Incremental Property Directed Reachability

Max Blankestijn and Alfons Laarman(✉)

Leiden University, Leiden Institute for Advanced Computer Science, Leiden, The Netherlands

max@blankestijn.com, a.w.laarman@liacs.leidenuniv.nl

Abstract. Property Directed Reachability (PDR) is a widely used technique for formal verification of hardware and software systems. This paper presents an incremental version of PDR (IPDR), which enables the automatic verification of system instances of incremental complexity. The proposed algorithm leverages the concept of incremental SAT solvers to reuse verification results from previously verified system instances, thereby accelerating the verification process. The new algorithm supports both incremental constraining and relaxing; i.e., starting from an over-constrained instance that is gradually relaxed.

To validate the effectiveness of the proposed algorithm, we implemented IPDR and experimentally evaluate it on two different problem domains. First, we consider a circuit pebbling problem, where the number of pebbles is both constrained and relaxed. Second, we explore parallel program instances, progressively increasing the allowed number of interleavings. The experimental results demonstrate significant performance improvements compared to Z3's PDR implementation SPACER. Experiments also show that the incremental approach succeeds in reusing a substantial amount of clauses between instances, for both the constraining and relaxing algorithm.

1 Introduction

Symbolic model checking based on satisfiability has revolutionized automated verification. Initially, symbolic model checkers were based on (binary) decision diagrams [13,42]. While they enabled the study of large software and hardware systems [14,18,20], they were inevitably limited by memory constraints because decision diagrams represent all satisfying assignments explicitly. *Bounded Model Checking* (BMC) [5,6] alleviated the need for decision diagrams by encoding the behavior of a system directly into propositional logic, in a way similar to the reductions provided by Cook [19] and Levin [39] much earlier (who could have foreseen this future application of the theory?). BMC, in turn, is limited by the depth of the system under verification, since the encoding explicitly 'unrolls' the transition relation for each time step of the computation and each unrolling requires another copy of the state variables. The introduction of the IC3 algorithm [10,12], later known as Property Directed Reachability (PDR) [23,35,52],

Y. Li and S. Tahar (Eds.): ICFEM 2023, LNCS 14308, pp. 208–227, 2023.
https://doi.org/10.1007/978-981-99-7584-6_13

circumvents this unrolling by using small SAT solver queries to incrementally construct an inductive invariant from the property.

This work is inspired by the success of BMC and other verification methods based on *incremental SAT solving* [1,24,45,53]. In order to reduce the search space, modern SAT solvers learn new clauses, further constraining the original problem, from contradictions arising during the search for satisfying assignments [1,50]. BMC can exploit the power of clause learning by incrementally increasing the hardness of the problem instance, while retaining learned clauses from 'easier' instances [53]. A natural parameter here is the unrolling depth: Incremental BMC increases the unrolling bound to generate a new problem instance. This approach has shown to yield multiple orders of magnitude runtime improvements [7], which often translates into unrollings that are multiple times longer.

A natural question is whether PDR can also benefit from incremental SAT solving. However, since PDR does not unroll the transition relation, we need new parameters to gradually increase the hardness of instances and exploit incremental solvers. Moreover, the standard PDR algorithm requires an extension to reuse information learned in previous runs. We provide both these parameters and a new incremental PDR algorithm.

For instance, an increasing parameter to consider, other than the unrolling depth, is the number of parallel threads in a system. However, it is not always clear how a system with fewer threads relates to a larger one, since it is not necessarily either an over- or under-approximation: Interaction between threads can remove behavior in some systems, while the new thread also introduces new behavior. And the incremental SAT solving requires either a relaxing or a constraining of problem instances. Therefore, we focus here on bounding the number of interleavings in a parallel program. Research has shown that most bugs occur after a limited number of interleavings [29,49], so an incremental PDR algorithm can exploit this parameter by solving a 'relaxed' instance bounded to $\ell + 1$ interleavings by reusing the previous results (learned clauses in the SAT solver) from solving an instance of ℓ interleavings.

Another interesting application of incremental PDR is for optimization problems. An example is the PSPACE-complete circuit pebbling problem [40], used to study memory bounds. It asks whether a circuit, viewed as a graph, can be 'pebbled' using p pebbles and the optimization problem is to find the lowest p. These pebbles model the memory required to store signals at the internal wires of the circuit (the working memory): An outgoing wire of a gate can be 'pebbled' if its incoming wires are and pebbles can always be removed (memory erasure). In the reversible pebbling problem [2], pebbles can only be removed if all incoming wires are pebbled, is relevant for reversible and quantum computations. An incremental PDR algorithm can potentially solve a 'relaxed' instance with $p+1$ pebbles faster by reusing the results from a previous run on p pebbles. Moreover, it could approximate the number of pebbles from above by solving a 'constrained' instance with $p - 1$ pebbles, by reusing the same previous results

(we could start for example with p equal the number of gates in the circuit, which is always enough to successfully pebble it).

In Sect. 3, we introduce an incremental PDR (IPDR) algorithm that can both handle relaxing, as well as constraining systems. It runs an adapted PDR algorithm multiple times on instances of increasing (or decreasing) 'constraintness'. We show how the PDR algorithm can be adapted to reuse the internal state from a previous run to achieve this. Moreover, IPDR can combine both constraining and relaxing approaches in a binary search strategy in order to solve optimization problems, such as the pebbling problem.

We emphasize here that the incremental approach of IPDR is orthogonal to the incrementality already found in the original PDR algorithm: Its original name IC3 comes from $(\text{IC})^3$, which stands for: "Incremental Construction of Inductive Clauses for Indubitable Correctness." This refers to the internals of the PDR algorithm which maintain a sequence of increasingly constrained formulas (clause sets) which ultimately converge to the desired inductive invariant. This sequence is also extended *incrementally*. However, IPDR, in addition, incrementally grows a sequence of problem instances that are increasingly (or decreasingly) constrained. This sequence exists in between runs of the adapted PDR algorithm. Both approaches exploit incremental SAT solver capabilities: Or rather, IPDR uses incremental SAT solving in two different and orthogonal ways: inside PDR runs and in between PDR runs on incremental instances.

In Sect. 5, we give an open source implementation of an IPDR-based model checker. We experimentally compare IPDR with Z3's PDR implementation SPACER [22,36] and in our own PDR implementation. From the results, we draw two separate conclusions: 1) for the relaxing of interleavings, IPDR can reuse a large amount of information between increments, which gives roughly a 30% performance gain with respect to our own naive PDR implementation, while outperforming SPACER as well, and 2) for the pebbling problems, when gradually constraining the system, IPDR again reuses many clauses between incremental instances and can achieve performance gains of around 50% with respect to SPACER and our naive PDR implementation, but relaxing or binary search (using both relaxing and constraining) does not help.

