EasyChair Preprint

№ XXX

Model Checking of Verilog RTL using IC3 with Syntax-guided Abstraction

Aman Goel and Karem Sakallah

March 23, 2019

Model Checking of Verilog RTL using IC3 with Syntax-guided Abstraction

Aman Goel and Karem Sakallah

University of Michigan, Ann Arbor
{amangoel,karem}@umich.edu

Abstract. While bit-level IC3-based algorithms for hardware model checking represent a major advance over prior approaches, their reliance on propositional clause learning poses scalability issues for RTL designs with wide datapaths and complex word-level operations. In this paper we present a novel technique that combines IC3 with *syntax-guided abstraction* (SA) to allow scalable word-level model checking using SMT solvers. SA defines the abstraction implicitly from the syntax of the input problem, has high granularity and an abstract state-space size completely independent of the bit widths of the design's registers. We show how to efficiently integrate IC3 with SA, and demonstrate its effectiveness on a suite of open-source and industrial Verilog RTL designs. Additionally, SA aligns easily with data abstraction using uninterpreted functions. We demonstrate how IC3+SA with data abstraction allows reasoning that is completely independent of the bit width of variables, and becomes scalable irrespective of the state-space size or complexity of operations.

1 Introduction

IC3 [13] (also known as PDR [25]) is arguably the most successful technique for hardware model checking. Bit-level engines using IC3 (e.g. ABC [8], IIMC [14], PDTRAV [16], AVY [49]) have shown exceptional performance in hardware model checking competitions (HWMCC) [10]. As the size and complexity of the problem increases, the bit-level IC3 algorithm suffers from two main scalability issues: poor SAT solver performance, and learning too many weak propositional frame restrictions. Several techniques have been proposed to address these challenges (e.g. [48,20,50,40,31,32,33]), including different ways of adding a layer of *abstraction refinement* [37,22] to reduce the burden on reasoning engines. Approaches like [20,40] suggest raising IC3 to the word level by exploiting high-level information missing at the bit level. These techniques replace bit-level reasoning using SAT solvers with word-level clause learning in *first order logic* (FOL) using SMT [7] solvers.

The Averroes system [40,39] demonstrated how EUF abstraction [15,5,4] can be exploited to perform word-level IC3 on control-centric Verilog RTL designs. The technique performed backward reachability using a weakest precondition algorithm, effectively causing an *implicit* unrolling of the transition relation which leads to poor performance and possible non-termination in some situations. This

typically happens when the property being checked is strongly dependent on data operations, for which EUF abstraction is ill-suited, and leads to an excessive number of *data* refinement iterations to repair the abstraction.

In this paper we address these issues by extending the Averroes approach beyond control-centric problems using *syntax-guided abstraction* (SA). Inspired by EUF abstraction, SA implicitly creates an abstraction using the terms present in the syntax of the problem yielding an abstract domain whose size is completely independent of the bit widths of the registers or the *sequential depth* [43] of the design. SA offers high granularity and captures *all equality* relations among the terms present in the syntax of the problem, while also interpreting data operations. Any *spurious* behavior is eliminated by adding new terms that were missing in the original problem. We show how to efficiently combine IC3 with SA (IC3+SA), and extend IC3+SA with data abstraction using uninterpreted functions (UF). IC3+SA with data abstraction allows for abstract reasoning that is completely independent of the design's bit widths and offers scalability irrespective of the problem size or complexity of operations.

Our main contributions are as follows:

- We present syntax-guided abstraction to implicitly capture the most relevant details from the syntax of the system with negligible computation cost.
- We present an efficient syntax-guided cube generalization procedure for word-level IC3 that is quantifier-free, doesn't require any solver calls, and does not perform any implicit or explicit unrolling of the transition relation.
- We suggest a fully incremental procedure to refine SA and eliminate any spurious behavior in the abstract domain.
- We show how IC3+SA can be easily extended with data abstraction using UF for complete and scalable model checking on control-intensive problems.

The paper is organized as follows: Sec. 2 presents the relevant background to describe the detailed SA approach in Sec. 3. Sec. 4 shows how SA is integrated within the IC3 framework, and the correctness of this method is proved in Sec. 5. Sec. 6 covers implementation details and presents an experimental evaluation on a diverse set of RTL benchmarks. The paper concludes with a brief survey of related work in Sec. 7, and a discussion of future directions in Sec. 8.

2 Background

2.1 Notation

Our setting is standard first-order logic with the notions of *sort*, *universe*, *signature*, and *structure* defined in the usual way [7]. A *term* is a constant symbol, or an n-ary function symbol applied to n terms. An *atom* is $\top$, $\bot$ or an n-ary predicate symbol applied to n terms. A *literal* is an atom or its negation, a *cube* is a conjunction of literals, and a *clause* is a negation of a cube, i.e., a disjunction of literals. A quantifier-free *formula* is a literal or the application of logical connectives to formulas.

We will refer to all terms with a non-boolean range as *words*, and refer to words with 0-arity as *ground* words. A *partition assignment* for a formula φ is defined as a boolean assignment to each predicate in φ, and a set of partitions

(one for each sort) dividing the words in φ into equivalence classes. An interpretation $\mathcal{I}$ assigns a meaning to terms by means of a uniquely determined (total) mapping ($[\![\_]\!]^{\mathcal{I}}$) of such terms into the universe of its structure. A model of a formula φ for an interpretation $\mathcal{I}$ is a structure that satisfies φ (i.e. $[\![\varphi]\!]^{\mathcal{I}} = \top$). For example, the interpretation for the theory of free sort and function symbols (call it $\mathcal{I}_P$) maps terms into the universe of partition assignments. The interpretation for the theory of bitvectors (call it $\mathcal{I}_B$) maps terms into a universe composed of bitvector assignments.

Given a transition system, we will use primes to represent a variable after a single transition step. Given a set of variables X, X' is the set obtained by replacing each variable in X with its primed version. We will use φ (resp. φ') as a shorthand for a formula $\varphi(X)$ (resp. $\varphi(X')$).

2.2 Model Checking

A model checking problem $\mathcal{P}$ can be described by a 4-tuple $\langle X, I, T, P\rangle$, where X denotes the set of present state variables, $I(X)$ is a formula representing the initial states, $T(X, X')$ is a formula for the transition relation, and $P(X)$ is a formula for a given *safety* property. Given $\mathcal{P}$, the model checking problem can be stated as follows: either prove that $P(X)$ holds for any sequence of executions starting from a state in $I(X)$, or disprove $P(X)$ by producing a counterexample.

We assume that T is expressed as a conjunction of equalities that express next-state variables as functions of present-state variables. Input variables are conveniently modeled as state variables whose corresponding next states are completely unconstrained. Our focus is on verifying Verilog RTL designs which we encode as *finite* transition systems that are naturally expressed in the `QF_BV` theory of SMT-LIB.

3 Syntax-guided Abstraction

Predicate abstraction (PA) [28] encodes the abstract state space using a set of predicates whose boolean assignments encode the abstract states. In contrast, syntax-guided abstraction (SA) encodes the abstract state space using the set of terms present in the word-level syntax of the problem. Abstract states in SA correspond to partition assignments that capture the equality relations among the problem's terms. The relevant parts of the abstract transition relation in both *implicit* PA [20,21] and SA are constructed incrementally, as needed, during the reachability search using bitvector queries. We will use $\mathcal{P}$ to denote the original *concrete* problem and $\hat{\mathcal{P}}$ to denote its *syntactically-abstracted* version. Models in $\mathcal{P}$ use the interpretation $\mathcal{I}_B$, i.e. exact bitvector assignments, whereas models in $\hat{\mathcal{P}}$ use the $\mathcal{I}_P$ interpretation, i.e. partition assignments. Effectively, SA hides away irrelevant bit-level details and is able to infer higher-level equality relations among the words in the problem description.

