Leiden University

Master's Thesis

Solving incrementally harder instances with Incremental Property Directed Reachability

Author:
Max Blankestijn

Supervisor:
Dr. Alfons Laarman
Dr. Tim Coopmans
Arend-Jan Quist, MSc

A thesis submitted in fulfilment of the requirements for the degree of Master of Computer Science

in the

Theory Group
Leiden Institute of Advanced Computers Science (LIACS)

October 11, 2023

Abstract

Property Directed Reachability (PDR) is a widely used technique for formal verification of hardware and software systems. This thesis 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 number of clauses between instances, for both the constraining and relaxing algorithm.

Contents

1 Introduction

Our lives are becoming increasingly intertwined with digital systems. One only needs to look around to find an abundance of computing devices around us, crucial to the internet, embedded in medical devices, and many more examples to count. The number of these devices increases every day, and our dependence on such systems with it. Their users often put a blind trust within these systems, despite the fact that their failure can come at a great cost. In order for that trust to be warranted, it is vital that the producers of such a system are able to verify that they behave as we expect them to.

Having a fault in the wrong system in the wrong place can have catastrophic consequences, as was clearly demonstrated on the 4th of June 1996. The European Space Agency (ESA) launched the first test flight for the Ariane-5 rocket. A conversion between a 64-bit floating point number and a 16-bit integer triggered a hardware exception, halting some of its systems. As a result, Ariane-5 veered of course 37 seconds into its launch. The rocket broke apart due to the difference in launch angle and finally self-destructed. Post-launch simulations have faithfully reproduced the system failure, indicating that this software bug could have been detected beforehand.

The demand for high-speed computing only complicates matters further. This has introduced a need for parallel computing paradigms, which allow multiple computational cores to work on the same problem. The systems to be considered are only growing to be more complex.

1.1 The problem of software verification

Program testing can be used to show the presence of bugs, but never to show their absence! – Edsger W. Dijkstra

The intuitive first step to gain confidence in the correctness of a system is testing, where the behaviour of a system is observed when given a variety of inputs. As is pointed out by Dijkstra, this method is often incomplete. It is often unfeasible to consider all different configurations of components and possible inputs of a system, requiring one to only consider a limited subset of them.

Formal verification of software is a discipline that can rigorously prove the absence of bugs, but doing so is hard. There are various formal verification methods, one of which is an elaborate proof that is written out with pen and paper. There are software proof-assistants, but these still require plenty of work to formulate proof techniques. An automated way of verifying software is thus greatly desired, and is delivered by *model checking*.

A model checker is given a system to verify and some specification of its desired behaviour. It then reports whether or not the system adheres to the specification. To do this, both the system and the desired behaviour must be mathematically described. The model checker then exhaustively analyses the behaviour of the system. State-of-the art model checkers encode systems symbolically into logic and use powerful SAT- and SMT-solvers to reason over the system.

Bounded Model Checking (BMC) [8, 10] solves a problem instance by incrementally increasing the number of steps the system-under-verification can take, slowly increasing the hardness of the problem. Incremental SAT-solving techniques [9, 41, 71, 80, 83] are able to retain learned information from solving previous instances and reuse them for related instances. Figure 1.1 [83] shows how incremental SAT-solving is able to reduce the cumulative runtime of solving a series of bounded formulae when incrementally increasing the bound.

BMC does still have its own limitations. Whenever BMC performs a new iteration, it has to extend the logical formula that describes the system to compute a greater state-space. As these formulae become substantially large, they start to become a bottleneck. This is called the *unrolling* of the transition relation.

The PDR [17, 39] algorithm is a state-of-the-art method that circumvents this unrolling by incrementally producing refinements of the property until it is invariant. These can be computed with a series of much smaller SAT-queries than BMC performs. Each query reason over only one step of the transition relation.

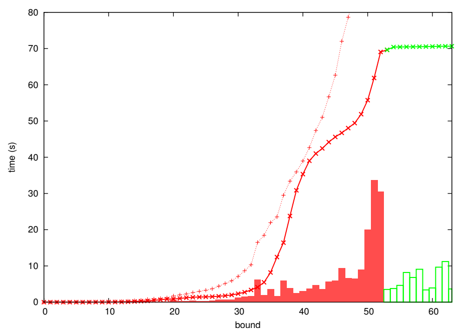
Automating the verification process removes the requirement for the user to come up with complicated proofs techniques, but the model check is limited by a phenomenon known as the *state-explosion*. As the number of variables grows, the state-space of the system grows exponentially. Model checking can problems quickly run into problems with memory space and long run-times. So even for finite systems, model checking remains a hard problem.

There are also cases where a system is infinite in size. In the case of a software system, the size of some data-structure in the algorithm could be infinite, i.e. a stack that has no maximum size, the algorithm could be recursive, or it could be a parallel algorithm that must work for any number of parallel processes. The methods verifying such a system are, unsurprisingly, undecidable in the general case.

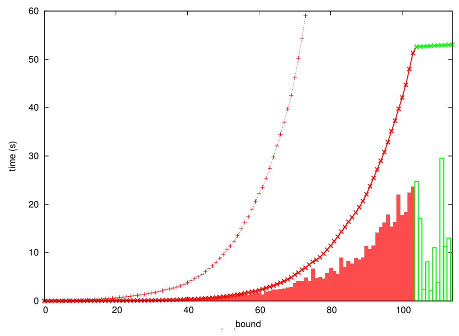
1.2 Approach

This thesis draws inspiration from BMC and other incremental SAT-solving techniques in order to leverage them for IC3. BMC's unrolling depth is a natural parameter to incrementally increase the hardness of a problem and leverage incremental SAT. PDR performs no unrolling and thus needs new parameters. The question arises: *can PDR take similarities between instances n and $n+1$ of the same problem to reuse the results of the evaluation of instance n for instance $n+1$?*

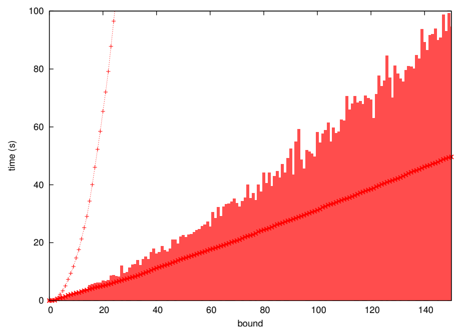
One interesting, and perhaps natural, parameter to consider is the number of parallel threads in a system. However, systems with fewer threads are not



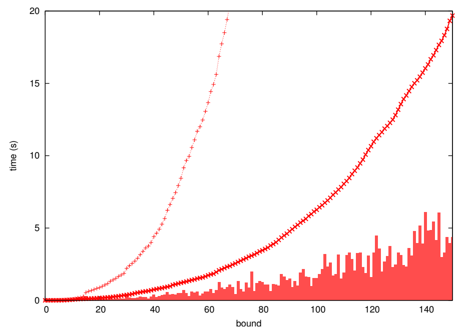
(a) Benchmark for irst.dme6 [7]



(b) Benchmark for bc57sensors.p2neg [7]



(c) Benchmark for eijk.S1238.S [7]



(d) Benchmark for abp4.ptimoneg [7]

Figure 1.1: Benchmarks from Incremental Satisfiability for Bounded Model Checking [83]. The bars denote runtimes for solving a formula with a corresponding bound, red is UNSAT and green is SAT. The plusses + denote cumulative non-incremental runtimes for that bound k: the runtimes of bound 0 to k added together. The crosses $\times$ denotes the runtimes that an incremental solver reported for proving a formula up to a certain bound.

always under approximations of systems with more: adding threads may remove behaviour in some systems while adding new behaviour. However, incremental SAT demands a direct relaxing or constraining of the system: if behaviour is removed then none may be added and vice versa. Thus, we instead focus on bounding the number of interleavings in a parallel system. Previous work has already shown that most bugs in a system occur after only a limited number of interleavings [22, 78]. Starting verification for l interleavings and incrementally increasing to $l + 1$ can exploit this to prove systems correct up to a certain number of interleavings. Giving a reasonable confidence into the correctness of the system. Section 5.2 implements this for the well known Peterson's Algorithm [73].

Another interesting application is for optimisation problems. For instance the Pebbling Problem [63], a PSPACE-complete problem used to study memory bounds. A circuit is interpreted as a graph, which can be pebbled using p

‘pebbles’. A pebble models the working memory of a circuit: the wires on which to store signals. An outgoing wire of a gate can only be pebbled if its incoming wire are. The optimisation problem is to find the lowest possible p for which the graph can be pebbled. An incremental PDR algorithm can solve a relaxed instance with $p + 1$ pebbles or a relaxed instance with $p - 1$ after solving an instance for p pebbles. This allows the algorithm to traverse the search space and find an optimum. Section 5.3 implements a constraining and relaxing algorithm the Reversible Pebbling Game and a binary search algorithm that combines both.

This work adapts PDR to allow previously learned information to be used in subsequent runs and provides such incremental parameters.

1.3 Contributions

In this thesis, we give the new *Incremental PDR* algorithm (IPDR), detailed in Section 4. IPDR is extension of the original algorithm by Aaron R. Bradley [17] that allows previously learned information to be reused in monotonically increasing or decreasing instances.

IPDR requires a reformulation of the original PDR [17]. We modify PDR to accept a PDR-state as input. This state contains the data structures of the PDR algorithm and asserts their properties to ensure correct execution of PDR. These properties are based on invariants maintained by Bradley in the original PDR.

We also provide an open-source implementation [13], that is publicly available on GitHub[1]. A paper, based on this work, has been accepted for publication [14, 15].

This implementation was experimentally evaluated in Section 6. The relaxing version of IPDR, which works on monotonically increasing systems, is evaluated in Section 6.3 using Peterson’s Algorithm [53, 73] as a system-under-verification. Results showed consistent speedups when using relaxing IPDR to incrementally increase the number of allowed interleavings. The constraining version, which works on monotonically decreasing systems, is evaluated in Section 6.3 alongside the relaxing version using the Reversible Pebbling Game [4]. The game is put as an optimisation problem to optimise the number qubits used in quantum circuits [68] by finding minimal pebbling strategies. A set of circuits [66] serves the inputs for the pebbling game. Results show that constraining IPDR consistently reduces workloads for the pebbling game, although it seems to be a bade use-case for relaxing IPDR. The pebbling game can also combine the relaxing and constraining algorithm to perform a binary search for the minimal strategy. Results are in line with a combined performance of constraining and relaxing PDR.

[1] https://github.com/Majeux/ipdr

1.4 Overview

This thesis is set up as follows.

- Section 2 will introduce several background subjects, including propositional logic, and model checking. As PDR uses propositional logic to encode a transition system and uses SAT-solvers to query the system. This section will include how this encoding relates to the, perhaps more broadly understood, set theoretic domain.
- Section 3 will explain the PDR algorithm in detail. The high-level operations of the algorithm will then be explained using sets to describe the system and the algorithm's data-structures. There are come specific low-level operations that directly leverage the Boolean encoding of the system that are explained in propositional logic.
- Section 4 modifies the original PDR algorithm to retain clauses and gives the IPDR algorithm uses to incrementally prove finite instances of a given system.
- Section 5 details design decisions made for the PDR and IPDR implementations and certain optimisations from related work that were adapted. It also includes encodings for the Reversible Pebbling Game and for Peterson's Algorithm into propositional logic.
- Section 6 contains the experimental evaluation of IPDR for both case studies. The section first displays and discusses the over runtimes of IPDR for the Reversible Pebbling Game and for Peterson's Algorithm.
- Section 7 goes into further detail of the results from Section 6 and investigates more detailed statistics that were gathered during the experimental runs.
- Section 8 lists several other works that operate within the same domain as IPDR or are state-of-the-art competitors to the algorithm given in this thesis.
- Section 9 will give contain our closing remarks on the results and summarise how well IPDR is able to reuse information on a series of problem instances of increasing difficulty.
- Section 10 lists some work that could be done to further IPDR that fell outside of the scope of this thesis, including work to improve on its performance and to establish proper use cases for IPDR.

2 Background

2.1 Propositional logic

Formulae in propositional logic [16, 82] are defined over a set of the Boolean variables $X = \{x_1, x_2, \ldots, x_n\} \in \mathbb{B}^n$, where $\mathbb{B} \triangleq \{0, 1\}$. A formula $F(X)$ represents a function $F \colon \mathbb{B}^n \to \mathbb{B}$.

A Boolean formula φ can be constructed through the following grammar:

$$\varphi := 0 \mid 1 \mid x \mid \neg\varphi \mid (\varphi_1 \lor \varphi_2) \mid (\varphi_1 \land \varphi_2) \mid (\varphi_1 \Rightarrow \varphi_2) \mid (\varphi_1 \Leftrightarrow \varphi_2)$$

0, 1 and x are atoms, which are combined with the Boolean connectives: negation $\neg$, conjunction $\land$, disjunction $\lor$, implication $\Rightarrow$ and if-and-only-if $\Leftrightarrow$. Parenthesis can often times be omitted by employing a precedence order on the operators. From highest to lowest, this order is: $\neg, \land, \lor, \Rightarrow, \Leftrightarrow$.

We define the following concepts according to standard notions from the literature on satisfiability solving [9].

Definition 2.1. A *literal* is any Boolean variable x or its negation $\neg x$.

Definition 2.2. A *cube* is a conjunction $\land$ of literals.

Definition 2.3. A *clause* is a disjunction $\lor$ of literals.

Note that the negation of a cube can be represented as a clause, via De Morgan's law: the negation of a conjunction is equal to a disjunction of its negated literals, and vice versa.

Cubes and clauses are often treated as sets of literals when reasoning over what literals are included in them. From this perspective, it could be said that

Example 2.1 (Cubes & clauses as sets). Let $a = x_1 \land \neg x_2 \land x_3$ and $b = \neg x_1 \lor x_2 \lor x_3$. It is said that $x_1 \in a$ and that $\neg x_1 \in b$.

One may quantify them using $\exists a_i \in a \colon \ldots$ or $\forall a_i \in a \colon \ldots$.

De Morgan's law could be presented as: $a = \bigvee_{a_i \in a} \neg a_i = \neg x_1 \lor x_2 \lor \neg x_3$.

A function $\mathcal{V} \colon X \to \mathbb{B}$ is a *truth assignment* and is used to evaluate a Boolean formula $F(X)$ as defined in Definition 2.4. If from an assignment $\mathcal{V}$ follows that $[\![F]\!]^{\mathcal{V}} = 1$, it is said that $\mathcal{V}$ *satisfies* $F(X)$.

Definition 2.4 (Boolean Semantics). Let X be a set of Boolean variables and $\mathcal{V}\colon X \to \mathbb{B}$. A propositional formula $F(X)$ has a value $[\![F(X)]\!]^{\mathcal{V}}$, which is given by:

$$\begin{aligned}
[\![x]\!]^{\mathcal{V}} &\triangleq \mathcal{V}(x)\\
[\![\neg\varphi]\!]^{\mathcal{V}} &\triangleq 1 - [\![\varphi]\!]^{\mathcal{V}}\\
[\![\varphi \wedge \psi]\!]^{\mathcal{V}} &\triangleq 1 \text{ if } [\![\varphi]\!]^{\mathcal{V}} = [\![\psi]\!]^{\mathcal{V}} = 1 \text{ else } 0\\
[\![\varphi \vee \psi]\!]^{\mathcal{V}} &\triangleq 1 \text{ if } [\![\varphi]\!]^{\mathcal{V}} = 1 \text{ or } [\![\psi]\!]^{\mathcal{V}} = 1 \text{ else } 0\\
[\![\varphi \Rightarrow \psi]\!]^{\mathcal{V}} &\triangleq [\![\neg\varphi \vee \psi]\!]^{\mathcal{V}}\\
[\![\varphi \Leftrightarrow \psi]\!]^{\mathcal{V}} &\triangleq [\![\neg\varphi \Rightarrow \psi]\!]^{\mathcal{V}} = [\![\neg\psi \Rightarrow \varphi]\!]^{\mathcal{V}}
\end{aligned}$$

Writing a truth assignment as a relation, a set of pairs $(x, \mathcal{V}(x))$, can be cumbersome. Definition 2.5 declares some conventions to display truth assignments. Example 2.2 shows these conventions for a simple truth assignment.

Definition 2.5 (Denoting truth assignments). A truth assignment $\mathcal{V}\colon X \to \mathbb{B}$ for variables $X = x_1, x_2, \ldots x_n$ can be denoted as the following:

As a cube which is satisfiable only by this $\mathcal{V}$:

$$v = \bigwedge_{x_i \in X} \begin{cases} x_i & \text{if } \mathcal{V}(x_i) = 1 \\ \neg x_i & \text{if } \mathcal{V}(x_i) = 0 \end{cases}$$

Or as string of bits $b_1 b_2 \ldots b_n$ where $\forall 1 \leq i < en\colon b_i = 1 \Leftrightarrow \mathcal{V}(x_i) = 1$.

Example 2.2 (Denoting truth assignments). The truth assignment $\mathcal{V}\colon X \to \mathbb{B}$ is defined by $\mathcal{V}(x_1) = 1$, $\mathcal{V}(x_2) = 0$, $\mathcal{V}(x_3) = 0$.

This assignment can be denoted as a cube $c = x_1 \wedge \neg x_2 \wedge \neg x_3$ and as a bitvector 100.

2.2 Boolean satisfiability

The Boolean Satisfiability problem, or SAT, considers some Boolean formula $F(X)$ over the variables X and asks whether there exists an assignment $\mathcal{V}\colon X \to \mathbb{B}$ such that $[\![\varphi]\!]^{\mathcal{V}} = 1$. If such an assignment exists, F is said to be satisfiable, else it is unsatisfiable. Stephen Cook [30] proved SAT to be the first NP-complete problem in 1971. Most notably, this shows that any NP decision problem can be reduced to an instance of the SAT problem.

SAT-solvers are programs that solve the Boolean Satisfiability problem. They take as input a formula and output SAT if a satisfying assignment exists and

UNSAT if it does not.

The modern landscape [9] contains many different SAT-solvers, for example: zChaff, MiniSat and Z3 [36, 40, 70]. Most are descended from the Davis-Putman (DP) algorithm [34], later refined into the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [33]. DPLL is a backtracking algorithm that makes use of the structure of propositional formulae in the CNF-format, see Definition 2.6. As a result CNF-formulae are the de facto input format for SAT-solvers and otherwise formulae can be efficiently converted into CNF using known methods [74, 81].

