Synthesizing Environment Invariants for Modular Hardware Verification

Hongce Zhang[1], Weikun Yang[1], Grigory Fedyukovich[0000−0003−1727−4043][2], Aarti Gupta[1], and Sharad Malik[1]

[1] Princeton University, Princeton NJ 08544, USA
{hongcez,weikuny,aartig,sharad}@princeton.edu
[2] Florida State University, Tallahassee FL 32306, USA
grigory@cs.fsu.edu

Abstract. We automate synthesis of environment invariants for modular hardware verification in processors and application-specific accelerators, where functional equivalence is proved between a high-level specification and a low-level implementation. Invariants are generated and iteratively strengthened by reachability queries in a counterexample-guided abstraction refinement (CEGAR) loop. Within each iteration, we use a syntax-guided synthesis (SyGuS) technique for generating invariants, where we use novel grammars to capture high-level design insights and provide guidance in the search over candidate invariants. Our grammars explicitly capture the separation between control-related and data-related state variables in hardware designs to improve scalability of the enumerative search. We have implemented our SyGuS-based technique on top of an existing Constrained Horn Clause (CHC) solver and have developed a framework for hardware functional equivalence checking that can leverage other available tools and techniques for invariant generation. Our experiments show that our proposed SyGuS-based technique complements or outperforms existing property-directed reachability (PDR) techniques for invariant generation on practical hardware designs, including an AES block encryption accelerator, a Gaussian-Blur image processing accelerator and the PicoRV32 processor.

1 Introduction

This paper addresses hardware verification of processing cores in modern complex Systems-on-Chip (SoCs). These comprise general purpose processors and also application-specific hardware accelerators. Despite advances in automated verification, scalability with increasing design complexity remains elusive. For general-purpose processors, where the instruction set architecture (ISA) serves as a specification, a natural approach is to take advantage of the modular per-instruction specification and perform equivalence checking against a microarchitectural implementation on a per-instruction basis [9, 36, 41, 44]. However, increasingly, domain-specific hardware accelerators are being used in SoCs to meet power-performance requirements. Traditionally these accelerators do not

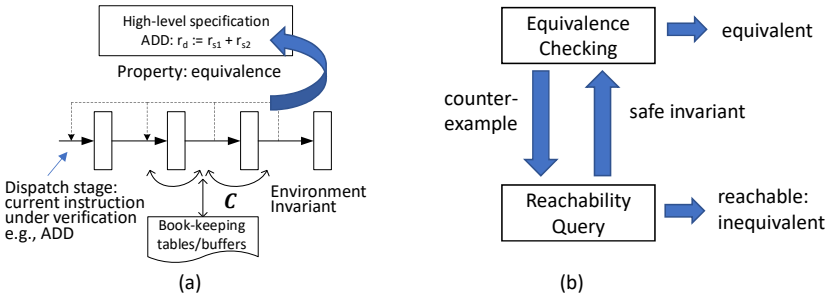

Fig. 1. (a) An example of environment invariants when verifying an `ADD` instruction in a pipelined processor. (b) Counterexample-guided environment invariant synthesis.

have an ISA or a high-level specification. Recent work has addressed this gap and proposed a generalization of the ISA referred to as an instruction-level abstraction (ILA) [33, 54, 55]. Similar to the ISA, an ILA provides a high-level modular specification that can be used for modular verification of accelerator implementations. Thus, per-instruction modular verification is applicable on general-purpose processors using ISA [37, 49, 50], as well as on accelerators using ILA [33].

Although per-instruction equivalence checking[3] helps improve scalability due to modularity, it has its own challenges. Each sub-task in verification checks whether a well-founded equivalence bisimulation (WEB) relation [41] holds between an ISA/ILA model and a low-level model (e.g., the register-transfer-level, or RTL implementation) when the same instruction is executed. The correspondence between states in the two models is specified by a refinement map, typically provided by the user. However, in each check, the given instruction starts to execute from an *arbitrary state* that is left by some previous (sequence of) instructions. For example, when a specified `ADD` instruction is in the dispatch stage in a processor, as shown in Figure 1(a), the state of the other pipeline stages (and other microarchitecture variables) constitutes the *environment*. If this environment is not in some consistent or reachable state, the equivalence check on the instruction may generate (spurious) counterexamples even when an implementation is correct. Thus, as in any modular verification method, one needs to model the environment adequately for the per-instruction equivalence checks to be successful.

Past efforts in processor verification have used a *flushing* abstraction [9] as a workaround to this problem. However, in general, modeling the environment constraints usually requires manual work [37, 41, 50]. Furthermore, a flushing abstraction is not readily available, and may not even be applicable, in the context of accelerator cores. Prior work on ILA-based verification of accelerators [33] also used manually-constructed environment invariants (after automatically checking their validity).

[3] Hereafter, we will use "equivalence checking" to refer to instruction-level functional equivalence checking.

1.1 Automatic Discovery of Environment Invariants

Our goal is to *automate* the process of discovering adequate environment abstractions for instruction-based equivalence checking. This would significantly reduce the human burden in applying verification (in other settings as well, described later in Section 6). One approach is to view this problem as relational program verification, and to automatically derive both environment and relational invariants (described in Section 2). We tried this approach and found that existing tools (e.g., Spacer [25, 38], FreqHorn [19, 20]) fail to solve these problems, likely due to large sizes of the hardware models and bit-precise reasoning required for equivalence checking (Section 5).

Instead, we adopt a counterexample-guided abstraction refinement (CEGAR) approach [12], where the environment is refined iteratively by blocking spurious counterexamples that are found during equivalence checking. Since counterexamples are often due to inconsistent (unreachable) states in the low-level implementations, we pose a reachability query to check whether the starting state of the counterexample is reachable in the implementation. If it is unreachable, we add invariants[4] generated during the reachability query, to provide generalizations that can potentially block a larger set of unreachable states.

Our top-level method using an equivalence checker and a reachability query engine is shown in Figure 1(b). While CEGAR-based approaches for refining environments have been used in other verification settings (e.g., in angelic [14] or depth-bounded [34] program verification, and also in hardware verification [40]), these have not been targeted at per-instruction equivalence checking in processors and accelerators, or customized for this purpose.

For invariant generation within each reachability query, we explored several existing techniques including Property Directed Reachability (PDR) [15], originally proposed by Bradley as IC3 [6]. PDR has been used successfully with Constrained Horn Clause (CHC) solvers on programs [25, 27], and with bit-level and word-level abstraction techniques in ABC [7, 28, 45] on hardware designs. Interestingly, we found in our experiments (Section 5) that accelerators are more challenging than processors for existing PDR-based tools. We conjecture this is due to two reasons: (1) accelerators tend to have wide bit-vectors (e.g., 128-bits), and word-level operations on wide bit-vectors are not handled well at the bit-level; and (2) control flow in accelerators is often more complex and software-like, in comparison to processors. These reasons make it harder for bit-level PDR and CHC-solvers (that support bit-vectors by bit-blasting) to converge with CEGAR.

1.2 SyGuS-based Invariant Generation

To overcome these additional challenges in our setting, we adopt syntax-guided synthesis (SyGuS) [1] for invariant generation. SyGuS has been applied very

[4] Our tool implementation can utilize general constraints in an environment abstraction, not necessarily invariants; however, we focus on invariant generation in this paper – hence we will use abstractions/constraints/invariants interchangeably when discussing the environment hereafter.

successfully in many applications, e.g., invariant generation in programs [22, 47] and program synthesis [2, 53]. In our method, candidates for invariants are generated by an *enumerative search* over a space of formulas restricted by a *grammar* (similar to prior work [20, 21]). When the grammar covers a small space of expressive formulas, then candidates can be enumerated efficiently and checked for invariance and safety using an off-the-shelf SMT solver.