Example 1: Let $\mathcal{P} = \langle\{u, v\}, (u = 1) \wedge (v = 1), (u' = ite(u < v, u + v, v + 1)) \wedge (v' = v + 1), ((u + v) \neq 1)\rangle$, where u, v are k-bit wide. $\mathcal{P}$ has 1 predicate $(u < v)$ and 5 words $(1, u, v, u + v, v + 1)$. Consider a concrete state $s := (u, v) = (1, 2)$. Its corresponding abstract state is obtained by evaluating the problem's

predicates and terms using the concrete state assignment and creating a partition assignment based on these evaluations. In this example, the abstract state is easily seen to be $\hat{s} := (u < v) \wedge \{\ 1, u \mid v \mid u + v, v + 1\ \}$[1].

The biggest advantage of SA is that the abstract state-space size is completely independent of the bit-width of variables while still accounting for all relations among terms in the original problem. Given $\mathcal{P}$ with, say, m total state bits, the concrete system has 2^m states. On the other hand, the total number of abstract states is bounded by $2^p \times B_n$, where p is the number of predicates in $\hat{\mathcal{P}}$, n is the number of words in $\hat{\mathcal{P}}$, and B_n is the n^{th} Bell number [46] (the number of unique partitions on n terms). For example, let $k = 16$ in Example 1. The size of the concrete state space is $2^{2\times16}$ i.e. ~ 4.2 billion, while the number of abstract states is $2^1 \times B_5 = 104$, completely independent of k.

Given a formula φ, we use concrete theory reasoning (i.e. `QF_BV`) for abstract SMT solving (similar to [20,21]) with the modification that the solution (i.e. model) of φ in the abstract domain is expressed as a partition assignment on terms in φ, i.e. for a partition assignment $\hat{s}$, $\hat{s} \models \varphi$ *iff* there exists a bitvector assignment s such that $s \models \varphi$ and $\hat{s} = \alpha(\varphi, s)$, where α is the abstraction function that converts a bitvector assignment s to a partition assignment on terms in φ. We perform a simple evaluation of each term in the formula to construct a partition assignment based on the bitvector assignment.

Example 2: Consider $\mathcal{P}$ from Example 1. Let $k = 2$. Consider the formula $\varphi = P \wedge T \wedge \neg P'$ and a satisfying concrete solution $s := (u, v, u', v') = (0, 2, 2, 3)$. Terms in φ evaluate as $(u < v, u+v, v+1, u'+v') = (\top, 2, 3, 1)$ under s, resulting in the abstract solution to be $\hat{s} := (u < v) \wedge \{\ u \mid 1, u'+v' \mid v, u+v, u' \mid v+1, v'\ \}$.

We can always construct a *unique* abstract solution $\hat{s}$ given a formula φ and its concrete solution s. Modern SMT solvers (e.g. [24,23]) have support to give the bitvector assignment for each term in the formula without any extra cost. Words with the same assigned value go in the same equivalence class of a partition, while different assignments mean different classes.

An abstract solution is *complete* if it contains all the terms in $\mathcal{P}$. The abstract state space is defined by the universe of complete abstract solutions. An abstract solution can be *projected* on any subset of symbols (a *projection set*) by *co-factoring* the solution to eliminate all terms with any symbol outside the projection set, i.e. by simply dropping terms from the partition assignment that contain symbol(s) outside the projection set. An abstract solution can be converted to an equivalent cube by adding all constraints needed to *cover* the solution.

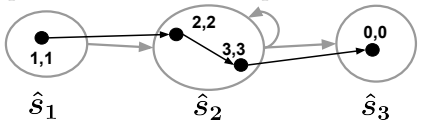
Example 3: Consider $\hat{s}$ from Example 2. $\hat{s}$ can be projected on the projection set $\sigma = \{+, 1, u', v'\}$ to get a *partial* abstract solution representing the destination states as $\hat{s}|_\sigma := \{\ 1, u' + v' \mid u' \mid v'\ \}$. The corresponding cube representation is $cube(\hat{s}|_\sigma) = ((u' + v') = 1) \wedge (u' \neq 1) \wedge (v' \neq 1) \wedge (u' \neq v')$.

[1] In this notation, vertical bars separate the equivalence classes of the partition. Thus $\{a, b|c\}$ should be interpreted to mean $\{\{a, b\}, \{c\}\}$ in the standard notation for partitions.

The SA abstract state space induces a partition on the concrete state space such that each concrete state is mapped to a single abstract state. An abstract state, thus, corresponds to a (possibly empty) set of concrete states causing the abstract transition relation to be *non-deterministic*. This abstraction is *sound* but may lead to *spurious* behavior.

Example 4: Consider the following abstract path from $\mathcal{P}$ in Example 2:

$\hat{s}_1 := \neg(u < v) \wedge \{\ 1, u, v \mid u + v, v + 1\ \}$
$\hat{s}_2 := \neg(u < v) \wedge \{\ 1 \mid u, v \mid u + v \mid v + 1\ \}$
$\hat{s}_3 := \neg(u < v) \wedge \{\ 1, v + 1 \mid u, v, u + v\ \}$



$\hat{s}_1$ has a concrete transition to $\hat{s}_2$, $\hat{s}_2$ can concretely transition to $\hat{s}_3$, though there isn't a continuous 2-step concrete path from $\hat{s}_1$ to $\hat{s}_3$ via $\hat{s}_2$. SA can be refined by adding new terms. For example, we can add the constant term 2 (or 3) to eliminate the spurious behavior of Example 4.

To better understand how SA compares to PA, consider $\mathcal{P}$ from Example 2. There are 16 concrete states in $\mathcal{P}$. The four predicates in $\mathcal{P}$, i.e. p_1: $(u = 1)$, p_2: $(v = 1)$, p_3: $((u + v) = 1)$ and p_4: $(u < v)$ are a natural choice as initial predicates for PA. Table 1 compares the abstract domain for SA and PA against the concrete state space. PA with p_{1-4} as predicates partitions the concrete states into 9 *feasible* abstract states. SA on the other hand offers higher expressiveness and partitions the concrete states into 13 feasible abstract states.