Definition 2.6. A formula F is in Conjunctive Normal Form (CNF) if it is a conjunction of clauses C. $\bigwedge_{(C \in F)} \bigvee_{(l \in C)} l$.

In addition to asserting that a formula F is satisfiable, many SAT-solvers are able to return a *witness* to satisfiability. [9] A witness can then be mapped to a satisfying assignment of the variables in F.

If F is unsatisfiable, many solvers are able to produce an *unsatisfiable core* [9]: a subset of all clauses in F, which are needed to prove unsatisfiability. Such a core is not necessarily minimal, but algorithms to find smaller core exist [46], minimally unsatisfiable core [37, 58, 62, 72], or cores of minimum size [65, 69, 84].

Incremental SAT-solvers specialise in solving a series of related queries [43, 83]. Such solvers may also allow a query to be extended with a list of *assumptions* [41, 71]. These are a set of literals, i.e. clauses of length 1, which are only asserted for one specific invocation of the SAT-solver. Efficient implementations of SAT-solving under assumptions are key for incremental SAT-solving, where the solutions to problems are obtained by multiple SAT-solver calls.

2.3 Set theory and logic

A propositional formula $\mathcal{F}(X)$, abbreviated to F, can be viewed as a set of satisfying assignments. This is used to interpret a propositional formula as both a set of assignments F and a CNF-formula $\mathcal{F}$ in this thesis. Lower case letters a, b, ... will be used to denote truth assignments from now on, which are represented by a cube $\mathfrak{a}, \mathfrak{b}, \ldots$. Sets of these elements are denoted by capital letters $A, B, \ldots$ and their respective propositional formulae are denoted by $\mathcal{A}, \mathcal{B}, \ldots$.

Definition 2.7 expresses the sets and formulae used in this convention formally. Example 2.3 displays an example for a small formula.

Definition 2.7 (Sets as logic). Given a propositional formula $\mathcal{F}(X)$, defined over the Boolean variables $X \in \mathbb{B}^n$. The set $F \subseteq \mathbb{B}^X$, where $F = \{\, \mathcal{V} \colon X \to \mathbb{B} \mid [\![F]\!]^{\mathcal{V}} = 1 \,\}$, describes all satisfying assignments. Or in other words: the set F such that $\mathcal{V} \in F \Leftrightarrow [\![F]\!]^{\mathcal{V}} = 1$.

Example 2.3. Consider the propositional formula $\mathcal{F} = x_1 \wedge x_2$, defined over the variables $X = \{ x_1,\ x_2,\ x_3 \}$.

F is satisfied by two assignments $a = 110$ and $b = 111$. It has a corresponding set $A = \{ a, b \}$.

Example 2.3 actually reveals a useful property about cubes and truth assignments. A cube containing some literals always includes all larger cubes that contain at least those literals, since the other literals remain free within the formula. This gives rise to Theorem 2.1

Theorem 2.1. Consider a set of Boolean variables $X = \{ x_1, x_2, \ldots x_n \}$ and two cubes over these variables $\mathscr{a}(X), \mathscr{b}(X)$. As per Definition 2.7, these cubes represent the sets of satisfying assignments a and b.

If the literals of $\mathscr{a}$ are a subset of those in $\mathscr{b}$, then b is a subset of a, i.e., $\mathscr{a} \subseteq \mathscr{b} \Leftrightarrow b \Rightarrow a \Leftrightarrow b \subseteq a$.

Several equivalences between operators in propositional logic and operators on sets arise from Definition 2.7. The set F to a formula $\mathcal{F}$ is defined that $\mathcal{V} \in F \Leftrightarrow [\![F]\!]^{\mathcal{V}}$, already giving an equivalence between the $\in$- and $[\![\bullet]\!]^{\mathcal{V}}$-operators. The operators in Table 2.1 show other useful equivalences between set theory and propositional logic.

Set		Logic
$s \in A$	$\Leftrightarrow$	$[\![\mathcal{A}]\!]^s$
$s \in \overline{A}$	$\Leftrightarrow$	$[\![\neg\mathcal{A}]\!]^s$
$s \in A \cap B$	$\Leftrightarrow$	$[\![\mathcal{A} \wedge \mathcal{B}]\!]^s$
$s \in A \cup B$	$\Leftrightarrow$	$[\![\mathcal{A} \vee \mathcal{B}]\!]^s$
$s \in A \setminus B$	$\Leftrightarrow$	$[\![\mathcal{A} \wedge \neg\mathcal{B}]\!]^s$
$A \subseteq B$	$\Leftrightarrow$	$\forall s\colon [\![\mathcal{A} \Rightarrow \mathcal{B}]\!]^s$

Table 2.1: Set theoretic operators and their logical counterparts.

These properties easily arise from Definition 2.7. Example 2.4 shows a derivation for the implication- and conjunction-operators. Showing the relation between the other operators is done similarly and is left as an exercise to the reader.

Example 2.4. The equivalence for subsets $\subseteq$ from Table 2.1 can be proven as follows:

$$\begin{aligned}
\forall s\colon [\![\mathcal{A} \Rightarrow \mathcal{B}]\!]^s = 1 &\Leftrightarrow \forall s\colon [\![\neg\mathcal{A} \vee \mathcal{B}]\!]^s && \text{(Definition 2.4)}\\
&\Leftrightarrow \forall s\colon ([\![\neg\mathcal{A}]\!]^s = 1 \text{ or } [\![\mathcal{B}]\!]^s = 1) && \text{(Definition 2.4)}\\
&\Leftrightarrow \forall s\colon (s \notin A \text{ or } s \in B) && \text{(Definition 2.7)}\\
&\Leftrightarrow A \subseteq B && \text{(subset)}
\end{aligned}$$

The equivalence for set intersection $\cap$ from Table 2.1 can be proven as follows:

$$\begin{aligned}
[\![\mathcal{A} \wedge \mathcal{B}]\!]^s = 1 &\Leftrightarrow [\![\mathcal{A} = 1]\!]^s \text{ and } [\![\mathcal{B} = 1]\!]^s && \text{(Definition 2.4)}\\
&\Leftrightarrow s \in A \text{ and } s \in B && \text{(Definition 2.7)}\\
&\Leftrightarrow s \in A \cap B && \text{(intersection)}
\end{aligned}$$

2.4 Model Checking

Model Checking [28, 76] is a paradigm of fully automated algorithms designed to verify safety properties for hardware and software systems. Such a system is encoded into a model that an algorithm can reason over and a desired property is provided. Over the last decades Model Checking has developed from proving only simple examples to being applied to realistic system designs [2].

Model checking typically focuses on two formalisms: Computation Tree Logic (CTL) [28] and Linear Time Logic (LTL) [75], a subset of the former branching-type time logic. CTL has a bias towards BDD-based model checking and LTL towards SAT-based model checking [9]. Since the latter is the focus of this thesis, only LTL is elaborated upon in the following Section 2.4.3.

2.4.1 Symbolic Model Checking

An initial response to the state-explosion-problem [29] came in the form of Symbolic Model Checking. Instead of explicitly building a graph of reachable states, Symbolic Model Checking represents its system using Boolean formulae. This allows them to be reasoned over with powerful reasoning engines like Binary Decision Diagrams (BDDs) [20, 67] and algorithms using SAT-solvers [8, 17, 39].

This thesis encodes a symbolic transition system given by Definition 2.8, which distinguishes between the current state of the system and the next state of the system after taking a step of the transition relation.

Definition 2.8 (Symbolic Transition System). 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 the Boolean variables $X = \{x_1, x_2, \ldots x_n\}$. A system state $s \in S$ is a truth assignment $X \to \mathbb{B}$ (Section 2.1).

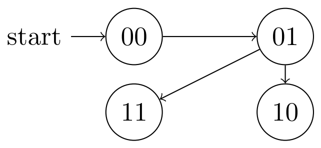


Figure 2.1: A simple transition system $TS = (\{00\}, \{x_1, x_2\}, \Delta)$.

- $I \subseteq S$ 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.

Propositional Logic Encoding. A transition system $TS \triangleq (X, I, \Delta)$ can be encoded into propositional logic, as described in Section 2.3, by interpreting states the system as truth assignments and sets of states as propositional formulae.

Each variable in X has a next-state counterpart, denoted as a primed variable. (Definition 2.9) Similarly, one may denote a next-state version of a propositional formula by "priming" each variable in that formula, see Example 2.5.

Definition 2.9 (Next-state Formulae). Given a $TS = (X, I, \Delta)$, defined over the variables $X = \{x_1, x_2, \dots x_n\}$, the next-state variables are denoted $X' = \{x'_1, x'_2, \dots x'_n\}$.

For a formula $\mathcal{F}(X)$, or $\mathcal{F}$ for short, its next-state version is obtained by substituting each x_i for its next-state counterpart x'_i. The resulting formula is denoted $\mathcal{F}(X')$, or $\mathcal{F}'$ for short.

Example 2.5 (Priming a Formula). Take a propositional formula $\mathcal{F}(X) = (a \wedge b) \vee \neg c$. Then $\mathcal{F}(X') = (a' \wedge b') \vee c'$.

The transition relation Δ is represented by a formula over both current- and next-state variables: $\mathcal{T}(X, X')$. This formula should only be satisfiable by a truth assignment $\mathcal{V}\colon (X \cup X') \to \mathbb{B}$, representing a pair of states $(p, q) \in S \times S'$. Various encodings for a transition relation may exist. Take for instance Example 2.6.

Example 2.6. Take the transition system $TS = (\{00\}, \{x_1, x_2\}, \Delta)$ in figure Figure 2.1. The initial state 00 denotes a truth assignment a where $a(x_1) = 0$ and $a(x_2) = 0$, the state 01 a truth assignment b where $b(x_1) = 0$ and $b(x_2) = 1$.

The transition from state 00 to 01 can be denoted as $\mathcal{T}_{ab} = \neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2'$. $\mathcal{T}_{ab}$ is easily seen to be satisfiable only by the assignment ab where $ab(x_1) = 0$, $ab(x_2) = 0$, $ab(x_1') = 0$ and $ab(x_2') = 1$.

The remainder of the transition formula can be constructed by creating a disjunction each possible transition:

$$\mathcal{T} = (\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2') \vee (\neg x_1 \wedge x_2 \wedge x_1' \wedge x_2') \vee (\neg x_1 \wedge \neg x_2 \wedge x_1' \wedge \neg x_2')$$

2.4.2 SAT-queries

Encoding the symbolic transition system into propositional logic enables the use of a SAT-solver to query transitions from sets of states. The following solver calls on the left are equivalent to the existential quantifiers on the right.

$$\begin{aligned} \mathsf{SAT}(\mathcal{A} \wedge \mathcal{T}) = 1 \quad &\Leftrightarrow \quad \exists s \in A.\Delta \\ \mathsf{SAT}(\mathcal{T} \wedge \mathcal{A}') = 1 \quad &\Leftrightarrow \quad \exists s \in \Delta.A \\ \mathsf{SAT}(\mathcal{A} \wedge \mathcal{T} \wedge \mathcal{B}') = 1 \quad &\Leftrightarrow \quad \exists (p,q) \in A.\Delta.B \end{aligned}$$

The assignments s and (p,q) can be obtained from the witness produced by the SAT-solver. Extracting them into states, assignments or cubes is an implementation detail.

Image and Preimage. Given a $TS = (X, I, \Delta)$, the image and preimage of Δ can be useful to reason over the reachable states of the transition system. This thesis will use the notations in Definition 2.10 to easily denote them.

Definition 2.10. For a set of states $A \subseteq S$, the image of A is defined as all states that are reachable from an A-state in one step:

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

The preimage is defined as the states that are backwards reachable from an A-state in one step:

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

The intersection between the two is denoted by:

$$A.\Delta.B \triangleq A.\Delta \cap \Delta.B \equiv \{\, p \in A, q \in B \mid \Delta(p,q) \,\}$$

2.4.3 Linear Temporal Logic

The propositional logic from Section 2.1 is lacking when trying to reason reason over the future state of a system and is thus often insufficient to provide interesting insights into its behaviour. LTL [75] extends regular propositional logic into *propositional temporal logic* that can do exactly that.

$[\![\text{true}]\!]^\pi$	$=$	1	$[\![\bigcirc\varphi]\!]^\pi$	$\Leftrightarrow$	$[\![\varphi]\!]^{\pi_{1..}}$
$[\![a]\!]^\pi$	$\Leftrightarrow$	$[\![a]\!]^{\pi_0}$	$[\![\Diamond\varphi]\!]^\pi$	$\Leftrightarrow$	$\exists i \geq 0\colon [\![\varphi]\!]^{\pi_{i..}}$
$[\![\neg\varphi]\!]^\pi$	$\Leftrightarrow$	$1 - [\![\varphi]\!]^\pi$	$[\![\Box\varphi]\!]^\pi$	$\Leftrightarrow$	$\forall i \geq 0\colon [\![\varphi]\!]^{\pi_{i..}}$
$[\![\varphi \wedge \psi]\!]^\pi$	$\Leftrightarrow$	$[\![\varphi]\!]^\pi = [\![\psi]\!]^\pi = 1$	$[\![\varphi \cup \psi]\!]^\pi$	$\Leftrightarrow$	$\exists i \geq 0\colon [\![\psi]\!]^{\pi_{i..}}$ and $\forall 0 \leq j < i\colon [\![\varphi]\!]^{\pi_{j..}}$

Table 2.2: Semantics of the propositional and temporal operators in LTL [2, 75]. For a path π and an LTL-formulae φ and ψ.

LTL-formulae typically consider systems without terminal states, and such a formula is satisfied by an infinite path over the given system. Note that any system with terminal states can easily be converted by introducing a sink-state with self-loops.

Definition 2.11. Given a symbolic transition (X, I, Δ), a sequence of states $\pi = \pi_0,\ \pi_1,\ \pi_2,\ \ldots$ is a *path* for *TS* if each state along the path leads the next under Δ.

$$\mathit{Paths}(\mathit{TS}) \triangleq \{\, \pi = \pi_0,\ \pi_1,\ \pi_2,\ \ldots \mid \forall \pi_i \in \pi\colon \Delta(\pi_i, \pi_{i+1}) \,\}$$

LTL-formulae are defined over a set of propositional Boolean variables and uses the propositional operations mentioned in Section 2.1, but extends them with the so-called *temporal operators*.

The semantics of an LTL-formula is evaluated by a path, where the propositional operators consider the first state on the path as shown in Table 2.2.

The temporal operators reason over the entire path. The *next* $\bigcirc$ operator states that a formula holds in the next step on the path. The *eventually* $\Diamond$ operator states that there is some point on the path where the formula holds. The *always* $\Box$ operator states that a formula holds now and in every future step. The *until* $\cup$ operator states that the right-hand formula holds at some point in the future, and that the left-hand formula holds up until that point. The exact semantics are given in Table 2.2.

An entire transition system is said to satisfy an LTL-formula if all paths that start in its initial state do (Definition 2.12).

Definition 2.12. When a transition system $\mathit{TS} = (X, I, \Delta)$ satisfies an LTL-formula φ, it is denoted as follows:

$$[\![\varphi(X)]\!]^{\mathit{TS}} \Leftrightarrow \forall \pi \in \mathit{Paths}(\mathit{TS})\colon (\pi_0 \in I \Rightarrow [\![\varphi]\!]^\pi)$$

Most interesting for this thesis is the always $\Box$ operator. A formula $\Box\varphi$ is called an *invariant* [2]. A safety property (nothing bad should ever happen) is often expressed with an invariant (always: nothing bad should happen). A safety property P can otherwise be described as set of states P. Conversely, the set $\overline{P}$ describes *bad states*. Once a bad state is reachable, the invariant is violated.

2.4.4 Bounded Model Checking

Bounded Model Checking (BMC) [8, 10] was a response to the state-explosion-problem for BDDs. Especially in BDDs, which store all satisfying assignments explicitly, this creates memory constraints [21, 24, 32]. in BMC, a symbolic transition system is encoded into propositional logic, which can be done similarly to earlier reductions by Cook [31] and Levin [60].

BMC seeks to disprove a property by searching for a counterexample in k steps of the transition relation. If no counterexample is found, k is incremented until one is found or until some predetermined bound is reached. These k steps are be computed by *unrolling* the transition relation. For each k, the formula describing the system is incrementally extended by another copy of the transition relation. In the context of an invariant $\Box\mathcal{P}$, any state along a path from the initial state that violates $\mathcal{P}$, or satisfies $\neg\mathcal{P}$, is a counterexample.

Example 2.7 (Unrolled transition relation (set)). Verifying if a symbolic transition system $TS \triangleq (X, I, \Delta)$ upholds a safety property P within $k = 3$ steps:

$$I.\Delta.\Delta.\Delta \cap \overline{P} = \varnothing$$

When encoding the system into logic, and using a SAT-solver to find counterexamples, each step requires their own set of variables to distinguish between them. Example 2.8 shows how an unrolled formula for only 3 steps already requires 4 sets of variables.

Example 2.8 (Unrolled transition relation (SAT)). Verifying if a symbolic transition system $TS \triangleq (X, \mathcal{I}, \mathcal{T})$ upholds a safety property $\mathcal{P}$ within $k = 3$ steps:

$$\mathcal{I}(X) \wedge \mathcal{T}(X, X') \wedge \mathcal{T}(X', X'') \wedge \mathcal{T}(X'', X''') \wedge \neg\mathcal{P}(X''')$$

BMC is naturally limited by the depth of the system-under-verification. As the bound k increases, the formulae for the system blows up in size requiring additional memory for additional variables and increasing runtimes of the underlying SAT-solver. Additionally, this means that BMC is not innately complete: if the search is stopped at a bound k, who is to say that a counterexample with $k+1$ steps does not exist? There are techniques to allow BMC to terminate by computing a *completeness threshold* [9, 25, 26, 41], although this is an additional problem to solve and requires additional computation.

2.4.5 Inductive invariance

Some property P can alternatively be proven to be an invariant for a system if it is an *inductive invariant*. Verifying if P is an inductive invariant is computationally much cheaper than techniques such as BMC, as it only considers a single step of the transition relation.

> **Definition 2.13 (Inductive invariant).** $F \subseteq S$ is an *inductive invariant* for $TS = (X, I, \Delta)$ if $F.\Delta \subseteq F$ (or if $\mathcal{F} \wedge \mathcal{T} \Rightarrow \mathcal{F}'$ in logic),

If P is an inductive invariant and the initial states of the system adhere also to it, then it becomes easy to see that P holds for every state in the system by induction (Theorem 2.2).

