Making PROGRESS in Property Directed Reachability

Tobias Seufert[1(✉)], Christoph Scholl[1], Arun Chandrasekharan[2], Sven Reimer[2], and Tobias Welp[2]

[1] University of Freiburg, Freiburg im Breisgau, Germany
{seufert,scholl}@informatik.uni-freiburg.de
[2] OneSpin Solutions, Munich, Germany
{arun.chandrasekharan,sven.reimer,tobias.welp}@onespin.com

Abstract. With *Proof-Guided Restriction Skipping* (PROGRESS) we present a fully automatic and complete approach for Hardware Model Checking under restrictions. We use the PROGRESS approach in the context of PDR/IC3 [9,18]. Our implementation of PDR/IC3 restricts input signals as well as state bits of a circuit to constants in order to quickly explore long execution paths of the design. We are able to identify spurious proofs of safety along the way and exploit information from these proofs to guide the relaxation of the restrictions. Hence, we greatly improve the capability of PDR to find counterexamples, especially with long error paths. In experiments with HWMCC benchmarks our approach is able to double the amount of detected deep counterexamples in comparison to Bounded Model Checking as well as in comparison to PDR.

1 Introduction

Lately, there have been many advances in the field of safety verification of sequential circuits. With modern solvers for the Boolean satisfiability problem (SAT), especially SAT-based Model Checking has become more and more popular. However, formal verification of systems with large state spaces remains a challenging problem.

A popular approach to counteract growing state spaces is by *abstraction* and abstraction refinement such as Counterexample-Guided Abstraction Refinement (CEGAR) [15,28,42]. This means that the behaviour of the circuit is overapproximated. For instance, variables representing the state of storage elements are handled as user inputs, disconnecting them from their transition function (called localization abstraction [42]). As a result, the underlying problem gets less complex and the state space is reduced. However, abstraction comes with the drawback of incompleteness. Over-approximating the behaviour of a circuit may lead to spurious counterexamples which are not valid in the original system. Proofs of safety though are also correct under over-approximation. CEGAR can

© Springer Nature Switzerland AG 2022
B. Finkbeiner and T. Wies (Eds.): VMCAI 2022, LNCS 13182, pp. 355–377, 2022.
https://doi.org/10.1007/978-3-030-94583-1_18

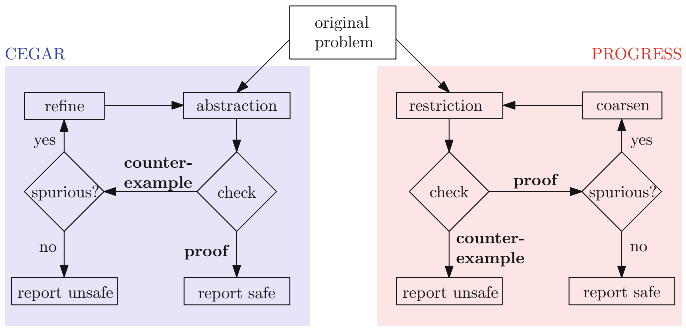

Fig. 1. Comparing CEGAR and PROGRESS.

be used to recover completeness: Based on the analysis of spurious counterexamples, the abstraction is subsequently refined until it terminates with a correct result.

In contrast to abstraction, it is also possible to introduce a *restriction* and under-approximate the behaviour of the system under verification. For instance, we may assume primary inputs of the sequential circuit as constants as well as consider only transitions from/to states with some latches fixed to constants. Intuitively, this makes sense if an engineer has prior knowledge of the system and only wants to consider parts of the system under some special control signals. A simple example could be the verification of the multiplier unit of an ALU - the engineer would restrict its control signal to 'multiply'. Another example is the search for a counterexample to the correctness of a processor with a pipelined multiplier. If a counterexample exists that does not make use of multiplication, then the internal pipeline registers of the multiplier can be fixed to constants without compromising the possibility to find such a counterexample. In practice, restriction requires a deep understanding of the circuit and the verification technology, and is usually custom made for the given system only. A counterexample found for an under-approximated system behavior is also valid wrt. the original system. However, a proof of safety may be spurious and incomplete. In general, abstraction and restriction are complementary techniques. While abstraction techniques mainly aim to improve the capabilities of finding a proof of safety, restrictions focus on certain parts of the system behaviour only and enable the examination of long error paths.

In this work, we pick up the idea of restrictions. Instead of restricting specific signals based on prior knowledge, we present a *fully automatic approach* that can be applied to any given circuit. We start with stringent restrictions to the system behaviour. If we find a counterexample under these restrictions, it is valid. In the case that we find an incomplete proof due to the restrictions, we apply a technique which we call *Proof-Guided Restriction Skipping* (PROGRESS). PROGRESS can be considered as the dual of CEGAR. For the relation of PROGRESS and CEGAR see also the illustration in Fig. 1. By analyzing the proof, we deduce restrictions which we have to skip in order to continue. This process is repeated and we coarsen the restrictions in each iteration. The loop ends either with a counterexample or with a proof for the complete (unrestricted) system. In that way we provide a *complete* model checking algorithm. Our concrete realization of the PROGRESS approach is based on Property Directed Reachability (PDR) also known as IC3 [9,18]. We call our implementation PROGRESS-PDR.

Apparently, restrictions are not only applicable in PDR. We chose PDR though, because it is widely considered as the strongest unbounded and complete method in the field of safety verification of sequential circuits. PDR considers only single instances of the transition relation and produces a large number of small and easy SAT problems. In contrast, Bounded Model Checking (BMC) [3] considers unrollings of the transition relation leading to more and more expensive SAT problems with each added unrolling. In our experiments on Hardware Model Checking Competition (HWMCC) benchmarks, we show that *PROGRESS-PDR* greatly improves the ability of finding counterexamples in PDR in general. Furthermore, we observe that PROGRESS-PDR is superior to BMC in finding counterexamples with long error paths.

PROGRESS-PDR also performs better than standard PDR on some safe problem instances and achieves a better overall performance.

In summary, our contributions are as follows.

- We present a novel paradigm called PROGRESS which is the dual of CEGAR and skips restrictions based on the analysis of spurious proofs.
- We introduce a fully automatic and complete Model Checking algorithm, applying the new paradigm in the context of PDR, leading to the PROGRESS-PDR approach.
- Additionally, we give an insight on how restrictions (instead of abstractions) affect the inner workings of PDR.
- Finally, we show that PROGRESS-PDR is significantly stronger than BMC as well as original PDR in finding deep counterexamples.

Structure of the Paper. In Sect. 2 we discuss related work and in Sect. 3 we give some preliminaries needed for this paper. We define restrictions in the context of PDR in Sect. 4, our algorithm including a restriction skipping loop in Sect. 5, and further implementation details in Sect. 6. An experimental evaluation is given in Sect. 7, and Sect. 8 summarizes the results with directions for future research.

2 Related Work

Lately, there have been many efforts to improve the efficiency of PDR [2,23,26,36]. Some of these extensions to PDR make use of abstraction [1,7,24,41] and abstraction refinement such as Counterexample-Guided Abstraction Refinement (CEGAR) [15,28].

Apart from counterexamples, also proofs (or both) have been used to guide abstractions [17,30,31]. In those approaches, proofs of safety up to a particular bound (time frame) in BMC are exploited in order to find abstractions whereas spurious BMC counterexamples are used to refine the abstraction.

