Syntax-Guided Synthesis for Lemma Generation in Hardware Model Checking

Hongce Zhang, Aarti Gupta, and Sharad Malik

Princeton University, Princeton NJ 08544, USA
{hongcez,aartig,sharad}@princeton.edu

Abstract. In this work we propose to use Syntax-Guided Synthesis (SyGuS) for lemma generation in a word-level IC3/PDR framework for bit-vector problems. Hardware model checking is moving from bit-level to word-level problems, and it is expected that model checkers can benefit when such high-level information is available. However, for bit-vectors, it is challenging to find a good word-level interpolation strategy for lemma generation, which hinders the use of word-level IC3/PDR algorithms. Our SyGuS-based procedure, SyGuS-$\mathcal{A}$PDR, is tightly integrated with an existing word-level IC3/PDR framework $\mathcal{A}$PDR. It includes a predefined grammar template and term production rules for generating candidate lemmas, and does not rely on any extra human inputs. Our experiments on benchmarks from the hardware model checking competition show that SyGuS-$\mathcal{A}$PDR can outperform state-of-the-art Constrained Horn Clause (CHC) solvers, including those that implement bit-level IC3/PDR. We also show that SyGuS-$\mathcal{A}$PDR and these CHC solvers can solve many instances faster than other leading word-level hardware model checkers that are not CHC-based. As a by-product of our work, we provide a translator `Btor2CHC` that enables the use of CHC solvers for general hardware model checking problems, and contribute representative bit-vector benchmarks to the CHC-solver community.

Keywords: hardware model checking · Syntax-Guided Synthesis (SyGuS) · bit-vector theory · lemma generation · CHC solver.

1 Introduction

Hardware bugs are circuit design errors that can cause malfunction or security breaches, which could further lead to system failures or economic losses. Compared to software bugs, hardware bugs tend to be more costly to fix due to the need for a physical replacement and high non-recurring expenses for respins. Therefore, it is very important to ensure the correctness of hardware designs before manufacturing. Model checking [18], which formally checks whether certain correctness properties hold in a state transition system, has been successfully applied in finding hardware bugs or proving there are no property violations.

In hardware model checking, descriptions of the circuit and properties to be checked are given as inputs to an automated tool. The design description, until

recently, was typically provided using a bit-level format called AIGER [7], which uses the and-inverter graph (AIG) representation. The AIGER format is compact and close to a post-logic-synthesis hardware implementation. However, it lacks word-level information which could be helpful in improving scalability of hardware model checking. Recently, in 2019, the hardware model checking competition (HWMCC) started to advocate use of a word-level description called Btor2 [49]. It follows similar principles as the bit-level AIGER format, but instead uses SMT-LIB2 [6] logics for bit-vectors and arrays. This format preserves the word-level information in the circuit description. For example, a 32-bit adder in Btor2 can be represented succinctly using a single "add" operator (namely the modular addition function `bvadd` in SMT-LIB2), whereas in AIGER format, it is bit-blasted into single-bit half and full adders represented using 378 AIG nodes. The Btor2 format allows model checkers to potentially take advantage of the high-level circuit structure.

Along with the Btor2 word-level format, there has been interest in using Constrained Horn Clauses (CHCs) to describe digital circuits and properties at the word-level, and CHC solvers have been used or developed to synthesize environment invariants [59]. Although CHC solvers have largely been used in software verification [13,24,30,35,39,48], the associated techniques to find invariants may also be helpful in hardware verification. Many CHC solvers can successfully find invariants in linear integer/real arithmetic (LIA/LRA) and array theories. For example, Spacer [39] extends the IC3/PDR algorithm [11, 21] to $\mathcal{A}$PDR [8, 32] (and also other variants) for LIA/LRA, where Craig interpolants are used to generate *lemmas* that are conjoined to construct an invariant. However, when it comes to supporting bit-vectors, the lack of a native word-level interpolation strategy hinders the use of $\mathcal{A}$PDR and similar techniques. Indeed, our experiments on bit-vector problems show that directly using word-level interpolants from an SMT solver for lemma generation in $\mathcal{A}$PDR can actually incur a performance loss.

In this paper, we propose our solution to address this problem. We propose to use Syntax-Guided Synthesis (SyGuS) [2] for generating lemmas for invariants, in a new method called SyGuS-$\mathcal{A}$PDR. It is tightly integrated with an IC3/PDR framework, where models from deductive reasoning in IC3/PDR are used to guide the generation of lemmas. In particular, it uses a general grammar template with predicate and term production rules without any need of extra human input, and where the search space of predicates and terms is pruned based on the deduced models. It also tightens previous frames in IC3/PDR to allow a larger set of lemma candidates to be considered. In addition to this tight integration, our method includes other known techniques [21, 37] specialized here to support word-level reasoning for bit-vectors—generalization of lemma candidates by extracting minimal UNSAT subset (MUS), and partial model generation in predecessor generalization. These features are summarized in Figure 1.

We have implemented our proposed SyGuS-$\mathcal{A}$PDR algorithm using SMT-Switch [44], which provides an interface to various SMT (Satisfiability Modulo Theory) solvers in the backend. We describe an extensive evaluation of SyGuS-

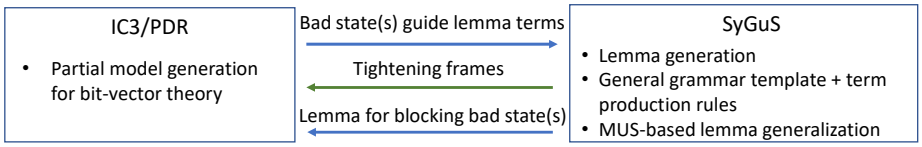


Fig. 1: Integration of IC3/PDR with SyGuS-based lemma generation

$\mathcal{A}$PDR against state-of-the-art CHC solvers and hardware model checkers on benchmarks from the bit-vector track of HWMCC'19. Our experiments show that SyGuS-$\mathcal{A}$PDR outperforms state-of-the-art CHC solvers with more solved instances. Furthermore, we show that CHC solvers can deliver better performance on a notable portion of the benchmarks, in comparison to other hardware model checkers that are not CHC-based. Finally, as part of this evaluation, we have developed a translator that can convert Btor2 to the standard CHC format. This enables other CHC solvers to be used on word-level hardware model checking problems.
Summary of contributions:

- We present a novel algorithm SyGuS-$\mathcal{A}$PDR that uses SyGuS-based lemma generation for word-level bit-vector reasoning in an IC3/PDR framework. It is distinctive in using a tight integration between the two, where: (1) the space of lemma candidates is guided both by a general grammar template and models provided by IC3/PDR, (2) existing functionality in IC3/PDR is used to tighten previous frames, which allows a larger set of lemma candidates to be considered.
- We have implemented SyGuS-$\mathcal{A}$PDR and provide an extensive empirical evaluation against other tools on the HWMCC'19 benchmarks.
- We enable application of CHC solvers on hardware model checking problems via a translation tool `Btor2CHC` developed as part of this work. We have made the translated HWMCC'19 benchmarks publicly available [58].

The paper is organized as follows. We start with some background in the next section, and describe a motivating example (Section 3). In Section 4, we present the SyGuS-$\mathcal{A}$PDR algorithm. Section 5 describes the experimental evaluation and results, followed by related work and conclusions.

2 Background and Notation

2.1 Constrained Horn Clauses (CHCs)

A Constrained Horn Clause is a first order logic (FOL) formula over some background theory $\mathcal{A}$ in the following form:

$$\forall v_1, v_2, ..., v_n, \phi(V) \wedge \left(\bigwedge_k p_k(V_k) \right) \rightarrow h(V_h) \quad (1)$$

Here $v_1, v_2, v_3, ..., v_n$ form the set of variables V from theory $\mathcal{A}$. ϕ is an interpreted constraint over the functions and variables in $\mathcal{A}$, and p_k and h are uninterpreted predicate symbols over sets of variables V_k and V_h, respectively. V_k and V_h are subsets of V, and can be empty. A CHC is satisfiable if there exists an interpretation $\mathcal{I}$ for the predicate symbols p_k and h that makes the formula valid. A set of CHCs is satisfiable if there exists an interpretation $\mathcal{I}$ for all the predicate symbols that make all CHC formulas valid.