Table 1: Mapping of abstract states on concrete states for SA and PA

Index	**SA**: partition assignment on $\{u < v, 1, u, v, u + v, v + 1\}$	**Concrete states**: (u, v)	**PA**: $p_1p_2p_3p_4$
1	$\neg(u < v) \wedge \{\ 1, v + 1 \mid u, v, u + v\ \}$	$(0, 0)$	0000
2	$\neg(u < v) \wedge \{\ 1, v + 1 \mid u, u + v \mid v\ \}$	$(2, 0)$, $(3, 0)$	
3	$\neg(u < v) \wedge \{\ 1 \mid u, v \mid u + v \mid v + 1\ \}$	$(2, 2)$, $(3, 3)$	
4	$(u < v) \wedge \{\ 1 \mid u \mid v, u + v \mid v + 1\ \}$	$(0, 2)$	0001
5	$(u < v) \wedge \{\ 1 \mid u, v + 1 \mid v, u + v\ \}$	$(0, 3)$	
6	$\neg(u < v) \wedge \{\ 1, u + v \mid u, v + 1 \mid v\ \}$	$(3, 2)$	0010
7	$(u < v) \wedge \{\ 1, u + v \mid u \mid v \mid v + 1\ \}$	$(2, 3)$	0011
8	$\neg(u < v) \wedge \{\ 1, v \mid u, v + 1 \mid u + v\ \}$	$(2, 1)$	0100
9	$\neg(u < v) \wedge \{\ 1, v \mid u \mid u + v \mid v + 1\ \}$	$(3, 1)$	
10	$(u < v) \wedge \{\ 1, v, u + v \mid u \mid v + 1\ \}$	$(0, 1)$	0111
11	$(u < v) \wedge \{\ 1, u \mid v \mid u + v, v + 1\ \}$	$(1, 2)$, $(1, 3)$	1001
12	$\neg(u < v) \wedge \{\ 1, u, u + v, v + 1 \mid v\ \}$	$(1, 0)$	1010
13	$\neg(u < v) \wedge \{\ 1, u, v \mid u + v, v + 1\ \}$	$(1, 1)$	1100
	others: $(2^1 \times B_5 - 13 = 91)$	infeasible	others: $(2^4 - 9 = 7)$

Syntax-guided abstraction has the following advantages over PA:

- Unlike predicate abstraction or its variants [28,6], SA is *implicitly* defined by the original syntax and does not require a user-specified set of initial predicates or solver queries to generate the abstract state space.
- By construction, SA accounts for all equality relations among terms in the syntax and offers higher granularity and expressiveness than implicit predicate abstraction [20,21], resulting in less spurious behavior.

– SA is refined by adding new terms that are absent in the original problem syntax, while PA relies on adding new predicates for refinement. Furthermore, equality propagation, which is at the heart of SA, allows all equality relations involving a newly-introduced term to be automatically detected; in PA such relations are discovered one by one in multiple refinement iterations.

SA+UF: SA uses bit-precise `QF_BV` queries and may not scale for large problems with complex operations. Since SA requires only a partition assignment on terms and not exact bitvector assignments, it aligns perfectly with data abstraction where data operations (like arithmetic, shift, etc.) are treated as uninterpreted functions [15,5,4,40]. SA+UF is most appropriate for control-centric properties where correctness is largely independent of data state. IC3 with SA+UF extends [40] and allows for efficient reasoning using `QF_UF` queries regardless of the bit-width of variables or complexity of data operations.

Example 5: Consider $\mathcal{P}$ from Example 1. Using SA+UF, the abstract problem becomes $\bar{\mathcal{P}} = \langle \bar{X}, \bar{I}, \bar{T}, \bar{P} \rangle$:

$$\bar{X} = \{\, \bar{u}, \bar{v} \,\} \qquad \bar{I} = (\bar{u} = \bar{1}) \wedge (\bar{v} = \bar{1}) \qquad \bar{P} = (ADD(\bar{u}, \bar{v}) \neq \bar{1})$$
$$\bar{T} = (\bar{u}' = ite(LT(\bar{u}, \bar{v}), ADD(\bar{u}, \bar{v}), ADD(\bar{v}, \bar{1})) \wedge (\bar{v}' = ADD(\bar{v}, \bar{1}))$$

SA+UF uses uninterpreted sorts instead of bitvectors (indicated by $^-$), and converts data operations (e.g. $<$, $+$) to UFs (e.g. LT, ADD) and ground terms to UFs with 0-arity.

4 IC3 with Syntax-guided Abstraction (IC3+SA)

IC3+SA uses SMT solving to raise reasoning from propositional to FOL, similar in spirit to [19,34,20,40,21]. The IC3+SA algorithm performs the core IC3 procedure in the syntactically-abstracted state space and tightens the abstraction using a typical CEGAR loop [37,22]. There are 2 key differences between IC3+SA and bit-level IC3.

– How to **generalize a satisfiable query** from a particular solver solution?
– How to **refine** spurious counterexamples?

Most other concepts in IC3 remain identical to the bit level and can be equivalently applied in IC3+SA using word-level clauses and SMT solvers (as elaborated in [19,11,40]).

4.1 Generalization of a Satisfiable Query

Consider a 1-step reachability query from frame m to a destination cube c (i.e. $SAT\ ?\ [F_m \wedge T \wedge c']$). If the query is satisfiable, it is essential for performance to *generalize* the particular solution returned by the solver into a generalized cube c_m (as indicated in [25,13,11]). For the propositional case, the authors of [25] suggest *ternary simulation* to generalize the particular solution into a cube. This generalization (as well as cube generalization suggested in the original IC3 algorithm [13]) ensures *strict continuity*.

Definition 1. *(Strict Continuity) Given a destination cube c, every state in the generalized cube c_m should have a transition under T to the destination cube c, i.e. $\forall_s\ SAT\ ?\ [s \wedge T \wedge c']$ is satisfiable, where $\{ s \in c_m \mid s \text{ is a state} \}$.*

Strict continuity is not necessary for IC3, though it is sufficient to guarantee "relaxed" *continuity*.

Definition 2. *(Continuity) Given a sequence of cubes* $\mathcal{C} = \langle c_m, \ldots, c_n \rangle$ *with* $c_n = \neg P$*, there exists a path* $\pi = \langle s_m, \ldots, s_n \rangle$ *such that* $\{ s_i \in c_i \mid s_i \text{ is a state} \}$ *for all* $i \in \{m, \ldots, n\}$.

For correctness, the necessary condition for any cube generalization procedure is to ensure *continuity* (Def. 2), i.e. there should exist a path from the generalized cube c_m to $\neg P$. After all, any cube with a continuous path to a bad state (i.e. a state satisfying $\neg P$) needs to be checked for reachability from the initial states.

It is unclear how to extend ternary simulation to word-level semantics, since ternary simulation inherently relies on modeling the system as a boolean circuit. Instead, we use a *syntax-guided* generalization technique that exploits the word-level structure of the problem to *cheaply* generalize a particular abstract solution. The procedure exploits structural *cone-of-influence* (COI) and model-based justification to identify *relevant* portions that are sufficient to *justify* the particular solution (similar to justification in test pattern generation [47]), and creates a projection set with relevant symbols. The particular solution is projected on these relevant symbols to get the generalized cube.