Compared to abstractions like localization abstraction [42], *restrictions* do not replace state bits (for instance) by free variables, but by constants, leading to simplifications of the transition relation by unit propagation. Exploiting restrictions is a typical method used in interactive bug hunting. However, to the best of our knowledge, restrictions have not been used so far in the context of a fully automatic and complete proof technique for verifying sequential circuits. Nevertheless, restrictions or under-approximations in general have already been used in various contexts related to bit-level hardware model checking. E.g. [32] defines both under- and over-approximations in the context of symbolic model checking for sequential circuits, but does not provide any refinement loop for manually chosen approximations. [16] considers a series of over- and under-approximations for state set collection as well as next state computation during model checking of real-time systems. The approximations become more and more precise until a proof using over-approximations or a refutation using under-approximations is found. The approximation techniques are tailored towards the computation of symbolic state set representations for timed systems and thus are not applicable in the context of hardware verification using SAT solving. Under-approximations as well as over-approximations were also considered for decision procedures for Presburger arithmetic [27] and for array and bit-vector theories [12,13]. Similar ideas have also been used in [11] for approximating floating-point operations in software verification.

Many of the approaches mentioned above use a refinement loop for approximations (as our approach). Whereas the CEGAR based approaches refine only over-approximations by different methods, [12,13,27] rely on an alternating generation of under-approximations (by bit-width restrictions relaxed by counterexamples from over-approximations) and over-approximations (derived from unsatisfiable solver calls for under-approximations). The refinement loop of [11] deviates from strict alternations of over- and under-approximations, but is restricted to floating-point operations. Our approach refines only under-approximations by restriction skipping, but it aims at (unbounded) safety verification of sequential circuits rather than solving combinational formulas as in [11–13,27]. In this context, we analyze possibly spurious inductive invariants in refinement steps skipping restrictions. From a technical point of view, this step is most closely related to the refinement step for under-approximations on the CNF layer introduced in [12].

3 Preliminaries

In the following, we introduce notations, the necessary background on finite state transition systems, and a basic review of PDR.

3.1 Basics and Notations

We discuss reachability analysis in finite state transition systems for the verification of invariant properties. In a finite state transition system we have a finite set of states and a transition relation which encodes transitions between states under certain inputs. States are obtained by assigning Boolean values to the (present) state variables $\vec{s} = (s_1, \ldots, s_m)$, inputs by assigning Boolean values to the input variables $\vec{i} = (i_1, \ldots, i_n)$. For representing transitions we introduce a second copy $\vec{s}'$ of the state variables, the so-called next state variables. The transition relation is then represented by a predicate $T(\vec{s}, \vec{i}, \vec{s}')$, the set of initial states by a predicate $I(\vec{s})$. The set of unsafe states are represented by a predicate $\neg P(\vec{s})$. For brevity, we often omit the arguments of the predicates and write them without parenthesis.

A *literal* represents a Boolean variable or its negation. *Cubes* are conjunctions of literals, *clauses* are disjunctions of literals. The negation of a cube is a clause and vice versa. A Boolean formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses. As usual, we often represent a clause as a set of literals and a CNF as a set of clauses. A cube $c = s_{i_1}^{\sigma_1} \wedge \ldots \wedge s_{i_k}^{\sigma_k}$ of literals over state variables with $i_j \in \{1, \ldots, m\}$, $\sigma_j \in \{0, 1\}$, $s_{i_j}^0 = \neg s_{i_j}$ and $s_{i_j}^1 = s_{i_j}$ represents the set of all states where s_{i_j} is assigned to σ_j for all $j = 1, \ldots, k$. We usually use letters c or $\hat{c}$ to denote cubes of literals over present state variables, d' or $\hat{d}'$ to denote cubes of literals over next state variables, and i to denote cubes of literals over input variables. By *minterms* (often named m) we denote cubes containing literals for *all* state variables. Minterms represent single states.

We assume that the transition relation T of a finite state transition system has been translated into CNF by standard methods like [40]. Modern SAT solvers [39] are able to check the satisfiability of Boolean formulas in CNF. Furthermore, SAT-based Model Checking heavily relies on *incremental* SAT solving [25]. Incremental SAT solvers allow for several queries on the same solver instance, reusing knowledge (e.g. conflict clauses) from previous runs. To each query so called *assumptions* can be added. These are literals which are conjoined to the solvers' internal CNF formula for exactly one query - and removed afterwards. In the case of an unsatisfiable solver call, most modern incremental SAT solvers are able to give a reason for the unsatisfiability, e.g. a so called *UNSAT-core* which contains a subset of the assumption literals which is sufficient to cause unsatisfiablity.

Reachability analysis (e.g. by PDR) often makes use of special properties of the transition relation T. E.g., when T results from a circuit, then it represents a *function*, i.e., it is *right-unique* and *left-total.* A relation $T(\vec{s}, \vec{i}, \vec{s}')$ is right-unique iff for all assignments $\vec{\sigma}$ to $\vec{s}$ and $\vec{\iota}$ to $\vec{i}$ there is *at most* one assignment $\vec{\sigma}'$ to $\vec{s}'$ such that $(\vec{\sigma}, \vec{\iota}, \vec{\sigma}') \in T$. $T(\vec{s}, \vec{i}, \vec{s}')$ is left-total iff for all assignments $\vec{\sigma}$ to $\vec{s}$ and $\vec{\iota}$ to $\vec{i}$ there is *at least* one assignment $\vec{\sigma}'$ to $\vec{s}'$ such that $(\vec{\sigma}, \vec{\iota}, \vec{\sigma}') \in T$.

3.2 An Overview of PDR

In this paper, we consider Property Directed Reachability (PDR) [18] (also called IC3 [9]). PDR produces stepwise reachability information in time frames without unrolling the transition relation as in Bounded Model Checking (BMC) [3]. Each time frame k corresponds to a predicate F_k represented as a set of clauses, leading

```
1 function Pdr($I$, $T$, $P$)
2     if BaseCases() = ‘Unsafe’ then return ‘Unsafe’
3     while $true$ do
4         if Strengthen() = ‘Unsafe’ then return ‘Unsafe’
5         $N \leftarrow N+1$, add new $F_N \leftarrow P$          /* New time frame. */
6         if Propagate() = ‘Safe’ then return ‘Safe’
```

Algorithm 1: PDR: main loop.

```
1 function Strengthen()
2     while $SAT?[F_N \wedge T \wedge \neg P']$ do /* SAT: error predecessor */
3         $m \leftarrow$ satisfying present state assignment
4         $c \leftarrow$ SatGeneralization($m$)
5         if ResolveRecursively($c, N$) = ‘Unsafe’ then return ‘Unsafe’
6     return ‘strengthened’          /* successfully strengthened. */
```

Algorithm 2: PDR: strengthen the trace.

to a ‘trace’ of predicates $F_0, \dots, F_N$ in main loop N of PDR.[1] F_0 is always equal to $I(\overrightarrow{s})$, for $k \geq 1$ F_k over-approximates the set of states which can be reached from $I(\overrightarrow{s})$ in up to k steps, and the state sets $F_0, \dots, F_N$ are monotonically increasing by construction.

The PDR main algorithm (see Algorithm 1) first excludes error paths of lengths 0 and 1 in procedure `BaseCases()` (line 2) by proving unsatisfiability of $I \wedge \neg P$ and $I \wedge T \wedge \neg P$. If there is no counterexample in `BaseCases()`, N is initialized to 1, F_1 is initialized to P and F_1 thus overapproximates the states reachable in up to one step. In general, PDR tries to prove the absence of error paths of length $N+1$ in the main loop N of Algorithm 1 by extracting single step predecessors of $\neg P(\overrightarrow{s})$. To do so, the procedure `Strengthen()` (Algorithm 2) is called in line 4. If a predecessor minterm m is detected in line 2 of Algorithm 2, it is extracted from the satisfying assignment. Furthermore, m is ‘generalized’ (line 4) to a cube c where c represents only predecessor states of the unsafe states. Now it has to be proven that there is no path from the initial states to c. To do so, the proof obligation (c, N) (also called *Counterexample To Induction* (CTI)) has to be recursively resolved by calling `ResolveRecursively`(c, N) (Algorithm 3) in line 5.