2.2 Hardware Model Checking using CHCs

Here we focus on safety properties in hardware model checking. A digital circuit can be viewed as a state transition system: $\langle V, \mathit{Init}, T \rangle$, where V is a set of state variables (along with the primed version of variables V' that denote next states), Init is a predicate representing initial states, and T is a transition relation. Note that for hardware model checking, the transition relation T is functional, i.e., $T(V, V') \coloneqq V' = \mathit{Next}(V)$. For a given safety property P, we would like to check if the transition system will ever reach a state (the bad state) where P does not hold. If all bad states are unreachable, we would like to get a proof showing P is valid. One such proof is an inductive invariant (Inv):

$$\mathit{Init}(V) \rightarrow \mathit{Inv}(V) \tag{2}$$
$$\mathit{Inv}(V) \wedge T(V, V') \rightarrow \mathit{Inv}(V') \tag{3}$$
$$\mathit{Inv}(V) \rightarrow P(V) \tag{4}$$

In other words, Inv should hold in the initial states (2), it should be inductive (3), and it should imply safety (4). These three constraints are in the form of CHCs (with an implicit universal quantification over all variables), where Inv is an uninterpreted predicate. If these CHCs are satisfiable, then the interpretation of Inv is the inductive invariant that forms the proof of safety.

2.3 IC3/PDR and $\mathcal{A}$PDR

The IC3/PDR algorithm [11, 21] constructs inductive invariants to check safety. It maintains a sequence of forward reachable sets of states (FRS): F_i, which are over-approximations of all reachable states in i steps. They satisfy the following properties.

$$F_0(V) = \mathit{Init}(V) \tag{5}$$
$$F_i(V) \wedge T(V, V') \rightarrow F_{i+1}(V') \tag{6}$$
$$F_i(V) \rightarrow F_{i+1}(V) \tag{7}$$
$$F_i(V) \rightarrow P(V) \tag{8}$$

The algorithm converges if at any point $F_{i+1}(V) \rightarrow F_i(V)$. As F_i is in the form of a conjunction of clauses (for bit-level PDR) or lemmas (for word-level PDR), when it is clear from the context, we interchangeably use F_i to refer to either the conjunction or the set of clauses/lemmas.

The procedure for constructing FRS can be viewed as iteratively blocking bad states or their predecessors (states that can reach bad states following the transitions) by applying the following rules in an indefinite order. (We refer the readers to [32] for details.)

- `Unreachable`. If $\exists i, F_{i+1} \rightarrow F_i$, the system is safe ($P$ holds) and the algorithm converges.
- `Unfold`. For the last FRS: F_N in the series, if $F_N \wedge T \rightarrow P'$, then extend the series with $F_{N+1} \leftarrow P$ and $N \leftarrow N + 1$.
- `Candidate`. For the last FRS: F_N in the series, if $\exists m, m \models F_N \wedge T \wedge \neg P'$, then we need to add $\langle m, N\rangle$ as a proof obligation (meaning that we would like to try blocking m at step N as it can lead to the failure of P).
- `Predecessor`. For a proof obligation $\langle m, i + 1\rangle$, according to the transition relation T, if there is a predecessor m_i of it at step i, then we will also add $\langle m_i, i\rangle$ to the proof obligation.
- `NewLemma`. For a proof obligation $\langle m, i+1\rangle$, if we found no predecessor of it at step i, then try to find a lemma l showing m is infeasible at $i+1$, and update all $F_j, j \leq i+1$ with l to remember this (explained in details in Section 4.1).
- `Push`. For a lemma l in F_i, see if it also holds at step $i + 1$.
- `ReQueue`. For a proof obligation $\langle m, i\rangle$, if we found it has no predecessor at $i - 1$, then also add m to the proof obligation at step $i + 1$.
- `Reachable`. If we get a proof obligation at step 0, then the system is unsafe. The algorithm stops.

For CHCs in different theories, the theory-dependent techniques used in the above procedures may vary. In particular, $\mathcal{A}$PDR [8] (in Spacer [39]) implements the two procedures—`Predecessor` and `NewLemma`—using model-based projection [39] and Craig interpolation [46], respectively, to adapt IC3/PDR for LIA/LRA theories.

2.4 CHC Solving Techniques for bit-vectors

As the hardware model checking problems require bit-vectors, here we focus our discussion on solving CHCs in bit-vector theory.

Bit-Blasting. The original IC3/PDR algorithm [11, 21] is applicable if the BV problems are *bit-blasted*, i.e., transformed into propositional logic with one Boolean variable for each bit in each bit-vector variable. This is the general approach implemented in Spacer [39] for bit-vectors. For the special case where a problem contains only arithmetic operators in the BV theory, Spacer can attempt the translation method described below.

Translation and Abstraction. Another approach for solving bit-vector problems is to translate them into another theory (e.g., LIA or LRA), derive a safe inductive invariant, and then port it back soundly to the bit-vector theory. This approach is discussed in related work [33] and implemented in the PDR engine in Spacer.

Table 1: A simple transition system: $\langle V, Init, T \rangle$

V	$\{a, b, c, e, i\}$, a, b, c, i: `(_BitVec 16)`, e : `Bool`
$Init$	$a = 0 \wedge b = 0 \wedge c = 0$
T	$a' = \text{ite}\,(e, i, a + 1) \wedge b' = \text{ite}\,(e, i, b) \wedge c' = \text{ite}\,(e, 0, c + 1)$

Another CHC solver, Eldarica [35], handles bit-vector theory through abstraction, as well as translation. It applies lazy Cartesian predicate abstraction [5, 29], in combination with a variant of counterexample-guided abstraction refinement (CEGAR) [4, 17]. Bit-vectors are lazily mapped to quantifier-free Presburger constraints, and then solved and interpolated by an SMT solver. Using the abstractions, it constructs an abstract reachability graph (ARG). To eliminate spurious counterexamples in the abstract reachability relation, it obtains additional predicates from Craig interpolation.

Learning-based Methods. There have also been other efforts that use learning-based or guess-and-check approaches for CHC solving, e.g., SynthHorn [61], FreqHorn [23–25], HoICE [13], Code2Inv [54]. However, to the best of our knowledge, these tools currently do not offer support for bit-vector theory.

3 A Motivating Example

We use an example to illustrate why word-level reasoning is beneficial and also how word-level interpolants for bit-vectors can fail to converge. Table 1 shows a simple transition system. This is a case simplified from a verification problem of a domain-specific accelerator design we encountered in our previous hardware verification work [59].

For simplicity of presentation, we use "+" to represent bit-vector addition: `bvadd`, which will wrap-around in the case of overflow and we use `ite` as the short form for "if-then-else". Variables a, b, and c correspond to three registers in the circuit, and e and i are primary inputs. All variables except e are 16-bit wide. a and c will count up if $e = \bot$ (false), while a and b will be loaded with input i and c will be cleared if $e = \top$ (true). A simple property to check can be, for example, if state $(a, b, c) = (6, 4, 1)$ is reachable. For a human looking at this transition system, it is not hard to find that $a = b + c$ is an inductive invariant. Initially a, b, c are all 0. Subsequently, if $e = \top$, then $a = b$ and $c = 0$, and the relation holds; if $e = \bot$, then a and c both increase by 1 (and may cause both sides of the equality to wrap-around) so the equality relation still holds. This relation is easy to find for a human analyzer, however, it turns out to be hard for a bit-level model checker because the bit-blasting breaks the word addition into bit-level operations and the invariant becomes much more complex.