The main novelty in our SyGuS-based method is the grammar used for generating candidates, and the filtering and prioritizing heuristics to prune the search space of candidates. Specifically, our grammar exploits the separation between data-related and control-related state variables that naturally exists in hardware designs for processing cores. Such "control-or-data" difference often affects how variables appear in invariants. For example, concrete values of data-related variables appear less frequently in environment invariants, whereas concrete values of control-related variables are more significant. Our SyGuS-based method generates small candidates (in term of formula size) and iteratively strengthens the learned invariant with relatively inductive candidates (those becoming inductive after assuming the learned candidates) until it is safe for a given query. This shows better scalability in comparison to searching for a single monolithic candidate that satisfies all the constraints, which is often used in a generic SyGuS procedure (e.g., in CVC4SY [3, 51]).

We have implemented our SyGuS-based method in a tool called GRAIN (**Gra**mmar-based **in**variant generator), developed on top of an existing CHC solver [20, 21]. To the best of our knowledge, this is the first SyGuS-based tool for synthesizing invariants on large hardware designs. We also provide a detailed experimental comparison with existing PDR-based and SyGuS-based tools for invariant generation. Our results show that GRAIN often complements or outperforms existing tools on practical hardware designs. Our overall approach is especially beneficial in enabling *automated* modular verification for accelerators, such as the AES block encryption accelerator and the Gaussian-Blur image processing accelerator reported in this paper.

In summary, the contributions of this paper are:

- We *automate* the generation of environment invariants for modular hardware equivalence checking to reduce human effort in relation to prior work. Our CEGAR-based approach leverages available techniques and tools for invariant generation.
- We propose a syntax-guided method for synthesizing environment invariants, with a novel grammar that leverages insights about hardware designs and uses pruning techniques to reduce the search space.
- We implement our SyGuS-based method as a prototype tool GRAIN on top of an existing CHC solver, and demonstrate its usefulness on a range of hardware designs that include accelerators and processors. To the best of our knowledge, this is the first SyGuS-based tool that has been applied for invariant generation on large hardware designs.
- We provide a detailed experimental comparison against existing PDR-based and SyGuS-based tools for generating invariants. This has identified their

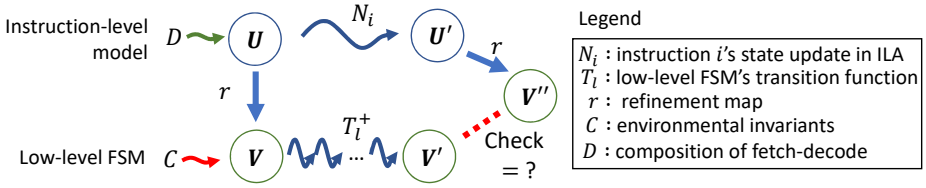

Fig. 2. The verification task in instruction-level modular verification.

key weaknesses in our application setting – inadequate handling of word-level operations (in bit-level PDR techniques), and poor scalability in the enumeration of complex candidates on large hardware designs (in existing SyGuS-based techniques).

We start by introducing instruction-level equivalence checking, and then present our top-level CEGAR-based method. In Section 4, we present our novel grammar and SyGuS-based method for generating invariants. We describe our tool GRAIN and present detailed evaluation results in Section 5, followed by a discussion of related work and conclusions.

2 Background and Preliminaries

2.1 Instruction-level Modular Verification

We consider checking equivalence of a low-level implementation, e.g., an RTL design in Verilog, against a formal instruction-level (ILA) specification [33]. The implementation is represented as a finite state machine (FSM): $\langle V, T_l, Init_l \rangle$, where V is a set of states, T_l is a transition relation representing the next-state function, and $Init_l$ is an initial state.

The ILA specification has program-visible (architectural) state variables S and input variables W (on its interface). Instructions to update S are modeled using a standard *fetch-decode-execution* style. In the following, $bvec_w$ denotes a bit-vector of width w, and $\mathbb{B} = \{true, false\}$. To simplify the presentation, we omit W in formulas hereafter, since inputs can be treated as free state variables.

- *fetch* function $F : S \to bvec_w$, maps states to an instruction word,
- *decode* predicate $\delta_i : bvec_w \to \mathbb{B}$, identifies if an instruction word corresponds to instruction i,
- *state update* function $N_i : S \to S$, specifies the effect of executing instruction i on the state.

We use predicate $D_i(\cdot) \stackrel{\text{def}}{=} \delta_i(F(\cdot))$ to denote the composition of *fetch* and *decode*. Thus, an instruction i is triggered only when D_i evaluates to true. (When clear from the context, we drop the subscript i.)

The equivalence verification task is shown in Fig. 2: an instruction i in the ILA model updates the architectural state from U to U', and correspondingly

the implementation transitions from state V to V'. A *refinement map* [41] r (could be one-to-many) maps the ILA states to the implementation states.

Informally, the verification task checks that the diagram commutes, i.e., starting from states that are matched by r, and updating both models by executing instruction i, the ending states U' and V' should also be matched by r.

The following details are needed to describe the starting and ending states in the models.

- We use predicate $D_i(U)$ to ensure that only one ILA instruction i executes.
- Although an ILA updates the state in a single transition N_i, the implementation could use multiple transitions T_l^+ to perform the same operation. Therefore, we use a *completion* predicate $E(V)$ (provided by a user) on the implementation state V to denote its ending state. (It is a common design practice to have an instruction commit signal in the low-level FSM).
- Predicate $C(V)$ on the starting state of the implementation represents the environment invariant that we seek to discover. Without such invariants, the implementation is free to start from inconsistent or unreachable states, those that no past instructions can reach. This often results in spurious counterexamples-to-equivalence, even when an implementation is correct.

More formally, the verification task for each instruction is the following:

Definition 1. *The two transition systems ILA and FSM start from arbitrary states U, V respectively, where $r(U,V) \wedge D_i(U) \wedge C(V)$ holds. After applying an instruction i, if their ending states U',V', (defined as $U' = N_i(U)$, $V' = T_l^+(V) \wedge E(V')$) are related by r, then they are equivalent for instruction i.*

To use off-the-shelf property verification tools, we can rephrase the task using a *product* transition system $\langle U \times V, T_p, Init_p \rangle$, where: $Init_p(U,V) = r(U,V) \wedge D(U) \wedge C(V)$, and $T_p\left((U,V),(U',V')\right) = T_h(U,U') \wedge T_l(V,V')$. Here, T_l is the transition relation of the low-level FSM, and T_h is a stuttering version of the ILA transition relation N whose first transition corresponds to state update of instruction i, after which the state remains unchanged. The equivalence check is represented as a property $\phi(U',V') \stackrel{\text{def}}{=} E(V') \implies r(U',V')$.

2.2 Checking Equivalence via Relational Program Verification

The ILA vs. FSM equivalence checking problem can be solved using techniques for relational program verification, where relational invariants are automatically derived. In the product state transition system, there are two invariants to find—the environment invariant $C(V)$, and an invariant $I(U, V)$ that can prove equivalence. Using the notations defined in the previous section, the equivalence checking problem can be formulated using constrained horn clauses (CHCs)[5]:

$$Init_l(V) \implies C(V) \tag{1}$$

$$C(V) \wedge T_l(V, V') \implies C(V') \tag{2}$$

[5] All CHC rules are considered to be universally quantified over the variables.

$$r(U,V) \wedge D(U) \wedge C(V) \implies I\,(U,\,V) \tag{3}$$
$$I\,(U,\,V) \wedge T_p\,(U \cup V,\, U' \cup V') \implies I\,(U',\,V') \tag{4}$$
$$I\,(U,\,V) \wedge \neg\phi\,(U,\,V) \implies \bot \tag{5}$$

The first two Horn rules define C to be closed in the transition relation of the low-level FSM. C is then used as an environment invariant in (3) to constrain arbitrary starting states in the product FSM to avoid infeasible states. Another relational invariant $I(U, V)$ is needed to prove safety with respect to the equivalence property ϕ.

