mathbb{P}}}[x_p := \mu(p)]$.

The predicate abstraction of a system $S \doteq \langle X, I, T\rangle$ is obtained by abstracting the initial and the transition conditions, i.e.

$$\widehat{S}_{\mathbb{P}} \doteq \langle X_{\mathbb{P}}, \widehat{I}_{\mathbb{P}}, \widehat{T}_{\mathbb{P}}\rangle$$

In the following, when clear from the context, we omit the $\mathbb{P}$ and write just $\widehat{\phi}$ instead of $\widehat{\phi}_{\mathbb{P}}$.

Example 2 Consider the transition system S of Example 1, the set of predicates $\mathbb{P} = \{c = 0, d = 0\}$ and the set of variables $X_{\mathbb{P}} = \{x_{c=0}, x_{d=0}\}$. The abstract system $\widehat{S}_{\mathbb{P}}$ is defined as: $\langle\{x_{d=0}, x_{c=0}\}, x_{d=0} \wedge x_{c=0}, (x_{d=0} \rightarrow (\neg x'_{d=0} \wedge x_{c=0} \leftrightarrow x'_{c=0})) \wedge (\neg x_{d=0} \rightarrow (x_{c=0} \rightarrow \neg x'_{c=0}))\rangle$.

By construction, $\widehat{S}_{\mathbb{P}} \models \widehat{P}_{\mathbb{P}}$ implies that $S \models P$. Clearly, the converse does not hold, i.e. it is possible that counterexamples in $\widehat{S}_{\mathbb{P}}$ have no counterpart in S (i.e. they are spurious).

Example 3 Consider Figure 2. The figure on the left represents a transition system and its abstraction: each circle is a state and solid arrows represent concrete transitions. States of the same color give equal values to the predicates. There are four abstract states, induced by two predicates; each abstract state corresponds to a different truth assignment (00, 01, 10, 11). Each abstract state encloses the concrete states of the corresponding color. The figure on the right depicts an abstract path (each black rectangle represents various copies of the concrete state space)

with four abstract transitions, 00→10→01→11→11. The sequence 00→10→01 has no counterpart in the concrete system because there does not exist a pair of concrete transitions that connects a concrete state in 00 to a state in 10, and then the same state in 10 to a state in 01.

A spurious abstract counterexample indicates that the abstraction is too coarse. CEGAR [23] is a popular approach to automatically refine an abstraction by extracting information from spurious counterexamples. In predicate abstraction, this is done by adding more predicates. Predicates can be discovered with several techniques, like interpolation, unsat core extraction, or weakest precondition, for which there is a wide literature (e.g. [33, 34, 2]).

The main issue with Predicate Abstraction is that most model checkers deal only with quantifier-free formulas. Thus, the computation of $\widehat{S}_{\mathbb{P}}$ requires the elimination of the existential quantifiers. Efficient algorithms for the computation of predicate abstractions (by means of All-SMT and extensions) have therefore received a lot of interest in the literature [44, 15, 17]. A different, but equally active, research direction has focused instead on approximation techniques in order to reduce the computational cost of eliminating the quantifiers. These approximations typically yield a simpler abstract transition relation, at the cost of including more abstract transitions (and thus potential refinements). In this sense, predicate abstraction as defined above is also called *minimal*, or precise, because it contains the minimal set of transitions given a set of predicates. Some of the most popular approximation techniques are *Cartesian abstraction* [3], *early quantification* [24], *maximum cube length* [1], *localization reduction* [43], *predicate partitioning* [39]. However, an upfront, eager computation of the abstract system is known to be very hard in practice.

2.5.2 Abstraction Refinement via Craig Interpolation

Given an ordered pair of formulas (φ, ψ) in a theory T, a (binary) Craig interpolant is a formula ι that satisfies the following constraints:

(i) $\varphi \models_T \iota$;
(ii) $\psi \wedge \iota \models_T \bot$; and
(iii) all the uninterpreted (in T) symbols occurring in ι occur in both φ and ψ.

The definition can be extended to an ordered sequence of formulas $\varphi_0, \ldots, \varphi_n$ such that $\bigwedge_i \varphi_i \models \bot$, obtaining a *sequence interpolant* $\iota_1, \ldots, \iota_n$ such that:

(i) $\bigwedge_{0 \leq k < i} \varphi_k \models \iota_i$;
(ii) $\iota_i \wedge \bigwedge_{i \leq k \leq n} \varphi_k \models \bot$;
(iii) $\varphi_{i-1} \wedge \iota_i \models \iota_{i+1}$ for all $1 \leq i < n$; and
(iv) all the uninterpreted (in T) symbols occurring in ι_i occur in both $\bigwedge_{0 \leq k < i} \varphi_k$ and $\bigwedge_{i \leq k \leq n} \varphi_k$.

Sequence interpolants can be efficiently computed via SMT techniques for various important theories (see e.g. [21]). One of their prominent applications in formal verification is in automatic refinement of predicate abstractions via CEGAR. The technique, introduced in [33] in the context of software model checking, is described in more detail in §3.3, where we adapt it to our setting.

2.5.3 Implicit predicate abstraction

The idea of implicit predicate abstraction [48] is to avoid the upfront computation of the abstract system, by encoding as a quantifier-free formula the existence of a path in the abstract space. This is based on the following formula:

$$EQ_{\mathbb{P}}(X, \overline{X}) \doteq \bigwedge_{p \in \mathbb{P}} p(X) \leftrightarrow p(\overline{X}) \tag{2}$$

which represent the relation between two concrete states (the interpretation of X and the interpretation of $\overline{X}$) that correspond to the same abstract state: the two states are in the same abstract state if they result in the same valuation to the predicates. In Figure 2 any two states satisfy the $EQ_{\mathbb{P}}$ relation if and only if they have the same color. The formula

$$Path^k_{\mathbb{P}} \doteq \bigwedge_{1 \leq h < k} \Big(T(\overline{X}^{h-1}, X^h) \wedge EQ_{\mathbb{P}}(X^h, \overline{X}^h) \Big) \wedge T(\overline{X}^{k-1}, X^k)$$

is satisfiable iff there exists a path of k steps in the abstract state space. Intuitively, rather than a sequence of contiguous transitions, the encoding represents a sequence of (possibly disconnected) transitions. The starting state of the h-th transition is represented by $\overline{X}^{h-1}$, while X^h is the target state. Subsequent transitions may be disconnected, in the sense that X^h and $\overline{X}^h$ are not forced to be equal, but every gap between two transitions is forced to lay in the same abstract state by $EQ_{\mathbb{P}}(X^h, \overline{X}^h)$.

$BMC^k_{\mathbb{P}}$ encodes the abstract bounded model checking problem, and is obtained from $Path^k_{\mathbb{P}}$ by adding the abstract initial and target conditions:

$$BMC^k_{\mathbb{P}} \doteq I(X^0) \wedge EQ_{\mathbb{P}}(X^0, \overline{X}^0) \wedge Path^k_{\mathbb{P}} \wedge EQ_{\mathbb{P}}(X^k, \overline{X}^k) \wedge \neg P(\overline{X}^k) \tag{3}$$

Interestingly, if we let $EQ_{\mathbb{P}}(X, \overline{X})$ be $\bigwedge_{x \in X}(x = \overline{x})$, i.e. we force the subsequent states to be the same, after substitution we obtain the BMC encoding in S without abstraction.

We remark that implicit abstraction is based on the definition of minimal predicate abstraction: although it does not compute the whole abstract transition relation upfront by quantification, it finds a counterexample only if it is a path in the minimal predicate abstraction of S. However, during the search, only the relevant constraints on the predicates are built: in other words, the lemmas found by the SMT solver may not yield the quantifier elimination of the minimal predicate abstraction, but may give an overapproximation that is sufficient to prove the abstract bounded model checking formula unsatisfiable. In this sense, the lemmas that are found by the SMT solver when checking the satisfiability of an abstract bounded model checking problem can be seen as an on-demand approximation of the minimal abstraction.

3 IC3 with Implicit Abstraction

3.1 The Abstract Space

The main idea of IC3IA is to mimic how IC3 would work on the abstract state space defined by a set of predicates $\mathbb{P}$, and use implicit abstraction to avoid the

explicit computation of the abstract transition relation. In IC3IA, clauses, frames and cubes are formulas over the set $X_{\mathbb{P}}$ of abstract variables.

The set $\mathbb{P}$ contains all predicates in the initial condition I and in the property P. Thus, I and P are Boolean combinations of predicates in $\mathbb{P}$. Note that if ϕ is a Boolean combination of predicates in $\mathbb{P}$, its abstraction $\widehat{\phi}$ is logically equivalent to

$$\phi[X_{\mathbb{P}}/\mathbb{P}] \ \wedge \ \exists X.H_{\mathbb{P}}(X, X_{\mathbb{P}})$$

where $\phi[X_{\mathbb{P}}/\mathbb{P}]$ denotes the Boolean formula obtained by substituting in ϕ the occurrences of each predicate $p(X) \in \mathbb{P}$ with the corresponding abstraction variable x_p. Observe that the second conjunct is independent of the formula being abstracted. Intuitively, it defines the mutual relationships between the abstract names of the predicates. In an SMT setting, μ is a interpretation of $X_{\mathbb{P}}$ satisfying $\exists X.H_{\mathbb{P}}(X, X_{\mathbb{P}})$ iff the induced evaluation of the predicates $\mathbb{P}$ is consistent with respect to the underlying theory (i.e., iff $\bigwedge_{p\in\mathbb{P}} p(X) \leftrightarrow \mu(x_p)$ is satisfiable). This means that computing $\exists X.H_{\mathbb{P}}(X, X_{\mathbb{P}})$ upfront is not required, but can be done on demand by an SMT solver. We use $\overline{\phi}$ to denote $\phi[X_{\mathbb{P}}/\mathbb{P}]$, i.e. the syntactic replacement of the predicates with the corresponding Boolean name variables.