On the other hand, when we directly use a word-level bit-vector interpolator [31] out-of-the-box, to generate lemmas for blocking $(a, b, c) = (6, 4, 1)$ or other models that lead to it, we actually get these lemmas: $l_1 : a = b \vee b \neq 4$,

$l_2 : a = b + 2 \vee a = b + 1 \vee a = b \vee b = 4$, $l_3 : (b, c) \neq (4, 65535)$. There are several issues with these lemmas: (1) They only hold for the frames explored so far. This is true for all three lemmas shown here. (2) Although some may look similar to an inductive invariant, they are not general enough (e.g., l_2). (3) Some are overly generalized, like l_3, which drops a and will become invalid after 65536 steps. In general, interpolation in BV theory is hard, because unlike LIA/LRA, there is no counterpart to the Farkas's lemma [22] in BV theory that can directly provide a word-level interpolant.

In this work, we address these issues by using SyGuS in a tight integration with an IC3/PDR framework, as described in detail in the next section.

4 Integrating SyGuS with IC3/PDR

A SyGuS-based guess-and-check approach is flexible, but it can be quite expensive when the search space of candidate lemmas is large, and enumerating through the candidates is expensive. We address these issues by using SyGuS in a tight integration with an IC3/PDR framework where the distinctive features are: (1) We use the models from IC3/PDR to guide and prune the search space of predicates in lemma candidates. (2) We provide a general grammar template and production rules to generate new terms in lemma candidates. These rules use hardware-specific insights as heuristics to prioritize the search and term generation. (3) We use procedures in IC3/PDR to tighten previous frames to allow using lemmas that are otherwise not considered. In addition, we use UNSAT core minimization to create a more general lemma from a set of predicates, and use partial model generation for predecessor generalization to support bit-vector theory in IC3/PDR on the word-level.

4.1 Lemma Formulation

In IC3/PDR, a lemma is needed when some previously generated bad state(s) m in F_{i+1} should be blocked because it has no predecessor in F_i, i.e., when the following implication is valid:

$$(F_i(V) \wedge T(V, V')) \vee \mathit{Init}(V') \rightarrow Q(V') \tag{9}$$

Here, F_i is a set of lemmas learned at step i, and $Q(V') \coloneqq \neg \bigwedge_k (V'_k = c_k)$, where c_k is the assignment to variable V_k in the bad state m. This is illustrated in Figure 2(a). To learn this fact for future use, we would like to add it (i.e., conjoin it) with F_{i+1}. Although formula Q can itself be conjoined with F_{i+1} (as well as all $F_j, j \leq i$, thanks to the monotonicity of the series of F), typical IC3/PDR procedures will try to find a stronger lemma l such that $l \rightarrow Q$, and conjoin l with F_{i+1} instead. It is hoped that l can potentially block more unreachable states.

For LIA/LRA, the $\mathcal{A}$PDR algorithm uses Craig interpolants to derive lemma l. For bit-vectors, although there are existing word-level interpolation methods [3, 31] using techniques like equality and uninterpreted function (EUF) layering, equality substitution, linear integer encoding and lazy bit-blasting etc.,

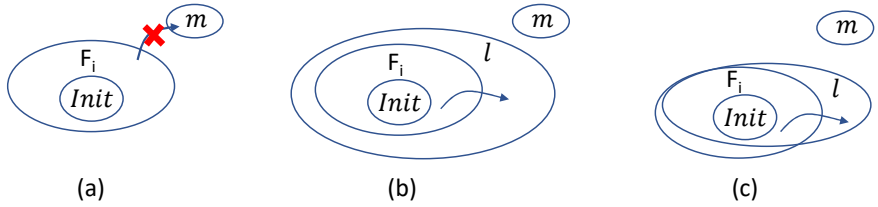


Fig. 2: (a) A bad state(s) m can be blocked when it has no predecessor in F_i. (b) Using constraint (10) for lemma generation and (c) using constraint (12) for lemma generation

our experiments in Section 5 show that using these interpolants actually incurs a performance loss and makes the word-level IC3/PDR slower than bit-blasting.

When viewing interpolation as constraint-based synthesis, the interpolator can be seen as trying to find a candidate l that satisfies the following two constraints (there is an implicit universal quantification over all variables):

$$(F_i(V) \wedge T(V, V')) \vee \mathit{Init}(V') \rightarrow l(V') \tag{10}$$

$$l(V') \rightarrow Q(V') \tag{11}$$

These requirements are sufficient but not necessary conditions for a lemma l. In fact, (10) can be relaxed to the following form (similar to what is used for inductive generalization in IC3 [11]):

$$(\boldsymbol{l(V)} \wedge F_i(V) \wedge T(V, V')) \vee \mathit{Init}(V') \rightarrow l(V') \tag{12}$$

The difference between (10) and (12) is that the latter applies l on the previous frame also. Any candidate l that satisfies (10) will also satisfy (12), but the reverse is not true. In fact, using constraint (12) allows finding a lemma l that can also tighten F_i at the same time, whereas (10) finds lemmas that contain all states in F_i and also all states that are one-step reachable from F_i. Therefore, using (12) is helpful if the previously generated lemmas in F_i are too weak. The difference between using (10) or (12) for lemma generation is illustrated in Figure 2(b) and (c).

As an example, if the over-approximation introduced by F_i is already too coarse, even if we can somehow "magically" guess a safe inductive invariant *Inv* correctly, *Inv* may not even hold for (10). On the other hand, choosing (12) instead of (10) will shift the cost to lemma generation. Note that while 12) is similar to prior work [11, 12], we target lemmas with bit-vectors rather than Boolean clauses. Our solution to this problem is to have a tightening procedure that will also generate lemmas in F_i while still using (10) as the constraint. This will be explained in Section 4.5. As we choose (10), our method can be viewed as SyGuS-based interpolation, combined with an additional tightening procedure (usually available) in the IC3/PDR framework.

$\langle \texttt{Cand} \rangle ::= \neg \langle \texttt{Conj} \rangle$

$\langle \texttt{Conj} \rangle ::= \langle \texttt{Pred} \rangle \mid \langle \texttt{Pred} \rangle \wedge \langle \texttt{Conj} \rangle$

$\langle \texttt{Pred} \rangle ::= \langle \texttt{Term} \rangle \ \mathit{Comparator}_{BV} \ \langle \texttt{Term} \rangle$

$\langle \texttt{Term} \rangle ::= \langle \texttt{Constant} \rangle \mid \langle \texttt{Variable} \rangle \mid \langle \texttt{Term} \rangle \ \mathit{Operator}_{BV} \ \langle \texttt{Term} \rangle$

Fig. 3: The grammar template for lemmas, where operators and terms are dynamically generated.

4.2 SyGuS-based Interpolation

The grammar template that we use for learning an interpolant, i.e., the lemma l, is shown in Figure 3. The top-level lemma candidate ($\langle \texttt{Cand} \rangle$) is a negated conjunction of predicates over BV theory. The predicates and terms used in the predicates are dynamically generated and pruned due to a tight integration of SyGuS with IC3/PDR. We would like to first give an overview of our SyGuS-based approach, and leave the discussion of operators and terms to Section 4.4.

At a high-level, our SyGus-based lemma generation procedure is shown in Algorithm 1). For given bad state(s) m, our method first checks if we have encountered m before (due to the `ReQueue` rule in $\mathcal{A}$PDR, it may have been blocked at some previous frame j, $j \leq i$). If so, the previously generated predicate set is reused (Line 2) to save the work of predicate generation. Otherwise, it will invoke the predicate generation procedure to get an initial set of predicates based on m (Line 4). For a set of predicates L, it will check if L is sufficient to generate a lemma (Line 5). If the current set is insufficient, it will try to tighten the previous frame first (Line 7). If after tightening, the current predicate set is still insufficient, it will then incrementally construct more predicates, while factoring

Algorithm 1: NewLemma(Init, F_i, T, m): Generating lemma when m has no predecessor in F_i

Input: Init: initial states, F_i: set of lemmas at step i, T: transition relation, m: the bad state to block at step $i+1$

Output: the lemma that blocks m

1 **if** m has been blockable on/before step i **then**
2 $L \leftarrow$ GetPreviouslyGeneratedPredSet (m) ;
3 **else**
4 $L \leftarrow$ GenInitialPredSet (m) ;
5 **while** $\neg$ PredSufficient (F_i, T, L) **do**
6 $(n, n') \leftarrow$ Model $(\ ((F_i \wedge T) \vee \mathit{Init}') \wedge \neg l_L\ )$;
7 **if** $\neg$ RecBlock (n, i, MAY) **then**
8 $L \leftarrow L \cup$ MorePred (m, n, n') ;
9 $\mathit{base} \leftarrow (F_i \wedge T) \vee \mathit{Init}'$;
10 $L \leftarrow$MUS $(L \cup \{\mathit{base}\})$;
11 **return** $\neg \bigwedge_{p \in L} p$;

in both m and the model (n, n') demonstrating L is insufficient (Line 8). After L finally becomes sufficient, it will invoke the MUS procedure (Line 10) to get a minimal set of predicates using UNSAT cores.