In general, a proof obligation(d, k) leads to new SAT calls $SAT?[F_{k-1} \wedge T \wedge d']$ (line 4 of Algorithm 3).[2] If this SAT query is unsatisfiable, then d has no predecessor in F_{k-1} and (after a possible generalization into $\hat{d}$ (line 8)) this cube can be blocked in F_k by $F_k \leftarrow F_k \wedge \neg\hat{d}$. (Since the sets $F_0, \dots, F_k$

[1] In the following we often identify predicates F_k with the state sets represented by them. We further identify the predicate T with the transition relation represented by it.

[2] It can be proven that strengthening the SAT query into $SAT?[\neg d \wedge F_{k-1} \wedge T \wedge d']$ by adding $\neg d$ does not affect the correctness of the overall method [9].

```
1  function ResolveRecursively($d, k$)
2      if $k = 0$ then /* Proof obligation in frame 0.                    */
3          return 'Unsafe'
4      while $SAT?[\neg d \wedge F_{k-1} \wedge T \wedge d']$ do /* SAT: predecessor in $F_{k-1}$   */
5          $\hat{m} \leftarrow$ satisfying present state assignment
6          $\hat{c} \leftarrow$ SatGeneralization($\hat{m}$)
7          if ResolveRecursively($\hat{c}, k-1$) = 'Unsafe' then return 'Unsafe'
8      $\hat{d} \leftarrow$ UnsatGeneralization($d$)   /* $d$ unreachable in up to $k$ steps */
9      $F_1 \leftarrow F_1 \wedge \neg\hat{d}, \ldots, F_k \leftarrow F_k \wedge \neg\hat{d}$
10     return 'resolved'
```

Algorithm 3: PDR: recursively resolve proof obligation (d, k).

```
1  function Propagate()
2      for $i \in \{1, \ldots, N-1\}$, $c$ blocked in $F_i$ do
3          if $\neg$ $SAT?[F_i \wedge T \wedge c']$ then
4              $F_{i+1} \leftarrow F_{i+1} \wedge \neg c$              /* UNSAT: push forward */
5          if $F_i \equiv F_{i+1}$ then
6              return 'Safe'                              /* Proof of safety. */
7      return 'propagated'
```

Algorithm 4: PDR: propagate blocked cubes forward.

are monotonically increasing by construction, $\neg\hat{d}$ can then be blocked from all previous F_i with $0 < i < k$ as well (line 9).) If the SAT query in line 4 is satisfiable, a new predecessor minterm $\hat{m}$ has been found (line 5), it is again generalized and a new proof obligation $(\hat{c}, k-1)$ at level $k-1$ is formed (lines 6, 7).

If all proof obligations have been recursively resolved and the SAT query from `Strengthen()` becomes unsatisfiable, then the trace is strong enough to prove the absence of counterexamples of length $N+1$. Then, Algorithm 1 increments N by 1 and initializes F_N by P. After that, Algorithm 1 tries to propagate all recently blocked cubes (learned clauses) into higher time frames by calling `Propagate()` (Algorithm 4) in line 6. PDR terminates, if a proof obligation in frame 0 is found in line 3 of `ResolveRecursively`, or if some F_i and F_{i+1} become equivalent in line 6 of `Propagate`. In the latter case an inductive invariant F_i has been found.

Function `SatGeneralization`(m) generalizes proof obligations originating from a satisfiable solver call and `UnsatGeneralization`(c) generalizes blocked cubes originating from an unsatisfiable one. Usually, `SatGeneralization` applies techniques like ternary simulation [18] or lifting [14,34], whereas `UnsatGeneralization` may subsequently remove literals from c and check if it still remains unreachable [18] ('literal dropping'). Apart from those generalizations, a few other optimizations contribute to the efficiency of PDR. We focus on one particular optimization which enables PDR to find counterexamples which are longer than the trace. When inserting the proof obligation (d, k), we know that we can reach $\neg P(\vec{s})$ from all states in the cube d. Therefore PDR can insert also proof obligations (d, l) with $k < l \leq N$, since traces from $I(\vec{s})$ to d of lengths larger than k should also be excluded, if the property holds. Thus, instead of recursive calls for proof obligations a queue of proof obligations is used and proof obligations are dequeued in smaller time frames first.

4 Restrictions

We consider the notion of *restrictions* in contrast to *abstractions*. While abstractions over-approximate the behaviour of the transition relation T, restrictions under-approximate it. We restrict variables to constant values.

Definition 1. *Consider a set of signal variables $V = \{s_1, \ldots, s_m, s'_1, \ldots, s'_m, i_1, \ldots, i_n\}$ (representing present resp. next state variables and input variables). A* restriction function *$\rho : V \mapsto \{0, 1\}$ maps signal variables to constants 0 or 1. A* restriction set *R is a subset of V, the set of restricted variables. A* restriction *for restriction function ρ and restriction set R is the function $\rho_R : R \mapsto \{0, 1\}$ with $\rho_R(v) = \rho(v)$ for all $v \in R$.*

Applying a restriction ρ_R with $R = \{v_1, \ldots, v_p\}$ to a transition relation T means replacing T by $T^{\rho_R} = T \wedge C^{\rho_R}$ with $C^{\rho_R} = \bigwedge_{i=1}^{p}(v_i \equiv \rho(v_i)) = \bigwedge_{i=1}^{p} v_i^{\rho(v_i)}$. Since our method fixes the restriction function ρ in the beginning and changes (reduces) only the set of restricted variables R, we will simply write C^R instead of C^{ρ_R} and T^R instead of T^{ρ_R} in the following. Obviously, every transition $(s, i, t) \in T^R$ is also a transition in T but not vice-versa. Thus, when considering a safety model checking problem, a counterexample under some restriction is also a counterexample in the original system. A proof of safety though could be spurious, since the restricted system may miss transitions which are present in the original system.

In the following we analyze how exactly restrictions affect the main ingredients of PDR.

4.1 Finding Proof Obligations

PDR proof obligations produced under a restricted transition relation T^R are also valid proof obligations under the original transition relation T. By definition, every single state of a proof obligation p under T^R reaches the unsafe states. Since every state which is reachable from p under T^R is also reachable under T (T^R under-approximates T), the proof obligation p is also a valid proof obligation under T. However, this does not hold vice versa. A proof obligation under T might not have a valid successor under T^R which then implies that it is not a predecessor of the unsafe states and therefore it is not a proof obligation under T^R. Thus, due to the restriction we may miss proof obligations, we may conclude the absence of error paths up to some length i prematurely, and we may miss counterexamples.

4.2 Generalizing Proof Obligations

Most commonly, generalization of proof obligations is done by applying lifting [14,34]. An important precondition for the correctness of lifting is left totality, which is (apart from right uniqueness) one part of the function property. This precondition is fulfilled, if T represents a digital circuit (which we assume in this paper) and therefore behaves like a function. When we encounter a proof obligation state m as a predecessor of proof obligation d under input i, lifting removes literals from m, leading to s, as long as the formula $s \wedge i \wedge T \wedge \neg d'$ remains unsatisfiable. If this formula is unsatisfiable, all s-states do not have successors under i into $\neg d'$. It is easy to see that we may add all s-states to the proof obligation then: It follows directly from left totality that if from some state there is no successor under i into $\neg d'$, there has to be a successor under i into d'. Therefore, since d is a proof obligation, all s-states can be considered proof as obligation states, i.e., predecessors of the unsafe states.

However, restricted state variables may break the left totality of the transition relation. Due to restricted state variables it is possible, e.g., that we may encounter dead-end states (with no successor at all). Consider a state $m = s \wedge l$ with a literal l. Assume that m is a proof obligation as a predecessor of proof obligation d under input i wrt. T^R. Now assume that $s \wedge \neg l$ violates the restriction and thus is a dead-end state under T^R. If T is right unique, then $s \wedge i \wedge T^R \wedge \neg d'$ is unsatisfiable and lifting would erroneously classify s as a proof obligation. Thus, applying lifting to restricted transition relations T^R may lead to spurious counterexamples.