When working in the abstract space of $\widehat{S}$, the critical step for IC3IA is repeatedly checking whether a clause c is inductive relative to the frame F (where c and F are both formulas over $X_{\mathbb{P}}$). This check, if encoded as $RelInd(F, \widehat{T}, c)$, would require the explicit construction of $\widehat{T}$. The key insight underlying IC3IA is to use implicit abstraction to perform the check without actually constructing the abstract transition relation $\widehat{T}$. This is done by checking the quantifier-free formula:

$$\begin{aligned} AbsRelInd(F, T, c, \mathbb{P}) \ \doteq \ & F(X_{\mathbb{P}}) \wedge c(X_{\mathbb{P}}) \wedge H_{\mathbb{P}}(X, X_{\mathbb{P}}) \wedge H_{\mathbb{P}}(X', X'_{\mathbb{P}}) \wedge \\ & EQ_{\mathbb{P}}(X, \overline{X}) \wedge T(\overline{X}, \overline{X}') \wedge EQ_{\mathbb{P}}(\overline{X}', X') \wedge \neg c(X'_{\mathbb{P}}) \quad (4) \end{aligned}$$

The correctness of this step is justified by the following theorem.

Theorem 1 *Consider a set $\mathbb{P}$ of predicates, and two formulas F and c over $X_{\mathbb{P}}$. Then $RelInd(F, \widehat{T}, c)$ and $AbsRelInd(F, T, c, \mathbb{P})$ are equisatisfiable. In particular, if $\mu \models AbsRelInd(F, T, c, \mathbb{P})$, then $\mu_{\mathbb{P}} \models RelInd(F, \widehat{T}, c)$, where $\mu_{\mathbb{P}}$ is the projection of μ over $X_{\mathbb{P}} \cup X'_{\mathbb{P}}$.*

Proof. Suppose $\mu \models AbsRelInd(F, T, c, \mathbb{P})$. Let us denote with $\bar{t}$ and t the projections of μ over $\overline{X} \cup \overline{X}'$ and over $X \cup X'$, respectively. Then $\bar{t} \models T$ and therefore the corresponding abstract transition $\widehat{\bar{t}} \models \widehat{T}$. Since $\mu \models EQ_{\mathbb{P}}(X, \overline{X}) \wedge EQ_{\mathbb{P}}(\overline{X}', X')$, $\widehat{t}$ and $\widehat{\bar{t}}$ are the same abstract transition and therefore $\widehat{t} \models \widehat{T}$. Since $\mu \models H_{\mathbb{P}}(X, X_{\mathbb{P}}) \wedge H_{\mathbb{P}}(X', X'_{\mathbb{P}})$, $\mu_{\mathbb{P}} = \widehat{t}$ and thus $\mu_{\mathbb{P}} \models \widehat{T}$. Note that $\mu_{\mathbb{P}} \models F \wedge c \wedge \neg c'$. Thus, $\mu_{\mathbb{P}} \models RelInd(\widehat{F}, \widehat{T}, \widehat{c})$.

For the other direction, suppose $t_{\mathbb{P}} \models RelInd(F, \widehat{T}, c)$. Then there exists an interpretation t of $X \cup X'$ such that $t \models T$ and $\widehat{t} = t_{\mathbb{P}}$. Therefore, $t_{\mathbb{P}} \cup t \models F(X_{\mathbb{P}}) \wedge c(X_{\mathbb{P}}) \wedge H_{\mathbb{P}}(X, X_{\mathbb{P}}) \wedge H_{\mathbb{P}}(X', X'_{\mathbb{P}}) \wedge EQ_{\mathbb{P}}(X, X) \wedge T(X, X') \wedge EQ_{\mathbb{P}}(X', X') \wedge \neg c(X')$, which concludes the proof. □

3.2 The algorithm

The IC3IA algorithm is shown in Figure 3. IC3IA has the same structure of IC3 (Fig. 1). The trace, i.e. the sequence of frames, is represented as the vector F.

```
bool IC3IA (I, T, P, ℙ):
1.  ℙ := ℙ ∪ {p | p is a predicate in I or in P}
2.  if not Ī ∧ H_ℙ(X, X_ℙ) ⊨ P̄: return FALSE  # violation in the initial states
3.  F_0 := Ī  # first elem of trace is the abstraction of init formula
4.  k := 1, F_k := ⊤  # add a new frame to the trace
5.  while TRUE:
        # blocking phase
6.      while F_k ∧ H_ℙ(X, X_ℙ) ∧ ¬P̄ ⊭ ⊥:
7.          extract a cube c(X_ℙ) from the model of F_k ∧ H_ℙ(X, X_ℙ) ∧ ¬P̄
8.          if not RECBLOCK(c, k):
                # ŝ_0 bwd-reached at depth 0, build π̂ = ŝ_0, ..., ŝ_k
9.              if not CONCRETIZABLE(I, T, P, ℙ, π̂):
10.                 ℙ := ℙ ∪ REFINE(I, T, P, ℙ, π̂)
11.             else return FALSE # counterexample found

        # propagation phase
12.     k := k + 1; F_k := ⊤
13.     for i := 1 to k − 1:
14.         for each clause c ∈ F_i:
15.             if AbsRelInd(F_i, T, c, ℙ) ⊨ ⊥:
16.                 add c to F_{i+1}
17.         if F_i = F_{i+1}:
                # property proved with inductive invariant F_i[ℙ/X_ℙ]
18.             return TRUE

bool RECBLOCK(s, i):
1.  if i = 0: return FALSE # reached initial states
2.  while AbsRelInd(F_{i−1}, T, ¬s, ℙ) ⊭ ⊥:
3.      extract a cube c(X_ℙ) from the model of AbsRelInd(F_{i−1}, T, ¬s, ℙ)
        # c is an (abstract) predecessor of s
4.      if not RECBLOCK(c, i − 1): return FALSE
5.  g := GENERALIZE(¬s, i)  # generalization using AbsRelInd
6.  add g to F_1 ... F_i
7.  return TRUE
```

Fig. 3 The IC3IA algorithm (with changes wrt. the Boolean IC3 in red).

The state of IC3IA is enriched by a set of predicates $\mathbb{P}$, that defines the current precision of the abstraction. The frames are sets of Boolean clauses over $X_{\mathbb{P}}$.

The algorithm consists of a loop, in which each iteration is divided into the blocking and the propagation phase.

The blocking phase starts by picking a Boolean cube $c(X_{\mathbb{P}})$ representing an abstract state in the last frame violating the property (line 6). This is recursively blocked along the trace by checking if $AbsRelInd(F_{i-1}, T, \neg c, \mathbb{P})$ is satisfiable. If the relative induction check succeeds, F_i is strengthened with a generalization of $\neg c$. If the check fails, the recursive blocking continues with an *abstract predecessor* of c, that is, a cube in $F_{i-1} \wedge \neg c$ that leads to c in one step. This recursive blocking results in either strengthening of the trace or in the generation of an *abstract counterexample*. If the counterexample can be simulated on the concrete transition system, then the algorithm terminates with a violation of the property. Otherwise, the algorithm refines the abstraction, adding new predicates to $\mathbb{P}$ so that the abstract counterexample is no more a path of the abstract system.

In the propagation phase, clauses of a frame F_i that are inductive relative to F_i using $\widehat{T}$ are propagated to the following frame F_{i+1}. As for IC3, if two consecutive frames are equal, we can conclude that the property is satisfied by the abstract transition system, and therefore also by the concrete one.

3.3 Simulation and refinement

When IC3IA finds an abstract counterexample $\widehat{\pi} \doteq \widehat{s_0}, \widehat{s_1}, \ldots, \widehat{s_k}$, in form of an interpretation of $\bigcup_{i=0}^{k} X^i_{\mathbb{P}}$, we check if it can be concretized, i.e. there exists a corresponding counterexample in S. The CONCRETIZABLE routine checks whether such a concrete counterexample exists, and if not the REFINE primitive is called to increase the precision of the abstraction, by adding new predicates to $\mathbb{P}$.

The abstract counterexample $\widehat{\pi}$ is simulated in the concrete system S encoding all the paths of S up to k steps, restricted to $\widehat{\pi}$.

$$\mathit{Simulate}(T, \mathbb{P}, \widehat{\pi}) \doteq \bigwedge_{0 \le i < k} T(X^i, X^{i+1}) \wedge \bigwedge_{0 < i \le k} \widehat{s_i}(X^i_{\mathbb{P}})[\mathbb{P}^i / X^i_{\mathbb{P}}] \tag{5}$$

If the formula is satisfiable, then the interpretation of the concrete variables $X^0, X^1, \ldots, X^k$ yield a concrete counterexample $s_0, s_1, \ldots, s_k$ that is a witnesses for $S \not\models P$ (note that $s_0 \models I$ and $s_k \models \neg P$ since $\widehat{s}_0 \models \overline{I}$ and $\widehat{s}_k \models \neg\overline{P}$). Otherwise, $\widehat{\pi}$ is spurious and the abstraction must be refined by adding new predicates.