Algorithm 1 Syntax-guided Generalization

```
1.  procedure GENERALIZE(ŝ, c′) ▷ ŝ is a particular abstract solution, c′ is a destination cube
2.      σ ← σ_refine                          ▷ initialize projection set (initially σ_refine = ∅)
3.      JustifyCOI(ŝ, c′, σ)                  ▷ build projection set σ
4.      σ ← σ − X′                            ▷ get rid of next state symbols
5.      ŝ|_σ ← Project(ŝ, σ)                  ▷ project ŝ on σ
6.      return cube(ŝ|_σ)                     ▷ convert to a cube and return

7.  procedure JUSTIFYCOI(ŝ, φ, σ)             ▷ φ is a FOL expression, σ is passed by reference
8.      if φ is a conditional operation then  ▷ if φ is an if-then-else expression
9.          ⟨cond, v⊤, v⊥⟩ ← BreakCondition(φ)   ▷ get condition and arguments
10.         JustifyCOI(ŝ, cond, σ)
11.         val ← Evaluate(cond, ŝ)           ▷ evaluate cond under ŝ
12.         JustifyCOI(ŝ, (val = ⊤) ? v⊤ : v⊥, σ)   ▷ recurse only on the relevant branch
13.     else if φ is a logical operation then
14.         val ← Evaluate(φ, ŝ)              ▷ evaluate φ under ŝ
15.         if IsControlling(val, φ) then ▷ if assigned a controlling value (⊥ for ∧, ⊤ for ∨)
16.             JustifyCOI(ŝ, GetControlling(φ, ŝ), σ)   ▷ recurse only on controlling arg.
17.         else
18.             for each a ∈ Argument(φ) do
19.                 JustifyCOI(ŝ, a, σ)
20.     else
21.         for each a ∈ Argument(φ) do
22.             JustifyCOI(ŝ, a, σ)
23.         if φ is a next state variable then
24.             JustifyCOI(ŝ, GetRelation(φ), σ)   ▷ get the next state relation for φ from T
25.         Add symbol(φ) to σ                ▷ add symbol of φ to the projection set
```

Alg. 1 presents the syntax-guided generalization procedure using COI with model-based justification. Given the particular abstract solution $\hat{s}$ and the destination cube c', the procedure traverses the *concrete* structural COI of c' and collects symbols encountered in the process (line 3, 7-25). The key idea is that

during the traversal we can syntactically prune away portions that are not important under the given particular solution (lines 12,16) and only visit portions that justify leading to the destination. Once the relevant symbols are collected, the algorithm projects $\hat{s}$ on these symbols to get the generalized cube (lines 5-6).

Alg. 1 guarantees abstract continuity (Def. 2), with the generalized cube always having an abstract path to $\neg P$ in $\hat{\mathcal{P}}$. The algorithm however does not guarantee strict continuity (Def. 1), as evident from the following example:

Example 6: Let $\mathcal{P} = \langle \{u, v, w\}, (u = 1) \wedge (v = 1) \wedge (w = 1), (u' = ite((u < v) \vee (v < w), u + v, v + 1)) \wedge (v' = v + 1) \wedge (w' = w + 1), ((u + v) \neq 1)) \rangle$, with u, v, w being 3-bit wide. Consider the following query and its particular solution:

$F_1 = P \qquad \varphi = F_1 \wedge T \wedge \neg P'$

$Q_1 := SAT\ ?\ [\varphi]$ gives SAT with solution s

$s = (u, v, w, u', v', w') = (0, 4, 2, 4, 5, 3) \qquad \hat{s} = \alpha(\varphi, s)$

$\hat{s} = (u < v) \wedge \neg(v < w) \wedge \{\ u \mid 1, u' + v' \mid w \mid w + 1 \mid v, u + v, u' \mid v + 1, v'\ \}$

`Generalize`($\hat{s}$, $\neg P'$) creates the generalized cube c_1 as follows:

$$\begin{aligned} \sigma &= \{\ +, u', v', 1, <, u, v\ \} - \{\ u', v', w'\ \} \\ &= \{\ +, <, u, v, 1\ \} \\ c_1 &= cube(\hat{s}|_\sigma) = (u < v) \wedge \{\ u \mid 1 \mid v, u + v \mid v + 1\ \} \end{aligned}$$

On careful analysis one can see that not all abstract states in c_1 have an abstract transition to the destination ($\neg P'$). For example, consider the abstract state $\hat{a}_1 = (u < v) \wedge \neg(v < w) \wedge \{\ u \mid 1, w \mid v, u + v, w + 1 \mid v + 1\ \}$. $\hat{a}_1$ is an abstract state in the cube c_1, but it does not have a transition under T to any destination state, i.e. $SAT\ ?\ [cube(\hat{a}_1) \wedge T \wedge \neg P']$ is UNSAT.

We believe *non-determinism* in the word-level abstract domain is the reason why Alg. 1 does not follow strict continuity. Even though Def. 1 is violated, Alg. 1 still guarantees continuity (Def. 2) in the abstract domain. This is because the `Generalize` algorithm ensures that all terms in $\hat{\mathcal{P}}$ that are required to lead to the destination c' under the particular abstract solution $\hat{s}$ are retained in the generalized cube $cube(\hat{s}|_\sigma)$ as is from $\hat{s}$. As a result, even though $cube(\hat{s}|_\sigma)$ has abstract states that do not have an abstract transition to the destination c', this cannot result in an abstract path discontinuity while still limiting to terms in $\hat{\mathcal{P}}$. The `Generalize` procedure acts as a quick sweeper that removes irrelevant terms that will never get involved with any query that satisfies $cube(\hat{s}|_\sigma) \wedge T \wedge c'$, and encodes the sufficient information using the relevant symbols in $cube(\hat{s}|_\sigma)$. The proposed generalization procedure has the following advantages:

- In contrast to solver-based methods suggested in [18,34,11], syntax-based generalization is inexpensive since it does not required any solver query.
- Since the generalization is driven from the syntactic cone of the destination, the procedure only captures the relevant information leading to a bad state.
- The technique guarantees continuity with no need for *lifting refinement* [11].
- Unlike [40,39], the technique does not use weakest preconditions (WP) for generalization, which can be regarded as implicitly unrolling the transition relation. WP-based techniques generate new terms through function compositions, which complicates the abstract state space and can often cascade to cause incompleteness and poor SMT solving.

Syntax-based generalization offers an inexpensive and effective procedure to expand a single solver solution to a set of solutions for word-level IC3. An identical generalization procedure can be used for SA+UF and possibly even for PA.

4.2 Refinement

Running IC3 in the abstract domain either generates an inductive invariant that proves the property to be true, or produces an abstract counterexample evidence $\mathcal{C}$. An abstract counterexample $\mathcal{C}$ of length $n+1$ is represented by a sequence of $n+1$ abstract cubes $\langle c_0, c_1, c_2, \ldots, c_n \rangle$, where $c_n = \neg P$.

We concretize $\mathcal{C}$ by restoring the interpretation to $\mathcal{I}_B$, i.e. exact bitvector assignments. $\mathcal{C}$ can be spurious when the terms in the original problem are insufficient to express the bit-precise nature of the concrete problem.

One way to identify spurious behavior in $\mathcal{C}$ is by checking the satisfiability of a single concrete path query along $\mathcal{C}$ with explicit unrolling, i.e. $SAT\ ?\ [I \wedge \big(\bigwedge_{i=0}^{n-1} c_i^i \wedge T^i\big) \wedge c_n^n]$ (where φ^i denotes the formula φ at i^{th} transition step) using `QF_BV` SMT solving. Checking satisfiability of such a query with multiple copies of T is not scalable in practice as the length of $\mathcal{C}$ increases. We instead perform *incremental* refinement *along the counterexample* (Alg. 2) which uses 1-step queries to perform *forward image computation* [29] along $\mathcal{C}$. We formulate at most n queries $Q_i := SAT\ ?\ [p_{i-1} \wedge c_{i-1} \wedge T \wedge c_i']$ $(1 \leq i \leq n)$ such that $p_0 = I$, and p_i equals the *symbolic post image* [29,42] of $p_{i-1} \wedge c_{i-1}$ under the solution of the query Q_i. To compute p_i after a satisfiable query Q_i (say with solution s), we use fresh symbolic constants to replace unconstrained variables at that step and *syntactically* evaluate T under s to get the symbolic post image of $p_{i-1} \wedge c_{i-1}$ for the next step (line 6). This generates new terms and results in an *implicit* unrolling of T, which in practice is simpler compared to explicit unrolling. We check for the satisfiability of Q_i in increasing order (from $i = 1$ to n) and stop as soon as a query is found unsatisfiable (lines 3-14). From the unsatisfiable query, we extract a minimal unsatisfiable subset [45,41] (MUS) m and get rid of any symbolic constant in m using substitution or rarely instantiation using last solver assigned value if substitution is not possible (lines 8-9). Since

