Formal Methods in Computer-Aided Design 2021

IC3 with Internal Signals

Rohit Dureja
IBM

Arie Gurfinkel
University of Waterloo

Alexander Ivrii
IBM

Yakir Vizel
The Technion

***Abstract*—IC3 is a highly-effective algorithm for formal hardware verification. It cleverly uses a SAT solver to compute an inductive invariant, an over-approximation of reachable states, of a hardware design. The invariant is computed in CNF as a conjunction of lemmas. This CNF representation over state variables, although efficient, leads to an obvious deficiency: IC3 is not effective for designs that do not have a concise CNF invariant over state variables. We show how to remedy this deficiency by extending traditional IC3 to learn invariants not only in terms of state variables, but also in terms of internal signals of the design. Our proposed method can learn significantly more compact invariants than IC3, while maintaining a highly-efficient CNF representation. We evaluate our technique on several industrial sequential equivalence checking (SEC) problems from IBM, SEC problems derived from designs in the Hardware Model Checking Competition (HWMCC) and SEC problems from academia. In addition, we evaluate it on HWMCC benchmarks. IC3 with internal signals is efficient for SEC and outperforms traditional IC3 on an important class of benchmarks.**

I. Introduction

IC3 [1], [2] is a powerful algorithm for formal hardware verification, and is the primary model-checking engine in various state-of-the-art formal verification tools. IC3, and its several variants [3], is especially useful for establishing system safety (i.e., discovering an inductive invariant). Whenever IC3 succeeds in proving safety, it finds an inductive invariant justifying the property. Traditionally, such an invariant is a conjunction of lemmas represented in CNF, each lemma is a disjunction of literals, and each literal is either a state variable or its negation. Conversely, IC3 does not succeed in proving a property when it is unable to find such an inductive invariant within the specified verification-resource limits. This can happen for one of two reasons: (i) a small inductive invariant exists but IC3 is unable to find it, or (ii) a small inductive invariant does not exist. It is difficult to determine which of these two cases is responsible for IC3 failing to prove a property. Most research on improving IC3 (e.g., [4]–[6]) focuses on quickly finding the inductive invariant. However, finding the inductive invariant quickly can only help if a (reasonably) small invariant exists in the first place.

A known Achilles heel of IC3 are model-checking problems for which any inductive invariant (over state variables) is necessarily exponential in size. For example, let $x_1, \ldots, x_n$ be state variables, and suppose that the set of reachable states is characterized by $\{x_1, \ldots, x_n \mid x_1 \oplus \cdots \oplus x_n = 1\}$, while the set of bad states is characterized by $\{x_1, \ldots, x_n \mid x_1 \oplus \cdots \oplus x_n = 0\}$. In this case the (only) inductive invariant is exponential in size and contains 2^{n-1} clauses that correspond to representing $x_1 \oplus \cdots \oplus x_n = 1$ in CNF. With $n = 3$, the inductive invariant contains four clauses: $(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3)$. A possible work-around is to extend the design with additional signals that are necessary to concisely represent an invariant. In this example, IC3 extended with a lemma over $z = x_1 \oplus \cdots \oplus x_n$, can find a tiny inductive invariant consisting of only a single unit-clause lemma: $(z = 1)$.

This leads to the question of which additional signals to consider. A possible solution is to consider variables that represent logic gates in the transition relation of the system model. We refer to these as *internal nets* or *innards*. Prior work [7] uses innards to extend ternary valued simulation of counterexamples to induction in IC3, which enables a succinct description of the set of states that IC3 must eventually block. In this paper, we propose an approach based on learning lemmas directly over innards that improves the performance of IC3 in establishing safety by finding more concise inductive invariants. Our method of learning lemmas over internal nets can be viewed as a form of inductive generalization. A lemma is first generalized as usual, and then literals corresponding to latches are replaced by internal nets. Specifically, whenever IC3 learns a lemma C over state variables, it also tries to learn an additional lemma C_2 over state variables and internal signals. To this end, we first extend C to a lemma C_1 that is logically equivalent to C but contains the literals of C and (certain) internal nets. We obtain C_2 by inductively generalizing C_1, while guiding the inductive generalization to remove state variables. It is guaranteed that C_2 is stronger than C. Therefore, C_2 blocks the same states (and maybe more) as C. We then add lemma C_2 to IC3's inductive trace, so that it can be used for predecessor queries and convergence checks. A major advantage of our approach is that it can be easily integrated with any existing mature IC3 implementation.

Our work is motivated by a challenging set of microprocessor verification problems that arise from the Aspect-Oriented Design (AOD) methodology used at IBM. The verification problem checks sequential equivalence of an original design against a new version of the design with added aspects (e.g., clock-gating, logging, or debug interfaces). The complex verification challenge is broken into many sub-tasks using a combination of the usual sequential equivalence checking (SEC) approaches, including k-induction, speculative reduction, and localization [8]–[12]. Verification sub-tasks that are not solved by these techniques are then checked using Interpolation-based Model Checking (IMC) or IC3. Traditional IC3 scales very poorly for these verification problems. On the other hand, IMC works rather well but is not stable – small changes in the

https://doi.org/

design negatively impact verification times. The proposed IC3 algorithm with internal signals significantly outperforms both IMC and traditional IC3.

The proprietary nature of IBM AOD verification problems prohibits detailed public disclosure. Nevertheless, we apply the IBM AOD sequential equivalence checking flow on two selected benchmarks from the Hardware Model Checking Competition (HWMCC) to validate equivalence between the original design and its retimed [13] versions. Each such equivalence-check generates hundreds of verification problems of which some are solved by k-induction, but a significant number remain unsolved. We note that IC3 with internal signals is more effective than traditional IC3 in solving the remaining equivalences for both SEC problems. We also apply our algorithm on a small set of publicly available SEC benchmarks [14] from academia, and note that our proposed algorithm is able to solve a higher number of equivalences compared to traditional IC3. This suggests that using internal nets in IC3 is especially effective for difficult sequential equivalence checking problems.

To further validate the efficacy of IC3 with internal signals, we apply the proposed algorithm to a variety of single-property benchmarks from HWMCC. However, the technique does not show a significant improvement unlike our experience with IBM AOD and other benchmarks. There are a few HWMCC benchmarks that are solved significantly faster and some that are uniquely solved by our algorithm, but overall, traditional IC3 is superior. Interestingly, the number of designs where the new technique succeeds increases in the latest competition editions that are based on word-level designs. This points to a deficiency of any benchmark set – the distribution of problems in the set does not necessarily correspond to their distribution in practice. Techniques that perform well on only a few benchmarks in the set, might actually be very effective in some practical application!