In Algorithm 1, GetPreviouslyGeneratedPredSet is simply to retrieve the cached predicates for the same model. GenInitialPredSet and PredSufficient will be described in Section 4.3. Our approach to generate new terms to make new predicates in MorePred is presented in Section 4.4. MUS is briefly discussed in Section 4.6, and RecBlock is the recursive blocking function provided by an IC3/PDR framework.

Note that our SyGuS-based method is tightly integrated with the IC3/PDR framework and uses the results from deductive solving to guide the generation of predicates (Line 4 and 8), and uses procedures in IC3/PDR to tighten previous frames to allow using lemmas that are otherwise not considered (Line 7). It checks the sufficiency of a set of predicates using a single SMT query to expedite candidate validation (Line 5) and uses UNSAT core minimization (Line 10) to construct a more general lemma.

Theorem 1. *For bit-vector problems, IC3/PDR converges when using Algorithm 1 for lemma generation.*

This is because the lemmas generated by Algorithm 1 will block the given bad states, and the rest follows from correctness of the original IC3/PDR algorithm.

4.3 Lemma Generation and Validation

Pruning based on bad state. The lemma generation procedure is invoked if formula (9) is found to be valid, and we need to find a candidate l that satisfies (10) and (11). Our SyGuS method starts with handling constraint (11) first. Note that l is in the form of $\neg(p_1 \wedge p_2 \wedge ... \wedge p_n)$ according to the grammar in Figure 3 and the equivalent contrapositive of (11) is $\neg Q \rightarrow \neg l$. Therefore, $\neg Q$ should imply every predicate p_1, p_2, ... p_n in l. For a given Q, our SyGuS approach will only construct predicates from the grammar that can be implied from $\neg Q$, i.e., model m. This is shown as pruning on the initial predicate set based on model m (Line 4, GetInitialPredSet in Algorithm 1).

Candidate Validation. After generating predicates that satisfy (11), we consider the constraint (10). Our method starts with a candidate set of simple syntactic structures and incrementally adds more complex predicates if the current set is not sufficient. To test if a set of predicates L is sufficient, we conjoin all predicates in L to form $l_L = \neg(\bigwedge_{p \in L} p)$, and check if l_L makes (10) valid. This is the PredSufficient procedure in Algorithm 1. In other words, if the following is UNSAT, it is adequate to construct a lemma:

$$((F_i(V) \wedge T(V, V')) \vee \mathit{Init}(V')) \wedge \neg l_L(V') \tag{13}$$

Pruning due to inadequate batch. If the batch is inadequate (i.e., formula (13) is satisfiable), we construct a formula $c := \bigwedge_i (V_i' = a_i)$, where a_i is the assignment to V_i' in the model of (13). Since c is constructed from the model,

$\forall p \in L, c \rightarrow p$. If there is a good set of predicates $L' \supset L$, then within L', there must be a predicate p_c such that $c \rightarrow \neg p_c$. Therefore, we can use the extracted formula c to guide the generation of new predicates. This constitutes additional pruning of the search space (Line 8 in Algorithm 1).

4.4 Generating Terms in the Grammar

As shown in Algorithm 1, our SyGuS method starts from an initial set of predicates (constructed from the initial set of terms) and incrementally generates new terms to create more predicates.

The Initial Set of Terms and Comparators. We extract the terms and predicates in the original problem from the syntax tree of the initial state predicate *Init*, the transition relation T, and the given property P), and select a subset to form the set of initial predicates. For a given Q (recall that Q encodes the model to block), all predicates containing only the variables in Q are added to the initial predicate set. While for the terms, only those that (a) contain only the variables in Q and (b) whose bit-width is less than a threshold H_p are added to the initial term set. We also expand the term set with additional constants when bit-widths are less than a threshold H_c. The rationale behind this strategy is that we want to find lemmas in the "control space" first, which is often beneficial as shown in previous work [42, 59]. Instead of relying on user input (as in [59]), we differentiate between the terms that are *likely* control- or data-related. *Specifically, we use a heuristic that terms with a larger bit-width are more likely to be data-related, and a specific constant for data-related terms would not be useful in the inductive invariants.* In our experiments, we empirically set thresholds $H_c = 4$ and $H_p = 8$, to balance between the expressiveness and the cost of extra predicates. For the bit-vector comparators in the predicates, initially we begin with only `Equal` and `NotEqual`.

Adding More Existing Terms and Comparators. If the predicates constructed from the initial terms and operators are not able to generate a lemma, in the next call to MorePred, we will add back all the existing terms that contain no variables outside Q. Comparators like `bvult` ("unsigned less than") and `bvule` ("unsigned less than or equal") will be added also if they exist in the original problem.

Generating New Terms. The above procedure adds terms and comparators that are already present in the original problem formulation. However, this set of terms is often insufficient. Therefore, we use a procedure for generating new terms based on the following three rules – Construct, Replace, and Bit-Blast.
Construct Rule. Assume that we already have a set of terms $\{t\}$ and operators $\{op\}$. For each operator op and vector of terms $\langle t_0, t_1, ... \rangle$, we will construct a new term `(op` t_0 t_1 `...)` if it is well-formed. In the examples we have seen in

the HWMCC benchmarks, the set of operators is usually not very large, which are mainly logic operators like `and`, arithmetic operators like `add`, some bit-field manipulations, and `ite`. *As a heuristic, we prioritize using bit-field extraction and arithmetic operators for wide terms, and will use only logical operators on the narrow ones (separated also by the bit-width threshold H_p).* Furthermore, we avoid having multiple terms that can be simplified to the same form, by using the rewriting capability of the underlying SMT solver implicitly. Some solvers, e.g., Boolector [49], simplify and rewrite the terms upon their creation. We create the terms in the solver, then retrieve the simplified form and compute a syntactic hash to detect duplicate terms. The construction process of this rule can be applied iteratively, where we use the newly generated terms to create more new terms. However, in our implementation, we restrict it to a single iteration per invocation, to avoid overwhelming the algorithm with too many predicates.

Replace Rule. This rule replaces a sub-term with another. Suppose there is an existing term t with the following form: `(op` t_0 t_1 `...)`. For each of the sub-terms (e.g., t_0), we will try to see if there is another term with the same sort that can replace it in t and result in a new term. Instead of trying all potential replacements, we look for replacement pairs using information from the transition relation as follows.

As stated in Algorithm 1, we invoke the MorePred procedure for new term production at times when the check (13) had returned SAT and we need more terms to form more predicates. At this point, we can evaluate the existing terms using the model of (13) and check which pairs of them have the same value. This is similar to running simulation to identify potential correlations of signals in a circuit. Terms are evaluated twice, once on the assignment to current state variables and once on the primed ones. In addition to finding correlation in the primed evaluation (where predicates become insufficient), we also detect correlation between the current state and the next state. For example, suppose that a term $t_1(V)$ evaluates to c_1 under M and another term $t_2(V')$ evaluates to c_2 under M. If $c_1 = c_2$, we will also identify t_1 and t_2 as a replacement pair. Here, the term t_1 being replaced is evaluated based on current state variables V and the term t_2 is evaluated as if it is on the next state variables V'. This difference on the current and next state variable set *allows us to find temporal correlations that are potential causes and effects.*

Bit-Blast Rule. For hardware model checking problems, it is also possible that no good word-level invariant exists or the desired word-level invariant cannot be generated from existing terms and operators. Therefore we keep this rule as a fallback option. When applied, it creates terms using the `extract` operator to extract every single bit of a state variable. We prioritize using it first on the state variables which have been used in other terms with an `extract` operator. The rationale for this heuristic is that if the original problem contains bit-fields that are extracted from signals derived from such a state variable, the state variable is likely to be more "bit-level" rather than "word-level." If we continuously apply this rule, eventually all bits from all state variables will be added to the term set.