Sadly, a given P is usually not innately invariant, but it could potentially be strengthened. As per the *Inductive Invariant Method* , Theorem 2.2, which was proven complete by J.W. de Bakker and L.G.L.T. Meertens [35]: there may exist a strengthening of the property $F \subseteq P$ that is an inductive invariant. BMC can be seen as computing the strongest possible inductive invariant: the reachable states $R = I \cup I.\Delta \cup I.\Delta.\Delta \cup \ldots$ and the showing that $R \subseteq P$. However, an inductive invariant stronger than P but weaker than R may exist.

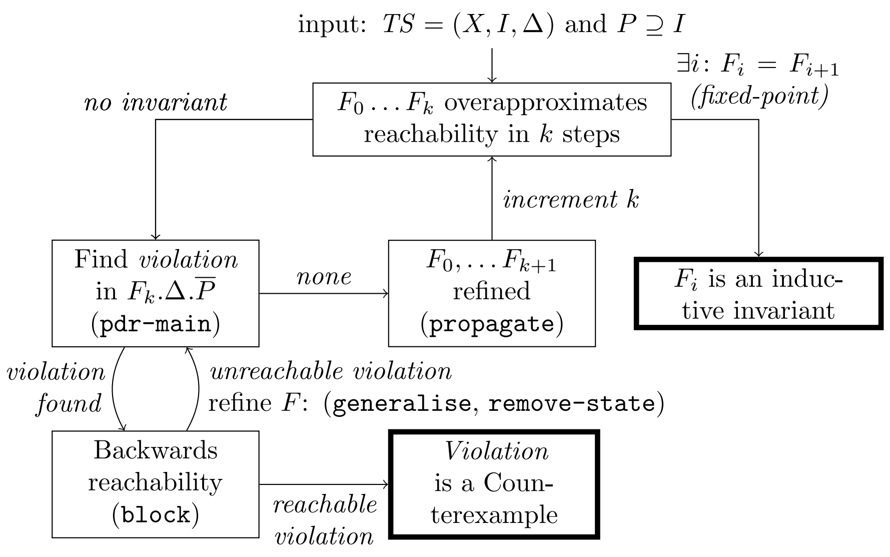
The *Inductive Invariant Method* is able to use a strengthening of P to prove its invariance as described in Theorem 2.2.

> **Theorem 2.2 (Inductive Invariant Method [35, 45, 48]).** A property $P \subseteq S$ is an invariant for $TS = (X, I, \Delta)$ if and only if there exists an inductive invariant F such that $I \subseteq F$ and $F \subseteq P$ (or $\mathcal{I} \Rightarrow \mathcal{F}$ and $\mathcal{F} \Rightarrow \mathcal{P}$ in logic).

3 Reformulating PDR with internal state

Finding an inductive invariant for Theorem 2.2 is a non-trivial task. The *"Incremental Construction of Inductive Clauses for Indubitable Correctness"* (IC3) algorithm, by Aaron R. Bradley [17], solves this problem through the concept of *relative inductiveness* (Definition 3.1). This allows such an invariant to be constructed in incrementally, without unrolling the transition relation as is done in BMC. IC3 was later simplified and improved by Een, Mishchenko and Brayton [39], who put forth the name *Property Directed Reachability* (PDR) for the algorithm.

> **Definition 3.1.** A formula $\mathcal{F}$ is *inductive relative* to $\mathcal{G}$ under a transition relation $\mathcal{T}$ if $\mathcal{F} \wedge \mathcal{G} \wedge \mathcal{T} \Rightarrow \mathcal{F}'$. Which can be represented as the following set-theoretic property using the conversion to the sets F and G from Section 2.3 and Section 2.4.1: $(F \cap G).\Delta \subseteq F$.



input: $TS = (X, I, \Delta)$ and $P \supseteq I$
$\exists i \colon F_i = F_{i+1}$ *(fixed-point)*
no invariant
$F_0 \dots F_k$ overapproximates reachability in k steps
increment k
Find *violation* in $F_k.\Delta.\overline{P}$ (`pdr-main`)
none
$F_0, \dots F_{k+1}$ refined (`propagate`)
F_i is an inductive invariant
violation found
unreachable violation
refine F: (`generalise`, `remove-state`)
Backwards reachability (`block`)
reachable violation
Violation is a Counterexample

Figure 3.1: High-level overview of PDR

This section implements a version of PDR following these two works, extended with an *internal state* in Section 3.1. *Incremental PDR* (IPDR) algorithm, introduced in Section 4, uses this state to share information in between multiple runs of PDR. The internal state contains the primary data structures used in PDR and maintains several invariant properties, derived from Bradley's original work [17], which assert that PDR is able to continue and produce a correct result. This section uses the notations from Section 2.3 to display the algorithm in set-theoretic notations wherever possible, instead of hiding the semantics for manipulating the input transition system behind SAT-solver calls and propositional formulae. Appendix B gives the exact propositional formulae and SAT-query equivalents to these notations. Where PDR specifically interacts with the propositional encoding, propositional logic is used explicitly.

Figure 3.1 gives a high-level overview of the algorithm., which takes a transition system TS from Definition 2.8 and property P as input. PDR builds a *sequence of candidate invariants* $F_0, F_1, \ldots \subseteq \mathbb{B}^X$ (Definition 3.3). Each candidate F_i is at least always inductive relative to their successor under Δ and overly approximate the states reachable in i steps or less.

PDR consists of two main loops. The major loop (Section 3.3) finds states in the last candidate F_k that prevent the sequence from being extended. These states are given to the minor loop, which performs backwards reachability on these states. If such a state is reachable, it reveals a *counterexample-trace* (Definition 3.2). If it is unreachable, the sequence of candidates is refined by removing unreachable states from the over-approximation reachable states. Once sufficiently refined, the sequence can be extended with a new entry $F_{k+1} = P$.

This process continues until PDR encounters a counterexample-trace or until the sequence converges into an inductive invariant.

> **Definition 3.2.** A *counterexample-trace*, for a transition system $TS = (X, I, \Delta)$ and a property P, is a path π of finite length n (Definition 2.11) which starts in an I-state and ends in an $\overline{P}$-state. I.e., $\pi_0 \in I$ and $\pi_n \in \overline{P}$.
>
> If there exists a counterexample-trace for TS, then $[\![\Box P]\!]^{TS}$ does not hold, following this definition and Definition 2.12.

On its own, this approach differs little from state enumeration. PDR abuses Boolean encoding of states as cubes to generalise reachability information: before removing an unreachable state, it drops literals from its cube while ensuring the resulting subcube remains unreachable. As mentioned in Theorem 2.1, these subcubes describe sets of states instead of single states. This allows PDR to eliminate multiple states at once, without requiring additional backwards searches and all the while reducing the size of the resulting Boolean formula.

The following sections will show how these parts operate in more detail.

3.1 Internal PDR state

PDR uses two main data structures during its execution the sequence of candidate inductive invariants F that model the reachable states and a priority queue O that is filled with states that must be proven unreachable, else the property P would fail. Bradley's [17] original work denotes their properties in pre-conditions, post-conditions and assertions for the functions that comprise PDR. This thesis records these properties as invariants in the *internal PDR state* in Definition 3.6. This will later allow IPDR to invoke PDR with a pre-existing PDR state.

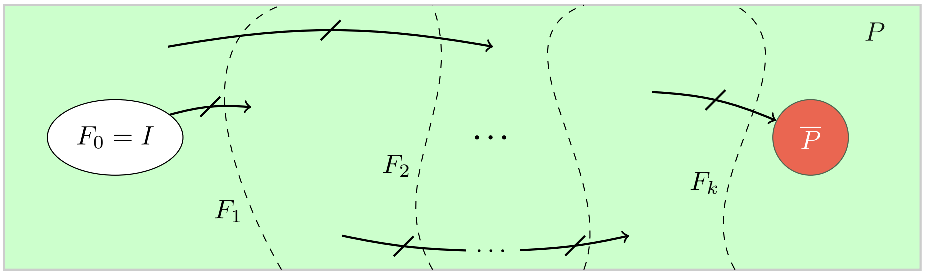


Figure 3.2: Φ (Definition 3.3) visualised. The box represents all states $\mathbb{B}^X$. Each F_i is included in the next, all $F_i \subseteq P$ and there are no transitions to a candidate F_{i+2} or larger. Note that in practice F is initialised such that $\forall i > k \colon F_i = P$.

Candidates. The sequence F is constructed such that each entry is inductive relative to its successor, representing an over-approximation to the set of reachable steps. To this end, F maintains the properties Φ from Definition 3.3. The first entry F_0 of the candidate sequence is set to always comprise exactly the initial states. As reachability is determined once a path to F_0 is found, this ensures that every encountered trace is sound. Every other candidate is initialised to P.

Definition 3.3. Given symbolic transition system $M = (X, I, \Delta)$ and a property $P \subseteq \mathbb{B}^X$. The *sequence of candidate inductive invariants* is defined as $F = \{F_0, F_1, F_2, \ldots\}$. It has the following properties Φ [17], which are maintained throughout the PDR algorithm:

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

These properties are visualised in Figure 3.2.

Some observations can be made from Φ. Firstly, Φ_3 states that all candidates are inductive relative to each other: by Φ_2, $F_i \cap F_{i+1} = F_i$ implying that Φ_3 actually states $(F_i \cap F_{i+1}).\Delta \subseteq F_{i+1}$. The sequence will ultimately be refined by finding clauses, representing negations of states, that are also inductive relative to F_{i+1}. These states can then be safely removed from F_i without compromising Φ.

Secondly, if at any point there exists a candidate $F_i = F_{i+1}$ then $F_i.\Delta \subseteq F_i$ by Φ_3. This reveals F_i to be an inductive invariant and, combined with Φ_0 and Φ_1, this satisfies the the inductive invariant method. Termination of PDR is thus determined as per Theorem 3.1. The remainder of the algorithm is now focused on removing states from this sequence and extending it with a new entry $F_{k+1} = P$ until such a candidate is found.

Theorem 3.1 (PDR Termination). Given a sequence of candidate inductive invariants $F_0, F_1, \ldots$, if $\exists 0 < i \leq k\colon F_i = F_{i+1}$ then $[\![\Box\, P]\!]^{TS}$ holds by the inductive invariant method Theorem 2.2.

Lastly, the relative inductiveness of each candidate F_i can be equated to overly approximating the set of states reachable in i steps or less R_i, i.e., $R_i \subseteq F_i$. By Φ_0, this implicitly holds for F_0 and via induction for every subsequent candidate: F_1 contains I and $I.\Delta$; F_2 contains I, F_1 and $F_1.\Delta$; and so on. This allows for a simple SAT-query to determine if a state is reachable (Theorem 3.2)

Theorem 3.2. Given a $TS = (X, I, \Delta)$, sequence of candidate invariants F and a state $s \in 2^X$. If $\overline{s}$ is inductive relative to some candidate, then s is unreachable within $i + 1$ steps of the transition relation.

If $|s| = |X|$, then this may be represented as $s \notin (F_i \setminus s).\Delta$. Else, by $(F_i \setminus s).\Delta \subseteq \overline{s}'$, or $\mathsf{SAT}(\mathcal{F}_i \wedge \neg s \wedge \Delta \wedge s') = 0$ as a SAT-query.

While each candidate F_i represents a set of states, it is represented as a CNF formula: the property to be verified conjoined with a set of clauses. Each clause represents the negation of a cube, in turn representing a set of states in TS that is being excluded from the candidate. PDR and IPDR both need to reason over the learned clauses in the candidates. They do this via the `clauses` function in Definition 3.4.

Definition 3.4 (Blocked states). Given a property $\mathcal{P}$ and a formula $\mathcal{F}(X)$ in propositional logic of the form $\mathcal{F} = \mathcal{P} \wedge \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \ldots \mathcal{C}_n$.

$$\texttt{clauses}(F_i) \triangleq \{\, \mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_n \,\}$$

In practice, Φ_2 can actually be represented more strongly. As will be shown in Section 3.4, not only are all candidates included in their successor, but they are build using the exact same clauses. When a set of states is removed from a candidate, and thus a clause added to its formula, this is also done for all previous candidates. This gives Lemma 3.1, which allows the equivalence of two adjacent frames to be asserted with a simple syntactic check[2].

Lemma 3.1. Given a sequence of candidate inductive invariants $F_0, F_1, \ldots$.

$$\forall 1 \leq i \leq k \colon \texttt{clauses}(F_{i+1}) \subseteq \texttt{clauses}(F_i)$$

Obligations. Following Theorem 3.2, the major iteration may only eliminate a state from F_k that reaches $\overline{P}$ if that state is unreachable in k steps from the initial states. Such states are inserted into the set of *proof obligations* O, which is designed to extend the knowledge of its elements' unreachability through the minor iteration. O has the properties Ω defined in Definition 3.5. In practice, O is implemented as a priority queue, where the front is an obligation (s, i) with a minimal i.

[2]This syntactic check circumvents potentially expensive semantic checks or SAT-solver invocations.

Definition 3.5 (Proof Obligations). $O \subseteq \mathbb{B}^X \times \mathbb{N}$ is a set of *proof obligations* adhering to the following properties:

$$\forall (s,i) \in O \colon 0 < i \leq k, \tag{Ω_1}$$
$$\forall (s,i) \in O \colon \overline{s} \textit{ is inductive relative to } F_{i-1} \colon s \notin (F_{i-1} \setminus s).\Delta, \tag{Ω_2}$$
$$\forall (s,i) \in O \colon \exists \pi \in \mathit{Paths}(\mathit{TS}), b \in \overline{P} \colon \pi = s, \pi_1, \pi_2, \ldots b \tag{Ω_3}$$

Ω_1 arises from the fact that only states for the current candidates are considered. Ω_2 states that s has already been proven to be unreachable up until the previous i, arising from the way the major iteration searches in incrementally increasing candidates and the way that the minor iteration pushes obligations forward. Ω_3 it not needed for the correctness of the backwards search, but encompasses the "property directed" nature of PDR: each state for which PDR seeks to determine reachability leads to a $\overline{P}$-state.

Internal State Tuple. The above data structures and their properties are gathered into the PDR state in Definition 3.6. If PDR is invoked with a valid state, it will produce a correct result. The remainder of the section will also show that if they hold before a function, it will hold afterwards.

Definition 3.6. Given a symbolic transition system $M = (X, I, \Delta)$ and a property $P \subseteq S$, a *valid PDR state* is a tuple $PDR = (M, P, k, F, O)$ satisfying:

- $F = \{ F_0, F_1, \ldots \}$ with $0 \leq k < 2^{|X|}$ is the sequence of candidate invariants, satisfying Φ from Definition 3.3
- O is a priority queue of proof obligations, satisfying Ω from Definition 3.5.

3.2 Initialisation of PDR

PDR as defined by Bradley [17] begins `pdr` by verifying reachability in 0 and 1 steps. Which allows a PDR state to satisfy Definition 3.6 for $k = 1$.

The queries $i \in I \cap \overline{P}$ and $(i, b) \in I.\Delta.\overline{P}$ reveal counterexample traces in the initial states and within one step. If non exist, then a valid PDR state for $k = 1$, $F_0 = I$, $F_1 = P$ and $O = \varnothing$ is returned.

Algorithm 1, containing the main loop of PDR, simply assumes that it has handed a valid PDR state. A traditional PDR run will have constructed this via an initialisation function verifying the above, while IPDR will use this to input more developed PDR states.

Algorithm 1: The major iteration of PDR

```
In  : A tuple $(M, P, k, F, O)$, satisfying Definition 3.6.
Out : A pair $(trace, PDR)$, where $trace$ is a counterexample to the
      invariance of $P$ and $PDR = (M, P, k, F, O)$ satisfies
      Definition 3.6.

function pdr$(M, P, k, F, O)$:
      ▷ In regular PDR the state is $(M, P, 1, \{ I, P, P, \dots \}, \varnothing)$, as per Section 3.2
1     for $k$ to $\infty$ do
2         forall $(o, p) \in F_k.\Delta.\overline{P}$ do ▷ Implemented as a SAT-query (Appendix B)
3             $O = O \cup (o, k-1)$
4             $trace \leftarrow$ block$(M, P, k, F, O)$
5             if $trace \neq \varnothing$ then
6                 return $(trace, (M, P, k, F, O))$
              ▷ $o \notin F_k$
7         $F \leftarrow$ propagate$(F, k, M)$
8         if $\exists 1 \leq i \leq k \colon F_i = F_{i+1}\colon$ then
9             return $(\varnothing, (M, P, k, F, O))$
```

3.3 Major iteration

The function `pdr` from Algorithm 1 performs the major iteration. Line 2 queries all states in the latest candidate F_k and line 3 adds them to the proof obligations O. If there are no more states left that lead to $\overline{P}$, then Φ hold for $k+1$.

Line 2 is implemented as a SAT-query $\mathsf{SAT}(\mathcal{F}_k \wedge \Delta \wedge \neg\mathcal{P})$. On a positive result, the witness reveals a transition (o, p). The state o is passed to the `block` function, further detailed in Section 3.4, by adding $(o, k-1)$ to O. This indicates that o is already inductive up until F_{k-2} (unreachable in $k-1$ steps) and must be proven inductive to F_{k-1} (unreachable in k steps).[3]

If an obligation (o, i) is unreachable within $i+1$ steps, that state is *blocked* in F_{i+1} and `block` returns $\varnothing$. When a state is blocked in some F_i, it is removed from all candidates $F_1, \dots F_i$.[4] If o is reachable, `block` returns a counterexample trace that leads to $\overline{P}$ through o and PDR terminates in line 6.

Once an obligation has been blocked, line 2 no longer discovers it. Once all obligations for F_k have been eliminated in this way, $F_k.\Delta \subseteq P$ now and thus Φ holds for the sequence $F_0, F_1, \dots F_k, P$. When k is incremented to $k+1$ in the next major iteration, this sequence will be considered with $F_{k+1} = P$.

As PDR executes `pdr` and `block`, F becomes more refined when states are

[3]This naturally maintains Ω_1 and Ω_3. Ω_2 holds for if $o \notin (F_{k-2} \setminus o).\Delta$ did not, then it would have been discovered and blocked in the previous major iteration.
[4]See Section 3.5 for details and how this maintains Φ.