2 Preliminaries

Given a set of Boolean variables $X = \{ x_1,\ x_2,\ \ldots,\ x_n \}$, a propositional formula $F(X)$ represents a function $F\colon \mathbb{B}^n \to \mathbb{B}$ where $\mathbb{B} \triangleq \{0, 1\}$. A literal is a variable x_i or its negation $\neg x_i$ (also written as $\overline{x_i}$). A clause is a disjunction of literals and a cube a conjunction of literals. A formula in conjunctive normal form (CNF) is a conjunction of clauses and a formula in disjunctive normal form (CNF) is a disjunction of cubes. A (truth) assignment v is a function $v\colon X \to \mathbb{B}$, which can also be expressed as a cube $\bigwedge_{x_i \in X} x_i \Leftrightarrow v(x_i)$. Vice versa, we can think of the formula F as a set of satisfying assignments $\{ v \in \mathbb{B}^X \mid F(v) = 1 \}$. We often use this duality to interpret a formula $F(X)$ as both a set of system states (satisfying assignments) and its CNF description (which we may explicitly denote with $\mathcal{F}$).

In symbolic model checking, a formula can represent the current set of states of the system under analysis (each satisfying assignment of the constraint formula represents one state). To reason over the next system states, i.e., the system states after the system performs a transition, we use primed variables, e.g., $F(X\prime)$, or more concisely: F'. So F' is obtained by taking every variable x_i in F, and replacing it with the corresponding x'_i. E.g., if $F = (a \wedge b) \vee (a \wedge \neg c)$ then $F' = (a' \wedge b') \vee (a' \wedge \neg c')$. A symbolic transition system (Definition 1) describes the behavior of discrete systems in Boolean logic over a set of Boolean variables X. Example 1 shows how to encode a simple system as an STS.

Definition 1 (Symbolic Transition System (STS)). *A symbolic transition system is a tuple* $TS \triangleq (X,\ I,\ \Delta)$ *where:*

- $S \triangleq \mathbb{B}^X$ *is the set of all system states defined over Boolean variables* $X = \{ x_1,\ x_2,\ \ldots,\ x_n \}$ *(wlog). A system state* $s \in S$ *is an assignment* $X \to \mathbb{B}$.
- $I \subseteq S$ *is a finite set of initial states of the system.*
- $\Delta \subseteq S \times S'$ *is the transition relation. Where* S' *represents the states* S *in the next state of the system. If there exists a pair of states* $(p, q) \in \Delta$*, this means that the system can go from state* p *to state* q *in a single step.*

Example 1. We construct an STS $(X,\ I,\ \Delta)$ for the system in Fig. 1b. First, we define its states over variables $X = \{ x_1, x_2 \}$, denoting 00 for $x_1 = x_2 = 0$, etc. We encode its states as $a = 00$, $b = 01$, $c = 10$ and $d = 11$. The initial states can de encoded as $I(x_1, x_2) = \overline{x_1} \wedge \overline{x_2}$. Finally, the transition relation is encoded as: $\Delta(x_1, x_2, x'_1, x'_2) = \overline{x_1} \wedge x'_1 \Leftrightarrow x_2 \wedge x'_2 \Leftrightarrow \overline{x_2}$ or alternatively as $(\overline{x_1} \wedge \overline{x_2} \wedge \overline{x'_1} \wedge x'_2) \vee (\overline{x_1} \wedge x_2 \wedge x'_1 \wedge \overline{x'_2})$. Both encodings can be efficiently transformed to CNF for use in SAT solver queries (see e.g. [47,51]).

A basic model checking task is to show that an STS $(X,\ I,\ \Delta)$ satisfies an invariant property $P \subseteq S$, i.e., that no state $s \notin P$ is reachable from the initial states I using the transitions encoded in Δ.

Image and Preimage. Given an STS $(X,\ I,\ \Delta)$, the image and preimage under $\Delta(X, X')$ can be used to reason over the reachable states of a transition system. For sets of states $A \subseteq S$, we define the (pre)image of A under Δ as the states (backwards) reachable from A-states in one step as follows.

$$A.\Delta \ \triangleq\ \{ q \in S \mid \exists p \in A \colon \Delta(p, q) \}$$

$$\Delta.A \ \triangleq\ \{ p \in S \mid \exists q \in A \colon \Delta(p, q) \}$$

Of course, a SAT solver can only query individual states (satisfying assignments) $s \in A, t \in B$ for formulas A, B. In practice, however, we will not compute (pre)images, but only whether the (pre)image is contained in a set of states, e.g., $A.\Delta \subseteq B$. This can be done with a SAT solver query $\textsc{sat}(A \wedge \Delta \wedge B')$, which returns false iff no state in A can transit to a state in B, and otherwise returns an example $s \in A, t \in B$ such that $\Delta(s, t) = 1$.

(a) STS Δ *(b) STS Γ*

Fig. 1. STS Δ relaxes Γ, since $\Delta \sqsupset \Gamma$. STS Γ constrains Δ, since $\Gamma \sqsubset \Delta$.

Model Checking using the Inductive Invariance Method. Given an STS (X, I, Δ) and a desired invariant property $P \subseteq S$, model checking can done by computing all reachable states $R = I \cup I.\Delta \cup I.\Delta.\Delta \cup \ldots$ and showing that $R \subseteq P$. This is the approach that BMC takes. Alternatively, Theorem 1 shows that we may also construct an *inductive invariant* (see Definition 2) $F \subseteq P$, which contains the initial states: That is, a strengthening of P, which contains I and is also inductive with respect to the transition relation Δ. E.g., the sets of states $F_1, F_2, \ldots, F_k$ in Fig. 2 prove unreachability of $\overline{P}$ provided they are inductive.

Definition 2 (Inductive Invariant). *$F \subseteq S$ is an inductive invariant for STS (X, I, Δ) if $F.\Delta \subseteq F$ (or $F \wedge \Delta \wedge \neg F' = 0$ as a SAT-solver query).*

Theorem 1 (Inductive Invariant Method [21,26,28]). *A property $P \subseteq S$ is an invariant for STS (X, I, Δ) if and only if there exists an inductive invariant F such that $I \subseteq F$ and $F \subseteq P$.*

In particular, R is the strongest possible inductive invariant for an STS (X, I, Δ) and all invariant properties P that hold for it. But in general, an invariant property $P \subseteq S$ is usually not initially inductive, even if it holds for the STS. For instance, in the case of a mutual exclusion protocol, the property that two processes do not end up in the critical section at the same time, can be violated in states where one process is already in the critical section and the other is about to enter it. However, in a correct protocol these states are unreachable. (And this information is contained exactly in the set of reachable states R.)

Constraining and Relaxing Symbolic Transition Systems. Consider the transition systems *(a)* and *(b)* in Fig. 1. System *(b)* can be obtained by removing transitions from system *(a)*. Intuitively speaking, system *(b)* behaves much the same as system *(a)* and if PDR were to collect clauses to describe the reachability in *(a)*, that information would seem useful when running PDR for system *(b)*. Definition 3 defines relaxing and constraining formally. For instance, if Δ encodes the transition relation of a pebbling problem with at most k pebbles, then we have $\Delta \sqsubseteq \Delta^{\uparrow}$ for $\Delta^{\uparrow}$ encoding the relaxed instance with $k + 1$ pebbles.