The REFINE$(I, T, P, \mathbb{P}, \pi)$ procedure is somewhat orthogonal to IC3IA, and can be done in various ways [33,34,2]. The only requirement is that the new set of predicates should be sufficient to remove the spurious counterexample. We now discuss the use of SMT-based interpolation to discover new predicates, similarly to [33]. We partition $\mathit{Simulate}(T, \mathbb{P}, \widehat{\pi})$ into a sequence of formulas $\varphi_0, \ldots, \varphi_k$ as follows:

$$\mathit{Simulate}(T, \mathbb{P}, \widehat{\pi}) \doteq \underbrace{\widehat{s_0}(X^0_{\mathbb{P}})[\mathbb{P}^0 / X^0_{\mathbb{P}}]}_{\varphi_0} \wedge \bigwedge_{1 \le i \le k} \underbrace{T(X^{i-1}, X^i) \wedge \widehat{s_i}(X^i_{\mathbb{P}})[\mathbb{P}^i / X^i_{\mathbb{P}}]}_{\varphi_i}$$

We then compute a sequence interpolant $\iota_1, \ldots, \iota_k$ for $\varphi_0, \ldots, \varphi_k$ (using the technique of [21]). Let $\iota_j[X/X^j]$ denote the formula obtained from ι_j by replacing all the variables $v^j \in X^j$ with $v \in X$, and let $\rho(\phi)$ be a function that returns the set of all the atomic formulas occurring in the formula ϕ (e.g. $\rho((x \ge 1) \wedge \neg(y + x = 0) \wedge b)$ returns $\{(x \ge 1), (y + x = 0), b\}$). We refine the abstraction by adding the set of predicates $\mathbb{P}^{\text{new}} \doteq \bigcup_{1 \le i \le k} \rho(\iota_i[X/\ X^i])$ to $\mathbb{P}$. As shown by the following theorem, the predicates will rule out the counterexample $\widehat{\pi}$ from the abstraction.[2]

Theorem 2 (Progress of interpolation-based refinement) *Let $S \doteq \langle I, T, P \rangle$, $\mathbb{P}$, $\pi \doteq s_0, \ldots, s_k$ be as above, such that $\mathit{Simulate}(T, \mathbb{P}, \widehat{\pi})$ is unsatisfiable. Let $\iota_1, \ldots, \iota_k$ be a sequence interpolant produced for the partitioning of $\mathit{Simulate}(T, \mathbb{P}, \widehat{\pi})$ into $\varphi_0, \ldots, \varphi_k$, and let $\mathbb{P}^{new} \doteq \bigcup_{1 \le i \le k} \rho(\iota_i[X/X^i])$. Then, $S_{\mathbb{P}^{new} \cup \mathbb{P}}$ contains no (abstract) counterexample path $\widehat{z_0}, \ldots, \widehat{z_k}$ such that $\widehat{z_i} \models \widehat{s_i}$ for all $0 \le i \le k$.*

[2] This is a reformulation of a well-known result stated (without proof) in [33], adapted to our context.

Proof.(Sketch) Since $\iota_1, \ldots, \iota_k$ is a sequence interpolant, we have that

$$\iota_i(X^i)[X/X^i] \wedge T(X, X') \wedge \widehat{s_{i-1}}(X_{\mathbb{P}})[\mathbb{P}/X_{\mathbb{P}}] \models \iota_{i+1}(X^{i+1})[X'/X^{i+1}]$$

for all $0 < i < k$. That is, each ι_{i+1} is an overapproximation of the image of $\iota_i \wedge \widehat{s_{i-1}}$ wrt. T. Together with the fact that $\bigwedge_{0 \leq k < i} \varphi_k \models \iota_i$, by induction it holds that each ι_i is an overapproximation of the states reachable in S with a path $z_0, \ldots, z_{i-1}$ such that, for all $0 \leq j \leq i-1$, $z_j \models \widehat{s_j}$. Since $\mathbb{P}^{\text{new}}$ contains all the atoms in the ι_i's and the predicate abstraction is minimal, it follows that $\widehat{\iota_i}_{\mathbb{P}^{\text{new}} \cup \mathbb{P}} \models \iota_i$, and that each $\widehat{\iota_i}$ is an overapproximation of the states reachable in $S_{\mathbb{P}^{\text{new}} \cup \mathbb{P}}$ with a path $\widehat{z}_0, \ldots, \widehat{z}_{i-1}$ such that, for all $0 \leq j \leq i-1$, $\widehat{z}_j \models \widehat{s_j}$. Since $\iota_k \models \bot$, $S_{\mathbb{P}^{\text{new}} \cup \mathbb{P}}$ contains no path $\widehat{z}_0, \ldots, \widehat{z_k}$ such that $\widehat{z_i} \models \widehat{\iota_i}$ for all i. □

3.4 Discussion

The IC3IA algorithm has the following distinguishing features. First, it is very close to the original, Boolean algorithm. The differences are: (i) IC3IA picks cubes instead of concrete states in the test to end the blocking phase (line 7); (ii) IC3IA uses *AbsRelInd* instead of *RelInd* (line 15, and line 2 in RECBLOCK); (iii) when a counterexample is found, instead of returning FALSE as in the Boolean case, it is checked for spuriousness (line 9), and, if needed, new predicates are added to refine the abstraction (line 10). In all these three points, IC3IA uses an SMT solver so that the facts asserted at the Boolean abstract level are ensured to be consistent with the SMT theories used in the predicates. However, IC3IA operates primarily on Boolean data structures (trace, clauses), and confines the theory reasoning within these checks. This makes it easy to implement and reuse the IC3 code, but at the same time it leverages the theory information available.

Second, IC3IA is highly incremental, in the sense that the set of predicates increases monotonically after a refinement (i.e. we always add new predicates to the existing set of predicates). Thus, the transition relation is monotonically strengthened (i.e. $\widehat{T}_{\mathbb{P} \cup \mathbb{P}^{\text{new}}} \models \widehat{T}_{\mathbb{P}}$). This allows us to *keep all the clauses* in the IC3IA frames after a refinement, enabling a fully incremental approach. Simulation and refinement can also be performed incrementally, reusing the same solver instance used for simulating the abstract counterexample.

Finally, concerning the initial set of predicates used by our algorithm, we require that it includes all the predicates of the initial states and of the safety property, as this simplifies the exposition, the proof of correctness, and the implementation of the procedure. However, we can easily limit the number of predicates in the initial abstraction by relying on simple model transformations. The original safety property P, which may be an arbitrary Boolean combination of predicates, can be replaced by a single Boolean variable p_b, ensuring that the invariant $P \leftrightarrow p_b$ holds in the transition system. For the initial state, it is sufficient to add an initial "reset" state (encoded with an additional Boolean variable r). The system is initially in the "reset" state (r is true in the initial states) and non-deterministically moves to an initial state of the original system, exiting the "reset" state (the transition relation states that $r \rightarrow (\neg r' \wedge I)$); from that point the system moves according to the original transition relation (i.e. $\neg r \rightarrow (\neg r' \wedge T)$); finally, the safety property is changed to always hold in the reset state (i.e. $\neg r \rightarrow p_b$). So, overall:

– the new initial condition is r;
– the new transition condition is $(\neg r') \wedge (r \rightarrow I') \wedge (\neg r \rightarrow T) \wedge (P' \rightarrow p'_b)$;
– the new property is $\neg r \rightarrow p_b$.

3.5 Example

Let us consider the transition system $S = \langle \{c, d\}, c = 0 \wedge d = 0, c' = c + d \wedge d' = d + 1 \rangle$ of Example 1. To give an intuition of the IC3IA behavior we show its main steps when proving the property $P \doteq (d \leq 3) \vee \neg(c \leq d)$ on S. We describe the first three iterations of the IC3IA main loop, showing all the important steps of the algorithm, like the blocking phase using the abstract relative induction check and the refinement.

IC3IA proves that the property holds performing 8 iterations of the IC3IA main loop, refining 4 times the abstraction and ending with a total of 9 predicates.

The initial set of predicates, taken from the initial formula and the property, is $\mathbb{P}_0 \doteq \{(c = 0), (d = 0), (d \leq 3), (c \leq d)\}$ and the initial status of the frames is $F_0 \doteq x_{c=0} \wedge x_{d=0}$.

First iteration. In the first iteration the algorithm checks that $F_0 \wedge H_{\mathbb{P}}(X, X_{\mathbb{P}}) \wedge \neg\overline{P}$ is unsatisfiable, adding the empty frame F_1.

Second iteration. IC3IA finds a pair $(c_0, 1)$ where $c_0 = x_{c=0} \neg x_{d=0} \wedge \neg x_{d\leq 3} \wedge x_{c\leq d}$ and such that $c_0 \wedge H_{\mathbb{P}}(X, X_{\mathbb{P}}) \models F_1 \wedge \neg P$. Then, IC3IA tries to block $(c_0, 1)$ in the frame F_0: c_0 is blocked by F_0, since $AbsRelInd(F_0, T, c_0, \mathbb{P}_0)$ is unsatisfiable. In fact, $AbsRelInd(F_0, T, c_0, \mathbb{P}_0)$ is the formula:

$$
\begin{aligned}
AbsRelInd(F_0, T, c_0, \mathbb{P}_0) \doteq\ & x_{c=0} \wedge x_{d=0} \wedge && [F_0(X_{\mathbb{P}})] \\
& x_{c=0} \wedge \neg x_{d=0} \wedge \neg x_{d\leq 3} \wedge x_{c\leq d} \wedge && [c_0(X_{\mathbb{P}})] \\
& x_{d=0} \leftrightarrow (d = 0) \wedge x_{c=0} \leftrightarrow (c = 0) \wedge && [H_{\mathbb{P}}(X, X_{\mathbb{P}})] \\
& x_{d\leq 3} \leftrightarrow (d \leq 3) \wedge x_{c\leq d} \leftrightarrow (c \leq d) \wedge \\
& x'_{d=0} \leftrightarrow (d' = 0) \wedge x'_{c=0} \leftrightarrow (c' = 0) \wedge && [H_{\mathbb{P}}(X', X'_{\mathbb{P}})] \\
& x'_{d\leq 3} \leftrightarrow (d' \leq 3) \wedge x'_{c\leq d} \leftrightarrow (c' \leq d') \wedge \\
& (d = 0) \leftrightarrow (\overline{d} = 0) \wedge (c = 0) \leftrightarrow (\overline{c} = 0) \wedge && [EQ_{\mathbb{P}}(X, \overline{X})] \\
& (d \leq 3) \leftrightarrow (\overline{d} \leq 3) \wedge (c \leq d) \leftrightarrow (\overline{c} \leq \overline{d}) \wedge \\
& (\overline{c'} = \overline{c} + \overline{d}) \wedge (\overline{d'} = \overline{d} + 1) \wedge && [T(\overline{X}, \overline{X}')] \\
& (\overline{d'} = 0) \leftrightarrow (d' = 0) \wedge (\overline{c'} = 0) \leftrightarrow (c' = 0) \wedge \\
& (\overline{d'} \leq 3) \leftrightarrow (d' \leq 3) \wedge (\overline{c'} \leq \overline{d'}) \leftrightarrow (c' \leq d') \wedge && [EQ_{\mathbb{P}}(\overline{X}', X')] \\
& \neg(x'_{c=0} \wedge \neg x'_{d=0} \wedge \neg x'_{d\leq 3} \wedge x'_{c\leq d}) && [\neg c_0(X'_{\mathbb{P}})]
\end{aligned}
$$

Since $AbsRelInd(F_0, T, c_0, \mathbb{P}_0) \models \bot$, IC3IA tries to generalize c_0 to block more states in the frame F_1. One possible generalization is $\neg x_{d\leq 3}$, since $AbsRelInd(F_0, T, \neg x_{d\leq 3}, \mathbb{P}_0) \models \bot$. IC3IA adds the negation of the generalized cube, $x_{d\leq 3}$, to F_1. Now the frame F_1 does not intersect the bad states $\neg P$, and thus IC3IA adds the frame F_2 and proceeds to the propagation phase (in this case there are no clauses in a frame that can be propagated to the successive frame).