Algorithm 2: The propagation phase

```
In  : $F = F_0, F_1, \ldots$ satisfying $\Phi$. A symbolic transition system
      $M = (X, I, \Delta)$.
Out : $F$, satisfying $\Phi$ and $\forall 0 \leq i \leq k, \forall \mathcal{C} \in$ clauses$(F_i)$:
      if $\mathcal{F}_i \wedge \Delta \Rightarrow \mathcal{C}$ then $\mathcal{C} \in$ clauses$(F_{i+1})$

function propagate($F, k, M$):
1   for $i \leftarrow 1$ to $k$ do
2       forall $\mathcal{C} \in$ clauses$(F_i)$ do          ▷ $\mathcal{C}$ describes a set of states $C = \overline{S}$
3           if SAT$(\mathcal{F}_i \wedge \Delta \wedge \neg\mathcal{C}') = 0$ then          ▷ $\mathcal{F}_i \wedge \Delta \Rightarrow \mathcal{C}'$
4               $\mathcal{F}_{i+1} \leftarrow \mathcal{F}_{i+1} \wedge \mathcal{C}$          ▷ $F_{i+1} \leftarrow F_{i+1} \setminus S$
5   return $F$ ▷ Modified through its CNF representation $\mathcal{F}$.
```

removed from the over-approximations. This may cause previously reachable states to become unreachable. Before continuing to the next iteration, the `propagate` function (Algorithm 2) uses this to push clauses further into the candidates without needing to perform backwards searches through `block`. For each candidate, it iterates over all clauses that were learned by PDR in line 2. Each clause $\mathcal{c}$ represents the negation of a state s ($s = \neg\mathcal{c}$). If s is not reachable from the current F_i, then it may be removed from F_{i+1} as well. The query in line 3 checks this using a simple SAT-query. During propagation can also verify if there are two equivalent, neighbouring candidates for the termination check on line 8 in Algorithm 1.

3.4 Minor iteration

The `block` from Algorithm 3 executes the minor iteration. It considers each obligation in O and performs backwards reachability on it to see if it can reach the initial states I by considering each its predecessors. These predecessors are also enqueued as new proof obligations. If an obligation reaches I, then it reveals a counterexample trace. If not, then F is refined with this new information and the obligation is removed from O. If all obligations have been handled, `block` reports that it could find no trace.

Each obligation $(s, n) \in O$ contains an integer n, which indicates that it has already been proven to be inductive to F_{n-1} and must now be proven inductive to F_n, lest the property P fails. O is treated as a priority queue, ensuring an obligation (s, n) with a minimal n is considered in each iteration. This makes it impossible for any new predecessor to be already represented in O, since any predecessor would be enqueued with $n - 1$., ensuring `block` terminates even if the system-under-verification M contains cycles.

Line 3 queries a predecessor p of s that is currently in F_n. If $F_n = F_0 = I$, then there exists a trace from I to $\overline{P}$ and `block` terminates with a trace. The trace

Algorithm 3: The minor iteration of PDR

```
In   : A tuple $(M, P, k, F, O)$ satisfying Definition 3.6, thus $M$ is a
       symbolic transition system $(X, I, \Delta)$. $F$ and $O$ are passed by
       reference.
Out  : $trace$, a counter-example to $[\![\Box\, P]\!]^{TS}$ if it exists.
Post : $F$ or $O$ are potentially changed (by reference) and $(M, P, k, F, O)$
       satisfies Definition 3.6.

function block($M, P, k, F, O$):
1     while $O \neq \varnothing$ do
2         $(s, n) \leftarrow O$.front()  ▷ $\forall (q, i) \in O : n \leq i$
3         if $\exists p \in F_n : \ \Delta(p, s)$ then  ▷ Implemented as SAT-query (Appendix B)
              ▷ $p$ is a predecessor to $s$
4             if $n = 0$ then
5                 return $trace(p, s, \ldots \overline{P})$  ▷ Counterexample! Trace reaches $I$
6             $O \leftarrow O \cup (p, n - 1)$
7         else
8             $m \leftarrow$ remove-cube($s, n, F, k, TS$)  ▷ $n \leq m \leq k, \ s \notin F_{m+1}$
9             $O \leftarrow O \setminus (s, n)$
10            if $m + 1 \leq k$ then
11                $O \leftarrow O \cup (s, m + 1)$
12    return $\varnothing$
```

can be constructed with some bookkeeping that is omitted from the discussion. For any other n, p becomes a new proof obligation $(p, n-1)$ on line 6.[5] After all, if p is reachable in $n-1$ steps, then s is reachable in n steps, which reveals a trace to $\overline{P}$. Since predecessors are enqueued at a lower level, (s, n) can only be finished once all of its predecessors are.

If s has no predecessor in F_n, the else-block on line 7 finishes the proof obligation. Without any transition from F_n to s, s may be removed from F_{n+1} without affecting Φ_3. The call to `remove-cube` on line 8 ensures that s is at least removed from $F_1, \ldots F_{n+1}$, and perhaps more states are removed from more candidates through generalisation (Section 3.5). This process may reveal $\overline{s}$ to inductive to a candidate of a higher level than n. The `remove-cube` function returns this level m, allowing s to be removed from $F_1, \ldots F_{m+1}$. The obligation is now removed from O on line line 9. Even then, s may still be part of a larger counterexample-trace. If s is reachable in more than $m+2$ steps, it would still lead to a non-P-state. This is captured in Lemma 3.2.

[5] It is already known that $\overline{p}$ is inductive relative to F_{n-2}. If it were not, then s would have a predecessor in F_{n-1}, contradicting Ω_2. This satisfies Ω_2 for p. Ω_1 and Ω_3 simply hold because $n - 1 < n$ and $\Delta(p, s)$.

Lemma 3.2 (Future obligations). If any proof-obligation (s, i) is fulfilled, an obligation $(s, i+1)$ will eventually be found and enqueued into O, or else P does not hold.

In order to push forward the reachability information of s as far as possible, it is enqueued as a new obligation $(s, m+1)$ on line 11, effectively replacing the previous. An obligation will only pushed as long as there already exists a candidate for $m+1$.[6] Otherwise it will need to wait to be discovered again after PDR has extended the the candidate sequence. This mechanism allows PDR to search deeply even at lower k and find traces of greater length than k. This improves performance in cases where there is a counterexample [39]. A side effect is that a found trace is not necessarily of minimal length.

3.5 Removing states

Algorithm 4 (`remove-cube`) takes a state s and a candidate F_i to which $\overline{s}$ is relatively inductive. However, F_i is only the lowest candidate to which, up until recently, $\overline{s}$ was not known to be inductive to. It may be inductive an even higher candidate. The loop on line 2 finds the highest candidate to which $\overline{s}$ is inductive. Once s is discovered in the post-image of F_i without s, $\overline{s}$ is known to be inductive to at most the previous candidate F_{i-1}.

PDR extrapolates more reachability information from s through the `generalise` function, which takes the state s and the candidate F_{i-1} and returns a set of states C, such that $\overline{C}$ is also inductive relative to F_{i-1}. The states in C may not necessarily lead to a violation of P like s, but they are equally unreachable and thus eligible to be removed from $F_1, \ldots F_i$. PDR now potentially remove multiple states, instead of just s, without needing to perform additional backwards searches. Lines 5 and 6 remove these states. In practice, this is done by appending its clause to each candidate: $\mathcal{F}_i \leftarrow \mathcal{F}_i \wedge \neg\mathcal{C}$.[7]

Algorithm 5 lists the `generalise` function. It uses the encoding of the state s in propositional logic as a cube $\mathcal{s}$. `generalise` constructs a subcube $\mathcal{C}$ of $\mathcal{s}$: a cube whose literals are a subset of those in $\mathcal{s}$. As per Theorem 2.1, this cube represents a set C containing s.

$\mathcal{C}$ is initialised to $\mathcal{s}$. `generalise` tries to drop each literal l from $\mathcal{C}$. The resulting smaller cube $\mathcal{C} \setminus l$ is verified by the `down` algorithm [18, 52]. If dropping l does not result in a cube that is inductive to F, then `down` returns *failure*, l is kept and

[6]As a new obligation cannot exceed k when increased and does not go under 1 when decreased in line 6, maintaining Ω_1. Ω_2 is satisfied since this has just been proven and Ω_3 is satisfied since the state s has not changed. F is not directly modified in `block`, so the function does not affect Φ. Section 3.5 shows that Φ is not modified by `remove-cube`.

[7]This maintains Φ_2 and Φ_3: either only the left-hand side of $\subseteq$ has a state removed or both have the same state removed. Φ_0 and Φ_1 are maintained implicitly as states are only removed, and not added, and never from F_0.

```
Algorithm 4: Remove a state from all eligible frames.
In   : $s \in \mathbb{B}^X$, where $\overline{s}$ is inductive relative to $F_n$ $(s \notin (F_n \setminus s).\Delta)$;
       $n \in \mathbb{N}$, where $0 < n < k$; $F = \{ F_0, F_1, \dots \}$ satisfying $\Phi$ for $k$; a
       symbolic transition system $M = (X, I, \Delta)$; $F$ is passed by
       reference.
Out  : The highest candidate $F_{i-1}$ for which $s \notin (F_{i-1} \setminus s).\Delta$.
Post: $F$ satisfies $\Phi$ for $k$, $s \notin F_n$, $s \notin F_i$

function remove-cube($s, n, F, k, M$):
1   $i \leftarrow n + 1$
2   while $s \notin (F_i \setminus s).\Delta$ do ▷ Implemented as a SAT-query (Appendix B)
3       $i \leftarrow i + 1$
    ▷ $F_i$ is the highest inductive candidate
4   $C \leftarrow$ generalise($s, F_{i-1}, k, M$) ▷ $s \in C \subseteq \mathbb{B}^X$, $\forall c \in C : c \notin F_{i-1}.\Delta$
5   for $j \leftarrow 1$ to $i$ do
6       $F_j \leftarrow F_j \setminus C$
7   return $i - 1$
```

`generalise` continues onto the next literal. Otherwise, it returns the smaller cube which replaces $\mathcal{C}$ and the next literal is considered.

In fact, `down` may produce a cube that is even smaller than $\mathcal{C} \setminus l$. The algorithm stores the reduced cube in $\mathcal{S}$ and enters a loop that potentially reduces $\mathcal{S}$, terminating once it is proven that $\mathcal{S}$ is inductive or that it cannot be. As $\mathcal{S}$ is reduced, it describes a larger set of states for the system. Line 9 detects if $\mathcal{S}$ describes one of the initial states, in which case removing it from $\mathcal{F}$ can never be inductive and `down` reports *failure*. Line 11 detects if $\neg\mathcal{S}$ has become inductive relative to $\mathcal{F}$, returning $\mathcal{S}$ if so.

If $\neg\mathcal{S}$ is not inductive, then the SAT-solver reveals a witness $\mathcal{Q}$ that is not yet in $\mathcal{S}$. Adding it to $\mathcal{S}$ may be enough to make $\neg\mathcal{S}$ inductive. This is done by taking all the literals that $\mathcal{S}$ and $\mathcal{Q}$ have in common, which creates a smaller cube that represents a larger set of states that includes the states in both S and Q. This prevents $\mathcal{Q}$ from being a future witness. The next iteration will reveal weather this new $\mathcal{S}$ causes `down` to terminate.

Algorithm 5: Find a minimal inductive subclause using the `down` [18] algorithm

```
In  : $s \in \mathbb{B}^X$, where $s \notin (F \setminus s).\Delta$, with a cube-representation $\mathcal{s}(X)$;
      $F \subseteq \mathbb{B}^X$, with a CNF-representation $\mathcal{F}(X)$; a symbolic transition
      system $M = (X, I, \Delta)$.
Out : A set $C \ni s$, represented by the cube $\mathcal{C}$, and $(F_i \setminus C).\Delta \subseteq \overline{C}$.

function generalise($s, F, M$):
1   $\mathcal{C} \leftarrow s$                              ▷ $s = s_1 \wedge s_2 \dots s_n$ (Definition 2.5)
2   for $l \in \mathcal{C}$ do
3       $\mathcal{S} \leftarrow$ down($\mathcal{C} \setminus l, F, M$)
4       if $\mathcal{S} \neq$ failure then
5           $\mathcal{C} \leftarrow \mathcal{S}$
6   return $C$ ▷ Set described by the cube $\mathcal{C}$ (Definition 2.5)
```

```
In  : A cube $\mathcal{C}(X)$; a CNF-formula $\mathcal{F}_i(X)$; a symbolic transition
      system $M = (X, I, \Delta)$.
Out : A cube $\mathcal{C}(X)$, where $\mathcal{C} \Rightarrow S$, $\mathcal{I} \Rightarrow \neg\mathcal{S}$ and $\mathcal{F}_i \wedge \neg S \wedge \mathcal{T} \Rightarrow \neg S'$;
      or failure if none exists.

function down($\mathcal{C}, F_i, M$):
7   $\mathcal{S} \leftarrow \mathcal{C}$
8   loop
9       if $\mathsf{SAT}(\mathcal{I} \wedge \mathcal{S}) = 1$ then ▷ $\mathcal{I} \not\Rightarrow \neg\mathcal{S}$
10          return failure
11      if $\mathsf{SAT}(\mathcal{F}_i \wedge \neg\mathcal{S} \wedge \mathcal{T} \wedge \mathcal{S}') = 0$ then ▷ $\mathcal{F}_i \wedge \neg\mathcal{S} \wedge \mathcal{T} \Rightarrow \neg\mathcal{S}'$
12          return $\mathcal{S}$
13      else
14          $Q \leftarrow \mathsf{SAT}-witness$                ▷ $Q \subseteq F_i \setminus S$ and $\Delta(Q, S) \neq \varnothing$
15          $\mathcal{S} \leftarrow \bigwedge_{l \in \mathcal{S} \cap Q} l$   ▷ cube containing common literals between $\mathcal{S}$ and $Q$
```

4 Incremental PDR for Constraining and Relaxing Systems

We now give the *Incremental PDR* algorithm (IPDR), which uses multiple executions of PDR and the internal state from Section 3 to solve problems that can be expressed as multiple related instances. Each instance is represented as a symbolic transition system that is monotonically related to the previous. IPDR takes clauses learned during a PDR run for one of these instances, and reuses them in the next. The clauses learned over the course of the PDR algorithm describe the reachability of states within an instance of the system. The intuition behind IPDR is that if subsequent instances are sufficiently similar, this information would useful when verifying these instances.

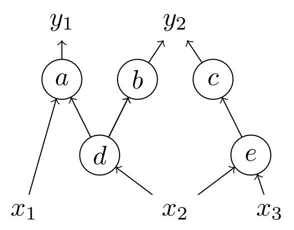
Monotonically decreasing or increasing systems will be denoted as being *constraining* or *relaxing*. Consider for instance the transition systems *(a)* and *(b)* from Figure C.1. One may obtain *(b)* by taking the transition relation describing *(a)* and removing two transitions from it, i.e., relaxing it. Intuitively, *(b)* behaves much the same as *(a)* and clauses describing reachability gathered by an execution of PDR for one system may be useful when executing PDR for the other. Definition 4.1 defines when two systems are constraining and relaxing formally.

Definition 4.1 (Constrained and Relaxing Transition System). A transition system $M_1 = (X, I_1, \Delta_1)$ is a *constrained* version of $M_2 = (X, I_2, \Delta_2)$, iff $I_1 \subseteq I_2$ and $\Delta_2 \subseteq \Delta_2$. This is denoted $M_1 \sqsubseteq M_2$. Vice versa, M_2 *relaxes* M_1, denoted $M_2 \sqsupseteq M_1$.

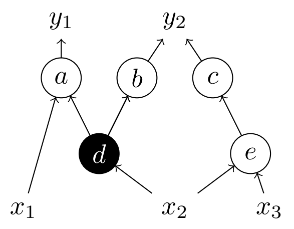
$\sqsubseteq$ forms a partial order, arising from the fact that $\subseteq$ forms a partial order. $M_1 = M_2$ iff $M_1 \sqsubseteq M_2$ and $M_1 \sqsupseteq M_2$. $M_1 \sqsubset M_2$ denotes that $M_1 \sqsubseteq M_2$ and $M_1 \neq M_2$. $\sqsupset$ Z denotes the same for $\sqsupseteq$.

For example, consider the of the Reversible Pebbling Game displayed in Figure 4.1. The game is described in more detail in Section 5.3, but for now simply note that each node in the graph can be marked or unmarked. There are two states reachable from the initial states, denoted *(b)* and *(c)*. One may divide the Pebbling Game into instances that bound the maximum number of nodes that can be marked at a time. Bounding the number of pebbles to 2 means both states are reachable, bounding it to 1 means only *(b)* is reachable. Solving the Pebbling Game under a tighter constrain is harder as transitions to board states that violate that bound are eliminated.

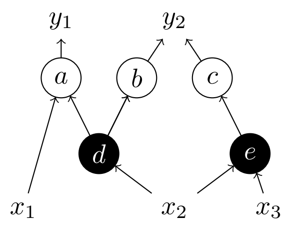
IPDR takes a sequence of symbolic transition systems as input, where subsequent entries are all *constraining* or all *relaxing*. This notion is more exactly formulated in Definition 4.2.



(a) The empty game state I.



(b) A reachable game state from I with one marked node.



(c) A reachable game state from I with two marked nodes.

Figure 4.1: A Directed Acyclic Graph, on which the Reversible Pebbling Game [4, 63] is played. White nodes are unmarked, black nodes are marked.

Definition 4.2 (Constraining and Relaxing Sequence). A sequence of symbolic transition systems $M_1, M_2, \dots M_z$ is called *constraining* if for each M_i, the next $M_{i+1} \sqsubset M_i$: if $M_1 \sqsupset M_2 \sqsupset \dots M_z$. It is called *relaxing* if $M_1 \sqsubset M_2 \sqsubset \dots M_z$.

The IPDR algorithm knows two variations that must be handled separately. Section 4.1 gives the *Constraining IPDR* algorithm (C-IPDR), which verifies a sequence of constraining systems. C-IPDR verifies each system incrementally and produces either a counterexample trace for M_z or an inductive invariant for the M_i with the lowest i if it exists. Section 4.2 gives the *Relaxing IPDR* algorithm (R-IPDR), which takes a sequence of relaxing systems. R-IPDR produces an inductive invariant for M_z

Each version has a main function (`ipdr-constrain` and `ipdr-relax`) which iterates over all instances and a function (`constrain` and `relax`) that takes the output PDR state and returns a new state that can be used in a constrained or relaxed instance.