The rest of the paper is organized as follows. Section II provides the necessary background. Section III describes motivating examples to highlight the core deficiency of IC3 addressed by our approach. Section IV describes the IC3 algorithm with internal signals, while Section V reports on our experimental evaluation. Section VI discusses related and future work, and Section VII concludes.

II. Background

A. Safety Verification Problem

We represent a finite state transition system S as a tuple $\langle i, x, \mathit{Init}(x), \mathit{Tr}(i,x,x')\rangle$, which consists of primary inputs i, state variables x, predicate $\mathit{Init}(x)$ defining the initial states, and predicate $\mathit{Tr}(i,x,x')$ defining the transition relation. Next-state variables are denoted as x'. We assume that Tr is represented as a *netlist*, that is, a directed acyclic graph with nodes corresponding to logic gates. Given the values of x and i, the values of x' may thus be uniquely computed by (constant) propagation – i.e., using Boolean or three-valued simulation. We say that a *net* is either an input, a state variable or a logic gate. We refer to state variables and their negations as *latches*, and to internal logic gates and their negations as *innards*. We say that an innard is *input-free* if it does not have any inputs in its combinational cone-of-influence.

A *clause* is a disjunction of literals, where each literal is either a net or its negation. We say that a clause is *over latches* to emphasize all the literals in the clause are latches. A Boolean formula in *Conjunctive Normal Form (CNF)* is a conjunction of clauses. A *cube* is a conjunction of literals. A Boolean formula in *Disjunctive Normal Form (DNF)* is a disjunction of cubes. It is often convenient to treat a clause or a cube as a set of literals, a CNF as a set of clauses, and DNF as a set of cubes. For example, given a CNF formula F, a clause c and a literal ℓ, we write $\ell \in c$ to mean that ℓ occurs in c, and $c \in F$ to mean that c occurs in F.

A *trace* is a sequence of Boolean valuations to the nets, starting with an initial state satisfying Init and with successive time-step valuations consistent with Tr. *Reachable states*, denoted by Reach, are states that can be reached on a trace. Let $\mathit{Bad}(x)$ be a predicate defining *bad* (or *unsafe*) states. The *safety verification problem* consists of checking whether $\mathit{Reach} \Rightarrow \neg \mathit{Bad}$, that is either finding a trace that leads to a state in Bad or showing that such a trace does not exist.

B. Traditional IC3

We give a very brief and high-level description of IC3, concentrating on the components that are relevant for this work. This description includes the classical IC3 algorithm [1], [2], and some of its variants such as [6]. In what follows, we refer to all these algorithms simply as IC3.

IC3 proves safety by finding a formula $\mathit{Inv}(x)$, called *a safe inductive invariant*, that satisfies the following conditions:

$$\mathit{Init}(x) \Rightarrow \mathit{Inv}(x) \tag{1}$$

$$(\mathit{Inv}(x) \wedge \exists i \cdot \mathit{Tr}(i,x,x')) \Rightarrow \mathit{Inv}(x') \tag{2}$$

$$\mathit{Inv}(x) \Rightarrow \neg \mathit{Bad}(x) \tag{3}$$

The computed formula $\mathit{Inv}(x)$ is in CNF over latches. Internally, IC3 maintains sets of clauses $F_0, F_1, \ldots$ called an *inductive trace*. Each F_k in a trace is called a *frame*, each clause $c \in F_k$ is called a *lemma*, and the index of a frame is called a *level*. We assume that F_0 is initialized to Init and that $\mathit{Init} \Rightarrow \neg \mathit{Bad}$. IC3 maintains the following invariant:

$$F_0 = \mathit{Init} \qquad F_{k+1} \subseteq F_k \qquad F_k \wedge \mathit{Tr} \Rightarrow F'_{k+1}$$

Note that the inductive trace maintained by IC3 is syntactically monotone, and each F_{k+1} is inductive relative to F_k. Let $\mathit{Reach}_{\leq k}$ denote the set of states reachable from Init in k steps or less. It holds that $\mathit{Reach}_{\leq k} \Rightarrow F_k$, i.e., F_k is an over-approximation of states reachable in k steps or less.

Additionally, IC3 maintains a queue of *proof obligations* (or *CTI's*) of the form $\langle m, k\rangle$ where m is a cube over latches and $k > 0$ is a *level*. At each point of the execution, it considers a proof obligation $\langle m, k\rangle$, and makes an *initial* query $\mathit{SAT?}(\mathit{Init} \wedge \neg m)$ that checks whether a state in m is an initial state, and a *predecessor* query $\mathit{SAT?}(\neg m \wedge F_{k-1} \wedge \mathit{Tr} \wedge m')$ that checks whether a state in m can be reached from a state in F_{k-1}. If both results are unsatisfiable, IC3 can add the lemma $\neg m$ to all F_j, for $j \leq k$, refining the inductive trace. However, for performance it is crucial to *inductively generalize* $\neg m$ first, finding a lemma $\varphi \subseteq \neg m$, that also satisfies $\mathit{Init} \Rightarrow \varphi$ and $\varphi \wedge F_{k-1} \wedge \mathit{Tr} \Rightarrow \varphi'$ (some IC3-variants such as `Quip` also keep an under-approximation of *Reach* and modify *Init* to include this under-approximation). The inductive generalization is typically done by removing literals from $\neg m$ while the two conditions remain satisfied. We refer the reader to [3] for more details.

IC3 periodically *pushes* all lemmas, by checking if a lemma $\varphi \in F_k \setminus F_{k+1}$ can be added to F_{k+1} as well. If at any point, $F_k = F_{k+1}$ and $F_k \Rightarrow \neg \mathit{Bad}$, then we can take $\mathit{Inv} = F_k$ as the safe inductive invariant.

III. Motivating Examples

In this section, we motivate our work with several examples. Each is a series of problems such that inductive invariants in CNF over latches grow exponentially, while the corresponding inductive invariants over latches and innards grow linearly. The examples are sketched briefly here, we provide full details with AIGER and source files in the companion repository.[1] Note that the examples are distilled to their essence. For some, the property itself is inductive. Thus, traditional IC3 that learns invariants over latches *and* the property is able to solve them. However, the illustrated problems remain when the examples are parts of a larger design, and the property is more complex and is no longer inductive on its own.