At this point, the set is guaranteed to be sufficient, and the algorithm degenerates to finding bit-level invariants in the worst case.

In our implementation, we apply these rules in the following order. First, we continuously apply **Replace** until it generates no more new terms or the predicate set becomes sufficient. Then we try **Construct**, where we prioritize different operators for wide and narrow terms and terminate if the prioritized terms are already sufficient. This is followed by another round of **Replace**, and finally **Bit-Blast** with the prioritization on variables that are more "bit-level." For any rule, if the prioritized terms are already sufficient, the procedure returns. Similar to the sharing of predicates among lemmas blocking the same model, the term set is also shared when two models in the proof-obligation have the same set of variables.

Example. Consider an illustration for creation of new terms according to these three rules. For the example transition system in Section 3, suppose at some point, there is a proof obligation: block model $(a, b, c) = (6, 4, 1)$ at F_2. The initial term set is listed in Figure 4. For F_1 there are already lemmas added using the these terms on variables a, b, c, however, they cannot generate sufficient predicates to block the model (predicates after initial pruning are shown in the figure). A model for (13) can be extracted: $(a, b, c, e, a', b', c') = (3, 3, 0, \perp, 4, 3, 1)$. Readers can check the assignments to the primed variables in the model make all initial predicates evaluate to true. When applying the **Replace** rule, existing terms will be evaluated on variable sets $(a, b, c) = (3, 3, 0)$ and $(a', b', c') = (4, 3, 1)$ to identify potential correlations between pairs. In this example, we find 7 possible replacement pairs, which result in 4 new terms, shown in the bottom-left table in Figure 4, where replacement is based on the correlated value in the first column and the replacement pair $\langle t_1, t_2 \rangle$ means t_1 is replaced by t_2. If these were not sufficient, we could further apply the **Construct** rule, and more new terms would be generated using operator `bvadd` on the existing terms. Finally if these were still not sufficient, we would fall back to **Bit-Blast**.

For the specific proof obligation here, the above term generation process will actually stop after the **Replace** rule, where the predicates with term $b + c$ will be sufficient, and the invariant $a = b + c$ will be discovered. (For this proof obligation, the term $b + 1$ also works, which can produce predicate $a = b + 1$, but it is not as general, and will be dropped by the MUS procedure.) Note that the discovery of $a = b + c$ in this example is not simply by chance. The two replacements needed – a to b, and 1 to c – are found by our method that looks for correlation between terms, which leads to finding this invariant. Interestingly, these correspond to the two cases in the induction step in our human reasoning process in Section 3.

4.5 Tightening Previous Frames in IC3/PDR

As we discussed in Section 4.1, using constraint (12) instead of (10) allows a larger space of lemmas. But on the other hand, having l on both sides of the implication breaks the monotonicity of predicate minimization. For (10), after a predicate is removed, (10) could stay UNSAT or become SAT. But once it

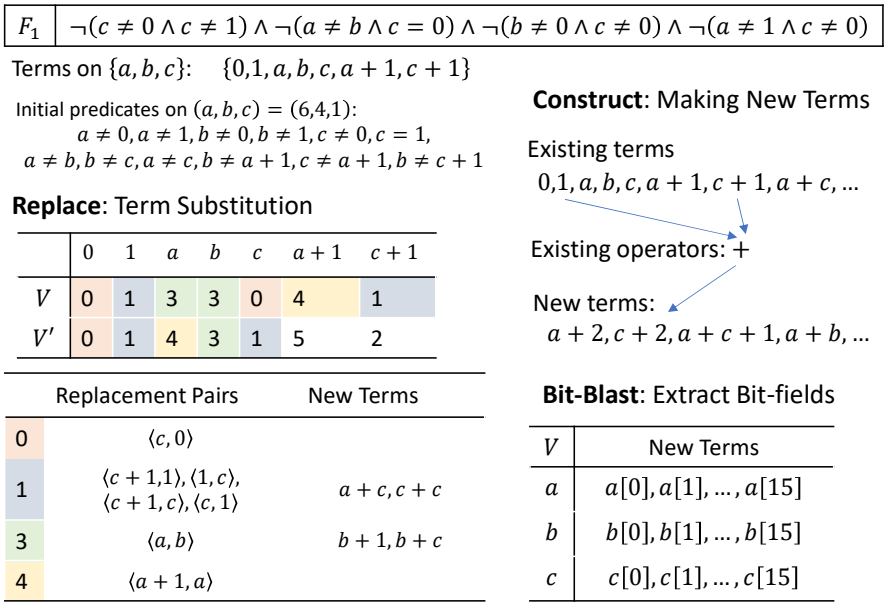


Fig. 4: An illustration of the three term generation rules.

becomes SAT, removing more predicates will not make (10) UNSAT. The same does not hold for (12), as removing predicates also shrinks the pre-image. This makes minimizing the set of predicates in (12) much harder. As a trade-off between allowing a larger space of lemmas and ease of minimizing the set of sufficient predicates, we choose the latter and decide to stick with constraint (10). To mitigate the associated problem—potentially missing a good lemma due to coarse frames, we add a procedure to tighten the previous frames, as described below.

We design a lazy approach to tighten previous frames (Line 7 in Algorithm 1). When the check (13) indicates that L is insufficient (i.e., we get a SAT result for (13)), instead of immediately generating more terms to construct more candidate predicates, we first check whether the current state variable assignment from the satisfiable model in (13) is blockable. This blocking operation will introduce new lemmas in the previous frame F_i and could potentially turn (10) into (12) by introducing the same lemma. This may then allow the predicates in L to be used. On the other hand, if the model cannot be blocked, it means no lemmas can be generated from the current predicate set, even using (12) instead of (10). So we will indeed need to construct more terms to enrich the set of predicates.

Instead of requiring any big change, the blocking operation suggested above can use an existing recursive blocking function utility available in an IC3/PDR framework (denoted as RecBlock in Algorithm 1). However, this blocking is different from blocking of models generated from (9), which *must* be blocked, otherwise P will fail. Thus, we need to distinguish between a "may-proof-obligation"

Algorithm 2: MUS(U): Minimizing the set U of UNSAT constraints

Input: U: $\{base\} \cup L$, a set of constraints
Output: U': a minimal UNSAT subset of U

```
1  while true do
2      U' ← UnsatCore(U) ;
3      if |U'| = |U| then
4          break;
5      U ← U' ;
6  Sort(U' − {base}) , by syntax complexity ;
7  for u ∈ (U' − {base}) do
8      if U' − {u} is UNSAT then
9          U' ← UnsatCore( U' − {u}) ;
10 return U';
```

and a "must-proof-obligation" for blocking a model. Note also that this distinction is not a special requirement of SyGuS-$\mathcal{A}$PDR. For example, in the existing IC3/PDR framework Quip [36], a "may-proof-obligation" arises due to failures of lemma pushing. Here, we simply reuse this facility to design our lazy frame tightening procedure.

4.6 Generalizing the Lemma by using UNSAT Cores

When constructing lemmas from a set of predicates, we would like to get a more general lemma using fewer predicates. This is done through minimal unsatisfiable subset (MUS) extraction from the unsatisfiable formula (13), where we treat each predicate $p_i \in L$ as an individual constraint, and the rest of the formula (*base*) as one constraint.