Third iteration. IC3IA finds a chain of pairs: $(\neg x_{c=0} \wedge \neg x_{d=0} \wedge \neg x_{d\leq 3} \wedge x_{c\leq d}, 2)$, $(x_{c=0} \wedge \neg x_{d=0} \wedge x_{d\leq 3} \wedge x_{c\leq d}, 1)$ and $(x_{c=0} \wedge x_{d=0} \wedge x_{d\leq 3} \wedge x_{c\leq d}, 0)$ by finding a satisfiable assignment to $\neg x_{c=0} \wedge \neg x_{d=0} \wedge \neg x_{d\leq 3} \wedge x_{c\leq d} \wedge H_{\mathbb{P}}(X, X_{\mathbb{P}}) \models F_2 \wedge \neg P$, and then recursively calling the function RecBlock. The last pair is at depth 0, hence IC3IA found an abstract counterexample. The counterexample path cannot be simulated on the concrete system, due to the transition from the second to the third state of the path. In the third state we have that the abstract path requires that $\neg(d \leq 3)$, but in the concrete system d must be lower or equal than 2 after two steps. The refinmement finds $(d \leq 2)$ as new predicate; now the abstraction is determined by the set of predicates $\mathbb{P}_1 \doteq \mathbb{P}_0 \cup \{(d \leq 2)\}$.

After the refinement, IC3IA checks if there exists another cube that violates P at frame F_2. The search still finds the pairs: $(\neg x_{c=0} \wedge \neg x_{d=0} \wedge \neg x_{d\leq 3} \wedge x_{c\leq d}, 2)$ and $(x_{c=0} \wedge \neg x_{d=0} \wedge x_{d\leq 3} \wedge x_{c\leq d} \wedge \neg x_{d\leq 2}, 1)$. $(x_{c=0} \wedge \neg x_{d=0} \wedge x_{d\leq 3} \wedge x_{c\leq d} \wedge \neg x_{d\leq 2}, 1)$ is blocked by F_0, and thus IC3IA adds $x_{d\leq 2}$ to F_1; then $(\neg x_{c=0} \wedge \neg x_{d=0} \wedge \neg x_{d\leq 3} \wedge x_{c\leq d}, 2)$ is blocked by F_1, thus IC3IA adds $x_{d\leq 3}$ to F_2. At this point F_2 satisfies the property and IC3IA adds the frame F_3, peforming the propagation phase (it still does not propagate any clause).

Final result The final set of predicates found by IC3IA is $\{(c = 0), (d = 0), (d \leq 3), (c \leq d), (d \leq 2), (d \leq 1), (1 \leq c), (3 \leq c)\}$ and the final inductive invariant is:

$$(\neg(c = 0) \vee (d \leq 2)) \wedge ((d \leq 1) \vee (1 \leq c)) \wedge ((c = 0) \vee \neg(d \leq 1)) \wedge$$
$$(\neg(c = 0) \vee (c \leq d)) \wedge ((d \leq 2) \vee (1 \leq c)) \wedge ((d \leq 2) \vee (3 \leq c)) \wedge ((d \leq 3) \vee \neg(c \leq d))$$

3.6 Correctness

In the following lemmas and proofs, we use $\mathit{Lift}_{\mathbb{P}}(X_{\mathbb{P}})$ to denote a Boolean formula equivalent to $\exists X.H_{\mathbb{P}}(X, X_{\mathbb{P}})$.

Lemma 1 (Invariants) *The following conditions are invariants of* IC3IA*:*

1. $F_0 \wedge \mathit{Lift}_{\mathbb{P}} \models \widehat{I}$;
2. *for all* $i < k$, $F_i \models F_{i+1}$;
3. *for all* $i < k$, $F_i \wedge \widehat{T}(X_{\mathbb{P}}, X'_{\mathbb{P}}) \models F_{i+1}$;
4. *for all* $i < k$, $F_i \wedge \mathit{Lift}_{\mathbb{P}} \models \widehat{P}$.

Proof. We prove now that the conditions (1-4) are loop invariants for the main IC3IA loop (line 5).

Note that $\widehat{I}$ is logically equivalent to $\overline{I} \wedge \exists X.H_{\mathbb{P}}(X, X_{\mathbb{P}})$, since I is a Boolean combination of predicates in $\mathbb{P}$. Thus, condition 1 holds initially since $F_0 = \overline{I}$. Moreover it is preserved by the loop, since F_0 is never changed.

For condition (4), note that when entering the loop it holds (from line 2) that $\overline{I} \wedge H_{\mathbb{P}}(X, X_{\mathbb{P}}) \models \overline{P}$, and thus $F_0 \wedge \mathit{Lift}_{\mathbb{P}} \models \widehat{P}$. Thus condition 4 holds at the beginning of the loop.

Note also that the frames can only be strengthened. Thus the condition 4 can be violated only at line 12. However, when the inner loop ends, we are guaranteed that $F_k \wedge H_{\mathbb{P}}(X, X_{\mathbb{P}}) \models P$ holds. Thus, condition 4 is preserved when k is increased.

The invariant conditions 2-3 hold trivially when entering the loop.

We now prove that they are preserved by the inner loop at line 6. The loop may change the content of a frame F_i adding a new clause $\neg s$ while recursively blocking

a cube (s, i) (line 6 of RecBlock). This happens when $i > 0$ and s can reach $\neg\widehat{P}$ in $k-i$ steps, and $AbsRelInd(F_{i-1}, T, \neg s, \mathbb{P})$ is unsatisfiable. By inductive hypothesis, $\widehat{P}$ is satisfied in the first $k-1$ steps. Thus, s is not reachable in $k-1-k+i = i-1$ steps. Since $AbsRelInd(F_{i-1}, T, \neg s, \mathbb{P})$ is unsatisfiable, by Theorem 1, s cannot be reached in i steps. Thus conditions 2-3 are preserved.

In the loop the set of predicates $\mathbb{P}$ may change at line 10. Note that the invariant conditions still hold in this case. Let $\mathbb{P}^{\text{new}} = \mathbb{P} \cup \text{Refine}(I, T, P, \mathbb{P}, \widehat{\pi})$. In particular, 3 holds because if $\mathbb{P} \subseteq \mathbb{P}^{\text{new}}$, then $\widehat{T}(X_{\mathbb{P}^{\text{new}}}, X'_{\mathbb{P}^{\text{new}}}) \models \widehat{T}(X_{\mathbb{P}}, X'_{\mathbb{P}})$. Finally, the propagation phase maintains all the invariants (2-3), by the definition of abstract relative induction $AbsRelInd(F_i, T, c, \mathbb{P}^{\text{new}})$ and Theorem 1. □

Lemma 2 *If* IC3ia$(I, T, P, \mathbb{P})$ *returns* True*, then* $\widehat{S_{\mathbb{P}}} \models \widehat{P_{\mathbb{P}}}$.

Proof. Let us define, for all $i < k$, $\widehat{F_i}$ as $F_i \wedge Lift_{\mathbb{P}}$. Thus, from Lemma 1, all the invariant conditions of the IC3 algorithm hold for the abstract frames: 1) $\widehat{F_0} = \widehat{I}$; for all $i < k$, 2) $\widehat{F_i} \models \widehat{F_{i+1}}$; 3) $\widehat{F_i} \wedge \widehat{T} \models \widehat{F'_{i+1}}$; and 4) $\widehat{F_i} \models \widehat{P}$.

By assumption IC3ia returns True and thus $F_{k-1} = F_k$, and thus $\widehat{F_{k-1}} = \widehat{F_k}$. Since the conditions (1-4) hold, we have that $\widehat{F_{k-1}}$ is an inductive invariant that proves $\widehat{S} \models \widehat{P}$. □

Lemma 3 (Abstract counterexample) *If* IC3ia *finds an abstract counterexample* $\widehat{\pi} \doteq \widehat{s_0}, \widehat{s_1}, \ldots, \widehat{s_k}$ *(line 8), then* $\widehat{\pi}$ *is a path of* $\widehat{S}$ *violating* $\widehat{P}$.

Proof. We show that $\widehat{\pi} \doteq \widehat{s_0}, \widehat{s_1}, \ldots, \widehat{s_k}$ is a path of $\widehat{S}$ violating $\widehat{P}$. For all i, $0 \leq i \leq k$, we have that $\widehat{s_i} \models F_i \wedge Lift_{\mathbb{P}}$ (by line 7). Since $\widehat{s_0} \models F_0 \wedge Lift_{\mathbb{P}}$ and by Lemma 1, $\widehat{s_0} \models \widehat{I}$. Moreover, $\widehat{s_k} \models \widehat{\neg P}$ (by line 7). Then, for all i, $0 \leq i < k$, $\widehat{s_i} \wedge \widehat{T} \models \widehat{s_{i+1}}$, since by Lemma 1 $F_i \wedge \widehat{T} \models F_{i+1}$. □

Theorem 3 (Soundness) *Let* $S = \langle X, I, T\rangle$ *be a transition system,* P *a safety property and* $\mathbb{P}$ *be a set of predicates over* X*. The result of* IC3ia$(I, T, P, \mathbb{P})$ *is correct.*

Proof. If IC3ia $(I, T, P, \mathbb{P})$ returns True, then $\widehat{S_{\mathbb{P}}} \models \widehat{P_{\mathbb{P}}}$ by Lemma 2, and thus $S \models P$. If IC3ia $(I, T, P, \mathbb{P})$ returns False, then the simulation of the abstract counterexample in the concrete system succeeded, and thus $S \not\models P$. □

Theorem 4 (Relative completeness) *Suppose that for some set* $\mathbb{P}$ *of predicates,* $\widehat{S_{\mathbb{P}}} \models \widehat{P_{\mathbb{P}}}$*. If, at a certain iteration of the main loop,* IC3ia *has* $\mathbb{P}$ *as set of predicates, then* IC3ia *returns* True.