The easiest way out is to just use proof obligation generalization techniques which do not depend on left totality, like don't care reasoning with ternary logic [18,20] or the Implication Graph Based Generalization (IGBG) from [34, 38]. IGBG traverses the implication graph of the SAT solver backwards and determines which assignments to present state variables were responsible for implying a particular next state valuation.

Another possibility is to extend lifting as follows. Assume that we have a proof obligation cube d and a predecessor state (minterm) m where $m \wedge i \wedge T^R \wedge d'$ is satisfiable. Standard lifting would verify SAT?$[m \wedge i \wedge T^R \wedge \neg d']$. We recall that m could be enlarged by adding states $\hat{m}$ for which $\hat{m} \wedge i \wedge T^R$ is already unsatisfiable. To work around this, we can alter lifting and employ the original *unconstrained* transition relation T. We consider the call SAT?$[m \wedge i \wedge T \wedge (\neg d' \vee \neg C^R)]$. Assume that this SAT instance is unsatisfiable even for a subcube s instead of m (resulting from the computation of an unsatisfiable core). Unsatisfiability implies that each satisfying assignment to $s \wedge i \wedge T$ satisfies $d' \wedge C^R$. Since T is left total, each s-state $\hat{m}$ has indeed a successor under i wrt. T. Thus, each s-state $\hat{m}$ has a successor in d under i wrt. $T^R = T \wedge C^R$, i.e., s is a proof obligation wrt. T^R. We will evaluate in Sect. 7 which variant achieves the best results.

4.3 Blocking Cubes

A cube d may be blocked in frame F_i once the formula $\neg d \wedge F_i \wedge T \wedge d'$ is unsatisfiable. However, the formula $\neg d \wedge F_i \wedge T^R \wedge d'$ with a restricted transition relation T^R, which under-approximates T, is more likely to be unsatisfiable. Thus, if d can be blocked under T it can also be blocked under T^R but not necessarily vice versa. Hence, we may encounter spuriously blocked cubes under restrictions.

4.4 Generalizing Blocked Cubes

Interestingly, with restrictions on next state variables there are constellations for which the literals can be immediately removed from a blocked cube without additional SAT checks when generalizing blocked cubes, i.e., when removing literals from d resulting in $\hat{d}$ such that $\neg\hat{d} \wedge F_i \wedge T^R \wedge \hat{d}'$ is still unsatisfiable. This can be easily seen when considering the idea of *literal dropping* [18,34] which is usually done after the extraction of an unsatisfiable core during generalization of blocked cubes. Literal dropping sequentially tries to remove literals l from blocked cubes. However, it is easy to see that all literals l occurring in d where the corresponding next state literal l' is in R and l' is consistent with the restrictions, i.e., $l' = {s_i'}^{\rho(s_i')}$ for some next state variable s_i', can immediately be removed from d: Let $d = \tilde{d} \wedge s_i^{\rho(s_i')}$, $s_i' \in R$, and let $\neg d \wedge F_i \wedge T^R \wedge d'$ be unsatisfiable. Since $\neg {s_i'}^{\rho(s_i')} \wedge C^R = 0$, $\neg(\tilde{d} \wedge \neg s_i^{\rho(s_i')}) \wedge F_i \wedge T^R \wedge \tilde{d}' \wedge \neg {s_i'}^{\rho(s_i')}$ is unsatisfiable as well and so $\neg\tilde{d} \wedge F_i \wedge T^R \wedge \tilde{d}'$ is unsatisfiable. Thus, additional SAT checks for removing literals with the mentioned property from d are not needed. As a result, restrictions may lead to more general blocked cubes.

However, a blocked cube (learned clause) may be spurious iff the unsatisfiability proof used to block or generalize it is based on restrictions or any other spurious cube blocked in F_i.

4.5 Overall Algorithm

Counterexamples under restrictions are valid for the original sequential circuit if the generalization of proof obligations is applied in a correct way (see Sect. 4.2). Since proofs based on restricted transition relations may be spurious, we will need a coarsening approach, which is able to detect spurious proofs and relax the restrictions accordingly, such that PDR will not find the same spurious proof again. The observations made above imply that we can re-use proof obligations with relaxed restrictions, but we have to be careful when re-using blocked cubes.

5 Skipping Restrictions by Analyzing a Spurious Proof

In this section we present the main idea of Proof-Guided Restriction Skipping (PROGRESS) in the context of PDR. Note that this idea does not really depend on using PDR, but is applicable to any verification method providing safety proofs in form of *safe inductive invariants* [9].

PDR decides that a system under verification is safe once it has found a CNF formula *Inv* which represents a safe inductive invariant. To act as a safe inductive invariant, *Inv* must satisfy certain requirements.

Definition 2. *A boolean formula Inv is a* safe inductive invariant*, iff* $I \implies \mathit{Inv}$, $\mathit{Inv} \wedge T \implies \mathit{Inv}'$*, and* $\mathit{Inv} \implies P$ *holds.*

In Sect. 4 we discussed how PDR under a *restriction* ρ_R may find spurious inductive invariants. Here we discuss how to detect whether a safe inductive invariant is spurious or not, i.e., whether it is a safe inductive invariant for the unrestricted system or not. Furthermore, we present an algorithm which detects restrictions that are responsible for a spurious proof and removes them from the set of restricted variables accordingly.

5.1 Detecting Spurious Proofs in PDR

PDR deduces safety of a system, if the CNF formulae of two adjacent time frames i and $i+1$ are equivalent, i.e., if $F_i \equiv F_{i+1}$, see Sect. 3.2. Hence, we assume that $\mathit{Inv} = F_i$. The first property $I \implies \mathit{Inv}$ holds by definition of PDR, since F_i over-approximates the states which are reachable from I within up to i steps. The second property $\mathit{Inv} \wedge T \implies \mathit{Inv}'$ is satisfied, since $F_i \wedge T \implies F'_{i+1}$ holds as an invariant of the PDR algorithm and $F_i \equiv F_{i+1}$ as well as $\mathit{Inv} = F_i$. The third property $\mathit{Inv} \implies P$ holds by definition of PDR based on [9], which initializes the time frame formula F_N with P in main loop N (see Sect. 3.2).

Now assume a restricted transition relation T^R. We assume that PDR with transition relation T^R does not find a counterexample, but a safe inductive invariant Inv. Note that we change only T into T^R by the restriction ρ, we do not change I and P. Therefore it immediately follows that $I \implies \mathit{Inv}$ and $\mathit{Inv} \implies P$. However, the property $\mathit{Inv} \wedge T^R \implies \mathit{Inv}'$ may hold only due to the restrictions. In order to detect whether Inv is also an invariant under the unrestricted transition relation T we can use a SAT solver: We insert $\bigwedge_{v \in R} v^{\rho(v)}$ as assumptions into the SAT solver [19] and call the SAT solver on the *unrestricted* transistion relation with $\mathit{Inv} \wedge T \wedge \neg \mathit{Inv}'$. The call will be unsatisfiable since assuming the restrictions is equivalent to using the restricted transition relation T^R. If the UNSAT-core over the assumptions contains at least one of the restricted variables, we conclude that the invariant may be spurious and may hold only due to the imposed restrictions. If not, the restrictions are not needed to prove that $\mathit{Inv} \wedge T \implies \mathit{Inv}'$, i.e., Inv is a safe inductive invariant wrt. T.