Algorithm 2 Refinement of SA

```
1.  procedure REFINE(Ĉ)
2.      p_0 ← I
3.      for i = 1 to n do
4.          ψ_i ← p_{i-1} ∧ c_{i-1} ∧ T ∧ c'_i
5.          if SAT ? [ψ_i]: solution s then
6.              p_i ← PostImage(p_{i-1} ∧ c_{i-1}, s)      ▷ compute image(p_{i-1} ∧ c_{i-1}) under s
7.          else                                            ▷ i.e. C is spurious
8.              m ← MUS(ψ_i)                                ▷ find MUS for the UNSAT query
9.              m ← Substitute(m)                           ▷ eliminate symbolic constants
10.             Φ ← ¬m
11.             T ← T ∧ Φ                                   ▷ conjoin axiom to T̂
12.             σ_new ← symbols(NewTerms(Φ))                ▷ find symbols in new terms
13.             σ_refine ← σ_refine ∪ σ_new                 ▷ add permanent symbols
14.             return ∅
15.     return C                                            ▷ i.e. C is a true counterexample
```

the unsatisfiability is due to a concrete *path infeasiblity*, m necessarily contains constraints from the forward image computation that include new terms generated from substitution. These new terms are important to eliminate the spurious counterexample. We add these newly discovered terms to the abstract domain by deriving a *refinement (path) axiom* by negating m (line 10). We refine the abstract problem $\hat{\mathcal{P}}$ by conjoining the refinement axiom to the transition relation T (line 11).

New terms created are crucial to eliminate spurious counterexamples. They were absent in the original problem and hence the abstract domain wasn't expressive enough to capture infeasibilities involving them. Adding the refinement axiom with these new terms automatically augments the abstract problem and makes them part of future iterations of IC3+SA. We add the symbols in the new terms as permanent members of all projection sets computed using Alg. 1 so as to ensure that future iterations of `Generalize` doesn't ambitiously generalize them away (lines 12-13). This is essential since these new terms are not part of the original problem syntax but are required to eliminate spurious counterexamples.

If all queries Q_i are satisfiable, it means that $\mathcal{C}$ is indeed including true counterexample(s) that disprove the property. One instance of a true counterexample can be easily retrieved by keeping tracking of solutions to the queries Q_i.

After learning a refinement axiom, IC3+SA *incrementally* resumes the abstract IC3 procedure from the last top frame. Since the abstraction refinement procedure is completely *monotonic* with each iteration making the abstract domain more precise and finer by adding new terms, we can reuse all of the reachability information and abstract clauses from previous iterations.
The refinement procedure provides the following advantages:

- All concrete queries involve a single instance of the transition relation and avoids explicit unrolling.
- There is no path explosion since the refinement is constrained to the paths along the abstract counterexample.
- Symbolic constants for unconstrained variables allow avoiding enumerative simulation on exact variable assignments returned by the solver.
- The procedure is completely incremental and allows reuse of all previous abstract clause learning.

SA+UF: Data abstraction using UF can introduce additional spurious behavior with inconsistencies resulting from the usage of UF instead of concrete data operations. Given $\mathcal{C} = \langle c_0, \ldots, c_n \rangle$, we can check for such inconsistencies using at most n concrete queries $Q_i := SAT\ ?\ [c_{i-1} \wedge T \wedge c_i']$ $(0 < i \leq n)$ in any order (similar to [40]). In the case any query returns UNSAT, we can learn a *refinement (data) axiom* to constrain T. Data axioms will never add any new term and therefore will never increase the size of the abstract state space. They eliminate spurious abstract states/transitions that got introduced due to data abstraction, while path axioms add more granularity.

5 Proof of Correctness

Inspired from [13,25,20], we list the properties on frames preserved by IC3+SA.

(p1) $F_0 = I$ (p2) $F_i \rightarrow P$
(p3) The clauses F_{i+1} is a subset of F_i for $i > 0$ (p4) $F_i \rightarrow F_{i+1}$
(p5) F_{i+1} is an over-approximation of the image of F_i

(p1-5) are true and preserved by the IC3 algorithm [13,25]. After a refinement iteration, all frame clauses remain valid since the refinement procedure (Alg. 2) is monotonic with respect to the terms describing the abstract state space. After each refinement iteration $T_{new} \models T$, implying $F_{new} \models F$, preserving (p1-5).

Lemma 1. *(Correctness) If IC3+SA($\mathcal{P}$) returns an invariant Φ, then Φ is inductive and $\Phi \rightarrow P$ under $\mathcal{P}$.*

Proof. From the IC3 algorithm, let F_{conv} be the frame that reached the fixed point (i.e. $F_{conv} = F_{conv+1}$). Let $\Phi = F_{conv}$. Due to (p5) and (p2), Φ is inductive and $\Phi \rightarrow P$.

Lemma 2. *(Correctness) If IC3+SA($\mathcal{P}$) returns a counterexample $\mathcal{C}$, then $\mathcal{C}$ has a path under T starting from I and violating P.*

Proof. Let $\mathcal{C} = \langle c_0, \ldots, c_n \rangle$. By construction, $c_n = \neg P$. From Sec. 4.1, $\mathcal{C}$ is abstractly continuous. The refinement procedure (Alg. 2) will return $\mathcal{C}$ iff $(I \wedge \big(\bigwedge_{i=0}^{n-1} c_i^i \wedge T^i\big) \wedge c_n^n)$ is satisfiable, implying $\mathcal{C}$ is concretely continuous and a true counterexample.

Lemma 3. *(Termination) IC3+SA($\mathcal{P}$) will eventually terminate.*

Proof. For a given abstract problem $\hat{\mathcal{P}}$, the IC3 algorithm will eventually terminate since all abstract queries are decidable, the number of abstract states is finite, and the maximum number of frames is bounded by the number of abstract states (due to (p2-5)). Each refinement iteration introduces new term(s) making the abstract state space more precise with respect to the concrete state space. The number of new terms that can be added is limited by the sequential depth of the concrete problem $\mathcal{P}$ (which is finite), making the number of refinement iterations finite. Hence, IC3+SA will eventually terminate.

Theorem 1. *IC3+SA($\mathcal{P}$) is sound and complete.*

Proof. From Lemmas 1, 2 and 3, IC3+SA is sound and complete.

6 Implementation and Evaluation

We implemented IC3+SA in C++ in the Averroes system [40]. We made a complete rewrite to the frontend and backend of Averroes, with a primary focus on model checking of Verilog RTL. The new version [26] (Averroes 2, or *avr* in short) uses *yosys* [51] as the preprocessor frontend to allow direct translation of Verilog RTL and SystemVerilog assertions (SVA) into a word-level model checking problem. *yosys* parses the Verilog RTL, removes any hierarchy, and

exports the flat word-level design to *avr* in the .ilang format[2]. *avr* uses Yices 2 [24] (version 2.6) for solving abstract SMT queries and Z3 [23] (version 2.5) for concrete SMT queries.