Proof. Let us consider the case in which, at a certain iteration of the main loop, $\mathbb{P}$ is as defined in the premises of theorem. At every following iteration of the loop, IC3ia either finds an abstract counterexample $\widehat{\pi}$ or strengthens a frame F_i with a new clause over $X_{\mathbb{P}}$. The first case is not possible, since, by Lemma 3, $\widehat{\pi}$ would be a path of $\widehat{S}$ violating the property. Therefore, at every iteration, IC3ia strengthens some frame with a new clause. Since the number of clauses over $X_{\mathbb{P}}$ is finite and, by Lemma 1, for all i, $F_i \models F_{i+1}$, IC3ia will eventually find that $F_i = F_{i+1}$ for some i and return True. □

Theorem 5 (Progress) *Let* $S \doteq \langle X, I, T\rangle$ *be a transition system,* P *a safety property and* $\mathbb{P}$ *a set of predicates over* X*, such that* I, T, P *and the predicates* $\mathbb{P}$ *are quantifier-free formulas in a first-order theory* $\mathcal{T}$ *whose ground satisfiability is decidable. Then* IC3ia$(I, T, P, \mathbb{P})$ *either returns* True*, or it finds an abstract counterexample* $\widehat{\pi}$.

Proof. Since by hypothesis the satisfiability of quantifier-free formulas in $\mathcal{T}$ is decidable, each SMT call in IC3IA terminates. Since $\mathbb{P}$ (and so $X_{\mathbb{P}}$) is fixed and finite, the number of possible cubes and clauses examined and generated during the blocking phase is finite. Then, the theorem holds by Theorem 1 and the completeness of IC3 for finite-state systems. □

4 Related Work

Among the existing abstraction techniques, predicate abstraction [26] has been successfully applied to the verification of infinite-state transition systems, such as software [46]. Implicit abstraction [48] was first used with k-induction to avoid the explicit computation of the abstract system. In our work, we exploit implicit abstraction in IC3 to avoid theory-specific generalization techniques, widening the applicability of IC3 to transition systems expressed over some background theories. Moreover, we provided the first integration of implicit abstraction in a CEGAR loop.

The IC3 [12] algorithm has been widely applied to the hardware domain [25, 16] to prove safety and also as a backend to prove liveness [11]. In [49], IC3 is combined with a lazy abstraction technique in the context of hardware verification. The approach has some similarities with our work, but it is limited to Boolean systems, it uses a "visible variables" abstraction rather than PA, and applies a modified concrete version of IC3 for refinement. Similarly, in [5], IC3 is combined with localization reduction for the verification of hardware designs, but is limited to Boolean systems and the integration is shallow in the sense that it consists in exploiting the over-approximations of incomplete run of IC3 to refine the abstraction.

Several approaches adapted the original IC3 algorithm to deal with infinite-state systems [18,35,42,36,52,8,9,45,41,37]. The techniques presented in [18,35, 9] extend IC3 to verify systems described in the linear real arithmetic theory. In contrast to these approaches, we do not rely on theory specific generalization procedures. In [18], a possibly expensive real quantifier elimination is adopted. In [35], the generalization is based on interpolation and does not exploit relative induction. In [9], the frames are restricted to be convex polyhedra, while quantifier elimination and polyhedral abstract interpretation are integrated for the generalization procedure. Differently from IC3IA, extending these methods to different theories requires ad-hoc techniques.

The approaches presented in [42,36] are restricted to timed automata, exploiting the abstraction given by the region graph or by the clock zones. While we could restrict the set of predicates used by IC3IA to regions/zones, our technique is applicable to a much broader class of systems, and it also allows us to apply conservative abstractions.

IC3 was also generalized to the bit-vector theory in [52]. The approach is based on an ad-hoc extension, that may not handle efficiently some bit-vector operators. Instead, our approach is not specific for bit-vectors.

In [18,45], IC3 was extended to exploit the control-flow graph of software programs. As shown also in [19], the approach is orthogonal to the usage of predicate abstraction and can be exploited also with IC3IA.

CTIGAR [8] is perhaps the most closely related approach, since it also embeds an abstraction refinement scheme in IC3. In CTIGAR the simulation and refinement may be performed after each generalization of a counterexample to induction and after each relative induction check, while in IC3IA we delay the simulation and refinement only when we find an abstract counterexample. Another key difference is the precision of the abstraction. In our approach, at each step of the algorithm, we analyze a precise predicate abstraction of the concrete system; CTIGAR relies on cartesian abstraction, that is not precise in general. As a consequence, CTIGAR may require more refinement steps; furthermore, in the refinement, CTIGAR has to consider generic Boolean combinations of predicates, instead of single predicates.

The main focus of [37,41] is to verify programs that manipulate the heap, while we focus on programs that can be expressed in the LRA, BV, or LIA theory. As in our work, [37] analyzes the abstract state space using IC3 without computing the abstract system explicitly. Differently from us, there is no automatic refinement of the abstraction as soon as the algorithm finds a spurious counterexample. A key aspect of IC3IA is to automatically refine the abstraction by adding new predicates incrementally, without restarting the analysis from scratch. In principle, our incremental refinement could be also applied to [37]. In [41] the authors employ a "diagram based abstraction" to infer universally quantified invariants. A diagram is an existentially quantified formula that represents the set of all the extensions of a finite model of a formula. The diagram is used to abstract a bad cube and a counterexample to induction in IC3, providing a mean to infer an universally quantified invariant. This approach does not require the refinement of the abstraction.

Interesting variants of IC3 have been presented in [30,50]. These focus on finite-state systems and the techniques are orthogonal to the proposed in this paper.

5 Experimental Evaluation

We have implemented the algorithms described in the previous sections within NUXMV [14], on top of the SMT-based extension of IC3 presented in [18]. We rely on MATHSAT [20] as backend SMT solver. The discovery of new predicates for abstraction refinement is performed using the interpolation procedures implemented in MATHSAT, following [33]. For inductive clause generalization, we use the simple iterative procedure described in [25] (however, since we use an SMT solver instead of a SAT solver, our inductive generalization is still modulo theory).

We organize our experimental evaluation by first comparing IC3IA with other approaches, and then analyzing some of its features. All the experiments have been performed on a cluster of 64-bit Linux machines with a 2.7 Ghz Intel Xeon X5650 CPU (our implementation is not parallel, we distributed the execution of the experiments on different machines) with a memory limit set to 3Gb and a time limit of 1200 seconds (unless otherwise specified). The tools and benchmarks used in the experiments are available at `http://es.fbk.eu/people/griggio/papers/fmsd-ic3ia.tar.bz2`.

Algorithm/Tool	LRA	BV	LIA	CFG(LIA)
IC3IA	✓	✓	✓	✓
CTIGAR-REIMPL	✓	✓	✓	✓
IC3QE	✓			
Z3	✓	✓	✓	✓
CTIGAR-CAV14				✓
PKIND			✓	
KIND2-PDR			✓	
KIND2-PARALLEL			✓	
JKIND-IC3IA			✓	
ABC-PDR		✓		
ABC-DPROVE		✓		
NUXMV-IC3-SAT		✓		

Table 1 Applicability of the tools. The first column (**Algorithm/Tools**) shows the list of tools considered in the experimental evaluation, while the other columns (LRA, BV, LIA, CFG(LIA)) tells what benchmark family (see the "Benchmark" section) are supported by each tool (a ✓indicates that the tool supports the benchmark set).

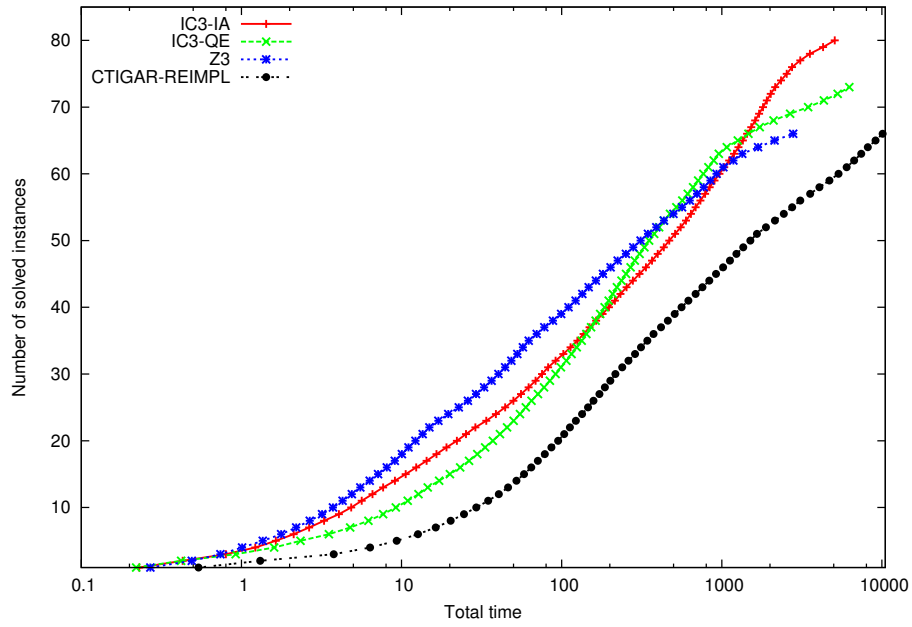
5.1 Tools

We compare the following tools, whose properties are summarized in Table 1:

- IC3IA: the implementation of the algorithm described in Section 3;
- CTIGAR-REIMPL: our reimplementation of the CTIGAR algorithm [8] within the same software platform as IC3IA;
- IC3QE: the IC3 extension for infinite state systems over LRA presented in [18]. IC3QE is based on underapproximated quantifier elimination, and only supports the LRA theory;
- Z3: the IC3 extension for SMT described in [35], as implemented in the latest version of the Z3 solver;
- PKIND: an induction-based model checker for Lustre programs [40], using the LIA theory for expressing constraints;
- KIND2-PDR: the implementation of IC3 for the LIA theory of KIND2, a model checker for Lustre programs[3]. This implementation of IC3 computes approximated pre-images, similarly to IC3QE. The description of the implementation of KIND2 may be found at `https://github.com/kind2-mc/kind2/blob/develop/doc/usr/content/1_techniques/1_techniques.md`.
- KIND2-PARALLEL: this configuration of the KIND2 model checker runs several algorithm in parallel, exchanging the discovered invariants. In this configuration, KIND2 runs in parallel k-induction, IC3, BMC and two processes that performs a template-based invariant generation.
- JKIND-IC3IA: the implementation of IC3IA for the LIA theory of JKIND, a model checker for Lustre programs[4]. This is an implementation of the algorithm described in the present paper made by a completelly independent team, using a different programming language, front-end, and underlying SMT solver.
- CTIGAR-CAV14: the original implementation of the CTIGAR algorithm [8]. The tool takes as input problems written as annotated programs in a subset of the C language, and only supports the LIA theory.