Example 1 (Parity) Let $x_1, \ldots, x_n$ be the latches. The set of reachable states is characterized by $\{x_1, \ldots, x_n \mid x_1 \oplus \cdots \oplus x_n = 1\}$. The set of bad states is characterized by $\{x_1, \ldots, x_n \mid x_1 \oplus \cdots \oplus x_n = 0\}$. Note that the only safe inductive invariant over latches has 2^{n-1} clauses representing $x_1 \oplus \cdots \oplus x_n = 1$ in CNF. Yet, there is a safe inductive invariant consisting of a single lemma, $(z = 1)$, for the innard $z = x_1 \oplus \cdots \oplus x_n$. □

Example 2 (from [15]) Consider two counters that count modulo-2^n, whose state bits are $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{n-1})$, respectively. Let i be an input. When $i = 0$ both counters keep their values; when $i = 1$ both counters increment their values by one modulo 2^n. Suppose that the initial state is $\{s \neq t\}$, and the bad state is $\{s = t\}$. The work [15] argues that any safe inductive invariant for the usual IC3 must contain at least 2^n lemmas. Furthermore, there is a much smaller safe inductive invariant for the *Reverse IC3* that consists of $2n$ lemmas required to represent $s = t$ in CNF. With innards, there is an inductive invariant consisting of a single lemma, $(z = 1)$, for the innard $z = (s \neq t)$. □

Example 3 (SEC) This example illustrates a sequential equivalence checking problem between an original and a retimed [13] design. Let the "original part" of the design consist of latches $x_1, \ldots, x_n$ and inputs $i_1, \ldots, i_n$, such that $\mathit{init}(x_k) = 0$ and $\mathit{next}(x_k) = i_k$ for $k = 1, \ldots, n$, and a net $z = x_1 \oplus \cdots \oplus x_n$. Let the "retimed part" of the design consist of a net $u = i_1 \oplus \cdots \oplus i_n$ and a latch v with $\mathit{init}(v) = 0$ and $\mathit{next}(v) = u$. Let the the bad state be $\{z \neq v\}$. The only safe inductive invariant is $v \leftrightarrow (x_1 \oplus \cdots \oplus x_n)$, that consists of 2^n lemmas in CNF. With innards, an alternative invariant requires only two lemmas: $v \rightarrow z$ and $z \rightarrow v$. □

Example 4 This example is motivated by the benchmark `rast-p16` from HWMCC'20. The design contains latches $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$, and innards $z_1 = x_1 \wedge y_1$, ..., $z_n = x_n \wedge y_n$. Assume that the lemma $C = (z_1 \vee \cdots \vee z_n)$ over innards is inductive. Representing C in CNF over latches requires 2^n lemmas. For example, for $n = 3$, the lemma $(z_1 \vee z_2 \vee z_3)$ is equivalent to 8 lemmas $(x_1 \vee x_2 \vee x_3)$, $(x_1 \vee x_2 \vee y_3)$, $(x_1 \vee y_2 \vee x_3)$, $(x_1 \vee y_2 \vee y_3)$, $(y_1 \vee x_2 \vee x_3)$, $(y_1 \vee x_2 \vee y_3)$, $(y_1 \vee y_2 \vee x_3)$, $(y_1 \vee y_2 \vee y_3)$. □

IV. Finding lemmas over innards

In this section, we provide an overview of our approach (Sec. IV-A), followed by an algorithm for extending IC3 lemmas with innards (Sec. IV-B), and finally an algorithm for inductive generalization in the presence of innards (Sec. IV-C).

A. The overall approach

Traditional IC3 learns lemmas by inductively generalizing negations of blocked proof obligations. Both proof obligations and lemmas are over latches. These lemmas are then added to IC3's inductive trace and used in future predecessor and convergence checks. In our approach, proof obligations are also over latches (exactly the same as in traditional IC3), however, we extend learning lemmas over both latches and innards. Our results apply to arbitrary innards, but for simplicity of presentation in the rest of the paper, we restrict to input-free innards, calling them simply innards. Note that unlike [7], our restriction is for presentation only. Throughout the section, we use the following running example.

Example 5 Let w, x, y, z be latches and i be an input. Let

$$\begin{aligned} \mathit{Init} &\triangleq w \wedge x \wedge y \wedge z \\ \mathit{Tr} &\triangleq (w' = \neg w) \wedge (x' = w) \wedge (y' = w) \wedge \\ &\quad (g = x \wedge y) \wedge (h = g \wedge i) \wedge (z' = h) \end{aligned}$$

This design has two gates: $g = x \wedge y$ and $h = g \wedge i$, where g is input-free and h depends on the input i. Hence, the set of (input-free) innards is $\{g\}$. □

We extend IC3 to reason about innards in the initial state and the next state. To this end, let $\mathit{Tr}_{\mathit{inn}}$ be the part of the transition relation that defines innards, and let $\widehat{\mathit{Init}} \triangleq \mathit{Init} \wedge \mathit{Tr}_{\mathit{inn}}$ and $\widehat{\mathit{Tr}} \triangleq \mathit{Tr} \wedge \mathit{Tr}_{\mathit{inn}}'$. In Example 5,

$$\mathit{Tr}_{\mathit{inn}} = (g = x \wedge y) \quad \widehat{\mathit{Init}} = \mathit{Init} \wedge (g = x \wedge y)$$
$$\widehat{\mathit{Tr}} = \mathit{Tr} \wedge (g' = x' \wedge y')$$

where g' is a copy of g in "the next state". The following definition extends relative induction [1] to lemmas over latches and innards.

[1] https://github.com/agurfinkel/innard-benchmarks.

```
Input: Frame $k$, Lemma $C$ over latches, s.t. $C$ is
       inductive relative to $F_k$
Output: Lemma $C_2$ over latches and innards, s.t. $C_2$ is
        inductive relative to $F_k$
$C_1 \leftarrow$ ExtendLemma($C$)
$C_2 \leftarrow$ InductivelyGeneralize($k, C_1$)
return $C_2$
```

Fig. 1. Procedure LearnAdditionalLemma.

Definition 1 A lemma C over latches and innards is inductive relative to a set of lemmas G *iff* (i) $\widehat{Init} \Rightarrow C$, and (ii) $G \wedge \widehat{Tr} \wedge C \Rightarrow C'$.

Def. 1 generalizes the original definition: if a lemma C over latches is relatively inductive in the original sense of [1], then C is also relatively inductive by Def. 1. In what follows, by *relatively inductive*, we always mean Def. 1. Continuing our running example, let $C = (w \vee x)$ (note that C is over latches), and $C_1 = (w \vee x \vee g)$ (note that C_1 is over latches and innards). Then, both C and C_1 are inductive relative to $G = \top$. Note that $\widehat{Init} \Rightarrow C$, $\top \wedge \widehat{Tr} \wedge C \Rightarrow C'$, $\widehat{Init} \Rightarrow C_1$, $\top \wedge \widehat{Tr} \wedge C_1 \Rightarrow C'_1$ hold.