5.2 Restriction Skipping Loop

In Algorithm 5 we present a main ingredient of Proof-Guided Restriction Skipping (PROGRESS) which we call the *Restriction Skipping Loop*. We check the satisfiability of $\mathit{Inv} \wedge T \wedge \neg \mathit{Inv}'$ (line 4) with assumptions $\bigwedge_{v \in R} v^{\rho(v)}$ (line 3) as already mentioned above. If the SAT solver returns UNSAT, then the UNSAT-core of the SAT solver can then be used to guide the removal of restrictions. For removing restrictions we have implemented two options: The 'careful' approach removes exactly *one* restriction from the UNSAT core (line 13) and checks whether it was sufficient to break the possibly spurious invariant and the 'aggressive' approach removes *all* restrictions occurring in the UNSAT core at once

```
1  function RestrictionSkippingLoop(Inv)
2      while true do
3          Assume $\bigwedge_{v \in R} v^{\rho(v)}$        /* Assume restrictions. */
4          if SAT?[$Inv \wedge T \wedge \neg Inv'$] then
5              /* Invariant spurious, retracted enough to break it.   */
6              return Spurious
7          else
8              if UNSAT core contains no variable $v$ with $v \in R$ then
9                  /* No restriction in UNSAT core, correct proof.   */
10                 return Safe
11             else
12                 if careful then
13                     skip one restriction appearing in the UNSAT core from $R$
14                 else
15                     skip all restrictions appearing in the UNSAT core from $R$
```

Algorithm 5: The *Restriction Skipping Loop.*

(line 15). Removing restrictions just means adding transitions to the transition relation T^R. If we finally arrive at line 6, we have removed enough restrictions such that the resulting T^R contains at least one transition from Inv to $\neg Inv$. i.e., the safe invariant has been destroyed and in the overall algorithm we can start over with the reduced set R of restrictions.[3] If we arrive at line 10, we have been able to prove that removing more restrictions will never destroy the invariant and the (unrestricted) system is safe. The different strategies (line 13 vs. 15) will be subject to our empirical evaluation in Sect. 7.

6 Implementation of PROGRESS-PDR

We present our implementation of PDR, called *PROGRESS-PDR*, which implements *restrictions* and the *restriction skipping loop* from Sect. 5. The algorithm is shown in Algorithm 6. In the following we will discuss the different parts of the algorithm in more detail.

6.1 Combining PROGRESS-PDR with Standard PDR

PROGRESS-PDR is meant to supplement PDR's capabilities of finding deep counterexamples. Therefore, there may be instances (especially safe instances) for which standard PDR could be of better use. In order to profit from the

[3] In this case the Restriction Skipping Loop has finally computed an approximate solution to the partial MaxSAT [29] problem $\bigwedge_{v \in R} v^{\rho(v)} \wedge Inv \wedge T \wedge \neg Inv'$ with $\{v^{\rho(v)}\}$ as soft clauses.

```
1  function ProgressPdr()
2      $resPDR \leftarrow$ Pdr($I$, $T$, $P$, '$check\_stuck$')        /* Safe/Unsafe/Stuck */
3      if $resPDR \neq$ 'Stuck' then return $resPDR$
4      $N_{init} \leftarrow N$, $F_1^{init} \leftarrow F_1, \dots, F_N^{init} \leftarrow F_N$
5      $PO \leftarrow \emptyset$                          /* Discard proof obligations. */
6      /* $F_1, \dots, F_N$ remain for next PDR run. */
7      $R \leftarrow$ InitRestrictionSet(), $\rho \leftarrow$ InitRestrictionFunction()
8      HandleTrivialCases($R$)
9      $n_{spurious} \leftarrow 0$
10     while $true$ do
11         $resPDR \leftarrow$ Pdr($I$, $T^R$, $P$)            /* returns Safe or Unsafe */
12         if $resPDR =$ 'Unsafe' then return 'Unsafe'
13         if $resPDR =$ 'Safe' then /* Proof may be spurious! */
14             $resRSL \leftarrow$ RestrictionSkippingLoop($Inv_R$)
15             if $resRSL =$ 'Safe' then return 'Safe'
16             if $resRSL =$ 'Spurious' then /* Spurious, $R$ was reduced. */
17                 $n_{spurious} \leftarrow n_{spurious} + 1$
18                 if $n_{spurious} > c_{spurious}$ then
19                     $R \leftarrow \emptyset$                   /* Remove all restrictions. */
20                 /* Proof obligations $PO$ remain for next PDR run. */
21                 $N \leftarrow N_{init}$
22                 $F_1^{prev} \leftarrow F_1$
23                 if $R \neq \emptyset$ then
24                     $F_1 \leftarrow P, \dots, F_N \leftarrow P$
25                 else
26                     $F_1 \leftarrow F_1^{init}, \dots, F_N \leftarrow F_N^{init}$
27                 for $c$ blocked in $F_1^{prev}$ do
28                     if $\neg$ SAT?[$\neg c \wedge F_0 \wedge T^R \wedge c'$] then $F_1 \leftarrow F_1 \wedge \neg c$
29                 Propagate()
```

Algorithm 6: Overall approach *PROGRESS-PDR*.

advantages of both worlds, we start with a run of standard PDR (line 2 of Algorithm 6) and use restrictions and restriction skipping only if PDR gets 'stuck'. This is similar to the approach in [35] which employs k-induction if PDR starts to exhaustively enumerate states due to the lack of strong generalization. By 'stuck' we mean that PDR does not advance fast enough in the number of open time frames, i.e., within main loop N PDR is busy with handling proof obligation after proof obligation, but is not able to prove the absence of counterexamples of length N for a long time. In our implementation we heuristically detect such a situation as follows: Starting from a fixed initial number of time frames (we choose 3 in our implementation) we store the number of proof obligations n_{PO} that have been resolved so far at the moment when standard PDR is about to open a new time frame. If the next main iteration of PDR (that strengthens the new time frame) produces a larger number of proof obligations than n_{PO}, we consider the execution as 'stuck' and we will switch to PDR with restrictions. Note that in PROGRESS-PDR we can always re-use the complete trace with its clauses and time frames from the standard PDR execution which we have aborted, since the restriction gets stronger and not weaker, but the unresolved proof obligations from the aborted standard PDR run have to be discarded, since they will not be necessarily proof obligations in the following PDR run with restrictions (see lines 5, 6).

6.2 Choosing Appropriate Restrictions

Now we introduce restrictions, i.e., we choose a restriction function ρ and a set R of restricted variables (line 7). Choosing the most effective restrictions in order to find deep counterexamples is a challenging task. For our study we use the simplest possible method: We start with restrictions on all primary inputs, present state and next state variables and we initially restrict them with 0. (In our experiments we also consider a variant restricting primary inputs and present state variables only.) Advanced heuristics for choosing initial restrictions remain as future work. Apparently, too many restrictions may cause the verification problem to become trivial. For instance, the restricted transition relation T^R may even become empty. Therefore, after initializing R and ρ, we immediately call `HandleTrivialCases(`R`)` in line 8 which subsequently removes variables from R until $(I \wedge T^R)$, $(T^R \wedge P')$, as well as $(T^R \wedge \neg P')$ become satisfiable, i.e., until there is at least one transition in T^R starting from I (otherwise the restricted system is trivially safe), there is at least one transition in T^R leading to P (otherwise the restricted system is trivially unsafe, if there are any transitions from I in T^R), and there is at least one transition in T^R leading to $\neg P$ (otherwise the restricted system is trivially safe).