Definition 3 (Constrained and relaxed STSs). *An STS $M_1 = (X, I_1, \Delta_1)$ is a constrained version of STS $M_2 = (X, I_2, \Delta_2)$, denoted $M_1 \sqsubseteq M_2$, iff $I_1 \subseteq I_2$ and $\Delta_1 \subseteq \Delta_2$. Vice versa, we say M_2 relaxes M_1, or $M_2 \sqsupseteq M_1$.*

Consequently, $\sqsubseteq$ is a partial order, and $M_1 = M_2$ iff $M_1 \sqsubseteq M_2$ and $M_1 \sqsupseteq M_2$. We denote $M_1 \sqsubset M_2$ ($M_1 \sqsupset M_2$) when $M_1 \sqsupseteq M_2$ ($M_1 \sqsupseteq M_2$) and $M_1 \neq M_2$.

3 Incremental Property Directed Reachability

This section introduces a simplified PDR algorithm. The full version of PDR requires intricate interactions with the SAT solver to attain efficiency (i.e., we omit *generalization* and use set-based notation instead of SAT solver calls). We emphasize that this simplified PDR is not efficient as it treats individual states (and not their generalizations); nonetheless, this description suffices to define IPDR in such a way that it is also compatible with the full version of PDR (as we discuss in Sect. 3.1). For a full description of PDR, we refer to [11,12,23].

Section 3.1 extends PDR with an internal state that Incremental PDR (IPDR) utilizes to reuse information between PDR runs. IPDR, introduced in Sect. 3.2, takes a sequence of constraining (or relaxing) STS instances $M_1, \ldots, M_z$ such that $M_{i+1} \sqsubseteq M_i$ (or $M_{i+1} \sqsupseteq M_i$), solving them one by one with the extended PDR algorithm, while passing the internal state along to speed up subsequent PDR runs. Using the properties of simplified PDR (mostly the invariants defined in Definition 5 maintained by the algorithm that also hold in the full PDR algorithm), we demonstrate how IPDR correctly instantiates incremental PDR runs.

3.1 Extending PDR with an Internal State

Given a TST (X, I, Δ) and an invariant property $P \subseteq S$, constructing an inductive invariant according to Theorem 1 is non-trivial. The PDR algorithm [10,23] approaches this problem by using the concept of *relative inductivity* (Definition 4). To construct the inductive invariant, PDR maintains a sequence $F_0, F_1, \ldots F_k \subseteq S$ of candidate inductive invariants as defined in Definition 5 and illustrated in Fig. 2. The first candidate F_0 is invariably set to the initial states. Initially, the algorithm also sets $k = 0$ and assures that $I \subseteq P$ so the Φ conditions of Definition 5 are satisfied.

By virtue of the fact that PDR maintains the Φ properties as invariants, a few observations can be made. First, the candidates are relatively inductive to their neighbors: By Φ_2, we have $F_{i+1} \cap F_i = F_i$. Paired with Φ_3, this implies

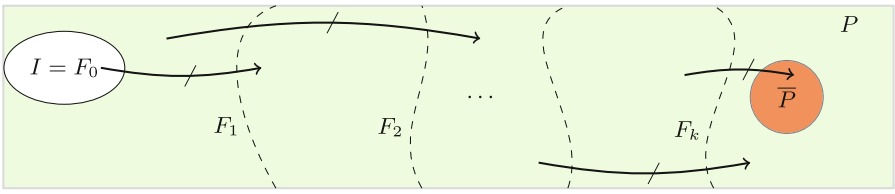

Fig. 2. The box represents all states $S = \{0,1\}^X = 1$. The candidates F_i visualized. All F_i for $i \leq k$ are a subset of P, each F_i is a subset of F_{i+1}, and there is no transition from a state in F_i to a state in F_{i+2}.

that F_{i+1} is inductive relative to F_i. It can also be shown that each candidate F_i over-approximates the set of states reachable within i steps (only states proved unreachable in i steps are blocked in F_i). It follows in turn that, in iteration k of the PDR algorithm, all counterexamples traces (Definition 6) of length k have been eliminated, because otherwise Φ_1 would not hold. Finally, whenever $F_i = F_{i+1}$ for some $i < k$, then Φ_3 implies that F_i is an inductive invariant. Because PDR maintains the candidates F_i as CNF formulas (the candidates are refined by blocking cubes, which is the same as conjoining clauses), this termination check can be done syntactically [12] (without expensive SAT solver call).

Towards incrementing k, the algorithm initializes F_{k+1} to $1 = S = \{0, 1\}^X$ for each k. To make candidate F_{k+1} satisfy Φ_1, PDR proceeds to remove states (block cubes) $s \in F_{k+1} \setminus P$ (initially $s \in \overline{P}$). But removing these states might invalidate Φ_2 and Φ_3. So the algorithm searches backwards to also refine previous candidates F_i. Once a candidate F_i (which overapproximates states reachable in i steps) is strong enough to show that some $s \in F_{i+1}$ cannot be reached in one step, F_{i+1} is refined by removing s by constraining the candidate with the negation of cube s (a clause). We now explain this process in more detail.

Definition 4 (Relative Inductivity). *A formula F is inductive relative to G under a transition relation Δ if $(F \cap G).\Delta \subseteq F$ (or $F \wedge G \wedge \Delta \wedge \neg F\prime = 0$).*

Definition 5. *The sequence of candidate inductive invariants, or simply the candidates, is defined as $F = \{F_0 = I, F_1, F_2, \ldots, F_k\}$. It has the following properties Φ [10], which are maintained throughout the PDR algorithm:*

$$F_0 = I, \tag{Φ_0}$$
$$\forall 0 \leq i \leq k\colon F_i \subseteq P, \tag{Φ_1}$$
$$\forall 0 \leq i < k\colon F_i \subseteq F_{i+1}, \tag{Φ_2}$$
$$\forall 0 \leq i < k\colon F_i.\Delta \subseteq F_{i+1} \tag{Φ_3}$$

Figure 3 gives the simplified PDR algorithm. PDR takes as inputs: an STS according to Definition 1, a property P and a sequence of candidate inductive invariants F, all initialized to 1. It then produces either an inductive invariant, which proves P to be an invariant of the system, or a *counterexample trace*, i.e., a path that shows a violation of P (see Definition 6). To do this, PDR iteratively extends the sequence of candidate inductive invariants $F_1, F_2, \ldots, F_k$ in a major loop (`pdr-main`). To maintain all invariants in Definition 5, the candidates are refined by a minor loop (`block`) within each major loop iteration. This loop uses a queue O of *proof obligations* $(s, i) \in S \times \mathbb{N}$; obligations to show that a state s is not reachable in $i + 1$ steps. If there is no $t \in F_i$ that can transit to s ($\Delta(t, s) = 1$), then (s, i) is removed from O and s is blocked in all candidates $F_1, F_2, \ldots F_{i+1}$, preserving Φ_3 (because the algorithm never adds an obligation (u, j) with $u \in I$ as this would constitute one end of a counterexample trace). The queue O is initialized with $(\overline{P}, k)$, because the algorithm wishes to refine F_{k+1} until it is a subset of P so that Φ_1 is satisfied when k is increased to $k+1$.[1]

[1] Without loss of generality, we may assume that $\overline{P}$ is a single state (i.e., a sink state).