The following lemma shows that using relatively inductive (in the sense of Def. 1) lemmas in IC3 is sound.

Lemma 1 (Soundness) *For any lemma C over latches and innards, if $\widehat{Init} \Rightarrow C$ and $F_k \wedge \widehat{Tr} \wedge C \Rightarrow C'$ hold, then C includes $R_{\leq k+1}$ (all the states reachable in up to $k+1$ steps from Init). In particular, C can be added to IC3's inductive trace up to the frame $k+1$.*

Our approach of learning lemmas over innards is a form of inductive generalization. Each time that IC3 blocks a proof obligation and learns a (relatively inductive) lemma over latches, we generalize it into an (additional) lemma over latches and innards. The overall algorithm LearnAdditionalLemma is shown in Fig. 1. We give a high-level overview of LearnAdditionalLemma, while the details of key functions are described in later sections. The approach consists of two steps:

Step 1: The procedure ExtendLemma extends lemma C (over latches) to a lemma $C_1 = C \vee C_0$ (over latches and innards) such that $Tr_{inn} \Rightarrow (C \Leftrightarrow C_1)$, i.e. C and C_1 are equivalent modulo Tr_{inn}. The details are in section IV-B. For instance, in our example lemmas $C = (w \vee x)$ and $C_1 = (w \vee x \vee g)$ are equivalent, given that $g = x \wedge y$. Indeed, modulo Tr_{inn}: $(w \vee x \vee g) \equiv (w \vee x \vee (x \wedge y)) \equiv (w \vee x)$. It also follows (see Lemma 1) that C_1 remains relatively inductive.

Step 2: The procedure InductivelyGeneralize inductively generalizes C_1 by removing literals, while prioritizing removal of latches (the original literals of C), and more generally trying to leave only the "intereresting" innards. The details are in section IV-C. In our example, lemma $C_1 = (w \vee x \vee g)$ can be generalized to $C_2 = (w \vee g)$.

By construction, it follows that C_2 remains inductive relative to F_k. Moreover, as $Tr_{inn} \Rightarrow (C \Leftrightarrow C_1)$, and $C_2 \Rightarrow C_1$, then C_2 is potentially stronger than the original lemma C (but the converse might not hold). In our example, $C_2 = (w \vee g)$ is equivalent to $(w \vee (x \wedge y)) = (w \vee x) \wedge (w \vee y)$, i.e. the lemma C_2 over latches and innards represents two different lemmas over latches only. It is also interesting to note that while the original lemma C was over latches $\{w, x\}$, the "additional" lemma $(w \vee y)$ is over a different set of latches $\{w, y\}$.

Whenever ExtendLemma does not add any innards to C, the procedure LearnAdditionalLemma stops immediately, without calling InductivelyGeneralize. However, note that even when ExtendLemma adds new literals, it is possible that InductivelyGeneralize removes them, resulting in the original lemma C! When LearnAdditionalLemma returns a lemma C_2 that is different from C, C_2 is also added to IC3's inductive trace (up to frame F_{k+1}), and hence is also used in future predecessor and pushing queries.

B. Extending lemmas with innards

The procedure ExtendLemma receives a lemma C over latches as input and returns a lemma C_1 over latches *and* innards as output. It iteratively finds innards z such that $Tr_{inn} \Rightarrow (z \Rightarrow C)$ and replaces C with $C \vee z$. It works as follows: instead of searching for an innard z that implies C, it searches for all innards $\neg z$ that are implied by $\neg C$ and take their negations. Specifically, given a lemma $C = (c_1 \vee \cdots \vee c_m)$, we set each $c_i \in C$ to 0 and find which innards are implied by constant propagation in the Tr_{inn} part of the netlist. The algorithm for constant propagation in a netlist is standard and is not presented here.

Going back to our running example, given a lemma $C = (w \vee x)$, we are looking for innards implied by the partial assignment $(w = 0) \wedge (x = 0)$. Since $g = x \wedge y$, by propagation we obtain that $g = 0$. Thus, modulo Tr_{inn}, $g \Rightarrow C$, and hence C is equivalent to $(C \vee g) = (w \vee x \vee g)$. Note that by not considering input-free innards only (recall, we consider only input-free innards for simplicity of presentation), then, by propagation, we would also obtain that $h = (g \wedge i) = 0$. This would allow us to extend C to $(C \vee g \vee h) = (w \vee x \vee g \vee h)$. The following lemma follows by construction.

Lemma 2 *Given lemma C over latches, the procedure* ExtendLemma *returns a lemma C_1 over latches and innards such that $Tr_{inn} \Rightarrow (C_1 \Leftrightarrow C)$.*

Corollary 1 *Let C and C_1 be lemmas over latches and innards respectively, such that (i) C is inductive relative to some G, and (ii) $Tr_{inn} \Rightarrow (C_1 \Leftrightarrow C)$. Then, C_1 is also inductive relative to G.*

We remark that extending lemmas with literals that imply it is closely related to *asymmetric literal addition* [16] in SAT. We also remark that the condition that the original lemma C is over latches is not essential, and ExtendLemma can be used to extend lemmas that already have innards in them. This may be potentially useful for additional IC3 extensions.

```
Input: Frame $k$, lemma $C$ over latches and innards, s.t.
       $C$ is inductive relative to $F_k$
Output: (Inductively generalized) lemma $C_2 \subseteq C$ over
        latches and innards, s.t. $C_2$ is inductive relative
        to $F_k$
 1 $C \leftarrow$ SortLemma($C$) // $C = \{c_1, \ldots, c_n\}$
 2 for $i = 1, \ldots, n$ do
 3     if $c_i$ has already been removed from $C$ then
           // do nothing
 4     else if $Tr_{inn} \Rightarrow ((C \setminus c_i) \Leftrightarrow C)$ then
 5         $C \leftarrow C \setminus c_i$
 6     else if $\widehat{Init} \Rightarrow C \setminus c_i$ and
       $F_k \wedge \widehat{Tr} \wedge (C \setminus c_i) \Rightarrow (C \setminus c_i)'$ then
 7         $C \leftarrow C \setminus c_i$
 8         for $j = i+1, \ldots, n$ do
 9             if $c_j$ not used in the above proofs then
10                 $C \leftarrow C \setminus c_j$
11             else
12                 break
13 return $C$
```

Fig. 2. Procedure InductivelyGeneralize: inductively generalizes lemmas over latches and innards.