Setup: We analyzed a total of 535 invariant checking problems (Verilog RTL files with SVA) that can be classified as follows:

- *opensource*: a set of 141 problems collected from benchmark suites accompanying tools *vcegar* [36] (#23), *v2c* [44] (#32) and *verilog2smv* [35] (#86). Problems include cores from picoJava, USB 1.1, CRC generation, Huffman coding, mutual exclusion algorithms, simple microprocessor, etc.
- *industry*: a set of 370 problems collected from industrial collaborators[3]. Of these, 124 were categorized as *easy* (code sizes between 155 and 761 lines; # of flip-flops between 514 and 931), and 235 as *challenging* (code sizes between 109 and 22065 lines; # of flip-flops between 6 and 7249). The remaining 11 problems involved sequential equivalence checking on a multiplier design before and after clock gating optimization.
- *crafted*: a set of 24 simple problems synthetically created for calibration (includes both control- and data-centric problems).

We compared the following techniques:

From ABC version 1.01 [8]:

- *pdr*: *pdr* is one of the best implementations of the bit-level IC3 algorithm.
- *dprove*: *dprove* employs a preprocessing stage using a portfolio of techniques (BMC [9], retiming, fraiging, simulation, interpolation, etc.) with carefully-tuned heuristics to quickly solve/reduce the problem. If the problem remains unsolved, *dprove* invokes *pdr* on the reduced problem.
- *pdr-nct*: the -nct flags configure *pdr* to use better generalization [30] and enable *localization* abstraction [33].

From nuXmv version 1.1.1 [17]:

- *nuxmv-ic3ia*: a word-level IC3 implementation in nuXmv using implicit predicate abstraction [20].

From *Averroes* version 2.0 [26]:

- *avr-ic3sa*: IC3+SA i.e. IC3 with syntax-guided abstraction.
- *avr-ic3sa-uf*: IC3+SA+UF i.e. IC3+SA with data abstraction using UF.

For comparison against bit-level IC3, we chose implementations from ABC since these have shown exceptional performance in HWMCC [10]. We also considered including other abstraction based IC3 techniques like *L-IC3* [48], *UFAR* [31] and *PDR-WLA* [32]. However, *PDR-WLA* was not able to process the designs due to input format issues, while *L-IC3* and *UFAR* do not have, to the best of our knowledge, a publicly available implementation. Techniques like [34,11,38,12] do not have implementations that can handle hardware designs.

We used *yosys* [51] as the common frontend. The Verilog designs and SVA were parsed by *yosys*, which removes any hierarchy and produces flat RTL Verilog. For *nuxmv-ic3ia* and *avr-**, the flat word-level format is syntactically ex-

[2] .ilang is a format for textual representation of the *yosys*'s design.

[3] We obtained these designs under non-disclosure agreements and, unfortunately, cannot make them publicly available.

ported by *yosys* into the equivalent word-level input formats used by these tools. Since ABC based tools cannot exploit word-level information and operate at the bit level, we used *yosys* to *synthesize* the flat RTL to an And-Inverter Graph (AIG) and exported to ABC in .blif format. All experiments were conducted on a cluster of 163 2.5 GHz Intel Xeon E5-2680v3 processors (cores) running 64-bit Linux. Each verification run was given exclusive access to a single core, with a memory limit of 16 GB and a time limit of 5 hours.

6.1 Results

Even though experimental evaluations are necessarily biased by the suite of problems used, we nonetheless believe that useful insights can still be gained from a careful analysis of the results. The raw data along with detailed benchmark statistics and plots along with *opensource* and *crafted* benchmarks can be retrieved from a publicly-accessible repository [1]. The three tool packages used ABC, nuXmv and Averroes 2 are publicly available from [2], [3] and [26] respectively.

The reader is referred to [27] for a summary on the performance of *avr-ic3sa-uf*, which demonstrates the effectiveness of IC3+SA with data abstraction. Here, we provide an in-depth analysis compared to *avr-ic3sa* to better understand the strengths and weaknesses of SA and SA+UF.

Table 2: Number of problems solved. TO: timed out, MO: out of memory, Unique: solved uniquely (not solved by others), IN: *industry*, OS: *opensource*, CR: *crafted*

Tool	**Solved** (535)	**TO**	**MO**	**Error**	**Unique**	**IN** (370)	**OS** (141)	**CR** (24)
pdr	466	69	**0**	**0**	1	308	137	21
dprove	477	57	**0**	1	3	315	**138**	**24**
pdr-nct	466	68	1	**0**	1	308	137	21
nuxmv-ic3ia	389	92	46	8	0	232	133	**24**
avr-ic3sa	461	69	5	**0**	0	302	135	**24**
avr-ic3sa-uf	**526**	**0**	9	**0**	**52**	**368**	134	**24**

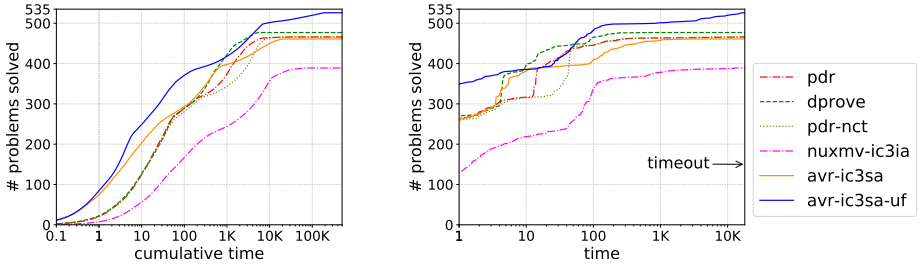


Fig. 1: Survival plot comparing the number of problems solved versus time

Aggregate results: Table 2 and Fig. 1 provide an overview on the performance of each tool. Overall, techniques from ABC, nuXmv and Averroes 2 solved 480, 389 and 527 problems respectively in total. IC3+SA with data abstraction (*avr-ic3sa-uf*) performed the best, particularly in the *industry* category. The performance of *avr-ic3sa* is competitive to ABC tools even though ABC tools have a highly tuned and efficient implementation developed over years of innovation.

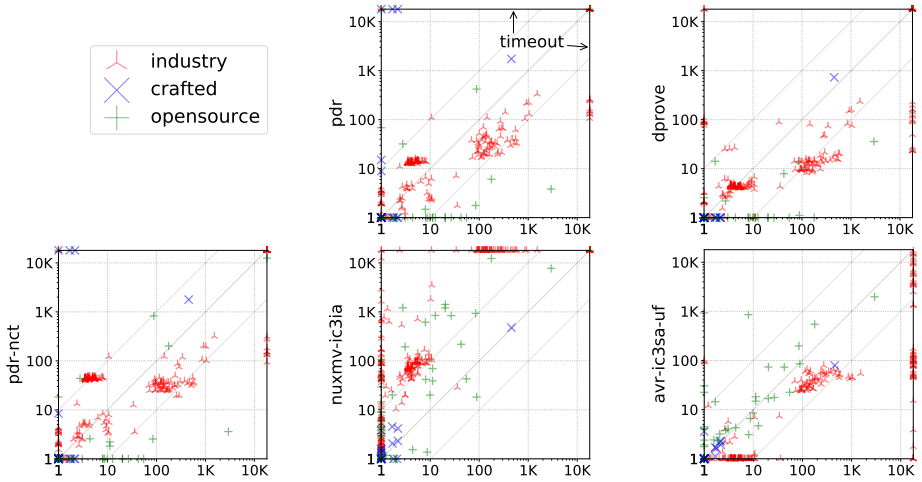


Fig. 2: *avr-ic3sa* runtime comparisons. *avr-ic3sa*'s times are better (resp. worse) above (resp. below) the diagonal.