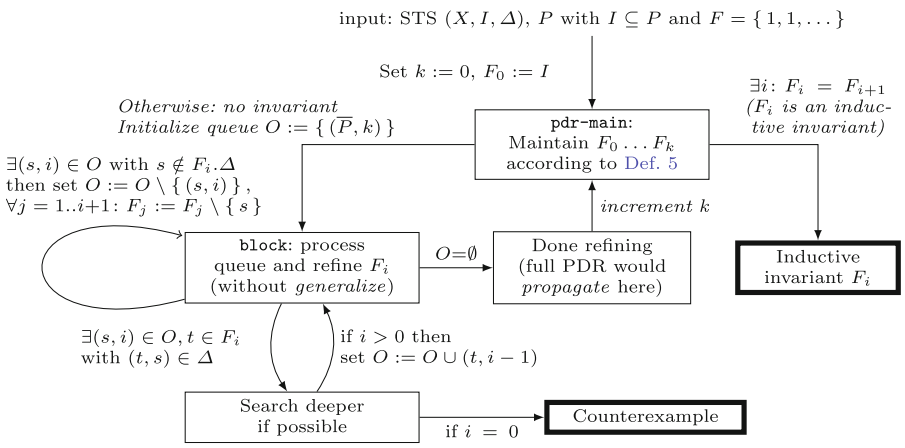

Fig. 3. A simplified PDR algorithm

Once the minor loop completes the search by refining candidates, the major loop continues by incrementing k.

Definition 6 (Counterexample trace). *A counterexample trace for an STS (X, I, Δ) and a property P is a path $\pi_0, \pi_1, \dots, \pi_m \in S$ with $\pi_0 \in I$ and $\pi_m \in \overline{P}$.*

We now extend PDR with an internal state for restarting the algorithm. The goal of the internal state is to let PDR suspend the search upon encountering a counterexample or an inductive invariant, to restart it later on a constrained or relaxed instance of the STS. PDR uses two main data structures during its execution the sequence of candidate inductive invariants F and a priority queue O to track outstanding proof-obligations. Definition 7 gives a *valid PDR state* that also records invariants maintained by the proof obligation queue O.

Definition 7 (Valid PDR state). *Given an STS $M = (X, I, \Delta)$ and a property $P \subseteq \{0,1\}^X$, a valid PDR state is a tuple (M, P, k, F, O) that satisfies:*

- *$F = F_0, F_1, \dots, F_k$ with $0 \leq k < 2^{|X|}$ is a sequence of candidate invariants that adheres to the properties Φ from Definition 5, and*
- *$O \subseteq S \times \mathbb{N}$ is a queue (set) of proof obligations adhering to the properties:*

$$\forall (s,\ i) \in O\colon 0 < i \leq k \qquad (\Omega_1)$$

$$\forall (s,\ i) \in O\colon (s,i) = (\overline{P}, k)\ \vee\ \exists (t, i+1) \in O\colon \Delta(s,t) = 1 \qquad (\Omega_2)$$

$$\forall (s,\ i) \in O\colon s \notin F_i \qquad (\Omega_3)$$

The properties Ω follow from the fact that the minor loop (`block`) basically performs a backwards search starting from $\overline{P}$ states. The property Ω_2 in particular requires that all proof obligations lead to a $\overline{P}$ state (via other proof obligations). Only Ω_3 is non-trivial in this respect: It follows from the fact that

Algorithm 1: Propagation at the level of formulas and SAT queries

In : A sequence $F_0, F_1, \ldots, F_k$ (in CNF form $\mathcal{F}_i$) and STS $M = (X, I, \Delta)$.

function `propagate`$(F,\ TS)$

1 **for** $i \leftarrow 1$ **to** $k-1$ **do** ▷ CNF $\mathcal{F}_i$ (set of clauses) represents F_i:
2 **forall** $\mathcal{C} \in \mathcal{F}_i \setminus \mathcal{F}_{i+1}$ **do** ▷ Find the last $\mathcal{F}_i$ containing $\mathcal{C}$
3 **if** $\mathsf{SAT}(\mathcal{F}_i \wedge \Delta \wedge \neg\mathcal{C}') = 0$ **then**
4 $\mathcal{F}_{i+1} \leftarrow \mathcal{F}_{i+1} \cup \{\, \mathcal{C} \,\}$
5 **return** F ▷ Modified indirectly through its formula representation $\mathcal{F}$

in each major iteration k (`pdr-main`) the nonexistence of counterexample traces of length k has been proved, as noted above. Now if s would be in F_i, a counterexample trace of length k would exist, thus contradicting the Φ invariants of the algorithm.

The full version of PDR adds generalization of states and propagation of clauses in candidates F_i, which we briefly explain here, as IPDR uses propagation as well. In the above, the refinement of a candidate F_i is done by blocking (removing) a state $F_i := F_i \setminus \{\, s \,\}$ (in set notation). In reality, PDR maintains F_i as a CNF formula $\mathcal{F}_i$; a conjunction of clauses $\mathcal{C}$ which represent the removed states $\neg\mathcal{C}$. However, before blocking, states are first generalized by dropping literals from $\mathcal{C}$. The generalization g_s of state s can greatly strengthen a candidate F_i by blocking (removing) many states at once (since g_s is a subcube of s, we have $s \Rightarrow g_s$) that can all be proven unreachable in i steps (while taking care not to remove initial states!). So each (blocking) clause in $\mathcal{F}_i$ is a negated cube g_s, a subcube of s.

Generalization enables propagation (see Algorithm 1), as now other blocking clauses in $\mathcal{F}_i$ may be used to strengthen later candidates $\mathcal{F}_{i+1}$. Propagation pushes blocking clauses forward and is done during the minor search (`blocking`) and after it completes (see note in Fig. 3). Both generalization and propagation have been shown to preserve the Φ invariants [12]. They also do not affect the proof obligation queue. Therefore, Definition 7 holds in the full PDR algorithm.

We can now extend PDR with the internal state $Y = ((X, I, \Delta), P, k, F, O)$ from Definition 7, by modifying the inputs and initializations on the initial edge in Fig. 3. The inputs and variables are now all set to those provided by a valid internal state (obtained from a previous run). In the first `pdr-main` loop iteration, O should also not be reset to $\{\, (\overline{P}, k) \,\}$ but instead taken from Y. When the algorithm terminates with a counterexample or an invariant, it should also return the current internal PDR state. Note that the valid PDR state does not record the line at which the PDR algorithm currently is. Consequently, we will be able to restart it with a modified but valid PDR state to obtain a different result (e.g., to avoid a counterexample in the next run by further constraining the system).

3.2 Incremental Property Directed Reachability (IPDR)

IPDR takes a sequence of constraining (or relaxing) STS instances $M_1, \ldots, M_z$ such that $M_{i+1} \sqsubseteq M_i$ (or $M_i \sqsubseteq M_{i+1}$), solving them one by one with the extended PDR algorithm (`pdr-main`), while passing the internal state along to speed up subsequent PDR runs. Here we define relaxing (and constraining) IPDR as a loop around the PDR algorithm (extended with internal state). In relaxing IPDR, the outer loop terminates when PDR finds a counterexample trace. But when PDR finds an inductive invariant for the current instance M_i (wrt property P), IPDR relaxes the instance to some $M^{\uparrow} = M_{i+1} \sqsupseteq M$ and calls PDR iteratively. In constraining IPDR, the algorithm instead terminates when an inductive invariant is found and iterates on a constrained version of the STS when a counterexample is found. In both cases, we show how to modify the internal PDR state such that is a valid internal PDR state for the constrained or relaxed system.