4.1 Constraining IPDR

The Constraining IPDR algorithm (`ipdr-constrain` in Algorithm 6) takes multiple symbolic transitions systems $M_0, \dots M_z$ as input, all defined over the same variables as input, and a property P. Each system is constraining relative to the previous: $\forall 1 \leq i \leq z \colon M_{i+1} \sqsubset M_i$ (or equivalently $M_i \sqsupset M_{i+1}$). In practice,

```
Algorithm 6: Constraining IPDR
In  : $(M, P, k, F, O)$ satisfying Definition 3.6, where $M = (X, I, \Delta)$
      and $F = \{ F_0, F_1, \dots \}$. $M^{\downarrow} \sqsubset M$ where $M^{\downarrow} = (X, I^{\downarrow}, \Delta^{\downarrow})$.
Out : A PDR state satisfying Definition 3.6

function constrain($(M, P, k, F, O), M^{\downarrow}$):
1    $F^{\downarrow} \leftarrow \{ I^{\downarrow}, F_1, F_2, \dots \}$
2    $F^{\downarrow} \leftarrow$ propagate($F^{\downarrow}, k, M$)
3    return $(M^{\downarrow}, P, k, F^{\downarrow}, \varnothing)$

In  : $M_0 \sqsupset M_1 \sqsupset \dots M_z$ with each $M_i = (X, I_i, \Delta_i)$ and $P \subseteq \mathbb{B}^X$.
Out : A result containing the first inductive invariant encountered, or a
      counterexample trace for $M_z$.

function ipdr-constrain($M_0, M_1, \dots M_z, P$):
4    if $I \not\subseteq P$ then  return trace $= i \in I \cap \overline{P}$
5    if $I.\Delta \not\subseteq P$ then  return trace $= (i, p) \in I.\Delta.\overline{P}$
6    $(M, P, k, F, O)$, result $\leftarrow$ pdr($M, P, 1, \{ I, P, P \}, \varnothing$)
7    for $M_i \in \{ M_1, \dots M_z \}$ do
8        if result is an inductive invariant then
9            return result
10       $(M, P, k, F, O) \leftarrow$ constrain($(M, P, k, F, O),\ M_i$) ▷ satisfies Definition 3.6
11       $(M, P, k, F, O)$, result $\leftarrow$ pdr($M, P, k, F, O$) ▷ satisfies Definition 3.6
12   return result
```

each M_i need not be known beforehand and may be constructed based on the output of M_{i-1}.

PDR is first run normally with a state initialised as per Section 3.2 on line 6: with $k = 1$, $F_0 = I$, $F_1 = P$ and $O = \varnothing$. Calling PDR with this state returns an inductive invariant or a counterexample trace in *result*, and the PDR state after this run.

For every other M_i, IPDR will use the `constrain` function to take the latest PDR state *PDR* and convert it to one suitable for PDR for M_i that still satisfies Definition 3.6. `constrain` will also try to retain as many of the clauses in F as possible, the process and its validity are further detailed in Section 4.1.

With a valid PDR state, line 11 can perform PDR as described in Section 3, producing a new result and PDR state on with which M_{i+1} may be run. This continues until `ipdr-constrain` terminates.

`ipdr-constrain` may actually terminate early the moment it finds an inductive invariant. After all, this means that no counterexample trace exists. And if no trace exists, then it is impossible for PDR to find one in a constrained system. If there is a trace, then removing behaviour may remove transitions from that trace and invalidate it.

The example in Section C.1 shows this version of the algorithm in practice for a small system.

Copying Clauses. The `constrain` function (also found in Algorithm 6) takes a PDR state (M, P, k, F, O) as input and transition system $M^{\downarrow}$ that constrains M. Its goal is to return a new *PDR* wrapping $M^{\downarrow}$.

`constrain` first copies the sequence of candidates F to a new sequence $F^{\downarrow}$, which must uphold Φ from Definition 3.3. The only alteration made, is changing the first candidate to $I^{\downarrow}$. Intuitively, all states that were deemed unreachable under M will remain unreachable when behaviour is removed it. Formally, $F^{\downarrow} = \{ I^{\downarrow}, F_1, F_2, \dots \}$ can be seen to uphold Φ under $M^{\downarrow}$ as follows:

- Φ_0 is trivially holds by assigning $I^{\downarrow}$.
- Φ_1 holds as $F_0^{\downarrow} = I^{\downarrow} \subseteq I \subseteq$ and all other $F_1, \dots F_k$ are unchanged.
- Similarly, Φ_2 holds as well. Only F_0 is changed and $F_0^{\downarrow} = I^{\downarrow} \subseteq I \subseteq F_i$ by Φ_2 under M.
- For Φ_3, again, only the left hand side of the inequality is altered: $\forall 0 \leq i \leq k$: $F_i.\Delta^{\downarrow} \subseteq F_i.\Delta^{\downarrow} \subseteq F_{i+1}$.

Some of the states now blocked in $F^{\downarrow}$ that lead to a $\overline{P}$-state in F may not do so any longer, but this is of no consequence. Under the same logic for generalising states in Section 3.5, F may block unreachable states even if they do not lead to a $\overline{P}$-state.

The frontier of the algorithm k can be left as it was, as well. Which arises formally from the fact that the new $F^{\downarrow}$ upholds Φ and intuitively from the notion that with behaviour removed from M, PDR would not discover any new 1-step violation of P in any lower k.

If `ipdr-constrain` calls `constrain` to construct a new state, then PDR just terminated with a counterexample trace on line 5 in Algorithm 3. This means that O is not empty. Ω_1 and Ω_2 (Definition 3.5) hold if O were to be simply copied, as k remains unchanged and any inductive $\overline{s}$ remains inductive under a constrained transition relation. Ω_3 would not hold and its violation may cause traces to be found that do not lead to $\overline{P}$ anymore. The reachability of all obligations in O could feasibly be validated, but this would be done using the same SAT-queries as discovering the obligation anew and performing `block`. Thus, for the sake of simplicity, O is not copied over and $O^{\downarrow}$ is instead set to the empty set $\varnothing$.

A new PDR state $(M^{\downarrow}, P, k, F^{\downarrow}, \varnothing)$ now satisfies Definition 3.6. Before returning this state, it may pay dividends to execute the `propagate` from Algorithm 2 and

Section 3.3 on $F^{\downarrow}$. With some transitions removed from Δ, states that were previously found unreachable up to some number of steps i may now found to be unreachable up to some greater number. Note that the propagation may create two equal, adjacent candidates $F_i = F_{i+1}$ and reveal an inductive invariant. This may be detected here, after propagation and prevent the need to start PDR.

Adjusting Bradley's PDR. This implementation of IPDR places a restriction on the PDR algorithm used by `ipdr-main`, as explicitly given by Bradley's original work [17]. Bradley eagerly blocks obligations once they are discovered under the assumption that PDR will eventually remove them or terminate, at which point any inconsistencies this may cause are irrelevant. Because IPDR requires PDR to be in a valid state after terminating, the algorithm follows other implementations [1, 39] that followed Bradley's original work, which only block states in F once the obligation is fulfilled without noticeable decreases in performance.

4.2 Relaxing IPDR

The Relaxing IPDR algorithm (`ipdr-relax` in Algorithm 7) similarly takes a series of transition systems and a property. Only here, each system is relaxing relative to the previous: $\forall 0 \leq i \leq z \colon M_{i+1} \sqsupset M_i$ (or equivalently $M_i \sqsubset M_{i+1}$).

The initialisation and main structure of R-IPDR is similar to C-IPDR. Besides naturally preparing a new PDR state with the `relax` function instead of `constrain`, `ipdr-relax` has two major differences with `ipdr-constrain` when iterating over all $M_1, \dots M_z$.

Firstly, `ipdr-relax` now terminates early on line 15 if one of the M_i returns a trace. As long as PDR finds an inductive invariant, then adding behaviour may cause a trace to exist where it previously did not. But once a trace is found, this same trace is valid under any relaxed system.

Additional, R-IPDR must repeat the checks from initialisation on lines 10 and 11. After all, with I and Δ potentially larger than before: the initial states may have increased to include, or lead to, a $\overline{P}$-state. If these do not reveal a 0- or 1-step trace, `relax` converts the most recent PDR state into a valid one that wraps the new, relaxed M_i.

A relaxed run of PDR (with $M^{\uparrow}$) cannot implicitly continue with the last k, as this breaks Φ_3 under the new $\Delta^{\uparrow} \supseteq \Delta$. Since it is only know that $F_i.\Delta \subseteq F_i.\Delta^{\uparrow}$, it cannot be assumed that $F_i.\Delta^{\uparrow} \subseteq F_{i+1}$. Intuitively speaking: wherever the previous PDR run concluded that a state was unreachable, a new transition may reveal it to be reachable. Thus, `relax` cannot blindly copy F to $F^{\uparrow}$ and will explicitly validate each cube.

Lines 10 and 11 from `ipdr-relax` ensured that $\{I^{\uparrow}, P, P \dots\}$ satisfies Φ for the new $M^{\uparrow}$. This allows `relax` to begin with initialising $F^{\uparrow}$ to that sequence on line 1. Now that $F^{\uparrow}$ is in a valid state, `relax` follows a similar principle to the `propagate` function from Section 3.3 to push the learned clauses in F as far as possible in F. By Lemma 3.1, $\texttt{clauses}(F_i)$ contains all clauses that were learned by PDR, which are stored in *All* on line 2. The loop on line 3 first begins by checking for each clause in *All* that it does not block any of the new initial states and is inductive relative to I on line 5. If so, then the states described by $\neg\mathcal{C}$ are unreachable in 1 step and $\mathcal{C}$ is added to $\mathcal{F}_1$ on line 6. If a clause cannot be pushed any further, it is removed from *All* on line 8. The remainder of the loop then pushes each remaining clause as far as possible through the levels $1, 2, \ldots k$, similar to `propagate`.

The `relax` function is only called when PDR terminated with an inductive invariant, meaning that in the resulting PDR state $O = \varnothing$. This implicitly validates Ω and the new PDR state can simply use $\varnothing$.

The tuple $(M^{\uparrow}, P, 1, F^{\uparrow}, \varnothing)$ now satisfies Definition 3.6, where $F^{\uparrow}$ contains has blocked as many cubes from the previous PDR run as possible, although not all as could be done in C-IPDR.

The example in Section C.2 shows an example for this relaxing version.

Algorithm 7: Relaxing IPDR

```
In  : $(M, P, k, F, O)$ satisfying Definition 3.6, where $M = (X, I, \Delta)$
      and $F = \{ F_0, F_1, \dots \}$. $M^\uparrow \sqsupset M$ where $M^\uparrow = (X, I^\uparrow, \Delta^\uparrow)$.
Out : A PDR state satisfying Definition 3.6

function relax($(M, P, k, F, O), M^\uparrow$):
    $F^\uparrow \leftarrow \{ I^\uparrow, P, P, \dots \}$
    $All \leftarrow$ clauses$(F_1)$ ▷ by Lemma 3.1, this contains all clauses found by PDR
    for $i \leftarrow 0$ to $k$ do
        for $\mathcal{C} \in All$ do
            if SAT$(\mathcal{I}^\uparrow \wedge \neg\mathcal{C}) = 0$ and SAT$(\mathcal{F}_i^\uparrow \wedge \Delta^\uparrow \wedge \neg\mathcal{C}') = 0$ then
                ▷ $\mathcal{C}$ does not block $I$-states and $\neg\mathcal{C}$ is unreachable in $i + 1$ steps
                $\mathcal{F}_{i+1}^\uparrow \leftarrow \mathcal{F}_i^\uparrow \wedge \mathcal{C}$ ▷ $F_i \leftarrow F_i \setminus \neg\mathcal{C}$
            else
                $All \leftarrow All \setminus \mathcal{C}$
    return $(M^\uparrow, P, 1, F^\uparrow, \varnothing)$
```

```
In  : $M_0 \sqsubset M_1 \sqsubset \dots M_z$ with each $M_i = (X, I_i, \Delta_i)$ and $P \subseteq \mathbb{B}^X$.
Out : A result containing the first counterexample trace encountered,
      or an inductive invariant for $M_z$.

function ipdr-relax($M_1, M_2, \dots M_z, P$):
    if $I \not\subseteq P$ then  return $i \in I \cap \overline{P}$
    if $I.\Delta \not\subseteq P$ then  return $(i, p) \in I.\Delta.\overline{P}$
    $(M, P, k, F, O)$, result $\leftarrow$ pdr$(M, P, 1, \{ I, P \}, \varnothing)$
    for $M_i \in \{ M_1, \dots M_z \}$ do
        if result is a trace then
            return result
        if $\exists i \in I_i \cap \overline{P}$ then  return trace$(i)$
        if $\exists (i, p) \in \Delta_i(I_i, \overline{P})$ then  return trace$(i, p)$
        $(M, P, k, F, O) \leftarrow$ relax$((M, P, k, F, O), M_i)$ ▷ satisfies
          Definition 3.6
        $(M, P, k, F, O)$, result $\leftarrow$ pdr$(M, P, k, F, O)$ ▷ satisfies Definition 3.6
```

5 IPDR implementation

In this section, we discuss the implementation details of IPDR [13] in C++[8]. The algorithms in Section 4 have been implemented using our own PDR model checker, which is designed to expose the learned clauses in a desired manner. This model checker is largely implemented as described by described in Section 3, with additional optimisations from Bradley and related works. These optimisations are listed in Section 5.1. Section 5.2 shows how Peterson's Algorithm is encoded into propositional logic, and Section 5.3 does the same for the Reversible Pebbling Game.

5.1 Optimisations

There are many optimisations implemented in Bradley's original work [17] and in related work [39, 52] that is not critical the understanding of PDR in Section 3. This section lists some further implementation details that were used when implementing PDR and IPDR.

5.1.1 Delta-encoding

The candidate sequence F can be stored as a set of clauses B. These clauses are stored using a similar delta encoding to that used by Niklas Een, Alan Mishchenko and Robert Brayton [39].

Here every B_i stores the difference between the clauses present in F_i and F_{i+1}. To obtain all $\texttt{clauses}(F_i)$ for PDR, on takes all clauses belonging to subsequent B_j (Definition 5.1).

Definition 5.1 (Delta-blocked states). Given (M, P, k, F, O) satisfying Definition 3.6, where $F = \{ F_0, F_1, F_2, \dots \}$ satisfies Definition 3.3.

$$\forall 1 \leq i \leq k \colon B_i \triangleq \texttt{clauses}(F_i) \setminus \texttt{clauses}(F_{i+1})$$

Conversely, $\mathcal{F}_i$ is constructed by $\mathcal{F}_i = \mathcal{P} \land \bigwedge_i^k \bigwedge_{c \in B_i} c$.

Intuitively, a clause is only present in a set B_i if F_i is the latest set in which it is present. This is illustrated in Example 5.1.

The delta-encoding also provides a cheap method to compare two adjacent sets of blocked cubes: if some B_i is empty, then there is no difference between that and the next B_{i+1}. The PDR implementation uses this to detect inductive invariants after propagation.

[8] `https://github.com/Majeux/ipdr`

Example 5.1 (Delta-encoding). Consider a candidate sequence $F = \{F_0, F_1, F_2, F_3, P, P \dots\}$. The corresponding B-sets on the left store their clauses naively. The sets on the right use the delta-encoding as per Definition 5.1.

Normal B			**Delta − B**		
B_1	B_2	B_3	B_1	B_2	B_3
a	a			a	
b	b	b			b
c			c		

The candidates are represented as a CNF-formula $\varphi = \mathcal{P} \wedge \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \dots$ within a SAT-solver. When a cube $\mathcal{S}$ is blocked, it is negated into a clause $\mathcal{C}$ that is appended to φ.

Each clause is extended an especially reserved activation literal act_i, and is thus of the form $\mathcal{C} = \neg\mathcal{S} \vee \neg act_i$. The activation literal is used only for this purpose, and is thus not asserted to be true or false within φ. This means any SAT-query can implicitly satisfy the clause by asserting $\neg act_i$, effectively "removing" it from φ. The query $\mathsf{SAT}(\varphi \wedge act_i)$ "adds" the clause, as the solver is now forced to consider the remaining literals of the clause.

An activation literal act_i is used to denote when a clause belongs to the set B_i. When a cube removed from $F_1, F_2, \dots F_i$, the activation literal act_i is added to its clause.

While φ is loaded in an incremental SAT-solver, it is able to solver φ under a set of assumptions: literals that are temporarily asserted for that query only. A query over the candidate F_i can then be performed by adding the literals $act_i, act_{i+1}, \dots act_k$ to the assumptions, which effectively solves $\mathsf{SAT}(\varphi \wedge act_i \wedge act_{i+1} \wedge \dots act_k)$.

5.1.2 Subsumption

The number of clauses in a delta-encoded sequence is further reduced by eliminating *subsumed* clauses [17, 39], see Definition 5.2.

Definition 5.2 (Subsumed cubes or clauses). It is said a cube (or clause) $\mathcal{A}(X)$ *subsumes* a cube (or clause) $\mathcal{B}(X)$ if $lits(\mathcal{A}) \subseteq lits(\mathcal{B})$, where *lits* obtains all literals present in a cube (or clause).

If a clause $\mathcal{B}$ added to some candidate F_i is subsumed by another clause $\mathcal{A}$ in F_i, then $\mathcal{B}$ may be removed without affecting F_i. From a logical perspective, $\mathcal{A} \Rightarrow \mathcal{B}$. From a set-theoretic perspective, $B \subseteq A$ and thus blocks the same states as A. $\mathcal{B}$ is thus considered redundant, and slow down the SAT-solver [39].

Subsumption occurs often [39], so PDR removes subsumed clauses from candidates and periodically reinstates the formula in the SAT-solver to purge them.

Algorithm 8: Block a cube and remove all subsumed cubes from the delta-encoding

```
In  : $B = \{ B_1, B_2, \dots B_k \}$ satisfying Definition 5.1; a cube $\mathcal{S}(X)$;
      $l \in \mathbb{N}$ where $1 \leq l \leq k$.
Out : $B = \{ B_1, B_2, \dots B_k \}$ satisfying Definition 5.1 and
      $\forall 0 \leq i \leq l \colon \mathcal{S} \not\Rightarrow \bigwedge \texttt{clauses}(F_i)$

function remove-subsumed($B, \mathcal{S}, l$):
1   for $i \leftarrow 1$ to $l$ do
2       $B_i \leftarrow \{ \mathcal{C} \in B_i \mid \mathit{lits}(\neg\mathcal{S}) \not\subseteq \mathit{lits}(\mathcal{C}) \}$ ▷ remove subsumed clauses
3   $B_l \leftarrow B_l \cup \neg\mathcal{S}$ ▷ $\forall 1 \leq i \leq l \colon F_i \leftarrow F_i \setminus \mathcal{S}$
4   return $B$
```

Because all information is recorded in clauses, a syntactic check may quickly detect redundant clauses. Algorithm 8 gives a replacement function for the loop on line 5 in Algorithm 4 that removes set of states C from $F_1, \dots F_i$: `remove-subsumed`. `remove-subsumed` modifies the sequence of candidates through the delta-encoded representation B.