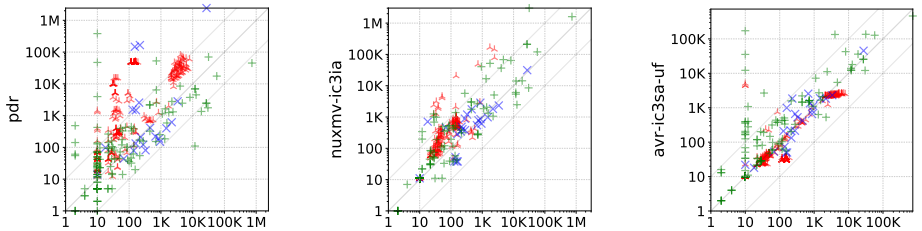


Fig. 3: Number of solver calls (SAT solver calls for *pdr*, SMT solver calls for others)

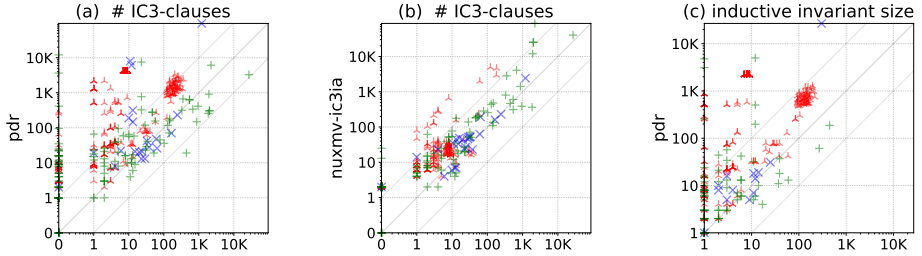


Fig. 4: IC3 statistics

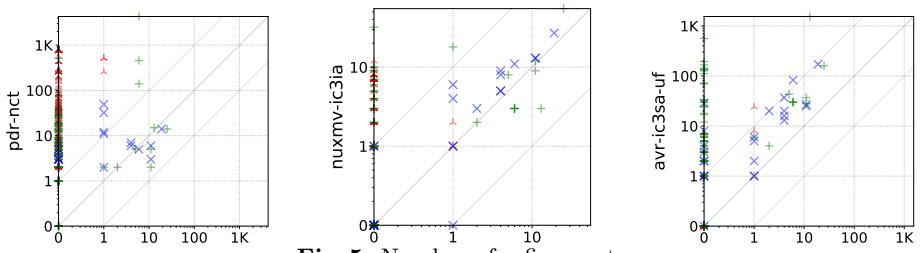


Fig. 5: Number of refinements

avr-ic3sa is always on x-axis. All plots exclude runs in which a tool reported an error or ran out of memory, and all runtime refer to CPU time in seconds.

Runtime comparison: Fig. 2 compares *avr-ic3sa*'s runtime against other tools. ABC tools marginally dominated *avr-ic3sa*, though there is a significant number where *avr-ic3sa* performed better. Bit-level techniques enjoy the advancements in hardware synthesis that can significantly reduce the complexity in the synthesized design, though they loose this advantage for larger and complex designs. Compared to *nuxmv-ic3ia*, *avr-ic3sa* shows good benefits and demonstrates the benefits of SA over implicit predicate abstraction [20]. Data abstraction helps *avr-ic3sa-uf* to outperform *avr-ic3sa* in the *industry* category (where the property is control intensive), while *avr-ic3sa* is better in the data-dependent *opensource* category.
Solver calls: Fig. 3 shows the comparison of the total number of solver calls. Bit-level IC3 (represented by *pdr*) makes orders-of-magnitude more SAT solver calls compared to the number of SMT calls made by word-level tools. Even with many more solver calls, bit-level techniques are competitive to word-level techniques w.r.t. runtime (Fig. 2), indicating the advancement gap between SAT versus SMT solving. Structural cube generalization and syntax-guided abstraction allows *avr-ic3sa* to require fewer solver calls than *nuxmv-ic3ia*. The large number of solver calls made by *avr-ic3sa-uf* compared to *avr-ic3sa* in the *opensource* category reflects the importance of correct abstraction procedure and suggests possible benefits from a hybrid abstraction on a subset of data operations that can tune automatically based on the nature of the property.
Clause learning: Fig. 4.a-b compares the number of frame clauses derived by *avr-ic3sa* versus *pdr* and *nuxmv-ic3ia*. *avr-ic3sa* requires orders-of-magnitude fewer clauses compared to *pdr*, showing the benefits of word-level clause learning as against weak propositional learning. Fewer frame clauses derived by *avr-ic3sa* as compared to *nuxmv-ic3ia* reflects that SA is better in capturing the important details of the problem compared to implicit predicate abstraction.
Number of refinements: Fig. 5 shows the comparison of the number of refinements required for the techniques that use an abstraction refinement procedure (*pdr-nct*, *nuxmv-ic3sa*, *avr-**). The number of refinements required by *avr-ic3sa* is the least compared to all others, demonstrating the effectiveness of syntax-guided abstraction. As expected, *avr-ic3sa-uf* has to undergo several refinement iterations for the data-dependent *opensource* category.
Invariant size: Model checking on Verilog RTL instead of post-synthesis netlist has the additional benefit of producing human-readable word-level inductive invariants. *avr-ic3sa* produces a concise and informative word-level inductive invariant with much fewer clauses than one produced by *pdr* (Fig. 4.c).

7 Related work

Several approaches from different domains have been suggested to extend the bit-level IC3 procedure. From the hardware domain, the authors of [48] suggest *lazy abstraction* using "visible variables". The authors of [31] use UF to abstract away expensive data operations, followed by bit-blasting. The authors of [32] use unconstrained new primary inputs to abstract away parts of the system. The authors of [33] suggest using *localization abstraction* to cut away irrelevant

logic. All these approaches [48,31,32,33] use bit-level IC3 as the core engine and suffer with the same scalability issues as with bit-level IC3.

Certain approaches propose performing word-level IC3 using SMT solvers. The authors in [50] generalize IC3 to the theory of bitvectors by using *polytopes* and interval simulation. The authors of [40,39] suggest performing word-level IC3 with data abstraction using uninterpreted functions.

Beyond hardware, different approaches propose to lift the IC3 procedure to richer logics and infinite state systems [19,34,11,38,12]. Our approach differ significantly from these techniques as IC3+SA does not rely on theory specific under-approximation of the pre-image, quantifier elimination, weakest preconditions or interpolation. The authors of [20,21] suggest using *implicit predicate abstraction* that performs a word-level IC3 procedure (IC3IA), and refines the abstraction by adding new predicates. Our approach is partly similar to IC3IA, but with better granularity and expressiveness resulting in fewer occurrences of spurious behavior. We also suggest an inexpensive syntax-driven cube generalization procedure for word-level IC3, along with a fully incremental refinement procedure without using multiple copies of the transition relation. Unlike CTIGAR [11], our cube generalization technique does not require any *lifting* solver query to eliminate non-essential symbols and still guarantees continuity.