Constraining IPDR Algorithm. Algorithm 2 shows constraining IPDR. Initially, `pdr-main` is called on the system M_1 wrapped in a valid PDR state (Line 5). Effectively, `pdr-main` is initialized like in Fig. 3. Then the algorithm considers the constrained systems $M_2, \ldots, M_z$ iteratively. If the previous `pdr-main` returned an inductive invariant, then the algorithm stops at Line 8, as further constraining the system is not necessary.[2] Otherwise, a more constrained system $M_i \sqsubset M$ or $M_{c+1} \sqsubset M$ is considered next (we may skip instances $M_i..M_c$ when a counterexample is found that is valid in M_c for $c \geq i$, as Line 10 does). However, we first update the PDR state for the constrained STS $M^{\downarrow} = M_i$ using the `constrain` operation. It resets O at Line 3 to satisfy Ω in the constrained system $M^{\downarrow}$ (repairing O would require at least as many operations as simply restarting the search). It also propagates blocking clauses between candidates $F_1, \ldots, F_k$ as constraining can potentially block them. Finally, `pdr-main` is called again for the updated PDR state and the IPDR continues to the next iteration (Line 12).

Constraining a system does not add behavior, therefore all the Φ invariants remain intact (intuitively: *for the constrained system*, the candidates F_i still are valid over-approximations of the reachable states in i steps and they disprove counterexamples of length i). The propagation does not change this, as it merely strengthens frames while preserving Φ. Because the queue O is emptied, Ω holds vacuously. We conclude that the `constrain` function indeed yields a new valid PDR state according to Definition 7. Consequently, the iterative call to `pdr-main` at Line 12 of Algorithm 2 can proceed incrementally checking the constrained system.

Relaxing IPDR Algorithm. Algorithm 3 shows relaxing IPDR. Like in the constraining version, initially, `pdr-main` is called on the system M_1 wrapped in a valid PDR state (Line 8). Effectively, `pdr-main` is initialized like in Fig. 3. Then

[2] This represents for instance the scenario when we find the minimum number of pebbles to successfully pebble a circuit by approximating the pebble count from above; reducing pebbles in each run until goal of pebbling the circuit is no longer possible.

Algorithm 2: Constraining IPDR (C-IPDR)

In : (M, P, k, F, O) with $F = \{ F_0, \ldots, F_k \}$ satisfying Def. 7 and $M^{\downarrow} \sqsubset M$
Out: A valid PDR *state* for STS $M^{\downarrow}$ according to Def. 7
function `constrain`$((M, P, k, F, O),\ M^{\downarrow})$

$F^{\downarrow} \leftarrow \{ I^{\downarrow}, F_1, F_2, \ldots, F_k \}$
$F^{\downarrow} \leftarrow$ `propagate`$(F^{\downarrow}, M^{\downarrow})$ ▷ See Alg. 1
return $\left(M^{\downarrow}, P, k, F^{\downarrow}, O := \emptyset\right)$ ▷ Repair Ω_2 from Def. 7 by setting $O := \emptyset$

In : $M_1 \sqsubset M_2 \sqsubset \cdots \sqsubset M_z$ with $M_i = (X, I_i,\ \Delta_i)$ and $P \subseteq \mathbb{B}^X$
Out: A counterexample trace or inductive invariant
function `ipdr_constrain`$(M_1, M_2, \ldots M_z,\ P)$

$F \leftarrow \{ F_0 := I_1, F_1 := 1 \}$
(M, P, k, F, O), *result* $\leftarrow$ `pdr-main`$((M_1, P, k := 0, F, O := \emptyset))$
for $M_i \in \{ M_2, \ldots M_z \}$ **do**
  if *result is an inductive invariant* **then**
    return *result*
  if *result is a trace valid in* M_c *for* $c \geq i$ **then**
    forward loop to $M_i := M_{c+1}$
  $(M, P, k, F, O) \leftarrow$ `constrain`$((M, P, k, F, O),\ M_i)$
  (M, P, k, F, O), result $\leftarrow$ `pdr-main`$((M, P, k, F, O))$
return *result*

Algorithm 3: Relaxing IPDR (R-IPDR)

In : (M, P, k, F, O) with $F = \{ F_0, .., F_k \}$ satisfying Def. 7 and $M^{\uparrow} \sqsupset M$ with $M^{\uparrow} = (X, I^{\uparrow},\ \Delta^{\uparrow})$.
Out: A valid PDR *state* for STS $M^{\uparrow}$ according to Def. 7
function `relax`$((M, P, k, F, O), M^{\uparrow})$

$F^{\uparrow} \leftarrow \{ F_0 := I^{\uparrow}, F_1 := 1, \ldots, F_k := 1 \}$ ▷ The tautology 1 is $\emptyset$ in CNF
for $i \leftarrow 1$ **to** $k - 1$ **do** ▷ As Alg. 1; try copy clauses from $\mathcal{F}_i$ to $\mathcal{F}_i^{\uparrow}$
  forall $\mathcal{C} \in \mathcal{F}_i$ **do** ▷ Access the candidates F in CNF form $\mathcal{F}$
    if $\mathsf{SAT}(\mathcal{I}^{\uparrow} \wedge \neg\mathcal{C}) = 0$ **then** ▷ Does $\neg\mathcal{C}$ not block new initial states?
      if $\mathsf{SAT}(\mathcal{F}_i^{\uparrow} \wedge \Delta^{\uparrow} \wedge \neg\mathcal{C}') = 0$ **then** $\mathcal{F}_{i+1}^{\uparrow} \leftarrow \mathcal{F}_{i+1}^{\uparrow} \cup \{ \mathcal{C} \}$
return $\left(M^{\uparrow}, P, k := 0, F^{\uparrow}, O\right)$

In : $M_1 \sqsupset M_2 \sqsupset \cdots \sqsupset M_z$ with $M_i = (X, I_i,\ \Delta_i)$ and $P \subseteq \mathbb{B}^X$
Out: A counterexample trace or inductive invariant
function `ipdr-relax`$(M_1, M_2, \ldots M_z,\ P)$

$F \leftarrow \{ F_0 := I_1, F_1 := 1 \}$
(M, P, k, F, O), *result* $\leftarrow$ `pdr-main`$((M_1, P, k := 0, F, O := \emptyset))$
for $M_i \in \{ M_2, \ldots M_z \}$ **do**
  if *result is a counterexample trace* **then**
    return *result*
  if $\exists s \in I_i^{\uparrow} \cap \overline{P}$ **then**
    return *trace* (s)
  $(M, P, k, F, O) \leftarrow$ `relax`$((M, P, k, F, O),\ M_i)$
  (M, P, k, F, O), result $\leftarrow$ `pdr-main`$((M, P, k, F, O))$
return *result*

the algorithm considers the constrained systems $M_2, \ldots, M_z$ iteratively. Now, if the previous `pdr-main` returned a counterexample trace, then the algorithm stops at Line 11, as further relaxing the system is not necessary.[3] Otherwise, the relaxed system $M_i \sqsubset M$ is considered next using the valid PDR state from the previous run. However, we first update the PDR state for the constrained STS $M^{\downarrow} = M_i$ using the `relax` operation. Here, it constructs a new set of candidate inductive invariants F. Finally, `pdr-main` is called again for the updated PDR state and the IPDR continues to the next iteration (Line 15).