C. Inductively generalizing lemmas with innards

Inductive generalization in traditional IC3 starts with a relatively inductive lemma C over latches (satisfying the conditions $Init \Rightarrow C$ and $F_k \wedge Tr \wedge C \Rightarrow C'$ with respect to a given frame F_k), and attempts to remove literals from C as long as C remains relatively inductive. The same procedure can be immediately applied to a lemma over latches and innards, once $\widehat{Init}$ and $\widehat{Tr}$ are used instead of $Init$ and Tr, respectively. However, we found that a naive application of inductive generalization gives poor results. In most cases, it simply removes the innards that were previously added by ExtendLemma, and therefore, ends up with the original lemma over latches. Moreover, regular inductive generalization does not exploit possible dependencies between innards.

Fig. 2 shows a variant of inductive generalization that is better suited for generalizing lemmas over innards. The first step (line 1), consists of sorting the nets in the lemma, from the nets that we want to remove most to the nets that we want to remove least. In particular, we want to prioritize removal of latches, so as to obtain a different lemma that we started with. In our current implementation, we sort the nets by their *logic level*, so that latches have the lowest level and deeper nets in general have higher level. This way deeper nets are considered "more interesting" and the algorithm attempts to remove shallower nets first. Other heuristics can be considered as well, e.g., sorting the nets by the *size of the supporting logic*, or even dynamic heuristics that measure the *activity* of a net in previously generalized lemmas.

The main loop (lines 3–12) corresponds to inductive generalization in regular IC3: essentially, we remove literals of C one by one, as long as C remains relatively inductive. We provide a detailed description of one iteration of the loop. Suppose that c_i is the literal under consideration.

1) Note that *multiple* literals can be removed from C in a single iteration of the loop (this optimization is also present in regular IC3 inductive generalization), so at the start of the iteration (line 3), we check if c_i has already been removed. If so, nothing needs to be done.

2) Lines 4–5 correspond to a special optimization that exploits dependencies between innards: in some cases, we can detect that c_i can be removed without requiring a SAT query. For instance, c_i can be removed when one of the following conditions holds:

(i) $c_i = a \wedge b$, with $a \in C$,

(ii) $c_i = a \vee b$, with $a, b \in C$, or

(iii) there is an innard $d \in C$ with $d = c_i \vee b$.

For example, suppose that $C = (a \vee c \vee d)$ and $\{d = (b \vee c)\} \in Tr_{inn}$. Then, modulo Tr_{inn}, $C \Leftrightarrow (C \setminus c)$, i.e. $(a \vee c \vee d)$ can be replaced by $(a \vee d)$. This closely corresponds to *hidden literal elimination* technique in SAT [17], and can be viewed as the inverse of the argument used in ExtendLemma.

3) Line 6 checks whether c_i can be removed using two SAT-queries. One query checks the validity of $\widehat{Init} \Rightarrow (C \setminus c_i)$, by checking whether $\widehat{Init} \wedge \neg(C \setminus c_i)$ is unsatisfiable. The other query checks the validity of $F_k \wedge \widehat{Tr} \wedge (C \setminus c_i) \Rightarrow (C' \setminus c_i')$ by checking whether $F_k \wedge \widehat{Tr} \wedge (C \setminus c_i) \wedge \neg(C' \setminus c_i')$ is unsatisfiable. If both of these queries are unsatisfiable, c_i can be removed.

4) IC3 has the following standard optimization based on considering which of the literals of $(C \setminus c_i)$ were potentially required for unsatisfiability: if $c_j \in C$ was not required for either checks, then c_j can be removed. This is typically implemented by passing the literals of $\neg(C \setminus c_i)$ via SAT *assumptions* and analyzing the set of *conflicting assumptions*; a mechanism supported by most modern SAT-solvers, following MINISAT [18]. However, simply removing all non-required literals regardless of their order in C is more likely to remove the "more interesting" literals that we want to keep. So, our variant of this optimization (lines 8–12) only removes non-required literals with respect to the order. As an example, suppose that $C = (c_1 \vee c_2 \vee c_3 \vee c_4 \vee c_5 \vee c_6)$ (in this order), and that only the literals c_4 and c_6 were potentially required for unsatisfiability queries involving $C \setminus c_1$. In addition to removing c_1, we also remove c_2 and c_3, but not c_5, and at the end of the iteration of the loop, $C = (c_4 \vee c_5 \vee c_6)$. Intuitively, this works better because leaving c_5 in the lemma increases the chances to remove c_5 and to leave c_6 (and not vice versa) on the following iterations of the loop. Lastly, in most cases an assumption-based SAT-solver applies assumptions in the order as they are given, hence, the assumptions appearing earlier are more likely to remain (while later assumptions are more likely to be removed). Therefore, when performing the SAT queries, we *reverse* the order of assumption literals, for instance when checking whether c_1 can be removed from $C = (c_1 \vee c_2 \vee c_3 \vee c_4 \vee c_5 \vee c_6)$, the assumptions are ordered from c_6 to c_2 (and not from c_2 to c_6).

Note that during the regular inductive generalization (i.e., when computing the original lemma over latches) it is beneficial to make multiple passes over the main loop (lines 3–12). However, when generalizing lemmas over innards, performing multiple passes has not proven to be useful, so we only perform a single pass.

Lemma 3 *Given a lemma C_1 over latches and innards, the* `InductivelyGeneralize` *procedure returns a lemma C_2 that is relatively inductive with respect to F_k.*

Going back to our running example, suppose that $C_1 = (w \lor x \lor g)$ is inductive relative to $F_k = \top$. The procedure `SortLemma` is not likely to change the order of nets, as the latches already appear first. On the first iteration of the main loop, we attempt to remove w, but this fails as the SAT query $\top \land \widehat{Tr} \land (x \lor g) \land \neg x' \land \neg g'$ is satisfiable. On the second iteration, we attempt to remove x, and succeed, reducing C_1 to $(w \lor g)$. Finally, we attempt to remove g, which again fails. The final lemma returned by the algorithm is $C_2 = (w \lor g)$.