5.1.3 Generalisation

Bradley [17] limits the amount of times that Algorithm 5 tries to drop a literal on line 3. When `down` fails to drop a literal three times, the subcube generated so far is returned by `generalise`. Bradley notes that while this subcube is not minimal, it is often sufficient and greatly reduces the time taken by generalisation.

Since his initial IC3 publication [17], Bradley has published an extension [52] to the `down` algorithm listed in Section 3.5. The original method tries to drop the literals of a cube and checks if the resulting cube is inductive. If not, this reveals a witness that prevents the generalisation of the cube, a counterexample-to-generalisation or a *ctg*. The original `down` coarsely removes the states of this witness from the cube under consideration to avoid this conflict. This may help PDR find a stronger clause, which may cause a proof to be discovered sooner [52]. The new method `ctg-down` (Algorithm 10) instead removes these *ctgs* through a similar mechanism to the one that removes counterexamples-to-induction. In fact, `ctg-down` recursively calls itself through `ctg-generalise` up to a certain depth.

This method brings along some additional parameters that limit how much focus is put upon blocking a *ctg*. The ctg_{max} setting limits how many *ctg*s are considered for each cube. The $depth_{max}$ setting limits how many times `ctg-generalise` is recursively called. With $depth_{max} = 1$, *ctg*s that interfere with the generalisation of *cti*s are considered. With higher $depth_{max}$, *ctg*s to those that interfere with other *ctg*s are considered.

Algorithm 9: A modified `generalise` algorithm that addresses counterexamples-to-generalisation [52]

```
In  : A cube $\mathcal{S}(X)$, representing a set of states $S \subseteq \mathbb{B}^X$, where
      $(F \setminus S).\Delta \subseteq \overline{S}$; $i \in \mathbb{N}$, where $i \leq k$; $PDR = (F, k, M)$, where
      $F = \{ F_0, F_1, F_2 \dots \}$ satisfies Definition 3.3 and $M = (X, I, \Delta)$ is
      a symbolic transition system.
Out : A set $C$, represented by the cube $\mathcal{C}$, where $S \subseteq C$ and
      $(F_i \setminus C).\Delta \subseteq \overline{C}$.

function ctg-generalise($\mathcal{S}, i, depth, PDR$):
    $\mathcal{C} \leftarrow \mathcal{S}$
    for $l \in \mathcal{C}$ do
        $\mathcal{X} \leftarrow$ ctg-down($\mathcal{C} \setminus l, i, depth, PDR$)
        if $\mathcal{X} \neq failure$ then
            $\mathcal{C} \leftarrow \mathcal{X}$
    return $C$ ▷ Set described by the cube $\mathcal{C}$ (Definition 2.5)
```

The call to `generalise`(s, F_{i-1}, k, M) on line 4 in Algorithm 4 is replaced by the call `ctg-generalise`$(\mathcal{S}, i-1, 1, (F, k, M))$, where $\mathcal{S}$ denotes the cube describing $\{ s \}$. Algorithm 10 begins much the same as Algorithm 5 up until line 7. There, $\mathcal{C}$ is not inductive. The algorithm automatically terminates early with a failure if the maximum depth is exceeded on line 8. If not, it considers the *ctg* $\mathcal{Q}$, obtained from the witness on . Unlike line 4, Algorithm 10 does not immediately join $\mathcal{S}$ and $\mathcal{Q}$, but tries to remove $\mathcal{Q}$ from F_i instead. Line 10 first checks if *ctg_max* has not been exceeded. Then $\mathcal{Q}$ must be inductive relative to F_{i-1}. If not, then it cannot be blocked in F_i and must be joined with $\mathcal{S}$ as normal. If it is inductive, then $\mathcal{Q}$ is blocked in a similar way to `remove-cube`: line 13 finds if there is a higher F_{j-1} to which $\mathcal{Q}$ is also inductive and line 15 recursively generalises it into a smaller cube. Line 16 removes the generalised cube from $F_1, \dots F_{j-1}$ through the delta-encoded set of blocked states B_{j-1}. Now that $\mathcal{Q}$ is blocked, it will no longer appear as a witness that prevents the generalisation of $\mathcal{S}$.

Bradley [52] sets ctg_{max} to small values $(2-5)$ and $depth_{max}$ to 1. Deviating from these values did not yield significant improvements in runtime in preliminary experiments. The implementation sets $ctg_{max} = 5$ and $depth_{max} = 1$ by default.

5.1.4 Skipping blocked cubes

After cubes have been generalised and blocked, some other outstanding proof-obligations may have been blocked by them. When a proof-obligation (s, n) is obtained on line 2 in Algorithm 3, it is first checked if $\neg s$ is already subsumed by a previously learned clause. If so, (s, n) is immediately dequeued $(O \leftarrow O \setminus (s, n))$ and a new obligation is considered instead. The subsumption check

Algorithm 10: Version of `down` that addresses counterexamples-to-generalisation [52]

In : A cube $\mathcal{C}(X)$; $i \in \mathbb{N}$, where $i \leq k$; $depth \in \mathbb{N}$; $PDR = (F, k, M)$, where $F = \{ F_0, F_1, F_2, \dots \}$ satisfies Definition 3.3, $k \in \mathbb{N}$, and $M = (X, I, \Delta)$ is a symbolic transition system.

Out : A cube $\mathcal{C}(X)$, where $\mathcal{C} \Rightarrow S$, $\mathcal{I} \Rightarrow \neg\mathcal{S}$ and $\mathcal{F}_i \wedge \neg S \wedge \mathcal{T} \Rightarrow \neg S'$; or *failure* if none exists.

```
function ctg-down($\mathcal{C}, i, depth, PDR$):
    $\mathcal{S} \leftarrow \mathcal{C}$, $ctgs \leftarrow 0$
    loop
        if $\mathsf{SAT}(\mathcal{I} \wedge \mathcal{S}) = 1$ then                         ▷ $I \cap S \neq \varnothing$
            return failure
        if $\mathsf{SAT}(\mathcal{F}_i \wedge \neg\mathcal{S} \wedge \mathcal{T} \wedge \mathcal{S}') = 0$ then              ▷ $(F_i \setminus S).\Delta \subseteq \overline{S}$
            return $\mathcal{S}$
        if $depth > depth_{max}$ then
            return failure
        $\mathcal{Q} \leftarrow \mathsf{SAT}-witness$               ▷ ctg: $Q \subseteq F_i \setminus S$ and $\Delta(Q, S) \neq \varnothing$
        if $ctgs < ctg_{max}$ and $i > 0$ and $\mathsf{SAT}(\mathcal{I} \wedge \mathcal{Q}) = 0$ and
          $\mathsf{SAT}(\mathcal{F}_{i-1} \wedge \neg\mathcal{Q} \wedge \Delta \wedge \mathcal{Q}') = 0$ then
            $ctgs \leftarrow ctgs + 1$
            $j \leftarrow i$
            while $\mathsf{SAT}(\mathcal{F}_j \wedge \neg\mathcal{Q} \wedge \Delta \wedge \mathcal{Q}') = 0$ do
                $j \leftarrow j + 1$                                  ▷ $(F_{j-1} \setminus Q).\Delta \subseteq \overline{Q}$
            $\mathcal{Q} \leftarrow$ ctg-generalise($\mathcal{Q}, j - 1, depth + 1, PDR$)
            $B_{j-1} \leftarrow B_{j-1} \cup \neg\mathcal{Q}$                ▷ $\neg Q$ is inductive relative to $F_{j-1}$
        else
            $ctgs \leftarrow 0$
            $\mathcal{S} \leftarrow \bigwedge_{l \in \mathcal{S} \cap \mathcal{Q}} l$     ▷ cube containing common literals between $\mathcal{S}$ and $\mathcal{Q}$
```

from Section 5.1.2 can be used to quickly detect if this is the case by comparing it against all cubes already blocked [39].

5.2 Peterson's Algorithm

Peterson's Algorithm, devised by Gary L. Peterson [73], ensures mutual exclusion between two or more processes. Each process has what is known as a *critical section*, which likely includes the usage of some shared resource contested by each process. If mutual exclusion is satisfied, no more than one process may be in its critical section at the same time.

An instance of Peterson's Algorithm, defined to ensure mutual exclusion for p

Algorithm 11: Peterson's Algorithm for mutual exclusion [73]

```
In  : $id \in \mathbb{N}$, representing the current process id.
      $p \in \mathbb{N}$, representing the total number of processes.

function peterson($id$, $p$):
1   $level \leftarrow \mathrm{int}[0..p]$ ▷ initialised to 0
2   $last \leftarrow \mathrm{int}[1..p]$ ▷ initialised to 0; $last[0]$ is unused
3   for $l \leftarrow 1$ to $p-1$ do
4       $level[id] \leftarrow l$
5       $last[l] \leftarrow id$
6       while $last[l] = id \land \exists k \neq id \colon level[k] \geq l$ do
            ▷ wait
7   ▷ Critical section
8   $level[id] \leftarrow 0$
```

processes, will be denoted as $Peter(p)$.

5.2.1 Encoding

The problem is most easily encoded using SMT-logic, but PDR requires propositional logic with individual literal. Bit-blasting conversions exist within Z3, but there seems no easy way to keep track of the variable renaming. To retain control over the variable names for the PDR and IPDR, the encoding uses the following encodings for various SMT encodings in Algorithm 12.

Integers. The integers count, prev and those in the arrays pc, level and last are each represented by multiple Boolean variables, denoted by a subscript. Within Algorithm 12 the assignment of a variable $x \in \mathbb{N}$ with $x = (x_0 x_1 \dots x_n)_2$ to an integer a is defined by

$$a = x \triangleq (a_0 = x_0) \land (a_1 = x_1) \land \dots (a_n = x_n)$$

Equivalence between two integers a and b is defined similarly by comparing each bit

$$a = b \triangleq (a_0 = b_0) \land (a_1 = b_1) \land \dots (a_n = b_n)$$

The individuals bits are compared using the =-operator. In practice, comparison between two bits $a, b \in \mathbb{B}$ is implemented as $(\neg a \lor b) \land (a \lor \neg b)$.

Arrays. Each index for the arrays pc and level are defined as a separate variable, using the integer representation mention above.

Index. Once, in $\mathcal{T}_2(i)$, the encoding must index one of the arrays with a variable. Normally they are predictably index by a constant, requiring no additional

Algorithm 12: A transition system $Peter(p, l)$ (Definition 5.3) and a set of bad states encoded as $\neg\mathcal{P}$. In practice, this encoding is translated into propositional logic as described in Section 5.2.1.

$$\mathcal{I}_p \triangleq \mathit{count} = 0 \;\wedge \bigwedge_{i=0}^{p-1}(\mathit{pc}[i] = 0 \wedge \mathit{level}[i] = 0) \wedge \bigwedge_{i=0}^{p-2} \mathit{last}[i] = 0$$

$$\neg\mathcal{P}_p \triangleq \left(\sum_{i=0}^{p-1} \mathit{pc}[i] = 4\right) > 1$$ ▷ implemented with Z3's cardinality solver [36]

$$\mathcal{T}_p \triangleq \bigvee_{i=0}^{p-1}(\mathit{prev}' = i) \wedge (\mathcal{T}_0(i) \vee \mathcal{T}_1(i) \vee \mathcal{T}_2(i) \vee \mathcal{T}_3(i) \vee \mathcal{T}_4(i))$$

$\mathcal{C}(l) \triangleq (\mathit{prev} = \mathit{prev}' \Rightarrow \mathit{count}' = \mathit{count}) \wedge$
$(\mathit{prev} \neq \mathit{prev}' \Rightarrow \mathit{count}' = \mathit{count} + 1) \;\wedge$ ▷ increment with Eq. 2
$\mathit{count}' \leq l$ ▷ bound check with Z3's cardinality solver [36]

$$\mathcal{T}_0(i) \triangleq (\mathit{pc}[i] = 0) \wedge (\mathit{pc}[i]' = 1) \wedge (\mathit{level}[i]' = 1) \wedge \lambda_i(\mathit{pc}, \mathit{level}) \wedge \lambda(\mathit{last})$$

$\mathcal{T}_1(i) \triangleq (\mathit{pc}[i] = 1) \wedge (\mathit{level}[i] < p \Rightarrow \mathit{pc}[i]' = 2) \wedge$
$(\mathit{level}[i] \geq p \Rightarrow \mathit{pc}[i]' = 4) \wedge \lambda_i(\mathit{pc}) \wedge \lambda(\mathit{level}, \mathit{last})$

$\mathcal{T}_2(i) \triangleq \mathit{pc}[i] = 2 \wedge \mathit{pc}[i]' = 3 \wedge \mathit{last}[\mathit{level}[i]]' = i \wedge$ ▷ index with Eq. 1
$\lambda_i(\mathit{pc}) \wedge \lambda(\mathit{level})$

$\varphi_3(i) \triangleq \mathit{last}[\mathit{level}[i]] = i \;\wedge$ ▷ index with Eq. 1
$\bigvee_{k \in [0,p) \setminus i} \mathit{level}[k] \geq \mathit{last}[i]$ ▷ if-then-else condition for $T_3(i)$

$\mathcal{T}_3(i) \triangleq (\mathit{pc}[i] = 3) \wedge \varphi_3(i) \Rightarrow (\mathit{pc}'_i = 3 \wedge \mathit{level}[i]' = \mathit{level}[i]) \wedge$
$\neg\varphi_3(i) \Rightarrow (\mathit{pc}'_i = 1 \wedge \mathit{level}[i]' = \mathit{level}[i] + 1) \;\wedge$ ▷ increment with Eq. 2
$\lambda_i(\mathit{pc}, \mathit{level}) \wedge \lambda(\mathit{last})$

$\mathcal{T}_4(i) \triangleq (\mathit{pc}[i] = 4) \wedge (\mathit{pc}[i]' = 0) \wedge (\mathit{level}[i] = p - 1) \wedge (\mathit{level}[i]' = 0) \wedge$
$\lambda_i(\mathit{pc}, \mathit{level}) \wedge \lambda(\mathit{last})$

logic. In order perform the assignment $\mathit{last}[\mathit{level}[i]]' = i$ the following formula is used

$$\bigwedge_{x=1}^{p-1} \begin{array}{l}(\mathit{level}[i] = x \Rightarrow \mathit{last}[x]' = i) \wedge \\ (\mathit{level}[i] \neq x \Rightarrow \mathit{last}[x]' = \mathit{last}[x])\end{array} \tag{1}$$

Note that in Algorithm 11 the last array only uses the indices between 1 and p, since on line 5 $l = \mathit{level}[id] \in [1, p-1]$. So Eq. 1 does not check for $x = 0$.

Less. For the purposes of this thesis, 4 bits are enough to represent the integers in the encoding. It thus uses a 4-bit comparator, which asserts for each bit of a and b that if all more significant bits are equal, then a's bit must be lesser than b's.

$$(a_0a_1a_2a_3)_2 < (b_0b_1b_2b_3)_2 \triangleq \begin{array}{l}(\neg a_3 \wedge b_3) \vee \\ (\neg a_2 \wedge b_2 \wedge a_3 = b_3) \vee \\ (\neg a_1 \wedge b_1 \wedge a_2 = b_2 \wedge a_3 = b_3) \vee \\ (\neg a_0 \wedge b_0 \wedge a_1 = b_1 \wedge a_2 = b_2 \wedge a_3 = b_3)\end{array}$$

Increment. The encoding only demands incrementing a value by 1, and no other additions. This is implemented by combining full adders [3] into a ripple-carry adder for two numbers a and b. Within the encoding only a is variable, b is set to a constant $1 = (1000)_2$ and the carry-in c_0 to 0, making the first part a half-adder.

The addition of the first bit, normally encoded as $a_0' = a_0 \oplus b_0$, can be simplified to $a_0' = a$ since b_0 is known to be 0. The carry-out, normally $c_1 = a_0 \wedge b_0$, similarly becomes $c_1 = a_0$.

For the remaining bits can assume that $b_i = 0$, reducing the full adder equations $a_i' = a_i \oplus b_i \oplus c_i$ and $c_{i+1} = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i)$ can similarly be reduced. Incrementing an integer $\mathbf{a} \in \mathbb{N}$ is thus implemented as

$$\mathbf{a}' = \mathbf{a} + 1 \triangleq \begin{array}{l} a_0' = a_0 \wedge c_1 = a_0 \wedge \\ \bigwedge_{i=1}^{3} (a_i' = a_i \neq c_i) \wedge (c_{i+1} = a_i \wedge c_i)\end{array} \tag{2}$$

Note that the $\oplus$-operator (xor) is equivalent to $\neq$, implemented as the negation of the $=$-operator between bits mentioned above.

Unchanged Variables. If a variable is assigned within Algorithm 11, it is implicit that all other variables remain at their current value. Within a Boolean encoding, this must be explicitly asserted by equating each variable to its next-state counterpart. For readability, this is encoded in the λ and λ_p functions, which take the integer arrays pc, level and last as arguments.

The first denotes that for each process, its version of the given variables are unchanged.

$$\lambda(\varphi_1, \varphi_2, \ldots \varphi_n) \triangleq \bigwedge_{i \in [1,n]} \bigwedge_{j \in [0,p)} \varphi_i[j] = \varphi_i[j]'$$

The second does the same, but excludes the variables with the index belonging to the process x. This is commonly used when one process modifies its variables and the others must not touch theirs.

$$\lambda_x(\varphi_1, \varphi_2, \dots \varphi_n) \triangleq \bigwedge_{i \in [1,n]} \bigwedge_{j \in [0,p) \setminus x} \varphi_i[j] = \varphi_i[j]'$$

5.2.2 Incremental algorithm

This section proposes an incremental algorithm to verify the mutual exclusion property [38] for instances of Peterson's Algorithm. Each instance $\mathit{Peter}(p, l)$ has a fixed number of processes p and allows at most l interleavings. Within the context of the parallel Peterson's Algorithm, an interleaving occurs when one process has been the latest to execute an instruction and another one executes one of theirs.

Definition 5.3 (Constrained Peterson Transition System). The tuple $\mathit{Peter}(p, l) = (X, \mathcal{I}_p, \mathcal{T}_p \wedge \mathcal{C}(l))$ represents a constrained transition system for Peterson's Algorithm [73]. It is encoded into propositional logic as per Algorithm 12.