The above formulation allows the use of existing CHC tools (e.g., Spacer [25, 38]) for simultaneously finding environment invariants C and checking equivalence property ϕ. However, this monolithic approach shows poor scalability as the CHC instances grow in size, as shown in our experiments (Section 5). This motivates our CEGAR-based approach for finding environment invariants, described in the next section.

3 Counterexample-Guided Invariant Synthesis

To improve scalability of the overall procedure, we propose finding environment invariants iteratively by using counterexample-guided abstraction refinement (CEGAR) [12]. Our CEGAR-based method to discover an environment invariant C (in the form of a conjunction of multiple invariants) is presented in Algorithm 1.

Algorithm 1: EqCheck-Inv-Syn(i, FSM, r): Equivalence Checking with Counterexample-Guided Abstraction Refinement for the Environment

Input: i: an instruction in ILA with its associated predicate and function, FSM: the low-level model, r: the refinement map.
Output: C: the environmental invariant; $res \in$ {Equivalent, Not-Equivalent}.

```
1 C ← ⊤;
2 while true do
3     cex ← EqCheck(i, FSM, r, C);
4     if cex = ∅ then return Equivalent, C;
5     V_start ← GetAssignment(cex);
6     result, Inv ← Reachability(FSM, V_start);
7     if result = reachable then return Not-Equivalent;
8     C ← C ∧ Inv;
```

It starts by initializing C to $\top$ (line 1). Then it iteratively checks equivalence (line 3) of the ILA and the low-level FSM (where they start from states that satisfy $r(U,V) \wedge D(U) \wedge C(V)$, as described earlier). If these models are not equivalent, a counterexample cex is returned, and the environment abstraction needs to be refined. From the counterexample trace cex, an assignment to the variables in the starting state V_{start} of the FSM is extracted (line 5), and the algorithm checks whether V_{start} is reachable in the FSM (line 6).

If the state V_{start} is unreachable, i.e., the safety property holds, then a formula that blocks V_{start} could be used to refine the environment invariant C. However, blocking each such counterexample individually could be expensive, and require many iterations for the algorithm to converge. Instead, our algorithm discovers a *safe inductive invariant* Inv as a proof of unreachability of V_{start} (in the REACHABILITY procedure on line 6).

Formally, for an FSM $\langle V, T_l, Init_l \rangle$ and a set of error states Bad, a safe inductive invariant is defined as a formula Inv such that the followings are valid:

$$Init_l(V) \implies Inv(V) \tag{6}$$

$$Inv(V) \wedge T_l(V, V') \implies Inv(V') \tag{7}$$

$$Inv(V) \wedge Bad(V) \implies \bot \tag{8}$$

In our case, $Bad \stackrel{\text{def}}{=} (V = V_{start})$. Thus, when $Bad(V)$ is unreachable, an invariant Inv is a strengthened constraint from $V \neq V_{start}$ (because $Inv(V) \implies V \neq V_{start}$ from (8)) This potentially blocks additional unreachable states, thereby requiring fewer iterations of the CEGAR loop to converge.

Note that the CEGAR approach decouples equivalence checking from environment invariant synthesis. This allows freely applying other tools and techniques in equivalence checking. In case the CEGAR-loop does not terminate due to time or resource limits, one can still get some useful invariants from the iterations that have completed.

Furthermore, we can leverage any existing technique or tool to discover safe inductive invariants during the reachability query. As we show in our detailed evaluations (Section 5), many CHC-based and SyGuS-based tools can be applied here. We found that existing reachability solvers based on bit-blasting tend to perform poorly on accelerators that require word-level reasoning on wide bit-vectors. In the next section, we present our novel SyGuS-based method for *word-level* invariant synthesis that is designed to overcome these limitations.

4 SyGuS for Word-level Invariant Synthesis

While SyGuS-based techniques have been successfully applied to synthesize loop invariants in software programs [22, 47], to the best of our knowledge, they have not been applied to generate invariants in large hardware designs before. In general, to use grammar-based enumeration of invariant candidates, one has to balance between the expressiveness of a grammar (to find adequate invariants) and the size of the related search space (for achieving tractability in practice). In addition, one needs to use pruning where possible during enumeration, to quickly eliminate non-promising candidates. In this section, we describe the design of our grammar, pruning techniques, and enumeration-based method targeted toward synthesis of word-level environment invariants for RTL hardware designs.

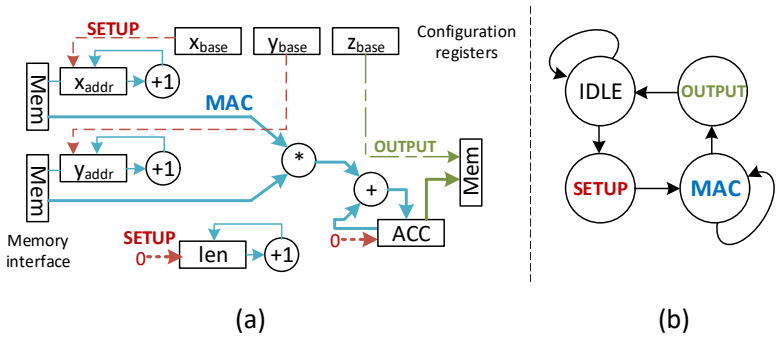

Fig. 3. Example of a HLSM model for a vector dot-product accelerator: (a) datapath, (b) control FSM. Paths in (a) are activated according to the state in (b).

4.1 Designing a Grammar for Environment Invariant Synthesis

One common design pattern in RTL processing cores is a separation between control and data. In particular, hardware accelerators often implement some high-level algorithm. It is typical to use a high-level state machine (HLSM) model [39], which is comprised of two interacting parts: a control finite state machine (FSM) and the datapath. The control FSM often tracks status signals (predicates) from the datapath, and triggers various datapath operations depending on the control state.

Example 1. Figure 3 shows a simplified view of an HLSM model for a vector dot-product accelerator that computes $\boldsymbol{z} = \boldsymbol{x} \cdot \boldsymbol{y}$. The datapath is shown on the left and the control FSM on the right. The input vectors are fetched from starting addresses in the configuration registers $\mathtt{x}_{base}$ and $\mathtt{y}_{base}$, and the dot-product result is stored in the address pointed by $\mathtt{z}_{base}$. The datapath may perform: (a) a multiplication-accumulation (MAC) operation, (b) set up the counters and the accumulator, (c) send the output, or (d) do nothing, when the control FSM is in `MAC`, `SETUP`, `OUTPUT`, or `IDLE` state, respectively. In the `SETUP` state, address counters are initialized to the base address, and length counter will be set to

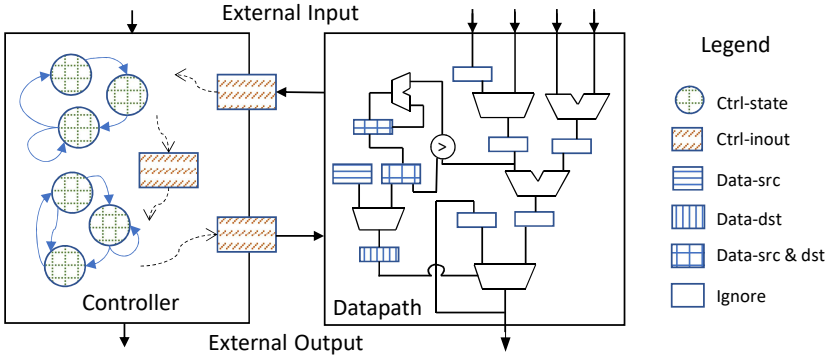

Fig. 4. Tags for state variables in the control FSM and datapath of a processing unit.