Our MUS procedure (shown in Algorithm 2) follows standard approaches, as it first computes a small UNSAT core by iteratively using UNSAT core extraction of the SMT solver until reaching a fixed-point of the core size [60] (Line 1-5). Then it further reduces the core size by trying to drop constraints. Here, we use a new heuristic based on the *syntax-complexity*, defined as the number of nodes in the syntax tree plus the occurrence of constants as an extra penalty. Our contraint-dropping is done iteratively in descending order of the syntax-complexity of constraints (Line 6-9). This allows us to get an MUS where the predicates have simpler syntactic structure and also fewer constants, which may generalize better in the overall algorithm.

4.7 Partial model generation for word-level reasoning in bit-vectors

We also propose to use partial model generation in the **Predecessor** procedure for *word-level* bit-vector reasoning in IC3/PDR. Our method can handle hardware model checking problems where the transition relation is functional. (We leave the general case of adapting it in model-based projection for bit-vectors to future work.)

Our implementation of partial model generation mimics the ternary simulation method used in the original PDR implementation [21], but at the word-level. For some bit-vector operators like `bvand` and `ite`, there is a masking effect. For example, consider a satisfiable SMT formula that contains a fragment `(bvand a b)`, if we know that variable `a` is assigned to all 0s in the model extracted from the query, then the assignment to `b` does not affect the evaluation of the fragment. If `b` does not appear elsewhere in the formula, we can remove the assignment to `b` and get a partial model while the formula still evaluates to `true` under the reduced set of assignments. Using partial model generation, we can derive a reduced set of variable assignments representing multiple bad states. This benefits the SyGuS-based interpolation because when we later generate lemma candidates to block it, we can limit the search space to candidates containing only those variables in the partial model.

5 Experimental Evaluations

We implemented the SyGuS-$\mathcal{A}$PDR methods on top of an $\mathcal{A}$PDR framework that we developed according to the algorithm presented in previous work [32]. We used the solver-agnostic interfacing library SMT-Switch [44], and used Boolector [49] for SMT queries and UNSAT core extraction.

5.1 Experiment Setup

Environment of the Experiments. The experiments were conducted on a cluster of machines with Xeon Gold 6142 CPUs running Springdale Linux 7.8, and each tool is allocated 8 cores and 64GB of memory. Similar to the HWMCC setting, we set the time-out limit to be one hour wall-clock time.

Benchmark Examples. We use the benchmarks from the bit-vector track of 2019's HWMCC. It has 317 test cases in the Btor2 format. We use our conversion tool `Btor2CHC` to convert them into CHCs.

Table 2: Number of Solved Instances

Solver		# Solved	Safe	Unsafe
Our work	SyGuS-$\mathcal{A}$PDR	126	**112**	14
CHC Solvers	BvItp	22	22	0
	Z3/Spacer	90	87	3
	Eldarica	4	4	0
HW Model Checkers	AVR	**157**	111	**46**
	CoSA2 (Pono)	137	96	41
	BtorMC	108	67	41
	CoNPS-btormc-THP	40	0	40

Tools for Comparison. We test our SyGuS-$\mathcal{A}$PDR tool against state-of-the-art CHC solvers on the HWMCC'19 benchmarks. For comparison, we also report the performance of word-level hardware model checkers that participated in HWMCC'19 (and were run with the same configuration). The tools we compared with are listed as follows.

- SyGuS-$\mathcal{A}$PDR is our tool that uses the syntax-guided lemma generation procedure described in Section 4.
- BvItp is a tool we constructed that uses the word-level interpolants [31] from MathSAT [16] out-of-the-box to generate lemmas in $\mathcal{A}$PDR.
- Spacer [32,40] is a state-of-the-art CHC solver and part of Z3 [19]. We test the newest release version 4.8.9, but it actually solves fewer instances (83 vs. 90) compared to an older version 4.8.7. We did not further investigate the reason of the performance degradation, but will report the results from 4.8.7.
- Eldarica is a CHC solver that makes use of counter-example-guided abstraction refinement (CEGAR) method. As Eldarica can use different interpolation abstraction templates, we start 4 parallel running engines each with a different template configuration and report the best result, using the latest release version 2.0.4.
- AVR (abstractly verifying reachability) is a collection of 11 parallel running engines including 3 variants of BMC and 8 variants of IC3 integrated with multiple abstraction techniques [27]. We use the binary release available from the Github tagged with `hwmcc19` for the experiment.
- CoSA2 (successor of CoSA [45], now named Pono) is a model checker based on the solver-agnostic framework SMT-Switch [44]. It runs four parallel engines: BMC, BMC simple-path, k-induction and the interpolation-based method [47]. We were unable to compile using the source code tagged with `hwmcc19`. Instead we use a development version with a commit hash `6d72613`. Our experiment results show it actually solves more instances than reported in HWMCC'19.
- BtorMC (version 3.2.0) is a tool based on the SMT solver Boolector [49], equipped with two engines: BMC and k-induction. In our experiment, we run two instances in parallel and record the shorter time.
- CoNPS-btormc-THP is from Norbert Manthey. It is a specially configured BtorMC using huge pages for mapping memory and is linked against a modified GlibC library. We obtained the tool from the author.

5.2 The Overall Result

We plot the wall-clock time vs. the number of solved instances in Figure 5. A table summarizing the number of solved instances is shown in Table 2. Our results on the hardware model checkers are mostly consistent with the results from the HWMCC'19 report [52], with minor difference which is probably due to difference in the machine configurations or the version of tools that we use.

Among the tools, AVR solves the most instances. Our SyGuS-$\mathcal{A}$PDR solves about the same number of safe instances as AVR, but fewer unsafe instances.

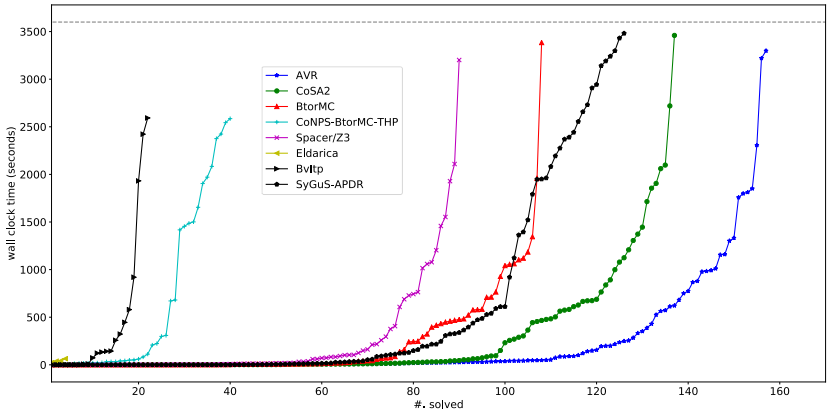
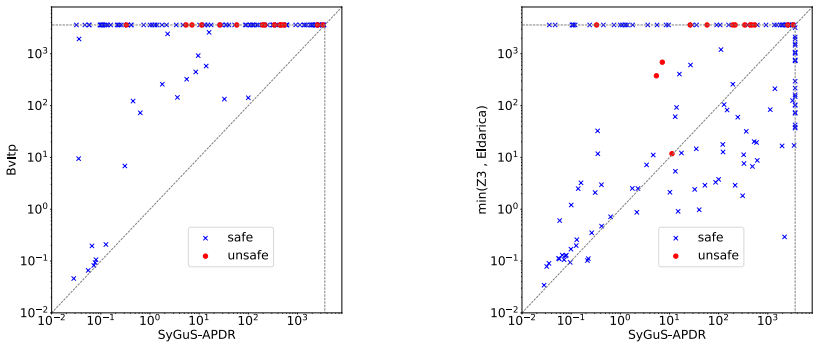


Fig. 5: Wall clock time vs. number of solved instances.

This is because in the unsafe case, even though a bad model leading to the violation of the property will become reachable at some point, in the first several frames, it is still blockable, and SyGuS-$\mathcal{A}$PDR will still try to construct lemmas to block it. We can make up for this disadvantage by having a BMC engine run in parallel with it, similar to what typical model checkers do. However, our focus is on lemma generation for proofs, so we leave this for future work.