After an instance terminates in iteration k, the `pdr-relax` function checks whether the Φ properties hold for $i = 0$ at Line 12. If not, IPDR found a short counterexample because of newly introduced (and erroneous) initial states in the relaxed instance $M^{\uparrow} = M_i$ and terminates. As relaxing introduces new transitions, the candidates F_i may no longer be strong enough to prove unreachability of states $\neg\mathcal{C}$ for each $\mathcal{C} \in \mathcal{F}_{i+1}$. Therefore, the new PDR-state created by the `relax` function will have to begin from $k = 0$ again in order to strengthen the frames enough to prove unreachability of $\overline{P}$ in multiple steps. Nonetheless, `relax` attempts to preserve as much as possible of the old candidate sequence in a new sequence $F^{\uparrow}$, so that once the PDR run on the relaxed system increases k, it potentially no longer starts with a frame $F_{k+1} = 1$ (or equivalently $\mathcal{F}_i = \emptyset$), but with a subset of the blocking clauses from the previous PDR run. This can be done through a mechanism similar to the propagation phase from Algorithm 1. From Line 2 in `relax`, blocking clauses $\mathcal{C}$ in the original candidates $\mathcal{F}_i$ (the CNF form of F_i) are inspected and copied to $\mathcal{F}_i^{\uparrow}$ if two conditions hold: 1) $\neg\mathcal{C}$ does not block a new initial state in $I^{\uparrow}$, and 2) the candidate $\mathcal{F}_{i-1}^{\uparrow}$ is strong enough to prove unreachability of $\neg\mathcal{C}$ in the relaxed STS (i.e., under transition relation $\Delta^{\uparrow}$).

Because k is set to 0, Φ trivially holds, but the pre-initialized frames are also sound for $\Delta^{\uparrow}$. Because relaxing IPDR only considers a PDR-state tuple when `pdr-main` terminates with an invariant, all outstanding obligations have been eliminated ($O = \emptyset$) before returning the inductive invariant (see Fig. 3). This vacuously satisfies Ω. We conclude that the `relax` function indeed yields a new valid PDR state satisfying Definition 7. Consequently, the iterative call to `pdr-main` at Line 15 of Algorithm 3 can proceed incrementally checking the relaxed system.

Binary Search with Relaxing and Constraining IPDR. Assuming the value of the target optimization parameter equals p, e.g., the minimal number of pebbles required to pebble a circuit, relaxing IPDR needs p PDR calls to find it. Assuming a sound upper bound b on p, e.g., the number of gates in the circuit, constraining IPDR takes $b-p$ calls. By combining the `pdr-relax` and `pdr-constrain` functions, a binary search algorithm takes only $\log(b)$ PDR calls, or $O(\log(p))$ in practice, since often $b = c \cdot p$ [43,48]. (We omit the details here).

[3] This represents for instance the scenario when we find a bug after increasing the number of interleavings in a parallel program.

4 Related Work

Well-structured transitions systems [25] provide another formalization of relaxed and constrained systems (Definition 3) that has been used to verify infinite-state systems like priced [38] and timed automata [37]. Other approaches to deal with infinite-state systems [3,8,15,30,35] extend PDR with SMT [1] using abstraction-refinement [4,17]. In the same vein, [27] extends PDR with symmetry reduction.

Context-bounded analysis [44] in concurrent programming deals with the study of programs by adding restrictions on the context switches of threads. Incrementally increasing parallel interleavings has been exploited for model checking in [29,49]. Reversible pebble game optimization was studied in [43,48].

5 Implementation and Experimental Evaluation

IPDR Implementation. We implemented IPDR in relaxing, constraining and binary search form. The open source implementation in C++ is available at GitHub.[4] It uses the SAT solver Z3 [22] and fully exploits its incremental solving capabilities for PDR (internally) and IPDR (in between incremental runs). It contains the following optimizations that have been discussed before for IC3 and PDR. (None of which interfere with the discussed modifications for IPDR.)

- The delta encoding of [23] avoids duplicating blocked clauses by only storing blocked clauses for the highest frame where it occurs.
- Subsumbtion checks [10,23] avoid storing redundant weaker blocked clauses.
- Generalization [10] with the later extension with the `down` algorithm [31] finds stronger clauses to block. This methods brings along some additional parameters `ctgs` and `max-ctgs`. After some preliminary testing, these parameters were set to 1 and 5 respectively; in line with the findings Bradley [31].
- Before handling a proof-obligation in the minor PDR loop, a subsumption check can quickly detect if newly added clauses already block it [23].
- We also preempt future obligations by re-queueing a proof obligation (s, i) as $(s, i+1)$, since it will have to be proven in later iterations anyway [23].

Benchmarks. As discussed in the introduction, we choose to apply IPDR to the optimization problem of reversible circuit pebbling. We encoded the transition relations as described in [43]. For a number of pebbles p between 1 and g (the number of gates in the circuit), we encode a separate system M_p. It is easy to see that adding more pebbles relaxes the problem, i.e., $M_i \sqsubset M_{i+1}$. We took circuits from the Reversible Logic Synthesis Benchmarks website [41], which lists several "families" of circuits of increasing size and complexity. We selected those circuits that could be completed within half an hour by all benchmarked algorithms.

For experimenting with increasing the number of interleavings [16,29,49], we encoded the Peterson mutual exclusion protocol [46]. We added a scheduler [49] to the encoding to bound the number of interleavings to ℓ, starting from zero.

[4] https://github.com/Majeux/pebbling-pdr.

Experimental Setup. All experiments were run on a computer with 16 GB of 2400 MHz DDR4 memory and an i7 6700T CPU. All benchmarks were performed ten times in a row providing a different random seed to the Z3 SAT-solver. We compare IPDR with our own naive PDR implementation that does not reuse information between runs and with the PDR implementation of Z3's [22]: SPACER [36], for which we re-encoded the systems in Horn clauses.

Results for Peterson. Figure 4 shows the runtimes for the Peterson protocol with 2, 3 and 4 processes with a timeout of four hours. The number of context switches that was feasible to run: `Peterson2` was verified up to a bound of 10 switches, `Peterson3` to a bound of 4 and `Peterson4` to 3. With our Horn clause encoding, SPACER is not competitive. With the most incremental steps, `Peterson2` achieved a speedup of almost a factor four over naive PDR. `Peterson3` and `Peterson4` both showed a similar improvement of around a factor 1,7.

Input	Naive PDR	SPACER	IPDR
`Peterson2`	39 s	307 s	10 s
`Peterson3`	599 s	15432 s	343 s
`Peterson4`	10321 s		6328 s

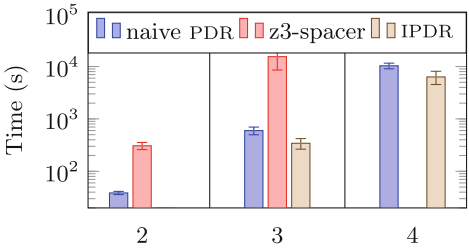

Fig. 4. Average runtimes with standard deviation for Peterson's protocol for naive PDR, SPACER and relaxing IPDR. For `Peterson4`, SPACER timed out.