Table 1. Tags of Variables, Invariants, and Grammar for Example 1

Tags	ctrl-state	`state`
	ctrl-inout	`<none>`
	data-src	$\texttt{x}_{base}$, $\texttt{y}_{base}$, $\texttt{z}_{base}$, `len`
	data-dst	$\texttt{x}_{addr}$, $\texttt{y}_{addr}$, `len`
	ignore	`ACC`
Invariants (automatically generated)		$\texttt{state} \neq IDLE \implies \texttt{x}_{addr} = \texttt{x}_{base} + \texttt{len}$
		$\texttt{state} \neq IDLE \implies \texttt{y}_{addr} = \texttt{y}_{base} + \texttt{len}$
Grammar		$\langle \texttt{Cand} \rangle ::= \langle \texttt{CSpred} \rangle \implies \langle \texttt{Dpred} \rangle$
		$\langle \texttt{CSpred} \rangle ::= \langle \texttt{CSvar} \rangle = \langle \texttt{Const}_C \rangle \mid \langle \texttt{CSvar} \rangle \neq \langle \texttt{Const}_C \rangle$
		$\langle \texttt{Dpred} \rangle ::= \langle \texttt{DDvar} \rangle = \langle \texttt{DSvar} \rangle + \langle \texttt{DSvar} \rangle$
		$\langle \texttt{Const}_C \rangle ::= 00 \mid 01 \mid 10 \mid 11$

zero. When the control FSM is in the `MAC` state, the address counter $\texttt{x}_{addr}$ will be incremented, and so will the vector length counter `len`. But their increments are the same, namely $\texttt{x}_{addr} = \texttt{x}_{base} + \texttt{len}$. Note this relation holds in all control states except `IDLE`. When the control FSM is in the `IDLE` state, there is no update of $\texttt{x}_{addr}$, but the $\texttt{x}_{base}$ register can be programmed to an arbitrary address. Similar relations can be found among $\texttt{y}_{addr}$, $\texttt{y}_{base}$ and `len`. We aim to find such relations in certain control states as environment invariants.

Our grammar builds on top of *tags* that are assigned to all state variables in an RTL design description (e.g., registers in Verilog). These tags are based on their role, shown pictorially in a generic HLSM model in Figure 4.

- `ctrl-state` (CS): state of a control FSM (there could be multiple FSMs)
- `ctrl-inout` (CIO): signals between control FSM and datapath, or between control FSMs
- `data-src` (DS): source in datapath operation
- `data-dst` (DD): destination in datapath operation
- `ignore`: none of the above, typically internal/temporary state variables.

In our experiments, we currently tag all variables manually, based on our knowledge of the designs (details described in Section 5.3). It does not seem too difficult to implement a simple analysis (over Verilog or an intermediate representation) for tagging variables automatically.

The main idea is to construct a grammar for formula expressions where these tags are used as "types" of the variables, to allow limited operators over certain types and to restrict the sets of variables during enumeration of expressions. In particular, state variables with the tag "ignore" are not considered in expressions at all. (Some variables could have multiple tags, e.g., DS and DD.) Incorrect tags may result in either an unnecessary enumeration overhead (e.g., DS tagged as CS) or missing candidates (e.g., variables incorrectly tagged as "ignore"). As a preview of the full grammar (described in detail in the next section), the tags, invariants, and relevant grammar snippets for Example 1 are shown in Table 1.

4.2 SyGuS Grammar

Our full grammar for generating invariant candidates is shown in Figure 5. The terminals of this grammar represent tagged variables, where `CSvar` represents a `ctrl-state` variable, `CIOvar` represents a `ctrl-inout` variable, and `DSvar` and `DDvar` represent a source and destination (`data-src`/`-dst`) in the datapath, respectively. (Recall that a variable can have multiple tags.)

$$
\begin{aligned}
\langle \texttt{Cand} \rangle &::= \langle \texttt{Ante} \rangle \implies \langle \texttt{Conseq} \rangle \\
\langle \texttt{Ante} \rangle &::= \langle \texttt{CSpred} \rangle \wedge \langle \texttt{Ante} \rangle \mid \mathit{true} \\
\langle \texttt{CSpred} \rangle &::= \langle \texttt{CSvar} \rangle = \langle \texttt{Const}_C \rangle \mid \langle \texttt{CSvar} \rangle \neq \langle \texttt{Const}_C \rangle \\
\langle \texttt{Conseq} \rangle &::= \langle \texttt{Disj} \rangle \mid \langle \texttt{Disj} \rangle \vee \langle \texttt{Conseq} \rangle \\
\langle \texttt{Disj} \rangle &::= \langle \texttt{CIOpred} \rangle \mid \langle \texttt{Dpred} \rangle \\
\langle \texttt{CIOpred} \rangle &::= \langle \texttt{CIOvar} \rangle = \langle \texttt{Const}_C \rangle \mid \langle \texttt{CIOvar} \rangle \neq \langle \texttt{Const}_C \rangle \\
\langle \texttt{Dpred} \rangle &::= \langle \texttt{DDvar} \rangle = \langle \texttt{Const}_D \rangle \mid \langle \texttt{DDvar} \rangle \neq \langle \texttt{Const}_D \rangle \\
&\quad \mid \langle \texttt{DDvar} \rangle = \mathit{op}\ \langle \texttt{DSvar} \rangle \mid \langle \texttt{DDvar} \rangle = \langle \texttt{DSvar} \rangle\ \mathit{op}\ \langle \texttt{DSvar} \rangle
\end{aligned}
$$

Fig. 5. The grammar for environment invariants.

The first (top-level) production rule of our grammar defines the shape of a candidate invariant to be an implication between an antecedent (`Ante`) and a consequent (`Conseq`). The antecedent is typically a condition on states of the control FSM(s). Thus, it is expressed as a conjunction of predicates that allow comparison of `CSvar` against constants (that define the control states). The consequent is a disjunction over predicates that allow comparison of `CIOvar` and `DDvar` against constants, or express datapath operations on `DSvar`. Since our invariant synthesis algorithm (to be outlined in Section 4.4) can discover conjunctive invariants, our grammar does not need to enumerate a top-level conjunction of implications. Furthermore, this also avoids a need to have disjunctions in the antecedent or conjunctions in the consequent.

Note that our grammar allows operators on only word-level variables that are data-related. We identified a small set of operators (negation, truncation, addition, subtraction) that is sufficient for our benchmarks. Intuitively, environment invariants are generally *independent* of the actual computation in the datapath. Although the datapath may be capable of using a rich set of operators, those needed for environment invariants tend to be fairly simple.

The sets of constants $\texttt{Const}_C$ and $\texttt{Const}_D$ (used with control and data variables, respectively) need not be the same. The set $\texttt{Const}_C$ can be derived from the Verilog descriptions— it is common practice for designers to define such constants using macros or parameters in Verilog. The concrete values of data variables are less important, and $\texttt{Const}_D$ can be populated with a few concrete values. In our implementation, we use all 0's and all 1's (of appropriate bit-widths that match the variables). One can also extend this set with data constants that appear in Verilog descriptions.

The grammar shown in Figure 5 is recursively defined to allow an arbitrary number of conjuncts (in `Ante`) and disjuncts (in `Conseq`). In practice, we instantiate it with a bound on each, and our experiments (in Sect. 5) show that a small bound of 2 is sufficient in our benchmark examples.

4.3 Candidate Enumeration

We enumerate over all allowed variables in the tagged sets for invariant candidates using the grammar. In addition, we use the following *meta*-production rules and heuristics to prune the set of enumerated candidates.