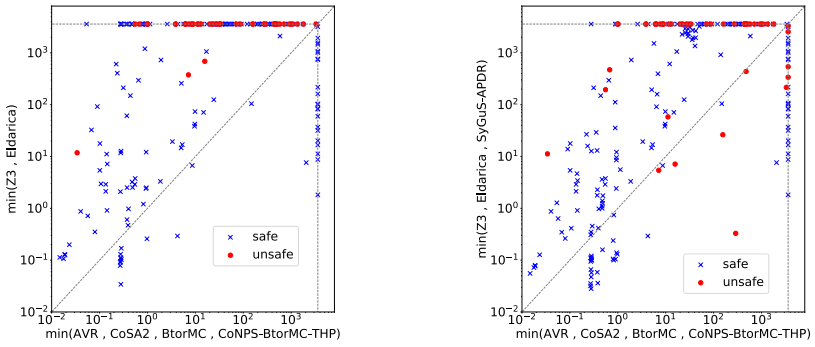
5.3 Effectiveness of SyGuS-$\mathcal{A}$PDR in Improving Lemmas

We plot the comparison between SyGuS-$\mathcal{A}$PDR and BvItp in Figure 6(a)—SyGuS-$\mathcal{A}$PDR shows a clear improvement over BvItp. In our experiments, we



(a) (b)

Fig. 6: Comparison of wall-clock time between SyGuS-$\mathcal{A}$PDR and (a) BvItp or (b) the faster time from Z3/Spacer or Eldarica.



(a) (b)

Fig. 7: Comparison of wall-clock time between word-level hardware model checkers and CHC solvers (a) without, or (b) with SyGuS-$\mathcal{A}$PDR.

found that the word-level bit-vector interpolants from MathSAT often contain conjunctions of a large number of equality relations in the form of $v = c$, where v is a state variable and c is a constant. This makes the interpolants very specific to the models, and they often trap the algorithm in the first few frames with hundreds or even thousands of lemmas. This explains why BvItp performs badly. And on the unsafe systems, it must reach a minimum bound to discover the shortest counterexample, therefore it is not able to find any unsafe instances. Removing some equalities in such an interpolant in BvItp, as we have attempted, often makes it no longer an interpolant. SyGuS-$\mathcal{A}$PDR, on the other hand, uses syntax-based guidance to steer the interpolants and can select simpler, and hopefully more general, predicates to mitigate such issues.

Figure 6(b) shows the comparison of SyGuS-$\mathcal{A}$PDR and the faster of either Z3/Spacer or Eldarica. Eldarica solves only 4 instances, two of which contain complex arithmetic operations in the transition relation and are solved by neither Z3/Spacer nor SyGuS-$\mathcal{A}$PDR. SyGuS-$\mathcal{A}$PDR solves 56 instances that are not solved by Z3/Spacer or Eldarica, and within the 70 instances that both categories solve, SyGuS-$\mathcal{A}$PDR runs faster on 36.

5.4 CHC Solvers vs. Hardware Model Checkers

We also compare the results from the two existing CHC solvers (referred to as the CHC group in the following text) with the collection of word-level model checkers that participated in HWMCC'19 (referred to as the HMC group). A comparison of solving time is shown in Figure 7. Although the CHC group solves fewer instances (92 vs. 196), there are 22 instances solved exclusively by CHC group. Among the 70 instances solved by both groups, the CHC group is faster on 16. This indicates that the CHC group has some complementary strengths that are worth further investigation.

For example, there is one test case `analog_estimation_convergence` where Spacer derives a safe inductive invariant in less than one second, whereas AVR does not converge within one hour. We took a closer look at the invariant produced by Z3/Spacer. It contains fragments with similar structure as a linear relation in LIA theory. This is likely an outcome of the translation technique, and it makes Z3/Spacer the fastest solver on this instance. When we also include SyGuS-$\mathcal{A}$PDR in the CHC group, the group now solves 148 (significant improvement from 92), where 27 are not solved by the HMC group. Among the 121 instances solved by both groups, CHC group is faster in 33 test cases.

Around the time of preparing the final version of the paper, two new tools become available: GSpacerbv [28] and a new version of AVR for HWMCC'20 competition (referred to as AVR-20). We conducted further experiments after paper submission, and for completeness, include a summary here. Detailed results can be found in [57]. GSpacerbv shows an improvement from Spacer thanks to its global guidance rules in lemma generation [41] and the model-based projection procedure for bit-vectors, yet it solves fewer instances than SyGuS-$\mathcal{A}$PDR (101 vs. 126). AVR-20 shows a great performance gain compared to its previous generation (249 vs. 157) as it doubles the portfolio size with more techniques integrated. Though, AVR-20 solves more safe instances than SyGuS-$\mathcal{A}$PDR (205 vs. 112), SyGuS-$\mathcal{A}$PDR runs faster on almost half of the instances it solves (50) and is there supplementary to the portfolio used in AVR-20.

6 Related Work

Enhancing the Interpolants. There are many existing works that aim to enhance the interpolants used in model checking. For example, Albarghouthi and McMillan [1] propose to reduce the number of disjuncts of linear inequality constraints to get simpler interpolants. Blicha *et al.* [10] propose to decompose the interpolants to mitigate the divergence problem. GSpacer [41] incorporates global guidance in the lemma. These works are mostly for interpolation in the infinite domain (e.g., LIA/LRA) theories.

In the bit-vector theory, there is no native word-level bit-vector interpolation strategy in the first place. Existing methods rely on EUF layering, translation to (non)-linear integer arithmetic, application of certain forms of quantifier elimination etc. [3, 31]. Additionally, compared to LIA/LRA, the bit-vector theory has a more diverse set of operators allowing bit-field manipulation as well as logical and arithmetic operations. This often introduces non-linear relations that are hard to translate to other theories.

In the LIA/LRA domain, the closest approach to ours is [43], which also uses templates to guide the generation of interpolants. It introduces interpolation abstractions in the SMT query but leaves the construction of interpolants completely to the solver, whereas SyGuS-$\mathcal{A}$PDR constructs the interpolants *outside* the solver, and therefore has more direct control on the generated lemma. Previous works [9, 14, 20] also construct interpolant outside the SAT/SMT solver, while SyGuS-$\mathcal{A}$PDR incorporates syntax guidance and further integrates it with

IC3/PDR framework to make use of models and procedures from deductive reasoning in IC3/PDR.

Word-Level IC3/PDR Algorithms for Bit-Vectors. Previous efforts on the word-level BV can be mainly categorized as: (1) adding an abstraction layer so that the core algorithm remains at the bit-level, (e.g., word-level abstraction [34], word-level predicate abstraction [38], IC3ia [15], data-path abstraction [42] and syntax-guided abstraction [26]) (2) using specific types of atomic reasoning units (ARUs) [55,56], or (3) translating the BV problem to another theory [33]. SyGuS-$\mathcal{A}$PDR differs from the existing works in that: (1) it does not need an explicit abstraction-refinement loop—the models in the proof obligations and the transition relation are all kept concrete and the interpretation of the predicates are always revealed to the solver; (2) the grammar allows lemmas that are in general more flexible compared to the ARUs; and (3) while translation is feasible for arithmetic and some related operations, it does not work for all the operators available in BV theory, especially bit manipulation operators. In comparison, SyGuS-$\mathcal{A}$PDR is native on the BV theory and supports all BV operators.

Syntax-Guided Inductive Invariant Synthesis. Syntax-guided synthesis has been applied on the inductive invariant synthesis problem before, e.g., LoopInvGen [50,51], cvc4sy [53], FreqHorn [23–25] and Grain [59]. A key feature of SyGuS-$\mathcal{A}$PDR is its tight integration with IC3/PDR framework, which allows use of *both deductive reasoning as well as grammars* to guide candidate lemma generation and prune the search space.

7 Conclusions and Future Work

In this work, we present our technique of using syntax-guided synthesis for lemma generation for unbounded hardware model checking. This is also an attempt to attack the challenges of BV interpolation with the help of a tighter integration with the IC3/PDR framework. Although our motivation for reasoning about problems in BV theory comes from hardware verification applications, the techniques we present may also benefit software verification, especially low-level software (e.g., device driver or firmware) where bit manipulation is essential.