Data abstraction using UF has been applied for both hardware [15,4,40,39] and software [5]. IC3+SA allows for an easy and scalable extension to data abstraction, and unlike [40,39], it does not face non-termination issues.

8 Conclusions and Future Work

Syntax-guided abstraction suggests an alternative way to raise bit-level IC3 procedure to the word level. SA is implicitly defined by the terms in the syntax of the problem and offers high granularity. We demonstrate how to integrate IC3 with SA efficiently, and propose a word-level structural cube generalization procedure without any need for additional solver queries or unrolling. We show the correctness of the technique and evaluate the effectiveness of the approach on a suite of open-source and industrial hardware problems.

Future work include extending SA to theories beyond bitvector, adding hybrid data abstraction on a subset of data operations, and performing a rigorous analysis against other model checking tools including techniques beyond IC3.

Acknowledgement. We would like to thank the reviewers for their valuable comments. The authors thank developers of Yosys [51], Yices 2 [24] and Z3 [23] for making their tools openly available. The authors thank Alberto Griggio for providing a custom version of nuXmv with detailed statistics output.

References

1. `https://github.com/aman-goel/nfm2019exp`
2. ABC: System for Sequential Logic Synthesis and Formal Verification. `https://github.com/berkeley-abc/abc`
3. The nuXmv model checker. `https://nuxmv.fbk.eu`

4. Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Reveal: A formal verification tool for verilog designs. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 343–352. Springer (2008)
5. Babić, D., Hu, A.J.: Structural abstraction of software verification conditions. In: International Conference on Computer Aided Verification. pp. 366–378. Springer (2007)
6. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking c programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 268–283. Springer (2001)
7. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
8. Berkeley Logic Synthesis and Verification Group: ABC: A system for sequential synthesis and verification. `http://www.eecs.berkeley.edu/~alanmi/abc/` (2017)
9. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y., et al.: Bounded model checking. Advances in computers **58**(11), 117–148 (2003)
10. Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: FMCAD. pp. 9–9 (2017)
11. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (ctigar). In: International Conference on Computer Aided Verification. pp. 831–848. Springer (2014)
12. Bjørner, N., Gurfinkel, A.: Property directed polyhedral abstraction. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 263–281. Springer (2015)
13. Bradley, A.R.: Sat-based model checking without unrolling. In: VMCAI. pp. 70–87 (2011)
14. Bradley, A.R., Somenzi, F., Hassan, Z.: IIMC: Incremental inductive model checker. `http://www.github.com/mgudemann/iimc`
15. Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: International Conference on Computer Aided Verification. pp. 68–80. Springer (1994)
16. Cabodi, G., Nocco, S., Quer, S.: The PdTRAV tool. `http://fmgroup.polito.it/index.php/download/viewcategory/3-pdtrav-package`
17. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: CAV. pp. 334–342 (2014)
18. Chockler, H., Ivrii, A., Matsliah, A., Moran, S., Nevo, Z.: Incremental formal verification of hardware. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design. pp. 135–143. FMCAD Inc (2011)
19. Cimatti, A., Griggio, A.: Software model checking via ic3. In: International Conference on Computer Aided Verification. pp. 277–293. Springer (2012)
20. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: TACAS. pp. 46–61 (2014)
21. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with ic3 and predicate abstraction. Formal Methods in System Design **49**(3), 190–218 (2016)
22. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. pp. 154–169 (2000)
23. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)

24. Dutertre, B.: Yices 2.2. In: International Conference on Computer Aided Verification. pp. 737–744. Springer (2014)
25. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD. pp. 125–134 (2011)
26. Goel, A., Sakallah, K.: Averroes 2. `http://www.github.com/aman-goel/avr`
27. Goel, A., Sakallah, K.: Empirical evaluation of ic3-based model checking techniques on verilog rtl designs. In: Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium (2019)
28. Graf, S., Saïdi, H.: Construction of abstract state graphs with pvs. In: International Conference on Computer Aided Verification. pp. 72–83. Springer (1997)
29. Gupta, A., Yang, Z., Ashar, P., Gupta, A.: Sat-based image computation with application in reachability analysis. In: International Conference on Formal Methods in Computer-Aided Design. pp. 391–408. Springer (2000)
30. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in IC3. In: FMCAD. pp. 157–164 (2013)
31. Ho, Y.S., Chauhan, P., Roy, P., Mishchenko, A., Brayton, R.: Efficient uninterpreted function abstraction and refinement for word-level model checking. In: FMCAD. pp. 65–72 (2016)
32. Ho, Y.S., Mishchenko, A., Brayton, R.: Property directed reachability with word-level abstraction. In: FMCAD. pp. 132–139 (2017)
33. Ho, Y.S., Mishchenko, A., Brayton, R., Eén, N.: Enhancing PDR / IC3 with localization abstraction (2017)
34. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 157–171. Springer (2012)
35. Irfan, A., Cimatti, A., Griggio, A., Roveri, M., Sebastiani, R.: Verilog2smv: a tool for word-level verification. In: Proceedings of the 2016 Conference on Design, Automation & Test in Europe. pp. 1156–1159. EDA Consortium (2016)
36. Jain, H., Kroening, D., Sharygina, N., Clarke, E.: Vcegar: Verilog counterexample guided abstraction refinement. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 583–586. Springer (2007)
37. Kurshan, R.P.: Computer-aided verification of coordinating processes. princeton series in computer science (1994)
38. Lange, T., Neuhäußer, M.R., Noll, T.: Ic3 software model checking on control flow automata. In: Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design. pp. 97–104. FMCAD Inc (2015)
39. Lee, S.: Unbounded scalable hardware verification. (2016)
40. Lee, S., Sakallah, K.A.: Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In: CAV. pp. 849–865 (2014)
41. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning **40**(1), 1–33 (2008)
42. McMillan, K.L.: Applications of craig interpolants in model checking. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 1–12. Springer (2005)
43. Mneimneh, M., Sakallah, K.: Sat-based sequential depth computation. In: Proceedings of the 2003 Asia and South Pacific Design Automation Conference. pp. 87–92. ACM (2003)
44. Mukherjee, R., Tautschnig, M., Kroening, D.: v2c–a verilog to c translator. In: TACAS. pp. 580–586 (2016)

45. Oh, Y., Mneimneh, M.N., Andraus, Z.S., Sakallah, K.A., Markov, I.L.: Amuse: a minimally-unsatisfiable subformula extractor. In: Proceedings of the 41st annual Design Automation Conference. pp. 518–523. ACM (2004)
46. Rota, G.C.: The number of partitions of a set. The American Mathematical Monthly **71**(5), 498–504 (1964)
47. Tafertshofer, P., Ganz, A.: Sat based atpg using fast justification and propagation in the implication graph. In: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design. pp. 139–146. IEEE Press (1999)
48. Vizel, Y., Grumberg, O., Shoham, S.: Lazy abstraction and sat-based reachability in hardware model checking. In: FMCAD. pp. 173–181 (2012)
49. Vizel, Y., Gurfinkel, A.: Interpolating property directed reachability. In: CAV. pp. 260–276 (2014)
50. Welp, T., Kuehlmann, A.: Qf bv model checking with property directed reachability. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 791–796. EDA Consortium (2013)
51. Wolf, C.: Yosys open synthesis suite. `http://www.clifford.at/yosys/`