Grouping (meta-rule). For verifying processor designs, we place additional restrictions during enumeration by using *grouping* over variables, whereby predicates on only the grouped variables are allowed to appear in the same clause in the antecedent or the consequent. For pipeline designs, the variables are grouped together if they are read in the same stage or written by the same stage. In a sense, combinations of grouped variables are likely to be more significant than combinations of unrelated variables. Enumerating clauses by choosing combinations of grouped variables can dramatically reduce the total number of invariant candidates to be checked.

Cone-of-Influence (meta-rule). Not all variables that are tagged need to appear in the invariants. For example, some datapath variables may not affect control-flow, i.e., they are outside the cone-of-influence (COI) of control states. Such variables can be dropped during enumeration. For a specific counterexample-to-equivalence, all variables might not appear in the COI of the equivalence property. The equivalence checker can identify the set of variables in the COI, so we enumerate only these variables and drop the rest.

Ordering the candidates (heuristic). The ordering of enumeration is important: if Inv_1 is inductive relative to Inv_2, then it is useful to first learn Inv_2 and then try Inv_1. Thus, we want to carefully choose an enumeration ordering that is efficient. Our heuristic is to respect the ordering of data-flow/control-flow in Verilog. For example, if there is a flow pattern like $a \rightarrow b$, $a \rightarrow c$, and $b \rightarrow c$, then relations between (a, b) and (a, c) are enumerated before (b, c). This allows the first two relations to set up some relation between b and c, thus making it more likely to be learned as a relative inductive invariant later in the ordering.

4.4 Enumerative SyGuS Solver

Our enumerative SyGuS solver method is shown in Algorithm 2. It takes as inputs the low-level FSM, grammar G, and an error state Bad (V_{start} from Section 3); and either successfully finds a safe inductive invariant Inv or fails (with result UNKNOWN).

We follow a standard *guess-and-check* paradigm for generating safe invariants (e.g., [19, 22]). The set of candidates, *CandSet*, is initialized with expressions enumerated from the given grammar G and pruning heuristics (line 1). The algorithm continues until either the error state is proved to be unreachable (line 3),

or there are no more candidates to process (line 4). A candidate must be implied by the initial state of the FSM (line 7), and it should be inductive relative to the already learned invariants (line 8), in order to be added to the *Learned* set. A candidate is not totally discarded if the inductiveness check fails, but is placed in the *2ndChance* set (line 10). We re-evaluate the inductiveness of such candidates by adding them back to the *CandSet* if their corresponding counterexample-to-induction (CTI, not to be confused with the counterexample-to-equivalence in Section 3) can be blocked by newly learned invariants (line 11).

Algorithm 2: Inv-Syn(FSM, Bad, G): Synthesize invariant to block Bad

Input: $FSM = \langle V \cup V', Init, t_l \rangle$: the low-level (FSM) model, Bad, and G: SyGuS grammar
Output: Safe inductive invariant Inv or Unknown

1 $CandSet \leftarrow$ Enumerate(G);
2 $Learned, 2ndChance \leftarrow \varnothing$;
3 **while** $Bad \wedge \bigwedge_{\ell \in Learned} \ell(V) \not\Longrightarrow \bot$ **do**
4 **if** $CandSet = \varnothing$ **then return** Unknown;
5 **for each** $cand \in CandSet$ **do**
6 $CandSet \leftarrow CandSet \setminus \{cand\}$;
7 **if** $Init(V) \not\Longrightarrow cand(V)$ **then Continue**;
8 **if** $cand(V) \wedge \bigwedge_{\ell \in Learned} \ell(V) \wedge t_l(V, V') \implies cand(V')$ **then**
9 $Learned \leftarrow Learned \cup \{cand\}$;
10 **else** $2ndChance \leftarrow 2ndChance \cup \{cand\}$;
11 **for** $cand \in 2ndChance$ **where** CTI($cand$) $\not\models \bigwedge_{\ell \in Learned} \ell(V)$ **do**
12 $2ndChance \leftarrow 2ndChance \setminus \{cand\}$;
13 $CandSet \leftarrow CandSet \cup \{cand\}$;
14 **return** $Learned$;

5 Experimental Evaluation and Comparison

We have developed Grain, a prototype implementation of our SyGuS-based method for invariant synthesis, on top of an existing CHC solver [19]. We have also developed a flexible CEGAR-based framework for equivalence checking, where we use Grain for synthesis of environment invariants. In this section, we describe an evaluation of these methods on benchmark examples, along with a comparison against other tools for invariant synthesis.

5.1 Methods Evaluated

For the purpose of detailed comparison, we consider the following five methods:

RelChc encodes the equivalence checking and environment invariant synthesis as a single CHC problem (Section 2.2), which is solved by Spacer [38]. Since

this does not use CEGAR explicitly in an outer loop, it serves as a top-level comparison for our CEGAR-based method.

PdrAbc uses our CEGAR-based approach with PDR-based techniques in the ABC tool [15] for solving the reachability query and generating safe invariants. We used Yosys [56] to parse Verilog descriptions and generated AIGER format [35] as input to ABC (since ABC's Verilog parser did not support all Verilog features in our designs). Due to this translation, we were unable to use word-level abstraction techniques in ABC [28].

PdrChc uses our CEGAR-based approach with the Spacer tool [38], a CHC-solver that uses PDR techniques to generate safe invariants. Again, we used Yosys [56] to parse Verilog descriptions and generate SMT-LIB2 [4] instances.

cvc4sy uses our CEGAR-based approach with the SyGuS procedure in CVC4 [51] to synthesize a function that satisfies the constraints (6)-(8). We use the same grammar and variable tagging as in Grain. The difference from Grain is that cvc4sy searches for a single expression that satisfies all three constraints at the same time, whereas Grain iteratively strengthens a candidate with more lemmas that are found inductive.

Grain uses our CEGAR-based approach with our SyGuS-based method for generating invariants.

5.2 Benchmark Examples

We applied all methods on five hardware designs (two synthetic and three from real-world) including processors and accelerators. The ILA specifications for some designs were developed in prior work [33, 55], where manually constructed environment invariants were used to prove equivalence against RTL designs.

Redundant Counters (RC). This example is a synthetic test case that implements a high-level specification of a 4-bit counter. The RTL implementation uses an extra counter for redundancy, storing it as 1's-complement. The RTL output is computed as $\mathit{BitwiseAnd}\,(\mathtt{c}_1, 15 - \mathtt{c}_2)$, which can be simplified as $\mathtt{c}_1$. This relation between $\mathtt{c}_1$ and $\mathtt{c}_2$ is not visible at the high level, and is the environment invariant that needs to be discovered by the synthesis process. This design is used as a small sanity check for our synthesis algorithm.

Simple Pipeline (SP). This design mimics the back-end of a simple pipelined processor. It has three stages (dispatch, execute, and write-back), and four 8-bit wide architectural registers. The instruction set has four instructions (`ADD`, `NOT`, `AND`, and `NOP`). The pipelined implementation has a scoreboard to track latest register values for data forwarding. The environment invariants need to capture the relation between the scoreboard and the intermediate stage registers among the three stages. The human-provided invariant contains 16 implications in conjunction, where each implication is not inductive by itself.

AES Block Encryption Accelerator (AES). The AES block encryption accelerator is a publicly available design from OpenCores.org [31]. An ILA specification was constructed in prior work [55], where a `START_ENCRYPT` command triggers a "load-compute-store" loop that works block-by-block. Although AES

is not the largest design we checked, it poses the most challenges: (a) it needs wide 128-bit state variables in the invariants, (b) the accelerator operation is like software, with a large maximum loop bound (4096), and (c) one of the (human-provided) invariants required a large number of conjunctions.