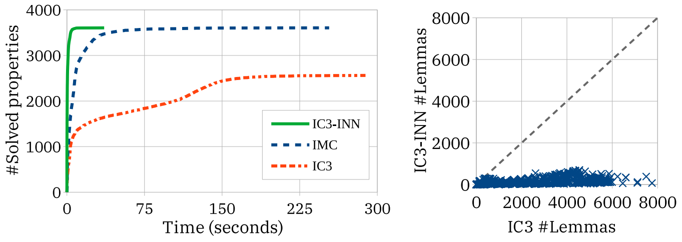
V. Experiments

In this section, we present our experimental results. The techniques described in this paper are implemented in the IBM formal verification tool *Rulebase: Sixthsense Edition* [19]. In what follows, we denote by IC3 the default variant of IC3 used by the tool (see [6]), and by IC3-INN the variant with the additional learning of lemmas over innards. For these experiments, we restrict to input-free innards. Table I summarizes the experiments. The table contains the benchmark set (explained in detail later), the number of instances in this set, time-limit per instance, and the data on performance of IC3 and IC3-INN. All the instances either are or expected to be unsatisfiable. For both IC3 and IC3-INN, we list the number of solved instances, and in parentheses – the number of uniquely solved instances (that is, not solved by the other configuration), and the cumulative runtime in seconds. Next, we describe each benchmark set in detail.

A. IBM-AOD-SEC

This set of benchmarks comes from checking sequential equivalence between two designs in the Aspect Oriented Design flow at IBM. This SEC problem is very challenging, and is traditionally solved as described in [8], [9], using speculative reduction to reduce the problem into multiple simpler (but still hard) sub-problems. These are then solved using a dedicated engine configuration consisting of combinational rewriting, k-induction, localization, and, eventually, a proof-based technique like IC3. Historically, Interpolation (IMC) was used for the final step. Generally IMC works well, but unfortunately, it's not stable – small changes in the design or in the solving configuration significantly affect verification times. While trying to find an alternative configuration, it was discovered that IC3 performs very poorly, while IC3-INN significantly outperforms all other approaches.

In total, there are 3 605 sub-problems. Each sub-problem contains 1–45 properties, 11–165 state elements, 126–2 290 inputs, and 754–15 924 gates. The (input-free) innards on average constitute 3% of the gates. For this experiment, we run both IC3 and IC3-INN with a time-limit of 300 seconds per problem. Referring to Table I, regular IC3 peforms very poorly: it can solve only 2 562 of the sub-problems and times out in the 1 043 remaining cases. On the other hand, IC3-INN performs extremely well: it can solve all of the problems, with the maximum run-time being only 36 seconds. Interestingly, IMC performs much better than IC3 on this set of problems and is also able to solve all problems (albeit about 13 times slower than IC3-INN). See the cactus plot in Fig. 3a for the detailed comparison between IC3, IC3-INN, and IMC.



(a) Solved vs. Runtime

(b) Invariant size

Fig. 3. Performance of IC3 and IC3-INN on AOD SEC benchmarks.

A further comparison consists of comparing the number of lemmas in the safe inductive invariants discovered by IC3 and IC3-INN respectively. The scatter plot Fig. 3b shows this data for the 2 562 instances solved by both configurations. We can see that IC3-INN discovers invariants that are significantly more compact, with the inductive invariants discovered by IC3-INN being on average 12× smaller than the invariants discovered by IC3. This partially explains the success of IC3-INN compared to IC3 on this set of benchmarks.

We also give data on the effectiveness of `LearnAdditionalLemma`, averaged across all 3 605 test-cases. On average, the original lemma C (over latches) has 7 latches; `ExtendLemma` adds 10 innards; `InductivelyGeneralize` shrinks the lemma to 2 latches and 1 innards. The average logic level of innards is 7. Thus, `LearnAdditionalLemma` is able to produce significantly shorter lemmas using deep innards in the design.

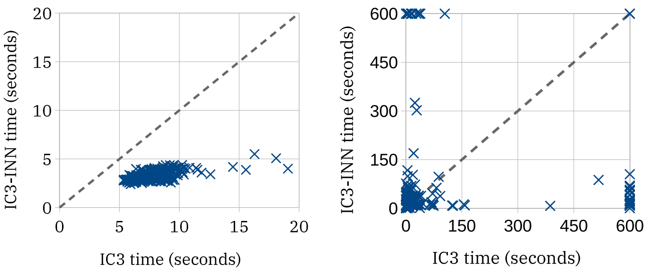
Unfortunately, this benchmark set is proprietary and cannot be publicly released at this time.

B. 6s119-SEC, 6s22-SEC

Inspired by the success of IC3-INN on internal IBM benchmarks, we tried to manually create similar test-cases starting from publicly available benchmarks. Specifically, we have taken several HWMCC designs, and created problems to check sequential equivalence between the original design and the retimed design [13]. We have further applied the SEC flow described above, consisting of breaking the main problem into multiple sub-problems using speculative reduction. It turns out that creating interesting benchmark sets in this way is non-trivial: in many cases the speculatively reduced problems turn out to be very easy, in many other cases some of these speculatively reduced problems turn out to be satisfiable (in

TABLE I
Summary of Experimental Results

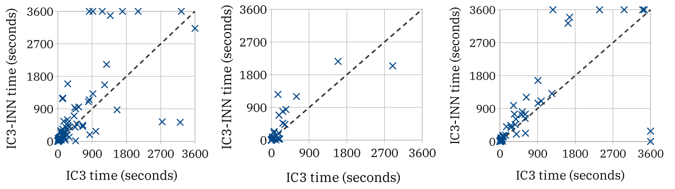
benchmarks	#instances	time-limit per instance	IC3 solved (unique)	IC3 time	IC3-INN solved (unique)	IC3-INN time
IBM-AOD-SEC	3 605	300	2 562 (0)	424 885	3 605 (1 043)	**2 465**
6s119-SEC	364	600	364 (0)	2 906	364 (0)	**1 207**
6s22-SEC	310	600	262 (22)	32 701	278 (38)	**24 774**
AES-SEC	16	3 600	13 (0)	11 186	15 (2)	**5 601**
HWMCC11	278	3 600	277 (6)	**40 186**	272 (1)	55 557
HWMCC17	76	3 600	76 (0)	**7 963**	76 (0)	11 221
HWMCC20	192	3 600	190 (5)	**35 907**	187 (2)	41 448



(a) 6s119-SEC

(b) 6s22-SEC

Fig. 4. Runtime of IC3 and IC3-INN on 6s119-SEC and 6s22-SEC.