Results for Pebbling. Figure 5 shows the benchmark results for the constraining (C-IPDR), relaxing (R-IPDR) and binary search versions of IPDR on the pebbling problem. In over half of the benchmarks, constraining IPDR achieves a speedup of around roughly 50% (a factor two) compared to naive PDR and SPACER. For larger benchmarks, however, SPACER has a clear advantage, reducing runtimes by a factor five, a trend that persists for the relaxing and binary search strategies as well. Relaxing IPDR appears to achieve little advantage over the other methods, a result contrary to what we observed for the Peterson protocol.

Finally, we see that speedups of the constraining version are preserved by binary search, but not improved. Internal statistics clearly show that a portion of the incremental runs complete extremely fast, because they are either over- or under-constrained. However, when the binary search approaches the optimal pebble count, the runs become expensive (a well-known threshold behavior in SAT solving [32–34]). The binary search is unable to reduce the number of expensive incremental runs close to the optimal number of pebbles, because it is now approached from both above and below.

Statistics. Figure 6 presents internal IPDR statistics for the `ham7tc` circuit (more details in [9]). These measurements reveal that constraining IPDR reduces the number of counterexamples to induction (CTIs) [10] by a factor six in incremental instances compared to naive PDR. This factor tends to reduce when approach-

	Input	Naive PDR	SPACER	IPDR	Speedup vs. naive	Speedup vs. SPACER
Constraining strategy	ham3tc	0.031 s	0.281 s	0.031 s	1%	89%
	mod5d1	1.942 s	2.850 s	1.067 s	45%	63%
	gf2^3mult_11_47	2.545 s	2.670 s	2.751 s	-8%	-3%
	nth_prime4_inc_d1	3.428 s	4.098 s	2.878 s	16%	30%
	4b15g_1	34.711 s	13.225 s	11.513 s	67%	13%
	4_49tc1	55.970 s	37.466 s	18.761 s	66%	50%
	5mod5tc	989.706 s	377.234 s	748.969 s	24%	-99%
	hwb4tc	30.292 s	30.153 s	9.865 s	67%	67%
	gf2^4mult_19_83	19.618 s	25.678 s	111.350 s	-468%	-334%
	rd73d2	836.655 s	351.181 s	499.460 s	40%	-42%
	mod5adders	57.610 s	57.699 s	34.781 s	40%	40%
	ham7tc	386.719 s	90.823 s	170.244 s	56%	-87%
	5bitadder	1396.920 s	341.008 s	964.889 s	31%	-183%
	gf2^5mult_29_129	760.006 s	242.379 s	1243.961 s	-64%	-413%
Relaxing strategy	ham3tc	0.041 s	0.144 s	0.039 s	6%	73%
	mod5d1	0.832 s	1.413 s	0.646 s	22%	54%
	gf2^3mult_11_47	1.323 s	1.416 s	1.159 s	12%	18%
	nth_prime4_inc_d1	2.647 s	2.433 s	2.806 s	-6%	-15%
	4b15g_1	16.901 s	8.773 s	19.870 s	-18%	-126%
	4_49tc1	43.018 s	19.833 s	63.133 s	-47%	-218%
	5mod5tc	845.274 s	324.640 s	747.978 s	12%	-130%
	hwb4tc	15.325 s	14.018 s	22.853 s	-49%	-63%
	gf2^4mult_19_83	5.755 s	8.441 s	5.366 s	7%	36%
	rd73d2	634.672 s	202.320 s	654.798 s	-3%	-224%
	mod5adders	45.037 s	38.639 s	28.258 s	37%	27%
	ham7tc	151.867 s	53.712 s	211.390 s	-39%	-294%
	5bitadder	639.886 s	91.152 s	474.993 s	26%	-421%
	gf2^5mult_29_129	211.999 s	64.588 s	255.076 s	-20%	-295%
Binary search strategy	ham3tc	0.135 s	0.284 s	0.043 s	68%	85%
	mod5d1	1.297 s	1.573 s	1.225 s	6%	22%
	gf2^3mult_11_47	2.124 s	1.861 s	3.169 s	-49%	-70%
	nth_prime4_inc_d1	3.776 s	4.239 s	3.041 s	19%	28%
	4b15g_1	28.433 s	10.814 s	12.944 s	54%	-20%
	4_49tc1	47.795 s	21.864 s	20.671 s	57%	5%
	5mod5tc	938.345 s	363.142 s	738.673 s	21%	-103%
	hwb4tc	19.572 s	17.900 s	8.961 s	54%	50%
	gf2^4mult_19_83	13.916 s	13.320 s	95.186 s	-584%	-615%
	rd73d2	714.732 s	279.362 s	594.911 s	17%	-113%
	mod5adders	45.037 s	39.628 s	29.790 s	34%	25%
	ham7tc	289.942 s	72.480 s	176.663 s	39%	-144%
	5bitadder	941.210 s	110.511 s	497.072 s	47%	-350%
	gf2^5mult_29_129	423.146 s	124.520 s	634.630 s	-50%	-410%

Fig. 5. Average runtimes the constraining, relaxing and binary search strategies to solve the pebbling problem with naive PDR, SPACER and IPDR. We omit standard deviations, as these are as insignificant as in Fig. 4. Speedup denotes the percentage runtime decrease or increase (gray) achieved by IPDR: $1 - \frac{\text{time(IPDR)}}{\text{time(other)}}$.

ing the optimal number of pebbles (threshold behavior that was observed elsewhere [32–34]). Nonetheless, the result is that IPDR is consistently about as fast for increment i as naive PDR for increment $i + 1$. For relaxing IPDR, we do not always observe this behavior (also not for instances with positive speedups), but R-IPDR is able to copy 60% of the blocked clauses between increments [9]. However, the copying process is expensive for the `ham7tc` circuit, which negates any performance benefits for the subsequent incremental IPDR run.

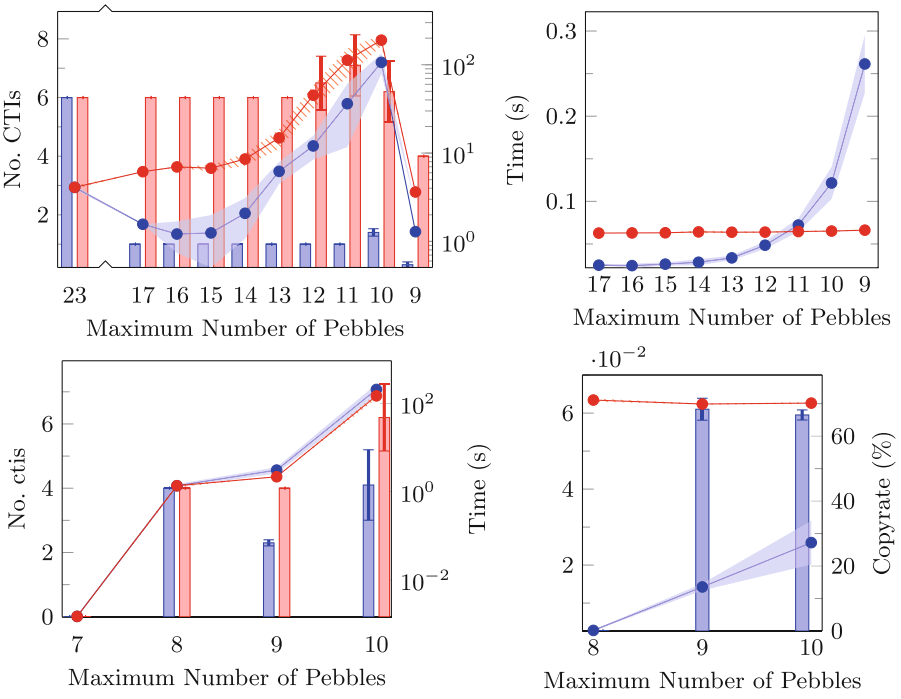

Fig. 6. Statistics for the constraining (top) and relaxing (bottom) `ham7tc` experiment: CTI count (bars left) and time (line left), percentage clauses copied between R-IPDR iterations (bars bottom right), copy time for R-IPDR (lines bottom right) and propagation time for C-IPDR (lines top right). For 23 pebbles, C-IPDR finds a trace with 18 pebbles, so it continues with 17 pebbles (Line 10).