The regular, unconstrained Peterson's Algorithm $\mathit{Peter}(p)$ would be represented by $\mathit{Peter}(p, \infty)$ in this notation.

It would perhaps seem natural to consider the maximum number of processes as a parameter for a series relaxing systems. However, $\mathit{Peter}(2)$, $\mathit{Peter}(3)$, etc. are not monotonically increasing. Consider the subset of two transition systems in Figure 5.1. $\mathit{Peter}(2)$ can enter the critical section from state-1 because $\mathit{level}[i] \geq 1$. In that same state, $\mathit{Peter}(3)$ may not since $\mathit{level}[i] < 2$. Instead, it has a transition that enters one of the waiting rooms. While $\mathit{Peter}(3)$ conceptually allows for more behaviour, it does so by replacing certain transitions, i.e., it does not increase monotonically as is demanded by Definition 4.1. This does not even consider possible changes in the encoding, and consequently the states of the system, as the number of variables increases.

Previous work [22, 78] has used under-approximations of programs based on bounded interleavings to perform Bounded Model Checking. This can be deemed a valid approach as most bugs in a system occur after only a relatively low number of interleavings [42].

Counting Interleavings. In order to limit the number of interleavings in the system, two variables prev and count are introduced in Algorithm 12 in addition to the expected variables for Peterson's Algorithm.

prev tracks which process fired in the previous transition, it is set to i for the next state in $\mathcal{T}_p$ whenever one of the transitions $\mathcal{T}_0(i)$, $\mathcal{T}_1(i)$, $\mathcal{T}_2(i)$, $\mathcal{T}_3(i)$, or

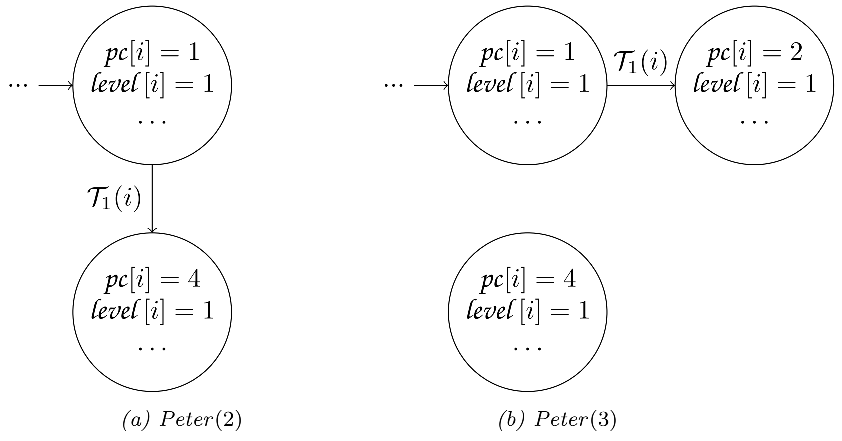


Figure 5.1: Outgoing transitions from state-1 for two instances $Peter(p)$ of Peterson's Algorithm, encoded as per Algorithm 11

$\mathcal{T}_4(i)$ are activated. Note that each transition is exclusive, as they demand that all program counters ***pc*** for each other process are unchanged.

$\mathcal{C}(l)$ uses the ***count*** variable to track the number of interleavings. If the previous process ***prev*** is different from the one currently firing ***prev***, ***count*** is incremented. Finally, it asserts that incrementing it does exceed the bound N.

Relaxing. The sequence of Peterson's Algorithm systems

$$Peter(p, 0) \sqsubset Peter(p, 1) \sqsubset \dots$$

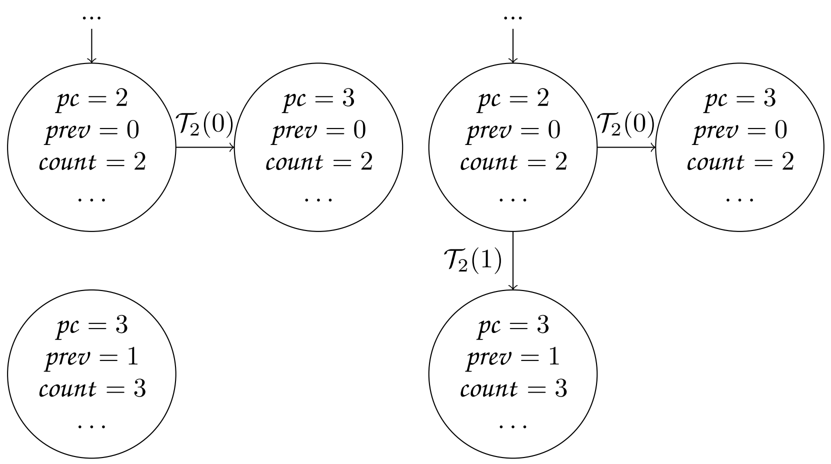
is relaxing as per Definition 4.1. The initial states are identical between each instance, as they are only dependent on the constant p.

Every state of the Peterson's Algorithm system $Peter(p, \infty)$ has p outgoing edges, representing one of the processes executing a step of the algorithm. For a bounded system $Peter(p, l)$, this is true as long as $\mathbf{count} < l$. If $\mathbf{count} = l$, then there is only one outgoing edge: the one that fires the transition corresponding to ***prev***. For a bound $l + 1$, there are now p outgoing edges when $\mathbf{count} = l$, and only one again when $\mathbf{count} = l + 1$. Figure 5.2 illustrates this for one state of systems $(2, 2)$ and $(2, 3)$.

Since the line $\mathbf{count}' \leq l$ is the only one that changes when, the difference between the transition relations $\mathcal{T}_p \wedge \mathcal{C}(l)$ and $\mathcal{T}_p \wedge \mathcal{C}(l + 1)$ is exactly these added transitions, meaning that $\mathcal{T}_p \wedge \mathcal{C}(l) \subset \mathcal{T}_p \wedge \mathcal{C}(l + 1)$ and $Peter(p, l) \sqsubset Peter(p, l + 1)$.

5.3 Reversible Pebbling Game

Giulia Meuli, Mathias Soeken, Martin Roetteler, Nikolaj Bjorner, and Giovanni De Micheli [68] likened Quantum Memory Management to the Reversible Peb-



(a) Peter$(2, 2)$ *(b) Peter*$(2, 3)$

Figure 5.2: Outgoing transitions from state-2 for two instances $Peter(p, l) = (X, \mathcal{I}_p, \mathcal{T}_p \wedge \mathcal{C}(l))$ *of Peterson's Algorithm, encoded as per Algorithm 11.*

bling Game [4, 63].

The Reversible Pebbling Game is defined by a tuple (G, O), consisting of an Directed-Acyclic-Graph $G = (V, E)$ and a set output vertices $O \subseteq V$. Vertices in the pebbling game can be marked as *pebbled* or *empty*, initially all vertices are empty. A *pebbling configuration* is a set $P \subseteq V$, which denotes if a vertex is pebbled in that configuration. The goal of the game is to enter the configuration where only the output vertices are pebbled: $P = O$. A vertex can be switched from empty to pebbled or vice versa, if each vertex that has an edge to it is also pebbled. This is denoted as *pebbling* and *unpebbling* a vertex respectively.

A *pebbling strategy* is a series of pebbling configurations $P_1 = \{\ \}, P_2, \ldots P_m = O$ obtained by pebbling or unpebbling one or more of the vertices V. For each configuration, $P_i \neq P_{i+1}$ and $E.(P_i \oplus P_{i+1}) \subseteq P_i$.

Giulia Meuli et al. [68] encode the pebbling game into propositional logic and pose the game as a satisfiability problem. This allows them to perform a SAT-based bounded model checking algorithm that can return valid pebbling strategies. The algorithm can constrain the maximum number of vertices which may be pebbled in any configuration in a strategy and uses multiple executions of this algorithm to find a strategy using the minimum number of pebbles.

5.3.1 Boolean encoding

A transition system described the pebbling game $TS = (X, I, \Delta)$ is encoded using Boolean formulae in Algorithm 13.

Every vertex v has a Boolean variable p_v, denoting whether or not v is pebbled

Algorithm 13: A CNF encoding for the Reversible Pebbling Game $(G = (V, E),\ O)$ into a transition system (X, I, Δ) and a set of bad states $\overline{P} \subseteq 2^X$. Inspired by Giulia Meuli et al. [68] and adapted for PDR.

$$\begin{aligned}
\mathcal{I} &\triangleq \textstyle\bigwedge_{v \in V} \neg p_v \\
\neg\mathcal{P} &\triangleq \textstyle\bigwedge_{v \in O} \quad p_v \wedge \bigwedge_{v \notin O} \neg p_v \\
\mathcal{T} &\triangleq \textstyle\bigwedge_{(v,w) \in E} \ (p_w \vee \neg p'_w \vee p_v) \wedge (\neg p_w \vee p'_w \vee p_v) \wedge \\
&\qquad\qquad\quad (p_w \vee \neg p'_w \vee p'_v) \wedge (\neg p_w \vee p'_w \vee p'_v) \wedge \\
\mathcal{C}(N) &\triangleq \sum_{v \in V} p_v \leq N \ \wedge \ \sum_{v \in V} p'_v \leq N \quad \triangleright \text{ implemented with Z3's cardinality solver [36]}
\end{aligned}$$

in the current state, and p'_v, which denotes it is pebbled in the next state. The sets of variables X and X' are simply the collection of these variables.

$\mathcal{I}$ encodes the initial configuration I of the transition system that describes the pebbling game: all vertices are empty.

$\mathcal{T}$ encodes the transition relation Δ. The clauses in Algorithm 13 are a CNF representation of the transition relation from Giulia Meuli et al. [68]

$$\mathcal{T} = \bigwedge_{(v,w) \in E} ((p_w \oplus p'_w) \Rightarrow (p_v \wedge p'_v))$$

adapted for PDR. It states that if a vertex p_w changes from being pebbled to unpebbled or vice versa, then its children must be pebbled in the current and next state.

$\neg\mathcal{P}$ encodes the bad states for PDR. By setting this to the final pebbling configuration, where only the vertices in O are pebbled, PDR will search for a trace that reaches it.

The cardinality constraint $\mathcal{C}$ limits both the current- and next-states to have at most N vertices pebbled. This constraint added to the transition relation, i.e., $\Delta = \mathcal{T} \wedge \mathcal{C}(N)$, so that every reachability query is performed under the constraint.

5.3.2 Incremental algorithm

The goal of the algorithm is to find a pebbling strategy that uses a minimal amount of pebbles. PDR is thus run multiple times, each time with a different constraint.

The range pebbling constraints to consider is $[|O|, |V|]$. As the final configuration O must be pebbled in any strategy and one can pebble at most all the vertices V.

Constraining. A constraining version of this algorithm begins with a constraint $\mathcal{C}(|V|)$. Whenever a trace is found that uses at most p pebbles, the constraint is lowered to $\mathcal{C}(p-1)$. If an invariant is found, then the previous trace is a strategy that uses the minimal amount of pebbles. If a trace is found when the constraint is $\mathcal{C}(|O|)$, then this trace must represent a minimal pebbling strategy.

Because $\mathcal{C}(N)$ enforces an upper bound on the number of pebbled vertices, a trace may actually use a number of pebbles $p \leq N$. This brings along an implicit optimisation where the algorithm may skip over some instances if $p < N$. Thus, the sequence of constraining systems $M_0 \sqsupset M_1 \sqsupset \dots M_z$, expected by `ipdr-constrain` in Algorithm 6, is constructed on demand. The first system is defined by $M_0 = (X, I, \mathcal{C}(|V|) \wedge \Delta)$ and the following systems by

$$\forall 1 \leq i \leq z \colon M_i = (X, I, \mathcal{C}(\pi_{i-1} - 1) \wedge \Delta)$$

where π_{i-1} denotes the maximum number of pebbles used in the trace from the previous result.

Relaxing. A relaxing version starts with $\mathcal{C}(|O|)$. Whenever PDR finds an invariant under $\mathcal{C}(N)$, the constrained is incremented to $\mathcal{C}(N+1)$. The first trace it finds is then represents a strategy with the minimal amount of pebbles.

The relaxing systems $M_0 \sqsubset M_1 \sqsubset \dots M_z$, expected by `ipdr-relax` in Algorithm 7, are known beforehand. The length is given by $z = |V| - |O|$ and each system is defined by

$$\forall 0 \leq i \leq z \colon M_i = (X, I, \mathcal{C}(|O| + i) \wedge \Delta)$$

Binary Search. Both the constraining and relaxing IPDR algorithms expect a valid PDR-state and produce one after completing an instance. Thus, the two approaches may combined into a binary search algorithm. This allows the algorithm to take larger incremental steps and switch direction if it goes over the lower or upper bound, while still reusing information from the previous run.

Algorithm 14 lists this algorithm. It begins by encoding the given DAG as described in Algorithm 13 and initiating the pdr-state in the same way as Algorithm 6 and 7.

If the initial (unconstrained) PDR run on line 4 returns an inductive invariant, then there is no pebbling strategy. If there is a trace, then the upper bound to search for a minimal pebbling strategy becomes one below the number of pebbles used. The lower bound is the number of pebbles in the final configuration O.

While there is at least one number in the range $[L, U]$, the loop on line 9 sets the constraint on the middle number m. Line 12 performs constraining IPDR if the new constraint is lower than the previous, and relaxing IPDR if it is larger.

Algorithm 14: Binary search IPDR for the Reversible Pebbling Game

```
In  : A DAG $G = (V, E)$ and a final pebbling configuration $O \subseteq V$.
Out : A trace that represents a pebbling strategy, or $\varnothing$ if none exists.

function binary-pebbling($G, O$):
    $(X, I, \Delta), P \leftarrow$ encode($G, O$) ▷ as per Algorithm 13
    $F \leftarrow \{ F_0 = I, F_1 = 1 \}$
    $(M, P, k, F, O) \leftarrow ((X, I, \Delta), P, 0, F, \varnothing)$ ▷ A pdr-state as per Definition 3.6
    $(M, P, k, F, O)$, result $\leftarrow$ ipdr-main($(M, P, k, F, O)$)
    if result is an inductive invariant then
        return $\varnothing$ ▷ no strategy exists
    $n \leftarrow |$trace from result$|$ ▷ track latest trace
    $L \leftarrow |O|,\ U \leftarrow n - 1$
    while $L \leq U$ do
        $m \leftarrow \frac{L+U}{2}$
        $M' \leftarrow (X, I, \mathcal{C}(m) \wedge \Delta)$
        if $m < n$ then
            $(M, P, k, F, O) \leftarrow$ constrain($(M, P, k, F, O),\ M'$)
        else
            $(M, P, k, F, O) \leftarrow$ relax($(M, P, k, F, O),\ M'$)
        $(M, P, k, F, O)$, result $\leftarrow$ ipdr-main($(M, P, k, F, O)$)
        if result is an inductive invariant then
            $L \leftarrow m + 1$
        else
            trace $\leftarrow$ result
            $U \leftarrow |$trace$| - 1$
        $n \leftarrow m$
    return trace ▷ the latest trace is the minimal
```

If the result from the PDR run is an invariant, then the binary needs to search for a higher constraint and the new lower bound is $m + 1$. If there is a trace, the search must continue below the number of pebbles of that trace.

When the loop concludes, the trace with the lowest number of pebbles is the last one recorded on algorithm 14 (since m only goes down after finding a trace).

6 Experiments

This section experimentally evaluates the IPDR algorithm for both constraining and relaxing systems from Section 4. The experiments investigate whether the workload for subsequent systems is reduced when using IPDR compared to regular PDR, and whether this results in a decrease in runtime.

6.1 Hardware

The experiments were run on a Dell OptiPlex 7040 Mini system. It uses 16 GB of 2400 MHz DDR4 memory and an Intel® Core™ i7-6700T CPU @ 2.80GHz $\times 8$.

6.2 Setup

A IPDR run executes the constraining or relaxing IPDR algorithm, implemented described by Algorithm 6 and Algorithm 7. The constraining version is denoted as C-IPDR and the relaxing version R-IPDR.

A *naive* IPDR run is used to compare against. These perform similar steps to the `ipdr-constrain` and `ipdr-relax` functions, but do not call the `constrain` and `relax` functions to copy clauses. A naive run is performed the same number of times as a corresponding normal IPDR run, with the same seeds.

IPDR and naive IPDR both use the internal model checker mentioned in Section 5.

A SPACER run is the same as a naive IPDR run, except that it uses Z3's [36] PDR implementation, Spacer [59], with its fixed point engine. Since the internal state of the spacer is not exposed to the user, implementing IPDR was not viable for this thesis.

Most experiments are run 10 times, where each execution has set the seed for the underlying Z3 SAT-solver to a random number. When runtimes only allowed a lower number of repetitions, for practical purposes, this is specified in the figures and discussion. Each run records several statistics, detailed below in Appendix A, that are aggregated into mean averages and standard deviations thereof.

6.2.1 Spacer encoding

Z3's SPACER engine [57, 59] requires the system to be encoded into Constrained Horn Clauses [44]. This thesis uses a general conversion from a transition system to horn clauses, detailed in Algorithm 15.

A function $\sigma : X \to \mathbb{B}$ is used to denote that a combination of variables represents a state in the system.

The *Initial* and *Transition* horn clauses are added to the fixed point solver and *Final* is given as a query. A SAT answer to the query indicates the existence of a trace describing a pebbling strategy.

The Pebbling Game, as presented by Giulia Meuli et al. [68] demands an encoding like this and cannot encode its transition into separate clauses. This instance of the pebbling game allows multiple pebblings and unpebblings in parallel as one step of the transition relation, giving the need to encode it into one horn clause.

Algorithm 15: A Constrained Horn Clause [44] encoding for a transition system (X, I, T) an a set of bad states $\overline{P} \subseteq 2^X$. These are represented by the formulae $\mathcal{I}$, $\mathcal{T}$ and $\neg\mathcal{P}$.

```
$Initial \triangleq \forall_X :\quad \mathcal{I}(X) \Rightarrow \sigma(X)$    ▷ added to solver
$Transition \triangleq \forall_{X,X'} :\quad \sigma(X) \wedge \mathcal{C}(N) \wedge \mathcal{T} \Rightarrow \sigma(X')$    ▷ added to solver
$Final \triangleq \exists_X :\quad \sigma(X) \wedge \neg\mathcal{P}(X) \Rightarrow \bot$    ▷ query for solver
```

Peterson's Algorithm does not have such a feature and any process choosing one of its transitions excludes any other combinations of process and transition. An encoding exists that more intuitively maps each possible transition to its own horn clause using Z3's SMT formulae. After some informal initial testing, this performed considerably slower than reusing Algorithm 12 with Algorithm 15. More efficient encodings may exist, but optimising those falls outside of the scope of this thesis.