(a) HWMCC'11

(b) HWMCC'17

(c) HWMCC'20

Fig. 5. Runtime of IC3 and IC3-INN on HWMCC benchmarks.

the real SEC flow this would trigger refinement and another speculative reduction). Nevertheless, we have created two benchmark sets 6s22-SEC and 6s119-SEC, available at https://github.com/agurfinkel/innard-benchmarks. The set 6s119-SEC consists of 364 rather easy problems, so that both IC3 and IC3-INN can solve all of them within 600 seconds, with IC3-INN being about 2.4× faster. The set 6s22-SEC consists of 310 problems, out of which IC3 can solve 262 problems and IC3-INN can solve 278 within 600 seconds. Please refer to Table I. Again, IC3-INN performs better than IC3, and is on average 1.3× faster. A more precise comparison is given in scatter plots in Fig. 4. A detailed comparison against IMC is not included as on both sets of problems IMC performs significantly worse than either IC3 or IC3-INN (for instance, within 600 seconds IMC cannot solve 64 out of 364 problems even for the easy set 6s119-SEC).

C. Other SEC benchmarks; AES-SEC

As far as we know, there are no publicly available large SEC benchmark sets. HWMCC competitions do include several SEC benchmarks. However, in general we do not know which benchmarks come from SEC or what kind of application they represent. We believe it would be valuable to have a dedicated repository for SEC benchmarks.

The AES-SEC benchmark set was used in [14]. We have obtained this set from the authors of [14] in BTOR format, and translated it to AIGER. The AIGER benchmarks are available at https://github.com/agurfinkel/innard-benchmarks. In total, there are 16 problems, 12 of which turn out to be very easy for both IC3 and IC3-INN. Out of the 4 remaining problems, IC3 can solve 1, and IC3-INN can solve 3. Please see Table I for details.

D. HWMCC benchmarks

We have run extensive experiments on the single-property benchmarks from HWMCC'11, HWMCC'17 and HWMCC'20 competitions (for the latter, we used the benchmarks in the AIGER format). In each case, we run simple combinational reductions prior to running IC3, and used the time-limit of 3 600 seconds. In Table I, we only report data for passing benchmarks that were solved either by IC3 or IC3-INN. In general, IC3-INN performs worse than IC3 both in terms of the number of properties solved and the total runtime. Detailed comparisons are presented as scatter plots in Fig. 5.

Table II presents data for 4 selected benchmarks. The benchmark *rast-p16* is very interesting: regular IC3 times out, yet IC3-INN solves the testcase in just 2 seconds. Futhermore, this benchmark was solved by relatively few tools in the HWMCC'20 competition. By closely examining the lemmas learned by IC3-INN exposed the pattern from Example 4 from Section III. In other words, IC3-INN learns lemmas over innards, each equivalent to a very large number of lemmas over latches. This potentially explains the success of IC3-INN in this case. Another noteworthy benchmark is *zipversa_composecrc_prf-p10*, which IC3-INN solves under 5 minutes, and which was solved only by one tool in the HWMCC'20 competition. The other two benchmarks exposed a certain inefficiency in our current implementation of IC3-INN. One can check that there are significantly more innards in the selected test-cases (and in HWMCC test-cases in general) as compared to IBM-AOD-SEC designs. The procedure `InductivelyGeneralize` starts taking a significant portion of the overall runtime, which negatively

TABLE II
SELECTED DESIGNS FROM HWMCC'20

Benchmark	#gates	#innards	IC3 time	IC3-INN time
rast-p16	3 019	332	timed-out	**2**
zipversa...prf-p10	1 688	694	timed-out	**282**
h_RCU	920	442	**3 410**	timed-out
dspfilters_fastfir...p45	21 301	5 289	**2 381**	timed-out

affects performance of IC3 when the lemmas over innards do not seem to help.

VI. RELATED AND FUTURE WORK

The technique presented in this paper can be viewed as an extension of regular IC3 that simply learns an additional lemma during inductive generalization. As such, it is reasonably easy to integrate it in an existing IC3 implementation. The main technical point being replacing $Init$ by $\widehat{Init}$ and Tr by $\widehat{Tr}$ in IC3's SAT queries. The key difference with other inductive generalization schemes (see for instance [3]) is that we are able to learn lemmas over both state variables and internal nets, which, in some cases, may exponentially reduce the size of the inductive invariant.

Backes and Riedel [7] also exploit internal nets in the design. However, the two approaches are very different: [7] uses input-free innards to generalize proof obligations (POBs), while we use arbitrary innards to generalize lemmas. Additionally, [7] uses only *input-free* innards (and, in fact, only the nets on the *boundary* between input-free and non input-free parts of the netlist), while we use all internal nets. Even more importantly, in our work the decision of which innards to include in the lemma was based on the ability to inductively generalize this lemma and not whether the innards are "boundary" or not. Above notwithstanding, it is interesting to combine the two approaches, i.e., to allow both proof-obligations and lemmas over internal nets. It is also interesting to more carefully integrate our approach with `Quip` [6]. `Quip` uses negations of lemmas as proof obligations, which would also introduce innards into POBs.

Another very interesting direction for further research is to extend the approach to learn lemmas over signals that are not present in the original netlist. Our framework allows such an extension: by including additional logic into the netlist (that is, creating additional innards), we would be able to learn lemmas over this new logic (even if this new logic is not in the cone-of-influence of the original problem!). This is closely related to implicit predicate abstraction of Tonetta et al. [20] that is used to lift propositional IC3 to SMT-based logics.

Finally, we believe that there is a lot of room to improve the current implementation. Currently, when there are many innards in the design, the procedure `InductivelyGeneralize` may require a large number of SAT queries, and, hence, may take a considerable portion of the overall runtime. Possibly, one can find better heuristics of which innards to consider (e.g., only to consider innards with high logic level, or only to consider *higher-priority* innards), or find more efficient procedures to perform inductive generalization (e.g., instead of the top-down approach that removes literals one can consider a bottom-up approach that adds literals). In the worst-case, if learning additional lemmas takes a considerable amount of time, but does not seem useful, the technique can be simply turned off.

A further extension of our approach is to allow lemmas to be arbitrary formulas, not restricted to clauses in CNF. This is commonly done in SMT-based extensions of IC3 algorithms. For example, Sally [21] uses arbitrary SMT-formulas as lemmas, and Spacer [22] uses clauses over complex First Order signature. However, these techniques are difficult to port efficiently in the context of Hardware Model Checker since they rely on dynamic cnfization that is common in SMT-solvers but not in SAT-solvers.

VII. CONCLUSION

Currently, IC3 is unquestionably the most effective technique for formal symbolic model checking. It has received a lot of research attention, and has been extended in variety of ways including better inductive generalization, better lemma management, and search direction. However, one significant hidden limitation remains – IC3 is limited to learning inductive invariants in CNF over the latches (i.e., state variables) of the design. Therefore, IC3 cannot be effective for any design whose invariant has no concise CNF representation. No improvements in core IC3 parts can solve this problem.