6.3 PDR with Restrictions

Now PDR is applied with the chosen restrictions on T (line 11). If we encounter a counterexample, we know (according to Sect. 4) that it is valid and terminate concluding that the design is *unsafe* (line 12). If we encounter a safe inductive invariant Inv_R, we check whether it is spurious (according to Sect. 5.1) (line 14). If it is a valid inductive invariant, we terminate concluding that the design is *safe* (line 15). If not, we retract with our Restriction Skipping Loop a number of restricted variables from R until Inv_R is not an inductive invariant anymore (line 13). Apparently, finding spurious safe inductive invariants, retracting restrictions accordingly, and starting over comes with additional cost. Therefore we count the number $n_{spurious}$ of coarsenings by restriction skipping and reset R to $\emptyset$ as soon as $n_{spurious}$ exceeds some upper limit $c_{spurious}$ (in our implementation we use $c_{spurious} = 20$) (line 19).

6.4 Re-using Information from Previous Restricted PDR Run

Before we start over PDR with the reduced restriction set R (and unchanged restriction function ρ), we prepare to re-use certain parts of the previous PDR run for the next one. For instance, we may re-use proof obligations (line 20 – we assume that the proof obligations from the previous run are stored in a set PO), since these are still valid predecessors of the unsafe states (see Sect. 4).

We are also able to re-use certain learned clauses (blocked cubes). We recall that cubes may be blocked early due to the restrictions on variables from R. Here, we call the set of restricted variables from the previous run (before we encountered a spurious proof and reduced it) R^{prev}. Assume that with R^{prev} cube c can be blocked in frame F_{i+1}, because the formula $\neg c \wedge F_i \wedge T^{R^{prev}} \wedge c'$ is unsatisfiable. If c has been blocked in the previous run with $T^{R^{prev}}$, and cannot be blocked in the run with T^R, we distinguish two cases: (1) There are transitions from $\neg c \wedge F_i$ to c' in T^R which had been removed from $T^{R^{prev}}$ by stronger restrictions. (2) Such transitions are not directly removed by restrictions in the previous run, but in the previous run F_i already contained spurious clauses which exclude valid predecessors of c' from the state space.

One option for re-using learned clauses would be similar to the technique from Sect. 5. For a blocked cube c we could compute by using assumptions and UNSAT-core analysis a subset of restrictions and clauses in F_i which are sufficient for making $\neg c \wedge F_i \wedge T^R \wedge c'$ unsatisfiable. If the same analysis had been done for the clauses in F_i, we could compute by transitivity the subset of restrictions which are directly or indirectly involved in the blocking of cube c. This directly reveals which learned clauses can be safely re-used in the new run with relaxed restrictions R. Such an analysis entails additional effort and an intensive bookkeeping. Moreover, it could over-estimate the set of restrictions needed for blocking a cube c. Therefore we prefer a much simpler approach for re-using learned clauses in this paper:

For the next PDR run we set the number N of open frames apart from F_0 to N_{init} which is the same number occurring in the initial standard PDR run that was stuck (see lines 4 and 21). Moreover, we open N additional frames $F_1, \ldots, F_N$ (apart from $F_0 = I$) for the next PDR run (line 24). Since in the initial standard PDR run is has been proven that there are no traces from I to $\neg P$ of lengths up to N under T, there are no traces from I to $\neg P$ of lengths up to N under T^R (which is an underapproximation of T) either. So P overapproximates the set of states reachable in up to N steps under T^R and thus it is sound to initialize $F_1, \ldots, F_N$ with P. Now we validate for all blocked cubes from the previous run (under R^{prev}) whether they can be blocked in the first time frame F_1 of our new run (under relaxed restrictions R) (line 28). (Note that due to monotonicity of the frames in PDR all cubes blocked in some arbitrary time frame F_i are blocked in F_1 as well.) We start a propagation phase (as in standard PDR) and try to subsequently block the cubes in higher time frames (line 29). Line 26 considers the special case $R = \emptyset$ where $T^R = T$. Apparently, we can restore all sets F_i from the initial standard PDR run in this case (see lines 4 and 26).

7 Experimental Results

We discuss the results of our approach on Hardware Model Checking Competition (HWMCC) benchmarks. All experiments have been performed on the

HWMCC benchmark sets (626 benchmarks in total) of the latest three competitions (2017, 2019 and 2020) [4,6,33]. For 2019 and 2020 we have only considered the AIGER [5] benchmarks (bit-vector track), since our tool does not support word-level verification. We limited the execution time to3600 s and set a memory limit of 7 GB. We used one core of an Intel Xeon CPU E5-2650v2 with 2.6 GHz. We provide a reproduction artifact under [37].

Our implementation of *PROGRESS-PDR* uses *IC3ref* [8] as its PDR core. Stand-alone IC3ref (without any preprocessing) is competitive [21,36], well-known and commonly referenced in the literature. Therefore, it is a perfect fit for the demonstration of our algorithm. To justify our selection of *IC3ref* [8] as the basis for our algorithmic extensions as well as comparisons we also compare our results to the PDR implementation of ABC[4] [10]. Moreover, we compare the results to the BMC implementation (bmc2) of ABC. We ran both ABC tools in their default configuration. We made only one change to IC3ref and replaced the lifting procedure for proof obligation generalization by IGBG from [34,38] (see Sect. 4.2). Replacing lifting in IC3ref has two advantages: IGBG is slightly better than the standard lifting implementation in IC3ref (which we have shown in previous work [38]). Moreover, IGBG requires for its correctness only right-uniqueness of the transition relation and is therefore able to cope with invariant constraints (imposed on HWMCC'19 and '20 benchmarks) without any changes as well as with restricted transition relations used in PROGRESS-PDR. In the following, we address the IC3ref implementation with IGBG and without restrictions as 'standard PDR' or just 'PDR'.

We use MiniSat v2.2.0 [19] as a SAT solver. Furthermore, we test whether PROGRESS-PDR is dominated by PDR with preprocessing the AIGER specifications with Cone-Of-Influence (COI) reduction. As an implementation for COI reduction we use the one from IIMC [22]. We also compare PROGRESS-PDR to BMC which uses the same interface to the AIGER model with the same transition relation (preprocessed with variable elimination of MiniSat).

Although the different methods have complementary strengths and weaknesses, we refrain from considering portfolio approaches combining different methods and rather focus on the contribution of single approaches for clearer comparisons. Moreover, we do not affect our comparisons by orthogonal methods like preprocessing with sequential circuit transformations.

7.1 Design Decisions for PROGRESS-PDR

First, in this section we justify our design decisions made for PROGRESS-PDR by analyzing the impact of different alternatives. We decided to use the following options: In PROGRESS-PDR we always re-use blocked cubes (learned clauses) after restriction skipping and check whether they can be blocked also with less restrictions. Moreover, we re-use proof obligations from the previous

[4] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, ABC 1.01 (downloaded Jul 13 2021). http://www.eecs.berkeley.edu/~alanmi/abc/.

Table 1. Different variants of PROGRESS-PDR.

Variant	Unsafe	Unsafe ($\geq$30)	Safe	MO/TO
Standard PDR	69	15	**293**	264
PROGRESS-PDR	**84**	**27**	**293**	249
Restricting all state variables + inputs	79	25	292	255
Restricting present state variables + inputs	82	26	290	255
Restricting only present state variables	77	22	290	259
No re-used blocked cubes	73	20	291	262
No re-used proof obligations	82	26	**293**	251
Aggressive retracting	77	23	289	260
Modified Lifting	77	22	284	265

run with more restrictions without needing additional checks. We restrict only state variables instead of restricting inputs. When encountering a spurious proof, we carefully retract restrictions within the restriction skipping loop from Sect. 5.2 (one-by-one until a spurious invariant is broken) instead of aggressively retracting all restrictions from the UNSAT-core of the SAT solver. Finally, we use IGBG for proof obligation generalization (instead of using the lifting approach of [14,34] with the extension from Sect. 4.2).