PicoRV32 Processor (Pico). The PicoRV32 processor [13] is a size-optimized RISC-V processor that implements the RISC-V RV32IMC instruction set. The processor is basically a multi-cycle implementation with an average CPI (cycle-per-instruction) of 4, but it also has some pipelining features, for example, instruction fetch can take place while another instruction is still executing.

Gaussian Blur Accelerator (GB). The Gaussian Blur image processing accelerator is a design generated by high-level synthesis (HLS) in Halide [48], for computing the convolution of an image with a Gaussian kernel. The accelerator streams in and out an image pixel-by-pixel, and buffers the pixels with internal memories (about 32Kb), while multiplication-accumulation (MAC) units are used for convolution. For environment invariant synthesis, we over-approximate the internal memories and MACs by replacing their outputs with free variables (since it is reasonable to expect that their values do not affect the environment invariants). However, we do not over-approximate them for equivalence checking, which is performed using Cadence JasperGold [10] that has built-in abstraction models for memory and computation units. Although the design size of this accelerator is the largest (in number of state bits), the environment invariants required are relatively simple.

5.3 Grammars used for SyGuS

The RTL designs and the instantiations of grammar we used for each benchmark example can be found in our Github repository [57], and our tools will be released as part of the ILAng verification framework [32]. Statistics for the benchmarks and grammars are reported in Table 2, where the last row reports the total number of candidates generated. We used a maximum bound of 2 for number of conjuncts/disjuncts (as shown). We used grouping in SP and PicoRV32 to prune the number of candidates – the number without grouping is 6137 and 255410, i.e., grouping reduced the number of candidates to 34% and 25%, respectively. We briefly summarize key points about tagging variables.

- Redundant Counters (RC): has only two variables, both are `data-src/-dst`.
- Simple Pipeline (SP): the scoreboard is tagged as `ctrl-state`, write-enable signals are `ctrl-inout`, and destination signals are `data-dst`. Grouping is used to group together the signals in the same stage of the pipeline.
- AES accelerator: computation in the datapath is ignored (plaintexts, ciphertexts, and keys), control FSM is kept, index and block counters are tagged as `ctrl-inout`, address and length registers are tagged as `data-src/-dst`.
- PicoRV32 processor: The ALU, register files, memory input/output data, and performance counters are ignored for environment invariants, while flags and control FSMs are kept. Flags are tagged as `ctrl-inout`, and grouped by the stage where they are set (decode/execute units, interrupts).

Table 2. Statistics of Benchmarks and SyGuS Grammars

Benchmarks	RC	SP	AES	PicoRV32	Gaussian Blur
#. state-bits	8	72	963†	1817	4840†
#. word-level state-vars	2	16	14†	149	176†
#. ctrl-state	-	4	3	30	4
#. ctrl-inout	-	2	2	34	8
#. data-src	2	-	2	-	-
#. data-dst	2	2	3	-	11
#. groups	-	2	-	3	-
Max. antecedent size	1	1	2	2	2
Max. consequent size	1	2	1	1	1
#. candidates	2	2112	22048	63663	19195

† For AES, this model abstracts the round-level computation; for Gaussian Blur, this model abstracts internal block RAMs and MACs.

– Gaussian Blur accelerator (GB): Since this design is generated by HLS, the naming of state variables follows some conventions. Control FSMs with names `ap_CS_fsm` are tagged as `ctrl-state`. Flag bits, tagged as `ctrl-inout`, have names like `xxx_full_n`, `xxx_empty_n`, or `exitcond_xxx`. Address pointers (with name `mOutPtr`), column or row counters (with names `col_reg_xxx` or `row_reg_xxx`) are tagged as `data-dst`.

5.4 Results of Experiments

The experiments were conducted on a laptop with 4-core i5-8300H processor and 32GB memory, except for Gaussian Blur which needs Cadence JasperGold (available on a server with 56 cores and 256GB memory). All other benchmarks used CoSA [42] for equivalence checking. Counterexamples are extracted by parsing the waveform generated by JasperGold or CoSA. We set the time-out limit for the CEGAR-loop to be 10 hours, which includes time for both equivalence checking and environment invariant synthesis.

The results are reported in Table 3 for the five methods (along columns) on the benchmark examples (along rows). We also report additional details (number of iterations for CEGAR-based methods and times for synthesis and equivalence checking). Note that our proposed method Grain successfully finds environment invariants in all benchmarks, and outperforms the other four methods on the three real-world designs (AES, Pico, GB).

5.5 Detailed Comparison and Discussion

CEGAR-loop vs. Monolithic CHC Query. As PdrChc uses the same CHC solver as RelChc, we use the results of the two as a comparison between CEGAR-loop and using a monolithic CHC query. RelChc succeeds on only

Table 3. Results of Experiments

		RelChc	CEGAR			
			PdrAbc	PdrChc	cvc4sy	Grain
Solver		Z3	ABC	Z3	CVC4	Z3
RC	# iter.	-	4	6	1	1
	$t_{syn}(s)$	-	30.2	11.5	0.2	1.2
	$t_{eq}(s)$	-	2.1	4.7	0.8	0.8
	$t_{total}(s)$	1.6	32.3	16.2	**1.0**	2.0
SP	# iter.	-	21	36	1	4
	$t_{syn}(s)$	-	1.9	2.9	O.O.M	134.4
	$t_{eq}(s)$	-	27.8	37.5	1.2	10.7
	$t_{total}(s)$	1035.2	**29.7**	40.4	O.O.M	145.1
AES	# iter.	-	2	2	4	5
	$t_{syn}(s)$	-	O.O.M.	T.O.	O.O.M.	912.3
	$t_{eq}(s)$	-	6.4	6.5	17.5	35.5
	$t_{total}(s)$	T.O.	O.O.M.	T.O.	O.O.M.	**947.8**
Pico	# iter.	-	3	149	1	9
	$t_{syn}(s)$	-	O.O.M.	3771.7	O.O.M.	4345.9
	$t_{eq}(s)$	-	87.2	4493.1	6.8	83.5
	$t_{total}(s)$	T.O.	O.O.M.	7864.8	O.O.M.	**4429.4**
GB	# iter.	-	176	7	8	3
	$t_{syn}(s)$	-	63.1	1292.4	161.9	414.5
	$t_{eq}(s)$	-	T.O.	1491.3	1653.2	631.5
	$t_{total}(s)$	T.O.	T.O.	2783.7	1815.1	**1046.0**

RC, SP, AES, Pico, GB denote the five benchmarks: Redundant Counters, Simple Pipeline, AES block encryption accelerator, PicoRV32 processor, and Gaussian Blur accelerator.
O.O.M indicates out-of-memory (> 32GB) and T.O. indicates time-out (> 10 hours). "# iter." reports the number of CEGAR iterations. For methods that did not converge within the time/memory limit, we report the last iteration it finished before it terminates.
RelChc does not use CEGAR, we only report total solving time. For all CEGAR methods, the total time (t_{total}) is the sum of time for synthesis (t_{syn}) and time for equivalence checking (t_{eq}).

the first two examples, both under 100 state bits. We looked at the invariants produced by the two methods and saw that RelChc generated invariants that were easier to understand. For example, for RC, it generated $\bigwedge_i \mathtt{c}_1[i] \oplus \mathtt{c}_2[i], i \in \{1, .., 4\}$ (where $\oplus$ is the XOR operator), which is more succinct than those generated by PdrChc. However, the overall results clearly show that CEGAR-based approaches are more scalable.

PDR-based methods: Spacer vs. ABC. Although Spacer and ABC both use PDR, their performance varies. For benchmarks where both succeeded,

PdrAbc requires less synthesis time and fewer CEGAR iterations. However, ABC seems to have bigger memory requirements – two of the five benchmarks failed due to out-of-memory error. For GB, using PdrAbc results in CEGAR-loop time-out. For the failing query of AES, we also tested gate-level abstraction techniques in ABC [28, 45], but these did not succeed (due to either timeout or too coarse abstraction). As mentioned earlier, due to our translation to AIGER, we were not able to test the word-level abstraction technique in ABC.