In this paper, we propose to address this limitation by extending IC3 to learn lemmas not only over latches, but also over internal signals, that we call *innards*. We show learning lemmas over innards is a natural generalization of *inductive generalization*. Instead of simply dropping literals to strengthen the lemma, we propose to replace literals by internal signals that are forced by them. We also propose several improvements to a naive strategy that lead to significantly improved performance.

Our work is motivated by a specialized set of Sequential Equivalence Checking (SEC) benchmarks at IBM. These benchmarks have been traditionally difficult for IC3, but not for Interpolation (IMC). However, the performance of interpolation was not stable – being affected by small changes in the verification flow. Our new implementation excels on these benchmarks and leads to an order of magnitude improvement in performance.

Unfortunately, similar performance gains do not manifest on the publicly available HWMCC benchmarks that are the de-facto metric for academic model checking research. We believe this shows deficiency in the currently available benchmarks. Techniques that might be effective in industry might be missed by researchers since they do not perform well on these benchmarks. To remedy this, we identified some publicly available benchmarks, and created new benchmarks based on SEC flow, that illustrate the advantage of our technique. We hope this can stimulate further research and improvements to IC3.

In the current work, we assume that the design is fixed, and use internal signals that are already available. We think that this opens an interesting direction by allowing IC3 to change the design by synthesizing new innards that are useful for a current verification run. This brings IC3 and interpolation much closely together, and also paves way for bringing algorithms from hardware verification to software verification, and/or to word level.

Acknowledgments

The authors would like to thank Jason Baumgartner, Robert Kanzelman, Raj Kumar Gajavelly, Ziv Nevo, Hongce Zhang, Sharad Malik, Alan Mishchenko, and Baruch Sterin. This work was supported, in part, by Individual Discovery Grant from the Natural Sciences and Engineering Research Council of Canada and IBM Faculty Fellowship.

References

[1] A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 70–87. [Online]. Available: https://doi.org/10.1007/978-3-642-18275-4_7

[2] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 125–134. [Online]. Available: http://dl.acm.org/citation.cfm?id=2157675

[3] A. Griggio and M. Roveri, "Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 35, no. 6, pp. 1026–1039, Jun 2016.

[4] Y. Vizel, O. Grumberg, and S. Shoham, "Lazy abstraction and sat-based reachability in hardware model checking," in *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, G. Cabodi and S. Singh, Eds. IEEE, 2012, pp. 173–181. [Online]. Available: http://ieeexplore.ieee.org/document/6462570/

[5] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in IC3," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 157–164. [Online]. Available: http://ieeexplore.ieee.org/document/6679405/

[6] A. Gurfinkel and A. Ivrii, "Pushing to the top," in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, R. Kaivola and T. Wahl, Eds. IEEE, 2015, pp. 65–72.

[7] J. D. Backes and M. D. Riedel, "Using cubes of non-state variables with property directed reachability," in *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, E. Macii, Ed. EDA Consortium San Jose, CA, USA / ACM DL, 2013, pp. 807–810. [Online]. Available: https://doi.org/10.7873/DATE.2013.171

[8] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations," in *24th International Conference on Computer Design (ICCD 2006), 1-4 October 2006, San Jose, CA, USA*. IEEE, 2006, pp. 259–266. [Online]. Available: https://doi.org/10.1109/ICCD.2006.4380826

[9] H. Mony, J. Baumgartner, A. Mishchenko, and R. K. Brayton, "Speculative reduction-based scalable redundancy identification," in *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, L. Benini, G. D. Micheli, B. M. Al-Hashimi, and W. Müller, Eds. IEEE, 2009, pp. 1674–1679. [Online]. Available: https://doi.org/10.1109/DATE.2009.5090932

[10] R. Brayton, N. Een, and A. Mishchenko, "Using speculation for sequential equivalence checking," in *21st International Workshop on Logic & Synthesis, IWLS 2012*, 2012.

[11] R. Dureja, J. Baumgartner, R. Kanzelman, M. Williams, and K. Y. Rozier, "Accelerating Parallel Verification via Complementary Property Partitioning and Strategy Exploration," in *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*. Haifa, Israel: IEEE/ACM, Sep. 2020.

[12] R. Dureja, J. Baumgartner, A. Ivrii, R. Kanzelman, and K. Y. Rozier, "Boosting Verification Scalability via Structural Grouping and Semantic Partitioning of Properties," in *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*. San Jose, California: IEEE/ACM, Oct. 2019. [Online]. Available: https://doi.org/10.23919%2Ffmcad.2019.8894265

[13] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Springer, 2001, pp. 104–117. [Online]. Available: https://doi.org/10.1007/3-540-44585-4_10

[14] H. Zhang, W. Yang, G. Fedyukovich, A. Gupta, and S. Malik, "Synthesizing environment invariants for modular hardware verification," in *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*, ser. Lecture Notes in Computer Science, D. Beyer and D. Zufferey, Eds., vol. 11990. Springer, 2020, pp. 202–225. [Online]. Available: https://doi.org/10.1007/978-3-030-39322-9_10

[15] T. Seufert and C. Scholl, "Sequential verification using reverse PDR," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2017, Bremen, Germany, February 8-9, 2017*, D. Große and R. Drechsler, Eds. Shaker Verlag, 2017, pp. 79–90.

[16] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," in *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364. Springer, 2012, pp. 355–370. [Online]. Available: https://doi.org/10.1007/978-3-642-31365-3_28

[17] M. Heule, M. Järvisalo, and A. Biere, "Efficient CNF simplification based on binary implication graphs," in *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, ser. Lecture Notes in Computer Science, K. A. Sakallah and L. Simon, Eds., vol. 6695. Springer, 2011, pp. 201–215. [Online]. Available: https://doi.org/10.1007/978-3-642-21581-0_17

[18] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518. [Online]. Available: https://doi.org/10.1007/978-3-540-24605-3_37

[19] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, ser. Lecture Notes in Computer Science, A. J. Hu and A. K. Martin, Eds., vol. 3312. Springer, 2004, pp. 159–173. [Online]. Available: https://doi.org/10.1007/978-3-540-30494-4_12

[20] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "IC3 modulo theories via implicit predicate abstraction," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 46–61. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_4

[21] D. Jovanovic and B. Dutertre, "Property-directed k-induction," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, R. Piskac and M. Talupur, Eds. IEEE, 2016, pp. 85–92.

[22] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-Based Model Checking for Recursive Programs," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 17–34.