In the first and second line of Table 1 we report the number of solved benchmarks for standard PDR as well as PROGRESS-PDR. In the second column 'unsafe' we report the number of benchmarks for which we found counterexamples, whereas in the third column 'unsafe ($\geq$30)' we report the number of benchmarks for which we found deep counterexamples with a path length greater or equal to 30. The fourth column 'safe' shows the number of benchmarks proved to be safe and the last column shows the number of benchmarks where the time or memory limit was exceeded.

We start our analysis by considering different sets of restricted variables. Whereas all variants outperform standard PDR in finding counterexamples and especially deep ones, we still observe some differences: In the third line, we present the results for restricting primary inputs in addition to the state variables. This configuration performs slightly worse. The fourth line shows the results for restricting primary inputs and present state variables, but no next state variables. Again, the results are slightly worse than the standard configuration with restrictions on present and next state variables. Interestingly though, when we do not restrict next state variables but *only* present state variables, we observed that it is beneficial to *also* restrict primary inputs (solving 5 more unsafe instances than with *only* restricting present state variables, see line 5).

We believe that restricting not only present state but also next state variables can be a powerful means, if we analyze rather loosely coupled circuits with inputs, which may deactivate irrelevant (at least for disproving the safety property) parts of the state space. These restrictions may simplify the verification problem drastically and also support the generalization of blocked cubes in PDR, see

Sect. 4.4. Although the results in Sect. 7.2 will show that (syntactical) Cone-of-Influence (COI) reduction does not have a significant impact on the considered benchmarks, the given safety property may not be influenced by certain state bits all the same (i.e., those state bits are not in the 'semantic COI'). Fixing those state bits to constants does not have an impact on the safety proof (apart from simplifications by unit propagation and reduced state spaces) and, apparently, it is often also not important to which value those state bits are fixed. However, we conjecture that restricting primary input variables as well would improve if we would replace our brute-force method of restricting inputs and state variables just by a fixed constant 0 by a more informed restriction variant exploiting user knowledge on inputs driving the design into potential error states. Nevertheless, we expect that varying the classes of variables to be restricted could make sense when considering different classes of benchmarks.

In the sixth line of Table 1 we show how the results change, if we do not re-use blocked cubes from previous runs. The results clearly show that this leads to significantly worse results - especially for deep counterexamples. Re-using cubes seems to be vital for PDR to be able to progress faster after restarting with less restrictions.

The seventh line shows a variant without re-using proof obligations from the previous run with more restrictions. Re-using proof obligations helps, but not as much as re-using blocked cubes. This could be due to the fact that proof obligations are generated relatively quickly and are therefore less valuable (in terms of computational effort) than blocked cubes which have been generalized with much more effort using a loop performing literal dropping.

As line 8 of Table 1 shows, aggressively retracting all restrictions from the UNSAT-core in case of a spurious proof (see Sect. 5.2) does not pay off. It seems to be beneficial to carefully keep as many restrictions as possible.

Finally, it turned out (see line 9 of Table 1) that the lifting approach of [14,34] with the extension from Sect. 4.2 is inferior to IGBG for proof obligation generalization. Solving 16 benchmarks less than PROGRESS-PDR with IGBG we conclude that adapting and using the standard lifting procedure is not worthwile.

7.2 PROGRESS-PDR vs. PDR

We compare PROGRESS-PDR against standard PDR. We present the overall results in Fig. 2 - including unsafe instances and also all kinds of counterexample depths. We depict the graphs for only unsafe benchmarks in Fig. 3. Furthermore, we also plot the graphs for unsafe benchmarks with counterexamples which are represented by error paths with a length greater or equal to 30 in Fig. 4. All comparisons show that our choice for 'standard PDR' (based on IC3ref) performs pretty similar to the PDR implementation in ABC.

PROGRESS-PDR greatly outperforms standard PDR on benchmarks with counterexamples present. The longer the error path, the stronger PROGRESS-PDR gets, nearly doubling the amount of deep counterexamples solved by standard PDR. Regarding safe benchmarks, PROGRESS-PDR and PDR both prove the absence of counterexamples in 293 problem instances. Interestingly though, these instances are not identical. PROGRESS-PDR solves safe instances that

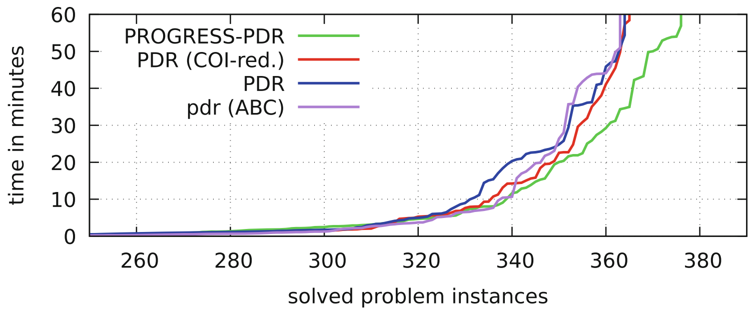

Fig. 2. Results on all instances.

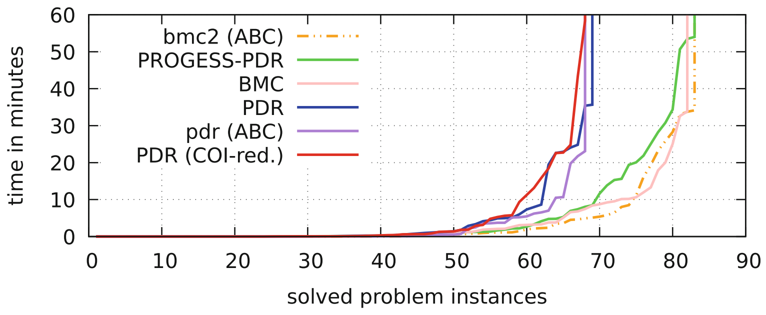

Fig. 3. Results on counterexamples.

PDR does not and vice versa. In summary, even though PROGRESS-PDR primarily aims to increase the capability of finding counterexamples, we can also observe an overall improvement.

We made the additional observation that for the instances solved by our Restriction Skipping Loop on average 49.3% of the total number of state variables were still under restriction after solving the instance and 5.65 restarts happened due to spurious proofs.

We also investigate whether our PROGRESS-PDR approach is dominated by simple COI reduction. It could be the case, that the remaining restrictions after some restriction skipping loops simply restrict variables which would have been removed by COI reduction anyway. However, our experimental results show that this can rarely be the case. The results for PDR and PDR with COI reduction are pretty similar in Figs. 2, 3, and 4 with really visible differences only in Fig. 2. This shows that COI reduction does not help for the considered benchmarks whereas PROGRESS-PDR does.

7.3 PROGRESS-PDR vs. BMC

To evaluate the effectiveness of PROGRESS-PDR we compare it to the most common SAT-based bug hunting technique, namely BMC. We depict our results

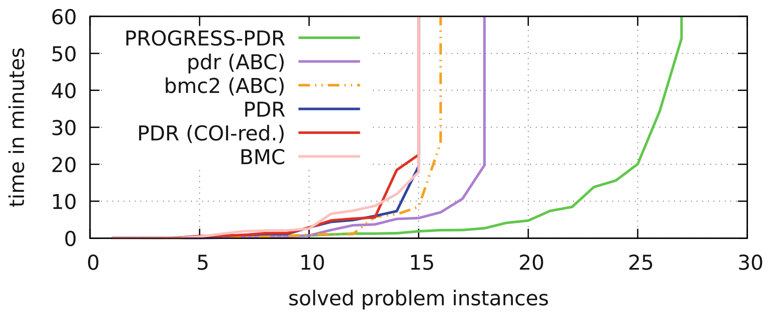

Fig. 4. Results on deep counterexamples.

in the cactus plot of Fig. 3 and for deep counterexamples (with an error path of length greater or equal to 30) in Fig. 4. While BMC achieves similar results as PROGRESS-PDR in the number of solved instances on the set of all unsafe benchmarks (benchmarks with counterexamples), it is greatly outperformed by PROGRESS-PDR on deep counterexamples. Even standard PDR is able to meet the results of BMC when it comes to deeper counterexamples. This can be explained by the size of SAT problems produced by circuit unrolling in BMC for deep counterexamples. Note that for our results COI reduction has been performed on the BMC instances, but it did not help much. As the results also show, exchanging our BMC implementation with the BMC implementation of ABC does not change the experimental findings above.