SyGuS-based methods: Grain vs. cvc4sy. One main difference between our method Grain and cvc4sy is how they construct the required invariant. Since cvc4sy is a generic SyGuS solver, its verifier requires a term enumerator to propose a candidate that satisfies all constraints (i.e., all three conditions for a safe inductive invariant) at the same time. In contrast, Grain is specialized for finding safe invariants for transition systems by incrementally conjoining relatively inductive candidates (inspired by PDR).

To study this further, we tested cvc4sy in more detail on the SP example. We instantiated the grammar with a fixed number of implications (based on known invariants) and asked cvc4sy to fill in the antecedent and consequent of each implication. This significantly shrinks the search space for cvc4sy. However, this still resulted in out-of-memory errors. This is likely due to a large number of syntactic constraints that the CVC4 term generator learns from failing candidates.

For AES, cvc4sy finished the first four iterations of the CEGAR-loop fairly fast, where the invariants contain at most one implication. In the fifth round, an invariant with many conjoined implications is needed, but it failed. cvc4sy is also successful on GB, where no invariant requires more than one implication. It seems that a large number of top-level conjunctions is an obstacle for cvc4sy, whereas Grain can handle this by candidate strengthening techniques.

Another difference is that Grain collects inductive candidates along the search for safe inductive invariants. These inductive invariants can block other infeasible counterexamples. Therefore, Grain requires fewer CEGAR iterations.

SyGuS-based vs. PDR-based techniques. When well-guided by grammars, Grain can deliver comparable performance to PDR-based approaches, and it outperforms them on large real-world designs. We believe this is mainly due to our emphasis on word-level invariants, which seem more difficult to derive in PdrAbc and PdrChc. For example, in AES, the failing query for PdrAbc and PdrChc needs an invariant: $\texttt{STATUS} \neq 0 \implies \texttt{IV} + \texttt{BLK\_CNT} = \texttt{AES\_CNT}$. This says that when the accelerator is operating, the current operation counter is the sum of a block counter and the initial value (IV), where the three variables in the consequent are all 128-bit. This relation is simple on the word-level but complex on the bit-level. Another challenge in accelerators is they contain loop structures similar to software. For example, the "load-compute-store" loop in AES can iterate as many as 4096 times. With a large number of variables after bit-blasting, it becomes harder for PDR to reach a fixpoint when computing forward reachability.

An interesting difference we noticed is that both PdrAbc and PdrChc sometimes produce invariants that refer to datapath variables. For example, instead

of a generalized invariant *Inv*, one may get $(D = v \implies Inv)$, where D is a datapath variable and v is some concrete value. The antecedent here is usually unnecessary, as in many processing units, the datapath variable can have an arbitrary value, and the consequent is a valid fact regardless of the value of D. These invariants could result in more CEGAR iterations and longer synthesis time in total. On the other hand, our grammars used in Grain target word-level expressions, place restrictions on variable sets, and do not enumerate concrete values on data-related variables. Therefore, they generate candidates that can produce a more general invariant.

For Grain, PicoRV32 is the hardest example due to a large number of state variables, which results in a large search space for enumeration. In this case, it would be beneficial for a human designer to provide additional insights to shrink the search space.

Lessons Learned and Potential Improvements. The above experiments show that PDR-based techniques sometimes suffer from an explosion of state bits due to bit-blasting, and sometimes generate invariants that are too specific to a query (thereby requiring more CEGAR iterations). By working on the word-level and using guidance on state variables to consider in candidate invariants, Grain can outperform PDR-based techniques under such situations. It would be interesting to investigate in future work whether the generalization step in PDR can benefit from guidance using grammars.

6 Related Work

Invariant Synthesis. Automatic generation of invariants has been studied extensively in verification. Among symbolic model checking techniques, IC3/PDR [6, 15] has demonstrated success for both hardware and software verification. It incrementally constructs an inductive invariant by iteratively removing counterexamples to induction. In software verification, several application of PDR [5, 11, 25, 29, 38] for linear arithmetic and arrays have been proposed. PDR engines that support bit-vectors often use bit-blasting and find fixpoints on the bit-level. As the data-width increases, the same word-level formula becomes larger on the bit-level. For hardware, PDR has been combined with various abstraction techniques [18, 45], including the use of word-level information to construct abstractions [28, 40]. In Grain, we continue this trend of leveraging word-level information to help with scalability, and extend it by considering roles of the variables (e.g., control or data) in the design.

Other work also targets bit-precise invariants for software [8, 26, 30] by lazily encoding the program in parts by using bit-vectors, along with light-weight theories (such as equality with uninterpreted functions, Presburger arithmetic, and linear rational arithmetic). These techniques, however, have not been evaluated on large hardware designs so far, and we expect they would require many refinement iterations before converging.

Syntax-Guided Synthesis [1] has also been used successfully for invariant generation, although not for large hardware designs so far. LoopInvGen [47] takes

a data-driven approach and learns features for loop invariance inference, whereas GRAIN relies more on the structure of hardware designs to provide guidance and enumeration heuristics. The CVC4SY solver [51] employs various advanced enumeration techniques from user-provided grammars, but attempts to generate a whole invariant *at once*, which has significant scalability implications. Liquid Fixpoint [52] has been used in IODINE [24] to generate invariants for constant-time property checking for hardware. It uses candidates from predicate abstraction rather than a grammar. The approach closest to ours is FREQHORN [19–21] that also generates individual lemmas first and then conjoins them together to derive an invariant. However it relies on various heuristics to automatically construct grammars (e.g., from the syntax and bounded semantics of the program). In contrast, our grammars leverage domain-specific knowledge of hardware designs.

Modular Hardware Verification. Our focus is on using instruction-level modularity in hardware equivalence checking, which has also been embraced in industrial practice [37, 50]. For modular verification of systems, in general, the specification and implementation are partitioned component-wise and assume-guarantee rules are used to reason about a component and its interaction with the environment [23, 41, 43]. Our environment invariants are also a form of environment assumptions, which we aim to discover automatically.

Other instruction-level verification efforts can also benefit from automatic generation of environment invariants. For example, Symbolic Quick-Error-Detection (S-QED) [17], unbounded protocol compliance verification [46], hardware information flow tracking [16] – all used some form of symbolic initial state constraints to avoid spurious counterexamples, where these constraints are manually constructed. Our methods for automated discovery of environment invariants can potentially benefit these applications by reducing human effort.

7 Conclusions

In this paper, we described techniques for automating the discovery of environment invariants for per-instruction modular hardware verification. We used an equivalence checker coupled with a CEGAR-based method to iteratively construct such invariants. We proposed a SyGuS-based method for invariant synthesis in each iteration, where we use a novel grammar to guide the search for invariant candidates. The grammar leverages domain-specific features in hardware designs, and can be tuned by a user. Our invariant synthesis approach is inspired by existing PDR-based and SyGuS-based techniques. It targets word-level invariants, to avoid dealing with complex relations at the bit-level, and constructs conjunctive invariants incrementally. Our detailed experiments demonstrate the effectiveness of our proposed CEGAR-based and SyGuS-based methods on several hardware designs including processors and accelerators.

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 Grants 1525936 and 1628926.