6 Conclusions

We introduced Incremental Property Directed Reachability (IPDR), which harnesses the strength of incremental SAT solvers to prove correctness of parameterized systems. Since PDR does not use unrolling like in bounded model checking, we identified other structural parameters for IPDR to exploit: a bound on the

number of interleavings in the parallel program and the maximum number of pebbles used to solve the pebbling game optimization problem.

With an open source implementation of IPDR, we demonstrated that the incremental approach can optimize pebbling games and model check parallel programs faster than SPACER [22,36]. Internal counters from the IPDR implementation reveal that ample of information is reused in both relaxing and constraining incremental runs. We therefore expect that further research on different parameters and other problem instances will reveal more benefits of the IPDR approach.

References

1. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
2. Bennett, C.: Time/space trade-offs for reversible computation. SIAM **18**, 766–776 (1989)
3. Beyer, D., Dangl, M.: Software verification with PDR: an implementation of the state of the art. In: TACAS 2020. LNCS, vol. 12078, pp. 3–21. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_1
4. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Cortellessa, V., Varró, D. (eds.) FASE 2013. LNCS, vol. 7793, pp. 146–162. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37057-1_11
5. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
6. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. In: Handbook of Satisfiability, vol. 185, no. 99 (2009)
7. Biere, A., Jussila, T. (eds.) Hardware Model Checking Competition 2007 (HWMCC07). LNCS, vol. 10867. Springer, Heidelberg (2007)
8. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (CTIGAR). In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 831–848. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_55
9. Blankestijn, M., Laarman, A.: Incremental property directed reachability. arXiv preprint arXiv:2308.12162 (2023)
10. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
11. Bradley, A.R.: Understanding IC3. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 1–14. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_1
12. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: FMCAD 2007, pp. 173–180. IEEE (2007)
13. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
14. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS, pp. 428–439 (1990)

15. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 277–293. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_23
16. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. STTT **2**, 279–287 (1999)
17. 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
18. Clarke, E., Henzinger, T., Veith, H., Bloem, R.: Handbook of Model Checking. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
19. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC, STOC 1971, pp. 151–158. ACM (1971)
20. Coudert, O., Madre, J.C.: A unified framework for the formal verification of sequential circuits. In: Kuehlmann, A. (ed.) The Best of ICCAD, pp. 39–50. Springer, Boston (2003). https://doi.org/10.1007/978-1-4615-0292-0_4
21. de Bakker, J.W., Meertens, L.G.L.T.: On the completeness of the inductive assertion method. J. Comput. Syst. Sci. **11**(3), 323–357 (1975)
22. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
23. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD 2011, pp. 125–134 (2011)
24. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. ENTCS **89**(4), 543–560 (2003)
25. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoret. Comput. Sci. **256**(1–2), 63–92 (2001)
26. Floyd, R.: Assigning meanings to programs. In: Colburn, T.R., Fetzer, J.H., Rankin, T.L. (eds.) Program Verification, pp. 65–81. Springer, Dordrecht (1993). https://doi.org/10.1007/978-94-011-1793-7_4
27. Goel, A., Sakallah, K.: On symmetry and quantification: a new approach to verify distributed protocols. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NFM 2021. LNCS, vol. 12673, pp. 131–150. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76384-8_9
28. Gribomont, E.P.: Atomicity refinement and trace reduction theorems. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 311–322. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61474-5_79
29. Grumberg, O., et al.: Proof-guided underapproximation-widening for multi-process systems. In: POPL, pp. 122–131. ACM (2005)
30. Günther, H., Laarman, A., Weissenbacher, G.: Vienna verification tool: IC3 for parallel software. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 954–957. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_69
31. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in IC3. In: 2013 Formal Methods in Computer-Aided Design, pp. 157–164 (2013)
32. Heule, M.: Schur number five. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32, no. 1 (2018)
33. Heule, M., Kullmann, O.: The science of brute force. Commun. ACM **60**(8), 70–79 (2017)

34. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 228–245. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_15
35. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13
36. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_2
37. Laarman, A., Olesen, M.C., Dalsgaard, A.E., Larsen, K.G., van de Pol, J.: Multi-core emptiness checking of timed Büchi automata using inclusion abstraction. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 968–983. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_69
38. Larsen, K., et al.: As cheap as possible: efficient cost-optimal reachability for priced timed automata. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 493–505. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_47
39. Levin, L.A.: Universal sequential search problems. Problemy Peredachi Informatsii **9**(3), 115–116 (1973)
40. Lingas, A.: A PSPACE complete problem related to a pebble game. In: Ausiello, G., Böhm, C. (eds.) ICALP 1978. LNCS, vol. 62, pp. 300–321. Springer, Heidelberg (1978). https://doi.org/10.1007/3-540-08860-1_22
41. Maslov, D.: Reversible Logic Synthesis Benchmarks Page. https://reversiblebenchmarks.github.io/. Accessed 24 July 2021
42. McMillan, K.L.: Symbolic Model Checking. Springer, New York (1993). https://doi.org/10.1007/978-1-4615-3190-6
43. Meuli, G., et al.: Reversible pebbling game for quantum memory management. In: DATE, pp. 288–291. IEEE (2019)
44. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455. ACM (2007)
45. Nadel, A., Ryvchin, V.: Efficient SAT solving under assumptions. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 242–255. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_19
46. Peterson, G.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)
47. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. **2**(3), 293–304 (1986)
48. Quist, A.-J., Laarman, A.: Optimizing quantum space using spooky pebble games. In: Kutrib, M., Meyer, U. (eds.) Reversible Computation. LNCS, vol. 13960, pp. 134–149. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-38100-3_10
49. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 82–97. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_9
50. Silva, J.P.M., Sakallah, K.A.: Grasp – a new search algorithm for satisfiability. In: CAD, pp. 220–227 (1997)

51. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J.H., Wrightson, G. (eds.) Automation of Reasoning, pp. 466–483. Springer, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_28
52. Welp, T., Kuehlmann, A.: QF_BV model checking with property directed reachability. In: DATE, pp. 791–796. EDA Consortium (2013)
53. Wieringa, S.: On incremental satisfiability and bounded model checking. In: CEUR Workshop Proceedings, vol. 832, pp. 13–21 (2011)