[3] `http://kind2-mc.github.io/kind2/`

[4] `https://github.com/agacek/jkind/`



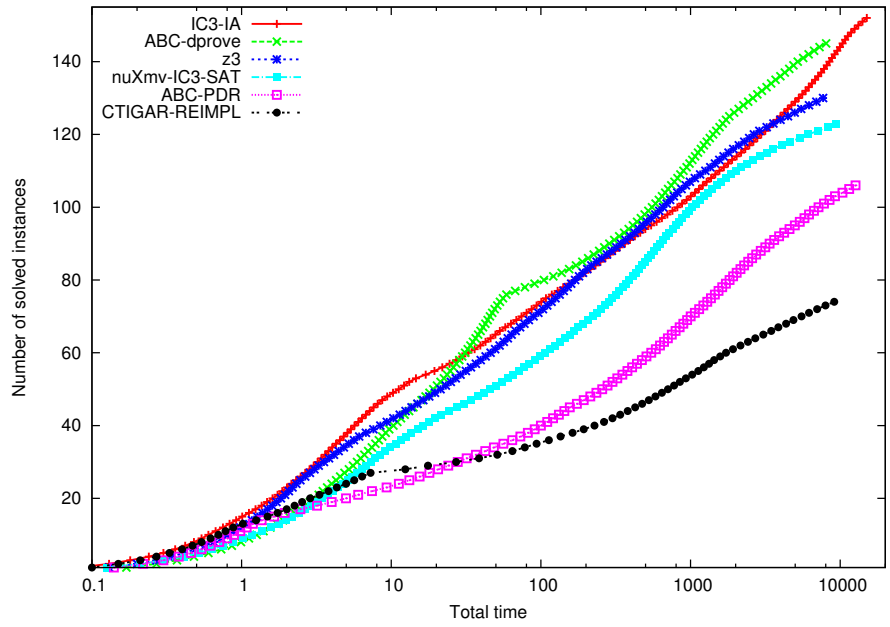
Algorithm/Tool	# Solved (out of 99)	Tot time
IC3IA	81	5072
IC3QE	74	6255
CTIGAR-REIMPL	68	11196
Z3	67	2793

Fig. 4 Experimental results on LRA benchmarks.

- ABC-PDR: the implementation of finite-state IC3, as available in ABC [25];
- ABC-DPROVE: the DPROVE algorithm of ABC, which combines various different techniques for bit-level verification (including IC3);
- NUXMV-IC3-SAT: the implementation of finite-state IC3, as available in the latest version of NUXMV [14].

With the exception of Z3, none of the tools is able to handle the full set of problems supported by IC3IA and CTIGAR-REIMPL.

Compared to the experimental evaluation in [19], we concentrate on IC3IA over transition systems, and do not include the results for the various versions of TreeIC3, that is able to deal with imperative-style programs, nor for the other recent adaptation of IC3 to control-flow automata presented in [45]. The availability of a CFG is somewhat orthogonal to the content of this paper. In fact Implicit Abstraction has been integrated within TreeIC3, and the results in [19] demonstrate significant advantages with respect to the concrete version. We also disregard ATMOC [42] since ATMOC targets timed systems. Anyway, the comparison against IC3IA in [19] was uniformly in favor of IC3IA.



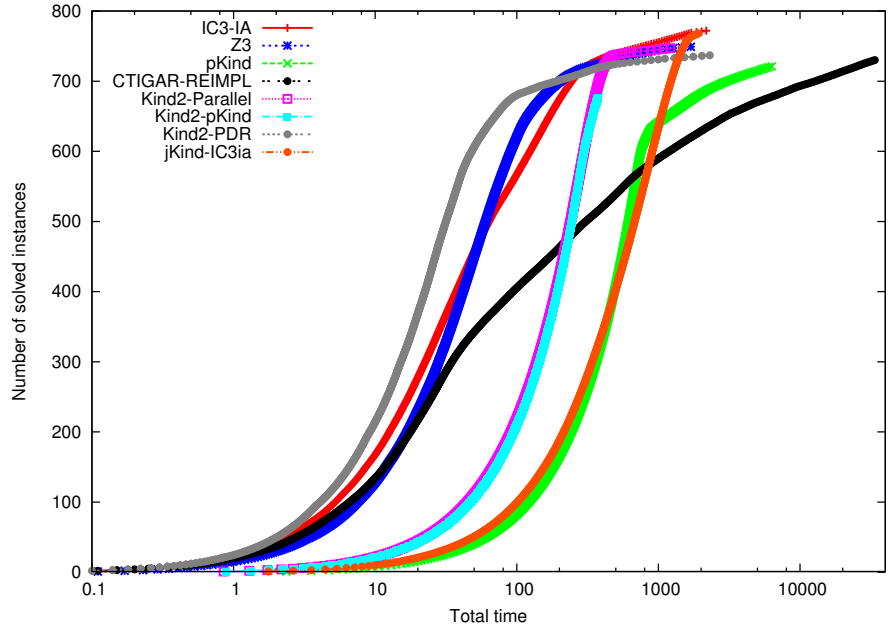
Algorithm/Tool	# Solved (out of 205)	Tot time
IC3IA	153	15050
ABC-DPROVE	146	8048
Z3	131	7674
NUXMV-IC3-SAT	124	9413
ABC-PDR	107	12654
CTIGAR-REIMPL	75	9125

Fig. 5 Experimental results on bit-vector benchmarks from software verification.

5.2 Benchmarks

We have collected benchmarks from several sources, and organized them in four groups:

- The symbolic transition systems used in [18], over the LRA theory. The **LRA** group of benchmarks consists of 99 instances. For this group, we compare IC3IA, IC3QE, Z3 and CTIGAR-REIMPL.
- Symbolic transition systems over bit-vectors, generated from software verification problems. The **BV** group of benchmarks consists of 205 benchmarks. More specifically, we used:
  - all the benchmarks used in [18], but using BV instead of LRA as background theory;
  - the instances of the `bitvector` set of the Software Verification Competition SV-COMP [6];
  - the instances from the test suite of InvGen [28], a subset of which was used also in [52].



Algorithm/Tool	# Solved (out of 790)	Tot time
IC3IA	773	2175
JKIND-IC3IA	769	1913
Z3	750	1691
CTIGAR-REIMPL	731	33719
PKIND	722	6297
KIND2-PDR	738	2305
KIND2-PARALLEL	748	1213

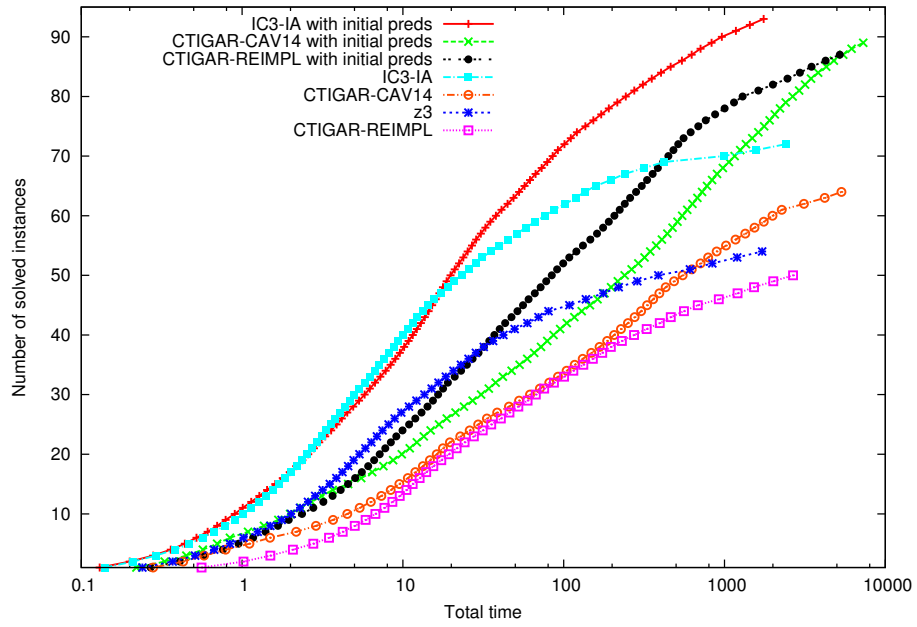
Fig. 6 Experimental results on LIA benchmarks from Lustre programs.

For this set of benchmarks, we compare IC3IA, CTIGAR-REIMPL, Z3, ABC-PDR, NUXMV-IC3-SAT, and ABC-DPROVE.