References

1. 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)
2. Alur, R., Singh, R., Fisman, D., Solar-Lezama, A.: Search-based program synthesis. Communications of the ACM **61**(12), 84–93 (2018)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. vol. 6806, pp. 171–177 (Jul 2011)
4. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. Handbook of Satisfiability pp. 825–885 (2009)
5. Bjørner, N., Gurfinkel, A.: Property directed polyhedral abstraction. In: VMCAI. pp. 263–281 (2015)
6. Bradley, A.R.: Sat-based model checking without unrolling. In: VMCAI. pp. 70–87 (2011)
7. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: CAV. pp. 24–40 (2010)
8. Bueno, D., Sakallah, K.A.: euforia: Complete Software Model Checking with Uninterpreted Functions. In: VMCAI. LNCS, vol. 11388, pp. 363–385. SV (2019)
9. Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: CAV. pp. 68–80. Springer (1994)
10. Cadence Design Systems, Inc.: Jaspergold: Formal property verification app (2019), http://www.jasper-da.com/products/jaspergold-apps/, accessed: 2019-09-20
11. Cimatti, A., Griggio, A.: Software model checking via IC3. In: CAV. LNCS, vol. 7358, pp. 277–293. SV (2012)
12. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. pp. 154–169 (2000)
13. Clifford Wolf: Picorv32 - a size-optimized RISC-V cpu (2019), https://github.com/cliffordwolf/picorv32, accessed: 2019-09-20
14. Das, A., Lahiri, S.K., Lal, A., Li, Y.: Angelic verification: Precise verification modulo unknowns. In: Computer Aided Verification (CAV), Part I. pp. 324–342 (2015)
15. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD. pp. 125–134 (2011)
16. Fadiheh, M.R., Stoffel, D., Barrett, C., Mitra, S., Kunz, W.: Processor hardware security vulnerabilities and their detection by unique program execution checking. In: DATE. pp. 994–999 (2019)
17. Fadiheh, M.R., Urdahl, J., Nuthakki, S.S., Mitra, S., Barrett, C., Stoffel, D., Kunz, W.: Symbolic quick error detection using symbolic initial state for pre-silicon verification. In: DATE. pp. 55–60 (2018)
18. Fan, K., Yang, M.J., Huang, C.Y.: Automatic Abstraction Refinement of TR for PDR. In: Asia and South Pacific Design Automation Conference. pp. 121–126 (2016)
19. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: TACAS. pp. 251–269 (2018)
20. Fedyukovich, G., Kaufman, S., Bodík, R.: Sampling invariants from frequency distributions. In: FMCAD. pp. 100–107 (2017)
21. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained horn clauses using syntax and data. In: FMCAD. pp. 170–178 (2018)
22. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A Robust Framework for Learning Invariants. CAV pp. 69–87 (2014)

23. Giannakopoulou, D., Namjoshi, K.S., Pasareanu, C.S.: Compositional reasoning. In: Handbook of Model Checking., pp. 345–383. Springer International Publishing (2018)
24. v. Gleissenthall, K., Kıcı, R.G., Stefan, D., Jhala, R.: IODINE: Verifying constant-time execution of hardware. In: USENIX Security Symposium. pp. 1411–1428 (2019)
25. Gurfinkel, A.: IC3, PDR, and Friends. Summer School on Formal Techniques (2015)
26. Gurfinkel, A., Belov, A., Marques-Silva, J.: Synthesizing safe bit-precise invariants. In: TACAS. LNCS, vol. 8413, pp. 93–108. SV (2014)
27. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: CAV. pp. 343–361 (2015)
28. Ho, Y.S., Mishchenko, A., Brayton, R.: Property directed reachability with word-level abstraction. In: FMCAD. pp. 132–139 (2017)
29. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: SAT. pp. 157–171 (2012)
30. Hojjat, H., Rümmer, P.: The ELDARICA Horn Solver. In: FMCAD. pp. 158–164. IEEE (2018)
31. Hsing, H.: Opencores.org tiny_aes project page. https://opencores.org/projects/tiny_aes (2014), accessed: 2019-09-20
32. Huang, B.Y., Zhang, H., Gupta, A., Malik, S.: ILAng: A modeling and verification platform for SoCs using instruction-level abstractions. In: TACAS. pp. 351–357 (2019)
33. Huang, B., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., Malik, S.: Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification. ACM Trans. Design Autom. Electr. Syst. **24**(1), 10:1–10:24 (2019)
34. Ivancic, F., Balakrishnan, G., Gupta, A., Sankaranarayanan, S., Maeda, N., Imoto, T., Pothengil, R., Hussain, M.: Scalable and scope-bounded software verification in varvel. Automated Software Engineering **22**(4), 517–559 (2015)
35. Jacobs, S.: Extended AIGER format for synthesis. arXiv preprint:1405.5793 (2014)
36. Jhala, R., McMillan, K.L.: Microarchitecture verification by compositional model checking. In: CAV (2001)
37. Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V.A., Reeber, E., Naik, A.: Replacing testing with formal verification in Intel Core^{TM}i7 processor execution engine validation. In: CAV. pp. 414–429 (2009)
38. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. FMSD **48**(3), 175–205 (2016)
39. Kuehlmann, A., Bergamaschi, R.A.: High-level state machine specification and synthesis. In: ICCD. pp. 536–539 (1992)
40. Lee, S., Sakallah, K.A.: Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In: CAV. pp. 849–865. Cham (2014)
41. Manolios, P., Srinivasan, S.K.: A refinement-based compositional reasoning framework for pipelined machine verification. IEEE Trans. VLSI Syst. **16**(4), 353–364 (2008)
42. Mattarei, C., Mann, M., Barrett, C., Daly, R.G., Huff, D., Hanrahan, P.: CoSA: Integrated verification for agile hardware design. In: FMCAD. IEEE (2018)
43. McMillan, K.L.: Verification of an implementation of tomasulo's algorithm by compositional model checking. In: CAV. pp. 110–121 (1998)

44. McMillan, K.L.: Modular specification and verification of a cache-coherent interface. In: FMCAD. pp. 109–116 (2016)
45. Mishchenko, A., Een, N., Brayton, R., Baumgartner, J., Mony, H., Nalla, P.: Gla: Gate-level abstraction revisited. In: DATE. pp. 1399–1404 (2013)
46. Nguyen, M.D., Thalmaier, M., Wedler, M., Bormann, J., Stoffel, D., Kunz, W.: Unbounded protocol compliance verification using interval property checking with invariants. IEEE Trans. on CAD of Integrated Circuits and Systems **27**(11), 2068–2082 (2008)
47. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: PLDI. pp. 42–56. ACM (2016)
48. Ragan-Kelley, J., Adams, A., Paris, S., Durand, F., Barnes, C., Amarasinghe, S.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: PLDI. pp. 519–530 (2013)
49. Reid, A.: Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In: FMCAD. pp. 161–168 (2017)
50. Reid, A., Chen, R., Deligiannis, A., Gilday, D., Hoyes, D., Keen, W., Pathirane, A., Shepherd, O., Vrabel, P., Zaidi, A.: End-to-end verification of ARM® processors with isa-formal. In: CAV. pp. 42–58 (2016)
51. 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)
52. Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. In: PLDI. pp. 159–169 (2008)
53. Si, X., Yang, Y., Dai, H., Naik, M., Song, L.: Learning a meta-solver for syntax-guided program synthesis. In: International Conference on Learning Representations (2019)
54. Subramanyan, P., Huang, B.Y., Vizel, Y., Gupta, A., Malik, S.: Template-based parameterized synthesis of uniform instruction-level abstractions for SoC verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **37**(8), 1692–1705 (2018)
55. Subramanyan, P., Vizel, Y., Ray, S., Malik, S.: Template-based synthesis of instruction-level abstractions for SoC verification. In: FMCAD. pp. 160–167 (2017)
56. Wolf, C.: Yosys open synthesis suite. http://www.clifford.at/yosys/, accessed: 2019-09-20
57. Zhang, H., Yang, W., Fedyukovich, G.: Benchmark examples for environment invariant synthesis. https://github.com/zhanghongce/vmcai2020-inv-syn-benchmarks, accessed: 2019-10-03