8 Conclusions and Future Work

With PROGRESS-PDR, we presented a *complete* and *fully automatic* approach for applying PDR to a system under restrictions. We introduced PROGRESS which - as a method that is dual to CEGAR - relaxes restrictions guided by spurious proofs. We were able to show that PROGRESS-PDR greatly improves upon PDR's capabilities of finding counterexamples, especially those with long error paths. Furthermore, our study shows that PROGRESS-PDR performs significantly better than BMC on *deep* counterexamples.

Our results indicate that restrictions can be a powerful tool for safety verification with PDR, even without domain knowledge on the structure of the circuit or the property under verification. We conjecture that our results could be further improved with such knowledge, for instance by distinguishing between control and data signals and by considering signals activating parts of the circuit which are relevant to a checked property.

User knowledge could also be beneficial for bug hunting by restricting internal signals other than primary inputs and state variables or by initially restricting signals with more sophisticated constraints instead of fixing signals to constants.

References

1. Baumgartner, J., Ivrii, A., Matsliah, A., Mony, H.: IC3-guided abstraction. In: FMCAD, pp. 182–185 (2012). https://ieeexplore.ieee.org/document/6462571/
2. Berryhill, R., Ivrii, A., Veira, N., Veneris, A.G.: Learning support sets in IC3 and quip: the good, the bad, and the ugly. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 140–147. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102252
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS, pp. 193–207 (1999). https://doi.org/10.1007/3-540-49059-0_14
4. Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition (2017). http://fmv.jku.at/hwmcc17/
5. Biere, A., Heljanko, K., Wieringa, S.: Aiger 1.9 and beyond (2011). http://fmv.jku.at/hwmcc11/beyond1.pdf
6. Biere, A., Preiner, M.: Hardware model checking competition (2019). http://fmv.jku.at/hwmcc19/
7. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction guided abstraction refinement (CTIGAR). In: CAV, pp. 831–848 (2014). https://doi.org/10.1007/978-3-319-08867-9_55
8. Bradley, A.: Ic3 reference implementation (2013). https://github.com/arbrad/IC3ref
9. Bradley, A.R.: Sat-based model checking without unrolling. In: VMCAI, pp. 70–87 (2011). https://doi.org/10.1007/978-3-642-18275-4_7
10. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
11. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15–18 November 2009, Austin, Texas, USA, pp. 69–76. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351141
12. Brummayer, R., Biere, A.: Effective bit-width and under-approximation. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2009. LNCS, vol. 5717, pp. 304–311. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04772-5_40
13. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_28
14. Chockler, H., Ivrii, A., Matsliah, A., Moran, S., Nevo, Z.: Incremental formal verification of hardware. In: FMCAD, pp. 135–143 (2011). http://dl.acm.org/citation.cfm?id=2157676
15. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
16. Dill, D.L., Wong-Toi, H.: Verification of real-time systems by successive over and under approximation. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 409–422. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60045-0_66

17. Eén, N., Mishchenko, A., Amla, N.: A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In: Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, 20–23 October 2010, pp. 181–188. IEEE (2010). https://ieeexplore.ieee.org/document/5770948/
18. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD, pp. 125–134 (2011). http://dl.acm.org/citation.cfm?id=2157675
19. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT, pp. 502–518 (2003). https://doi.org/10.1007/978-3-540-24605-3_37
20. Franzén, A.: Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT. Ph.D. thesis, University of Trento, Italy (2010). http://eprints-phd.biblio.unitn.it/345/
21. Griggio, A., Roveri, M.: Comparing different variants of the ic3 algorithm for hardware model checking. IEEE Trans. CAD Integr. Circuits Syst. **35**(6), 1026–1039 (2016). https://doi.org/10.1109/TCAD.2015.2481869
22. Hassan, Z., Bradley, A.R., Somenzi, F.: Incremental, inductive CTL model checking. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 532–547. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_38
23. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in IC3. In: FMCAD, pp. 157–164 (2013). https://ieeexplore.ieee.org/document/6679405/
24. Ho, Y., Mishchenko, A., Brayton, R.K.: Property directed reachability with word-level abstraction. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 132–139. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102251
25. Hooker, J.N.: Solving the incremental satisfiability problem. J. Log. Program. **15**(1 & 2), 177–186 (1993) **15**(1&2), 177–186 (1993) **15**(1&2), 177–186 (1993)
26. Ivrii, A., Gurfinkel, A.: Pushing to the top. In: FMCAD, pp. 65–72 (2015). https://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD15/papers/paper39.pdf
27. Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O.: Abstraction-based satisfiability solving of presburger arithmetic. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 308–320. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_24
28. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes. Princeton University Press, Princeton (1994)
29. Li, C.M., Manya, F.: Maxsat, hard and soft constraints. Handb. Satisf. **185**, 613–631 (2009)
30. McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_2
31. Mishchenko, A., Eén, N., Brayton, R.K., Baumgartner, J., Mony, H., Nalla, P.K.: GLA: gate-level abstraction revisited. In: Design, Automation and Test in Europe, DATE 13, Grenoble, France, 18–22 March 2013, pp. 1399–1404. EDA Consortium San Jose, CA, USA/ACM DL (2013). https://doi.org/10.7873/DATE.2013.286
32. Nopper, T., Scholl, C.: Symbolic model checking for incomplete designs with flexible modeling of unknowns. IEEE Trans. Comput. **62**(6), 1234–1254 (2013)
33. Preiner, M., Biere, A., Froleyks, N.: Hardware model checking competition 2020 (2020). http://fmv.jku.at/hwmcc20/
34. Ravi, K., Somenzi, F.: Minimal assignments for bounded model checking. In: TACAS, pp. 31–45 (2004). https://doi.org/10.1007/978-3-540-24730-2_3

35. Scheibler, K., Winterer, F., Seufert, T., Teige, T., Scholl, C., Becker, B.: ICP and IC3. In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2021 (2021). https://doi.org/10.23919/DATE51398.2021.9473970
36. Seufert, T., Scholl, C.: fbpdr: In-depth combination of forward and backward analysis in property directed reachability. In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, 25–29 March 2019, pp. 456–461. IEEE (2019). https://doi.org/10.23919/DATE.2019.8714819
37. Seufert, T., Scholl, C., Chandrasekharan, A., Reimer, S., Welp, T.: Reproduction artifact (2021). https://abs.informatik.uni-freiburg.de/src/projects_view.php?projectID=23
38. Seufert, T., Winterer, F., Scholl, C., Scheibler, K., Paxian, T., Becker, B.: Everything You Always Wanted to Know About Generalization of Proof Obligations in PDR. arXiv preprint arXiv:2105.09169 (2021). https://arxiv.org/abs/2105.09169
39. Silva, J.P.M., Sakallah, K.A.: GRASP-a new search algorithm for satisfiability. In: ICCAD, pp. 220–227 (1996). https://doi.org/10.1109/ICCAD.1996.569607
40. Tseitin, G.: On the complexity of derivations in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logics (1968)
41. Vizel, Y., Grumberg, O., Shoham, S.: Lazy abstraction and SAT-based reachability in hardware model checking. In: FMCAD, pp. 173–181 (2012). https://ieeexplore.ieee.org/document/6462570/
42. Wang, D., et al.: Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, 18–22 June 2001, pp. 35–40. ACM (2001). https://doi.org/10.1145/378239.378260