– Lustre programs from the PKIND suite [31], over the LIA theory. The **LIA** group of benchmarks has 790 instances (after removing duplicate instances from the original suite of 951 programs). On this benchmark set we compare IC3IA, CTIGAR-REIMPL, Z3, PKIND, KIND2-PDR and KIND2-PARALLEL.
– The C programs used in the CTIGAR paper [8]. The **CFG(LIA)** group consists of 110 instances. We compare IC3IA, CTIGAR, CTIGAR-REIMPL and Z3. For IC3IA, CTIGAR-REIMPL and Z3 we have produced a (straigthforward) symbolic encoding of the control-flow graphs (CFG) of the programs.

5.3 Results

The results for the various groups of benchmarks are reported in Figs. from 4 to 7. For each algorithm/tool considered, the plots show the number of solved instances (on the y axis) in the given total amount of time (on the x axis). A ranking of the



Algorithm/Tool	# Solved (out of 110)	Tot time
IC3IA with initial preds	94	1751
CTIGAR-CAV14 with intial preds	90	7290
CTIGAR-REIMPL with initial preds	88	5216
IC3IA	73	2405
CTIGAR-CAV14	65	5335
Z3	55	1712
CTIGAR-REIMPL	51	2667

Fig. 7 Experimental results on CFG(LIA) benchmarks from the CTIGAR suite.

tools (in terms of number of solved instances, and total time) is then displayed in the underlying tables.

Overall, IC3IA is the best performing tool in all the categories we have considered. It outperforms Z3 in all the categories, the highly tuned bit-level engines of ABC and NUXMV in the BV benchmarks, and KIND2-PARALLEL in the Lustre benchmarks. This is true also for the independent implementation of IC3 with Implicit Abstraction within the JKIND model checker, whose performance is very close to that of IC3IA, and better than all the other tools. These results clearly demonstrate the effectiveness of our approach.

Compared to IC3QE, the SMT-extension of IC3 presented in [18], IC3IA is not only more efficient (solving 7 more instances in a shorter total execution time), but also much more general, being able to handle transition systems expressed over various (combinations of) theories (LRA, LIA, bit-vectors), and not just LRA. We also remark that the performance of the IC3QE implementation has been greatly improved compared to the version of [18], thanks to a careful tuning of the approximated quantifier elimination routines used for computing preimages, and to the integration of the LRA-specific generalization technique of [35]. In fact, the results

for IC3QE correspond to the best configuration settings for the quantifier elimination procedure; other (apparently similar) settings produce significantly worse results. In contrast, the implementation of IC3IA did not require any particular tuning for achieving very good performance (see the discussion at the end of this Section).

5.4 An in-depth comparison between IC3IA and CTIGAR

The results across all categories of benchmarks show that the IC3IA algorithm is uniformly superior to the CTIGAR approach proposed in [8]. This is true not only for CTIGAR-REIMPL, our reimplementation of CTIGAR, but also for CTIGAR-CAV14, the original tool [8] (see Fig. 7).

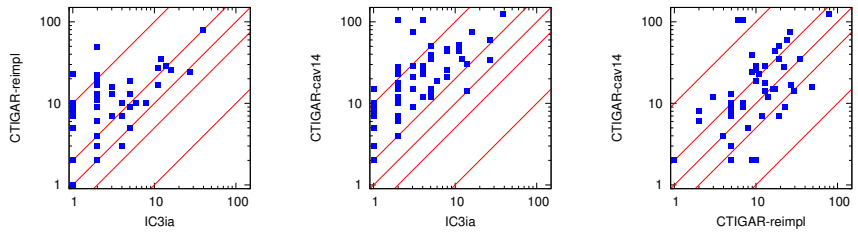
We now integrate the comparison with additional insights.

First, we point out that, although we have tried to follow the description of the algorithm in [8] as closely as possible, there are still some important differences between CTIGAR-REIMPL and the original implementation. Besides the different overall platform, front-end, and underlying SMT solver (we rely solely on MATHSAT, whereas in [8] both MATHSAT and Z3 are used), the most significant differences are in the refinement procedure. In [8], multiple different refinement strategies are discussed. In CTIGAR-REIMPL, we only implemented the one that was considered the best in [8]. We discover new predicates using interpolation, and we add both the top-level conjuncts and all the atoms occurring in interpolants as predicates. We use the "CCL" strategy of [8] for activating a refinement, with a threshold of 3 spurious transitions in a single trace. However, we have not implemented the "refinement state mining" predicate discovery technique described in [8], which is used to extract additional predicates that could not be discovered by interpolation.

Second, our implementation of CTIGAR supports the same set of problems as IC3IA, i.e. all the theories considered here (LRA, LIA, BV). It is thus more general than CTIGAR-CAV14.

Finally, consider that CTIGAR-CAV14 uses static analysis techniques to generate a set of initial predicates for the abstraction-refinement loop. These techniques are completely orthogonal to the underlying combination of IC3 with abstraction, and can be applied to both CTIGAR and to IC3IA.

In order to understand the impact of predicate initialization, for the fourth group of benchmarks we evaluated CTIGAR-CAV14, CTIGAR-REIMPL, and also IC3IA, both with and without the initial predicate discovery routines. In practice, this is done by extracting the predicates from a run of CTIGAR-CAV14, and then importing them in CTIGAR-REIMPL and in IC3IA. The results, reported in Fig. 7, clearly demonstrate that static analysis is extremely effective in identifying a good set of initial predicates: a significant performance boost is obtained for all the tools considered. In terms of number of instances gained, the biggest impact is for CTIGAR-REIMPL, which gains 37 instances. In fact, when using the computed initial set of predicates, CTIGAR-REIMPL can solve 57 instances without any refinement at all. However, IC3IA does even better in this respect, solving 72 instances without any refinement. In contrast, when no initial predicates are used, all instances of the set need at least 1 refinement step (with an median of 3 for IC3IA and 10 for CTIGAR-REIMPL).



# of refinements on instances solved by all tools (48):

	IC3ia	CTIGAR-reimpl	CTIGAR-cav14
Median	2	9	13
Average	4.58	13.08	23.41
Min	1	1	2
Max	39	79	123
1^{st} **quartile**	1	5	8
3^{rd} **quartile**	5	17	29
1^{st} **decile**	1	4	4
9^{th} **decile**	12	26	60

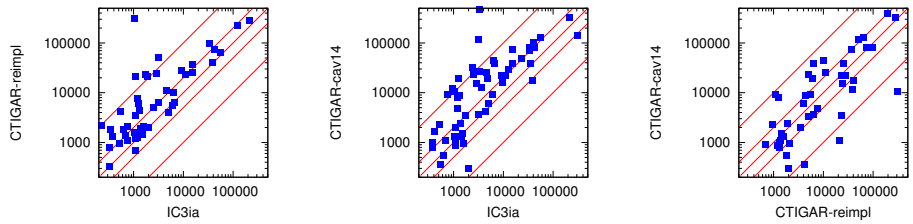
Fig. 8 Number of abstraction refinements on CFG(LIA) benchmarks from the CTIGAR suite.

We provide additional details in Figures 8 and 9, where we compare the three tools in terms of number of refinement steps and number of calls to the underlying SMT solver, over the 48 instances that can be solved by all the tools. We can see that in both cases IC3IA has a significant advantage compared to the two implementations of CTIGAR, which show very similar performance both in terms of number of refinements and number of SMT solver calls. We believe that these results can help explaining the performance advantage of IC3IA over CTIGAR for this class of instances.

5.5 Impact of redundant predicates on IC3IA

We now provide some additional insights on IC3IA. The interpolation-based refinement strategy may introduce more predicates than those actually needed to rule out a spurious counterexample. Our refinement procedure simply adds as predicates *all the atoms* found in the interpolants. However, interpolants generated by current SMT solvers are typically very redundant, so it often happens that not all the atoms are actually needed to refute the abstract trace. In principle, such redundant predicates might significantly hurt performance.

We thus implemented a procedure that identifies and removes (a subset of) redundant predicates after each successful refinement step, using the implicit abstraction framework. Suppose that IC3IA finds a spurious counterexample trace $\widehat{\pi} \doteq \widehat{s_0}, \widehat{s_1}, \ldots, \widehat{s_k}$ with the set of predicates $\mathbb{P}$, and that $\textsc{Refine}(I, T, P, \mathbb{P}, \pi)$ finds a set $\mathbb{P}^{\text{new}}$ of new predicates. The reduction procedure exploits the formula $\mathit{Simulate}(T, \mathbb{P}, \widehat{\pi})$, which is satisfiable if and only if the abstract path $\widehat{\pi}$ can be concretized (See Formula 5 in Section 3). If $\mathbb{P} \cup \mathbb{P}^{\text{new}}$ are sufficient to rule out the spurious counterexample, $\mathit{Simulate}(T, \mathbb{P} \cup \mathbb{P}^{\text{new}}, \widehat{\pi})$ is unsatisfiable. We ask the SMT solver to compute



# of SMT queries on instances solved by all tools (48):

	IC3ia	**CTIGAR-reimpl**	**CTIGAR-cav14**
Median	1524	5115	4816
Average	33373.31	29734.58	32367.50
Min	230	334	53
Max	1101826	313072	388806
1^{st} **quartile**	1041	1577	966
3^{rd} **quartile**	6407	23879	23039
1^{st} **decile**	380	1222	191
9^{th} **decile**	39429	72471	82808

Fig. 9 Number of SMT queries on CFG(LIA) benchmarks from the CTIGAR suite.

the unsatisfiable core of $\mathit{Simulate}(T, \mathbb{P} \cup \mathbb{P}^{\mathrm{new}}, \widehat{\pi})$, and we keep only the predicates of $\mathbb{P}^{\mathrm{new}}$ that appear in the unsatisfiable core.