To achieve better performance, our SyGuS-based lemma generation algorithm can be further integrated with other techniques, e.g., an abstraction refinement framework, or with other parallel running engines.

Acknowledgements. This work was supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA; by the DARPA POSH and DARPA SSITH programs; and by NSF Grant No. 1628926.

References

1. Albarghouthi, A., McMillan, K.L.: Beautiful interpolants. In: International Conference on Computer Aided Verification. pp. 313–329. Springer (2013)
2. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD. pp. 1–8 (2013)
3. Backeman, P., Rummer, P., Zeljic, A.: Bit-vector interpolation and quantifier elimination by lazy reduction. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–10 (2018)
4. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation. pp. 203–213 (2001)
5. 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)
6. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. Handbook of Satisfiability pp. 825–885 (2009)
7. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)
8. Bjørner, N., Gurfinkel, A.: Property directed polyhedral abstraction. In: VMCAI. pp. 263–281 (2015)
9. Bjørner, N., Gurfinkel, A., Korovin, K., Lahav, O.: Instantiations, zippers and epr interpolation. In: LPAR. pp. 35–41 (2013)
10. Blicha, M., Hyvärinen, A.E., Kofroň, J., Sharygina, N.: Decomposing Farkas interpolants. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 3–20. Springer (2019)
11. Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI. pp. 70–87 (2011)
12. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: Formal Methods in Computer Aided Design (FMCAD'07). pp. 173–180. IEEE (2007)
13. Champion, A., Kobayashi, N., Sato, R.: HoIce: An ICE-based non-linear horn clause solver. In: APLAS (2018)
14. Chockler, H., Ivrii, A., Matsliah, A.: Computing interpolants without proofs. In: Haifa verification conference. pp. 72–85. Springer (2012)
15. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 46–61. Springer (2014)
16. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S. (eds.) Proceedings of TACAS. LNCS, vol. 7795. Springer (2013)
17. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM) **50**(5), 752–794 (2003)
18. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT press (1999)
19. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340 (2008)

20. Drews, S., Albarghouthi, A.: Effectively propositional interpolants. In: International Conference on Computer Aided Verification. pp. 210–229. Springer (2016)
21. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD. pp. 125–134 (2011)
22. Farkas, J.: Theorie der einfachen ungleichungen. Journal für die reine und angewandte Mathematik **1902**(124), 1–27 (1902)
23. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: TACAS. pp. 251–269 (2018)
24. Fedyukovich, G., Kaufman, S., Bodík, R.: Sampling invariants from frequency distributions. In: FMCAD. pp. 100–107 (2017)
25. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained horn clauses using syntax and data. In: FMCAD. pp. 170–178 (2018)
26. Goel, A., Sakallah, K.: Model checking of Verilog RTL using IC3 with syntax-guided abstraction. In: NASA Formal Methods Symposium. pp. 166–185. Springer (2019)
27. Goel, A., Sakallah, K.: Avr: Abstractly verifying reachability. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 413–422. Springer International Publishing, Cham (2020)
28. Govind, H., Fedyukovich, G., Gurfinkel, A.: Word level property directed reachability. In: International Conference on Computer Aided Design (2020)
29. Graf, S., Saïdi, H.: Construction of abstract state graphs with pvs. In: International Conference on Computer Aided Verification. pp. 72–83. Springer (1997)
30. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI. pp. 405–416. ACM (2012)
31. Griggio, A.: Effective word-level interpolation for software verification. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design. p. 28–36. FMCAD '11, FMCAD Inc, Austin, Texas (2011)
32. Gurfinkel, A.: IC3, PDR, and friends. Summer School on Formal Techniques (2015)
33. Gurfinkel, A., Belov, A., Marques-Silva, J.: Synthesizing safe bit-precise invariants. In: TACAS. LNCS, vol. 8413, pp. 93–108. SV (2014)
34. Ho, Y.S., Mishchenko, A., Brayton, R.: Property directed reachability with word-level abstraction. In: FMCAD. pp. 132–139 (2017)
35. Hojjat, H., Rümmer, P.: The ELDARICA Horn Solver. In: FMCAD. pp. 158–164. IEEE (2018)
36. Ivrii, A., Gurfinkel, A.: Pushing to the top. In: 2015 Formal Methods in Computer-Aided Design (FMCAD). pp. 65–72 (2015)
37. Ivrii, A., Gurfinkel, A., Belov, A.: Small inductive safe invariants. In: 2014 Formal Methods in Computer-Aided Design (FMCAD). pp. 115–122. IEEE (2014)
38. Jain, H., Kroening, D., Sharygina, N., Clarke, E.: Word level predicate abstraction and refinement for verifying RTL Verilog. In: Proceedings of the 42nd annual Design Automation Conference. pp. 445–450 (2005)
39. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. FMSD **48**(3), 175–205 (2016)
40. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: CAV. pp. 846–862 (2013)
41. Krishnan, H.G.V., Chen, Y., Shoham, S., Gurfinkel, A.: Global guidance for local generalization in model checking. In: International Conference on Computer Aided Verification. pp. 101–125. Springer (2020)
42. Lee, S., Sakallah, K.A.: Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In: CAV. pp. 849–865. Cham (2014)

43. Leroux, J., Rümmer, P., Subotić, P.: Guiding Craig interpolation with domain-specific abstractions. Acta Informatica **53**(4), 387–424 (2016)
44. Mann, M., Wilson, A., Tinelli, C., Barrett, C.: SMT-Switch: a solver-agnostic C++ api for smt solving. arXiv preprint arXiv:2007.01374 (2020)
45. Mattarei, C., Mann, M., Barrett, C., Daly, R.G., Huff, D., Hanrahan, P.: CoSA: Integrated verification for agile hardware design. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–5. IEEE (2018)
46. McMillan, K.: Applications of Craig interpolation to model checking. In: International Workshop on Computer Science Logic. pp. 22–23. Springer (2004)
47. McMillan, K.L.: Interpolation and SAT-based model checking. In: International Conference on Computer Aided Verification. pp. 1–13. Springer (2003)
48. McMillan, K.L., Rybalchenko, A.: Solving constrained horn clauses using interpolation (2013)
49. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 587–595. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32, https://doi.org/10.1007/978-3-319-96145-3_32
50. Padhi, S., Millstein, T., Nori, A., Sharma, R.: Overfitting in synthesis: Theory and practice. In: International Conference on Computer Aided Verification. pp. 315–334. Springer (2019)
51. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: PLDI. pp. 42–56. ACM (2016)
52. Preiner, M., Biere, A.: Hardware model checking competition 2019. http://fmv.jku.at/hwmcc19/, accessed: 2020-09-10
53. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: CAV. pp. 74–83 (2019)
54. Si, X., Naik, A., Dai, H., Naik, M., Song, L.: Code2inv: A deep learning framework for program verification. In: International Conference on Computer Aided Verification. pp. 151–164. Springer (2020)
55. Welp, T., Kuehlmann, A.: QF_BV model checking with property directed reachability. In: 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 791–796. IEEE (2013)
56. Welp, T., Kuehlmann, A.: Property directed reachability for QF_BV with mixed type atomic reasoning units. In: Asia and South Pacific Design Automation Conference. pp. 738–743. IEEE (2014)
57. Zhang, H.: Figures for additional experiment results. https://github.com/zhanghongce/HWMCC19-in-CHC/blob/logs/figs/compare.md, accessed: 2020-11-14
58. Zhang, H.: HWMCC19 benchmark in constrained horn clauses. https://github.com/zhanghongce/HWMCC19-in-CHC, accessed: 2020-10-08
59. Zhang, H., Yang, W., Fedyukovich, G., Gupta, A., Malik, S.: Synthesizing environment invariants for modular hardware verification. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 202–225. Springer (2020)
60. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable Boolean formulas. In: International Conference on Theory and Applications of Satisfiability Testing (SAT) (2003)

61. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 707–721. PLDI 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3192366.3192416, https://doi.org/10.1145/3192366.3192416