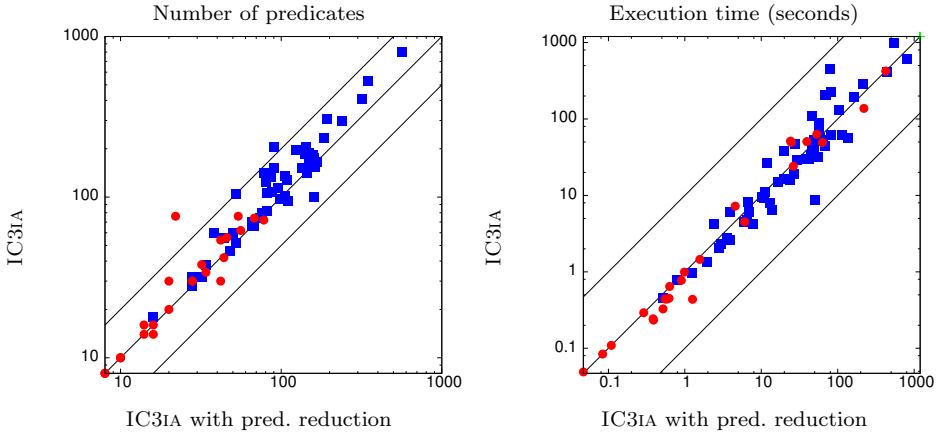


Fig. 10 Effects of predicates reduction on IC3IA. The plot on the left (resp. the right) shows the number of predicates discovered (resp. the total execution time in seconds) by IC3IA without predicate reduction, on the y axes, and with predicate reduction, on the x axes. In the graph each point corresponds to a single benchmark instance: a red circle represents an unsafe instance, where the property does not hold, while a blue square represents a safe instance.

In order to evaluate the effectiveness of this simple approach, we compared two versions of IC3IA with and without the reduction procedure. Figure 10 shows the results of the comparison with and without predicates reduction. The reduction

procedure is almost always effective in reducing the total number of predicates, although the number of predicate sometimes increases. In fact, it may happen that if two predicates are redundant, we remove the one that is necessary or maybe even sufficient to prove the property, while removing it requires to find new spurious counterexamples adding more predicates. Consider for example a counter c that increases by 1 from 0 to 10 satisfying the invariant $x \leq 11$; the predicate $c \leq 10$ would be sufficient to prove the property; suppose, however, we find the predicates $c \leq 0$ and $c \leq 10$ and we discard $c \leq 10$ and at the next iteration we add $c \leq 1$ and $c \leq 10$ and we discard again $c \leq 10$ and so on; at the end, we will have many more predicates than simply keeping in the first iteration $c \leq 10$.

Perhaps surprisingly, however, the effects on the execution time are not very big. Redundancy removal does seem to improve performance for the hard instances, but overall the two versions of IC3IA solve the same number of problems.

We conclude that our algorithm is much less sensitive to the number of predicates, compared to approaches based on an explicit computation of the abstract transition relation (e.g. via All-SMT). In fact, such approaches often show (not only in theory, but also in practice) an exponential increase in run time with the addition of new predicates. IC3IA manages to solve problems for which it discovers several hundreds of predicates, reaching the peak of 800 predicates and solving most of safe instances with more than a hundred predicates (see Figure 10, left). These numbers are typically out of reach for explicit abstraction techniques, which blow up with a few dozen predicates.

6 Conclusion

In this paper we have presented a new approach to the verification of infinite-state transition systems, based on an extension of IC3 with implicit predicate abstraction. The distinguishing feature of our technique is that it works in an abstract state space, since the counterexamples to induction and the relative inductive clauses are expressed solely with the abstraction predicates. This is enabled by the use of implicit abstraction to check (abstract) relative induction. Moreover, the refinement in our procedure is fully incremental, allowing to keep all the clauses found in the previous iterations.

The approach has two key advantages. First, it is very general: the implementations for the theories of LRA, BV, and LIA have been obtained with relatively little effort. Second, it is extremely effective, being able to efficiently deal with large numbers of predicates. Both advantages are confirmed by the experimental results, obtained on a wide set of benchmarks, also in comparison against dedicated verification engines.

In the future, we plan to apply the approach to other theories (e.g. arrays, non-linear arithmetic), investigating other forms of predicate discovery. Then, we plan to extend the approach to deal with temporal properties, by generalizing to the infinite-state case model checking techniques based on reduction to safety, such as K-Liveness [22] and liveness-to-safety [7].

Acknowledgments

This work was carried out within the D-MILS project, which is partially funded under the European Commission's Seventh Framework Programme (FP7).

References

1. T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI*, pages 203–213, 2001.
2. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In J.-P. Katoen and P. Stevens, editors, *TACS*, volume 2280 of *LNCS*, pages 158–172. Springer, 2002.
3. T. Ball, A. Podelski, and S.K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. *STTT*, 5(1):49–58, 2003.
4. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, volume 185, pages 825–885. IOS Press, 2009.
5. J. Baumgartner, A. Ivrii, A. Matsliah, and H. Mony. IC3-guided abstraction. In *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, pages 182–185, 2012.
6. D. Beyer. Second Competition on Software Verification - (Summary of SV-COMP 2013). In *TACAS*, volume 7795 of *LNCS*, pages 594–609. Springer, 2013.
7. A. Biere, C. Artho, and V. Schuppan. Liveness Checking as Safety Checking. *Electr. Notes Theor. Comput. Sci.*, 66(2):160–177, 2002.
8. J. Birgmeier, A. R. Bradley, and G. Weissenbacher. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In *CAV*, pages 831–848, 2014.
9. N. Bjørner and A. Gurfinkel. Property Directed Polyhedral Abstraction. In *VMCAI*, pages 263–281, 2015.
10. A. Bradley. IC3ref. `https://github.com/arbrad/IC3ref`.
11. A. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang. An Incremental Approach to Model Checking Progress Properties. In *FMCAD*, 2011.
12. A. R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
13. A. R. Bradley and Z. Manna. Checking safety by inductive generalization of counterexamples to induction. In *FMCAD*, pages 173–180. IEEE Computer Society, 2007.
14. R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv Symbolic Model Checker. In *CAV*, volume 8559 of *LNCS*, pages 334–342, 2014.
15. R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar. Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In *FMCAD*, pages 69–76. IEEE Computer Society, 2007.
16. H. Chokler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental Formal Verification of Hardware. In *FMCAD*, 2011.
17. A. Cimatti, A. Franzén, A. Griggio, K. Kalyanasundaram, and M. Roveri. Tighter integration of BDDs and SMT for Predicate Abstraction. In *DATE*, pages 1707–1712. IEEE, 2010.
18. A. Cimatti and A. Griggio. Software Model Checking via IC3. In *CAV*, pages 277–293, 2012.
19. A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 Modulo Theories via Implicit Predicate Abstraction. In *TACAS*, pages 46–61, 2014.
20. A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, volume 7795 of *LNCS*. Springer, 2013.
21. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient generation of Craig interpolants in Satisfiability Modulo Theories. *ACM Trans. Comput. Log.*, 12(1):7, 2010.
22. K. Claessen and N. Sörensson. A Liveness Checking Algorithm that Counts. In *FMCAD*, pages 52–59. IEEE, 2012.
23. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *J. ACM*, 50(5):752–794, 2003.
24. E. M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

25. N. Een, A. Mishchenko, and R. Brayton. Efficient Implementation of Property-Directed Reachability. In *FMCAD*, 2011.
26. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, pages 72–83, 1997.
27. Alberto Griggio and Marco Roveri. Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 35(6):1026–1039, 2016.
28. A. Gupta and A. Rybalchenko. InvGen: An Efficient Invariant Generator. In *CAV*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.
29. A. Gupta and O. Strichman. Abstraction Refinement for Bounded Model Checking. In *CAV*, pages 112–124, 2005.
30. A. Gurfinkel and A. Ivrii. Pushing to the Top. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 65–72, 2015.
31. G. Hagen and C. Tinelli. Scaling Up the Formal Verification of Lustre Programs with SMT-Based Techniques. In *FMCAD*, pages 1–9. IEEE, 2008.
32. Z. Hassan, A. R. Bradley, and F. Somenzi. Better generalization in IC3. In *FMCAD*, pages 157–164. IEEE, 2013.
33. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from Proofs. In *POPL*, pages 232–244, 2004.
34. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
35. K. Hoder and N. Bjørner. Generalized Property Directed Reachability. In *SAT*, pages 157–171, 2012.
36. T. Isenberg and H. Wehrheim. Timed Automata Verification via IC3 with Zones. In *ICFEM*, pages 203–218, 2014.
37. Shachar Itzhaky, Nikolaj Bjørner, Thomas W. Reps, Mooly Sagiv, and Aditya V. Thakur. Property-directed shape analysis. In *CAV*, pages 35–51, 2014.
38. H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke. Word level predicate abstraction and refinement for verifying RTL verilog. In *DAC*, pages 445–450, 2005.
39. H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke. Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog. In *DAC*, pages 445–450, 2005.
40. T. Kahsai and C. Tinelli. PKind: A Parallel K-induction Based Model Checker. In *PDMC*, volume 72 of *EPTCS*, pages 55–62, 2011.
41. Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. In *CAV*, pages 583–602, 2015.
42. R. Kindermann, T. A. Junttila, and I. Niemelä. SMT-Based Induction Methods for Timed Systems. In *FORMATS*, volume 7595 of *LNCS*, pages 171–187. Springer, 2012.
43. R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
44. S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT Techniques for Fast Predicate Abstraction. In *CAV*, pages 424–437, 2006.
45. T. Lange, M. R. Neuhäußer, and T. Noll. IC3 Software Model Checking on Control Flow Automata. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 97–104, 2015.
46. K. L. McMillan. Lazy Abstraction with Interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.
47. N. Sorensson and K. Claessen. Tip. `https://github.com/niklasso/tip`.
48. S. Tonetta. Abstract Model Checking without Computing the Abstraction. In *FM*, pages 89–105, 2009.
49. Y. Vizel, O. Grumberg, and S. Shoham. Lazy Abstraction and SAT-based Reachability in Hardware Model Checking. In *FMCAD*, pages 173–181. IEEE, 2012.
50. Y. Vizel and A. Gurfinkel. Interpolating Property Directed Reachability. In *CAV*, pages 260–276, 2014.
51. Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.
52. T. Welp and A. Kuehlmann. QF_BV model checking with property directed reachability. In *DATE*, pages 791–796, 2013.