6.3 Peterson's algorithm results

This experiment evaluates R-IPDR when proving instances of Peterson's Algorithm discussed in Section 5.2. An instance represents Peterson's Algorithm for a limited number of processes. Each instance is incrementally verified for correctness by allowing an increasing number interleavings.

Figure 6.1 shows the overall runtimes for three instances: $Peter(2)$, $Peter(3)$ and $Peter(4)$. These are defined as per Section 5.2. The runtimes for these instance increase exponentially for higher number of processes, making $Peter(4)$ the greatest instance that was feasible to run. The runtime also limited the bound on the number of context switches that was feasible to run: $Peter(2)$ and $Peter(3)$ was verified up to a bound of 10 switches, $Peter(3)$ to a bound of 4 and $Peter(4)$ to 3. Additionally, the $Peter(3)$ with SPACER was only repeated 3 times instead of 10 and $Peter(4)$ is omitted as it did not terminate in reasonable time.

All tested instances have a reduced runtime when using C-IPDR, compared to both the naive implementation and SPACER. $Peter(2)$ showed the greatest improvement of 74% when going from the naive implementation to R-IPDR. $Peter(3)$ and $Peter(4)$ achieve a similar speedup of 43% and 39% respectively. SPACER performed worse than both implementations. R-IPDR achieved speedups of 97% and 98% for $Peter(2)$ and $Peter(3)$ respectively. There is no data for SPACER for $Peter(4)$, but this was omitted due to it being considerably slower.

Table 6.1 also compares the runtime of using PDR to only verify the final instance of each experiment against using R-IPDR to incrementally verify all instances up

Input	Naive R-IPDR	SPACER	R-IPDR	Speedup vs. naive	Speedup vs. SPACER
Peter(2) *	38.654 s	307.348 s	10.144 s	(74%)	(97%)
Peter(3) *	598.523 s	15431.695 s	342.747 s	(43%)	(98%)
Peter(4)	10321.160 s		6328.118 s	(39%)	

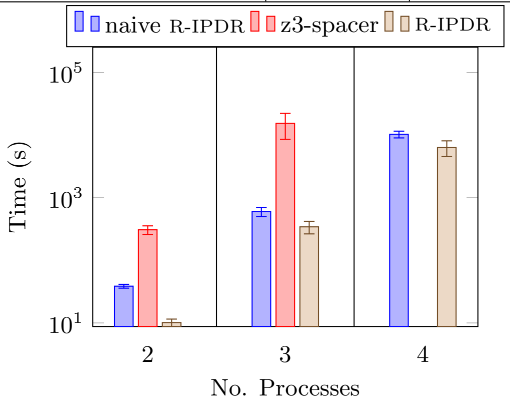


*Figure 6.1: Mean average runtimes and standard deviations for Peterson's Algorithm. A naive R-IPDR and SPACER implementation are compared against R-IPDR. Peter(2) was run 10 times and up to 10 interleavings, Peter(3) was run 10 times and up to 4 interleavings, and Peter(4) was run 3 times and up to 3 interleavings. The Speedup columns denote the percentage decrease from naive R-IPDR and SPACER to R-IPDR. Marked * inputs hung in certain cases of SPACER, see Appendix D. For Peter(4), SPACER was not feasible to run and has been omitted from the results.*

to that point. R-IPDR is initially slower, but becomes less so when more processes are added. For *Peter*(4), incrementally verifying 0 through 3 interleavings is faster than only use a run of PDR to verify 3 interleavings.

Input	PDR final only	R-IPDR total time	Speedup
Peter(2)	3.368 s	10.144 s	-201%
Peter(2) (4 interleavings)	3.726 s	6.317 s	-70%
Peter(3)	236.678 s	342.747 s	-45%
Peter(4)	8667.870 s	6328.118 s	27%

Table 6.1: Comparison of runtimes for only the final instance of an input using PDR and incrementally verifying up to that instance using R-IPDR. Peter(2) was also considered up to 4 interleavings as R-IPDR saturates at that point.

6.4 Reversible Pebbling Game

This experiment evaluates IPDR when solving the Reversible Pebbling Game introduced in Section 5.3, testing a constraining, relaxing and binary search algorithm to incrementally find a minimal pebbling strategy.

6.4.1 Benchmarks

The DAGs used for the Reversible Pebbling Problem in this thesis were obtained by generating a dependency graph for a quantum circuit by writing it in Static Single Assignment form [79], which ensures that variables are only assigned once. Every time an output is written to the same wire in the circuit, a new (unique) version of that variable is used represent the result. That result may then only be used as an input for a gate. For every gate, an edge is drawn from every input to a new output. The original inputs of the circuit are ignored in this, as they are always available.

The circuits themselves were taken from the Reversible Logic Synthesis Benchmarks Page [66]. This page lists several "Families" of circuits of increasing size and complexity. The circuits for the experiments are those which could be completed in a reasonable time in a round of preliminary testing on available hardware. Table 6.2 lists the circuits used and the sizes of the DAGs generated from it.

Within this listing and the experiments, the graphs are ordered based on their number of nodes.

Due to issues with exactly replicating the graphs from Meuli's original work and the fact that graph sizes increased rapidly in their benchmarks, the current circuits were chosen as inputs in order to examine the effects of IPDR. As the following results show, BMC seems to outperform PDR for this problem and would likely do so for those benchmarks as well.

	Edges	Nodes		Edges	Nodes
`ham3tc`	5	5	`gf2^5mult_29_129`	28	17
`mod5d1`	7	8	`rd73d2`	29	19
`gf2^3mult_11_47`	10	11	`4_49tc1`	31	20
`5mod5tc`	16	12	`mod5adders`	36	22
`gf2^4mult_19_83`	18	15	`hwb4tc`	37	23
`nth_prime4_inc_d1`	26	16	`ham7tc`	38	29
`4b15g_1`	28	17	`5bitadder`	44	29

Table 6.2: The circuits [66] used for the pebbling experiments and the number of edges and nodes in the generated DAGs. The names represent those of the `.tfc` files that detail the reversible circuits.

6.4.2 Constraining IPDR results

Figure 6.2 contains the overall results for the constraining version of IPDR. A table shows the mean average runtimes for each input, rounded to three decimal points. The IPDR result is listed along with the percentage decreases of IPDR compared to the other results, rounded to whole numbers. These are obtained by: $\frac{\textit{non-ipdr} - \textit{ipdr}}{|\textit{non-ipdr}|} \cdot 100$, rounded to whole numbers. A bar-graph displays these runtimes along with the standard deviation over all iterations. The workload tends to increase exponentially with increasing input sizes, graphs are displayed with a logarithmic scale.

Nearly all tested benchmarks achieve a speedup using the constraining IPDR algorithm over using the same PDR-based model checker without incremental functionality (naive C-IPDR). The decrease in runtime ranges from 16% to 67%. There are four exceptions. The `ham3tc` input shows a negligible difference of 1%. The `gf2^3mult_11_47`, `gf2^4mult_19_83` and `gf2^5mult_29_129` inputs all show an increase in runtime when using C-IPDR. These inputs are all derived from circuits that compute the same Galois Field Multiplication function for a variable input size [66].

C-IPDR outperforms the SPACER engine for the smaller-sized inputs, excepting the Galois Field Multiplication inputs mentioned before. For larger inputs where C-IPDR takes 100 seconds or more, SPACER becomes considerably faster. At those larger inputs, SPACER also outperforms the naive C-IPDR implementation.

6.4.3 Relaxing IPDR results

Figure 6.3 displays the overall results of R-IPDR, in the same manner as described in Section 6.4.2 for Figure 6.3.

For R-IPDR, many inputs show an increase in runtime when using R-IPDR, compared to naive R-IPDR. Compared to the runtimes of C-IPDR in Figure 6.2, R-IPDR has lower total runtimes for most inputs.

Of the bad inputs for C-IPDR, `gf2^3mult_11_47` and `gf2^4mult_19_83` now have

	Input	Naive C-IPDR	SPACER	C-IPDR	Speedup vs. naive	Speedup vs. SPACER
a	`ham3tc`	0.031 s	0.281 s	0.031 s	1%	89%
b	`mod5d1`	1.942 s	2.850 s	1.067 s	45%	63%
c	`gf2^3mult_11_47`	2.545 s	2.670 s	2.751 s	-8%	-3%
d	`nth_prime4_inc_d1`	3.428 s	4.098 s	2.878 s	16%	30%
e	`4b15g_1` *	34.711 s	13.225 s	11.513 s	67%	13%
f	`4_49tc1`	55.970 s	37.466 s	18.761 s	66%	50%
g	`5mod5tc`	989.706 s	377.234 s	748.969 s	24%	-99%
h	`hwb4tc`	30.292 s	30.153 s	9.865 s	67%	67%
i	`gf2^4mult_19_83`	19.618 s	25.678 s	111.350 s	-468%	-334%
j	`rd73d2` *	836.655 s	351.181 s	499.460 s	40%	-42%
k	`mod5adders` *	57.610 s	57.699 s	34.781 s	40%	40%
l	`ham7tc`	386.719 s	90.823 s	170.244 s	56%	-87%
m	`5bitadder`	1396.920 s	341.008 s	964.889 s	31%	-183%
n	`gf2^5mult_29_129`	760.006 s	242.379 s	1243.961 s	-64%	-413%

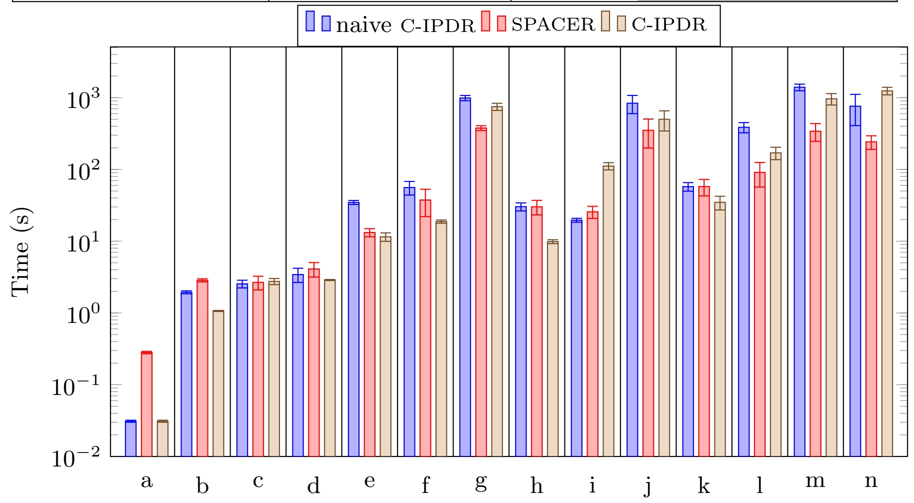


*Figure 6.2: Mean average runtimes and standard deviations (10 repetitions) for the constraining Pebbling Problem. A naive C-IPDR and SPACER implementation are compared against C-IPDR. The Speedup columns denote the percentage decrease from naive C-IPDR and SPACER to C-IPDR, grey cells indicate that C-IPDR slowed down. Marked * inputs hung in certain cases of SPACER, see Appendix D.*

	Input	Naive R-IPDR	SPACER	R-IPDR	Speedup vs. naive	Speedup vs. SPACER
a	`ham3tc`	0.041 s	0.144 s	0.039 s	6%	73%
b	`mod5d1`	0.832 s	1.413 s	0.646 s	22%	54%
c	`gf2^3mult_11_47`	1.323 s	1.416 s	1.159 s	12%	18%
d	`nth_prime4_inc_d1` *	2.647 s	2.433 s	2.806 s	-6%	-15%
e	`4b15g_1`	16.901 s	8.773 s	19.870 s	-18%	-126%
f	`4_49tc1`	43.018 s	19.833 s	63.133 s	-47%	-218%
g	`5mod5tc`	845.274 s	324.640 s	747.978 s	12%	-130%
h	`hwb4tc`	15.325 s	14.018 s	22.853 s	-49%	-63%
i	`gf2^4mult_19_83`	5.755 s	8.441 s	5.366 s	7%	36%
j	`rd73d2` *	634.672 s	202.320 s	654.798 s	-3%	-224%
k	`mod5adders`	45.037 s	38.639 s	28.258 s	37%	27%
l	`ham7tc`	151.867 s	53.712 s	211.390 s	-39%	-294%
m	`5bitadder`	639.886 s	91.152 s	474.993 s	26%	-421%
n	`gf2^5mult_29_129`	211.999 s	64.588 s	255.076 s	-20%	-295%

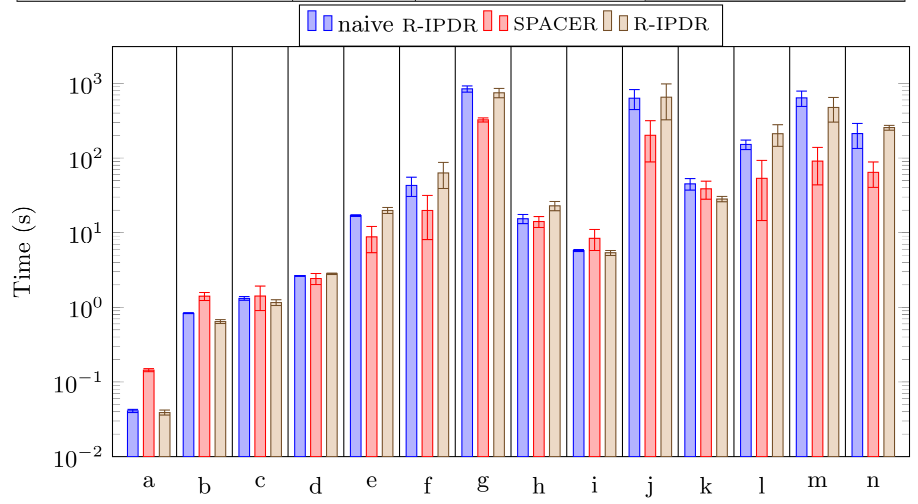


*Figure 6.3: Mean average runtimes and standard deviations (10 repetitions) for the relaxing Pebbling Problem. A naive R-IPDR and SPACER implementation are compared against R-IPDR. The Speedup columns denote the percentage decrease from naive R-IPDR and SPACER to R-IPDR, gray cells indicate that R-IPDR slowed down. Marked * inputs hung in certain cases of SPACER, see Appendix D.*

a slight decrease in runtime. Although, it should be noted that their runtimes are considerably lower than in C-IPDR.

Section 7.3 will consider more detailed statistics for some inputs. Some general remarks can be made that apply to all. Compared to the constraining algorithm, the relaxing algorithm tends to perform much fewer iterations. It seems that the minimum number of pebbles that can be used for a valid strategy lies much closer to the starting point of the relaxing algorithm than it does to that of the constraining algorithm. Furthermore, most relaxing iterations seem trivial to solve until the algorithm gets close to the first trace.

6.4.4 Binary Search results

Figure 6.3 displays the overall results of R-IPDR, in the same format as in Section 6.4.2 and Section 6.4.3.

When comparing the naive implementation of the binary search to IPDR, the speedup shows a similar pattern to C-IPDR in Figure 6.2. However, IPDR binary search runtimes are generally slower than C-IPDR.

The `5mod5tc`, `gf2^4mult_19_83`, `mod5adders`, `5bitadder` and `gf2^5mult_29_129` inputs show an improvement compared to C-IPDR. Although of those inputs, only `5mod5tc` has a (slight) improvement compared to R-IPDR.

	Input	Naive IPDR	SPACER	IPDR	Speedup vs. naive	Speedup vs. SPACER
a	`ham3tc`	0.135 s	0.284 s	0.043 s	68%	85%
b	`mod5d1`	1.297 s	1.573 s	1.225 s	6%	22%
c	`gf2^3mult_11_47`	2.124 s	1.861 s	3.169 s	-49%	-70%
d	`nth_prime4_inc_d1`	3.776 s	4.239 s	3.041 s	19%	28%
e	`4b15g_1` *	28.433 s	10.814 s	12.944 s	54%	-20%
f	`4_49tc1`	47.795 s	21.864 s	20.671 s	57%	5%
g	`5mod5tc`	938.345 s	363.142 s	738.673 s	21%	-103%
h	`hwb4tc`	19.572 s	17.900 s	8.961 s	54%	50%
i	`gf2^4mult_19_83`	13.916 s	13.320 s	95.186 s	-584%	-615%
j	`rd73d2`	714.732 s	279.362 s	594.911 s	17%	-113%
k	`mod5adders` *	45.037 s	39.628 s	29.790 s	34%	25%
l	`ham7tc`	289.942 s	72.480 s	176.663 s	39%	-144%
m	`5bitadder`	941.210 s	110.511 s	497.072 s	47%	-350%
n	`gf2^5mult_29_129`	423.146 s	124.520 s	634.630 s	-50%	-410%

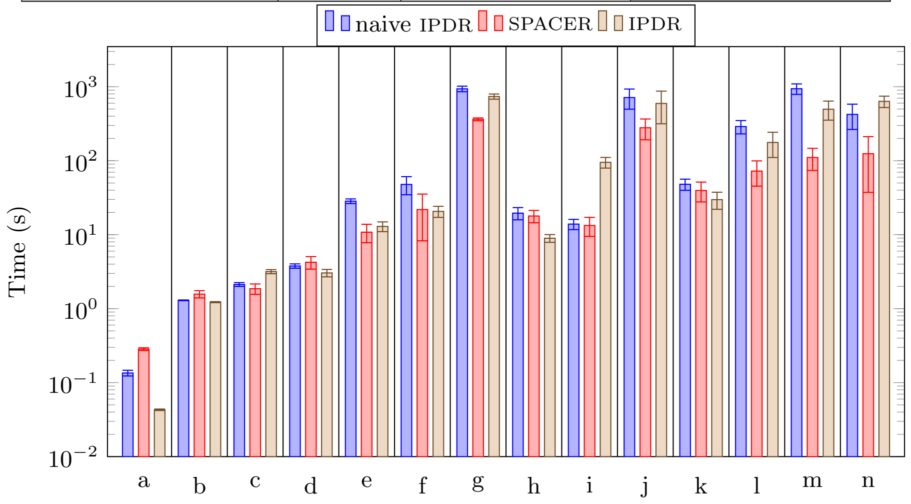


*Figure 6.4: Mean average runtimes and standard deviations (10 repetitions) for a binary search algorithm to solve the Pebbling Problem. A naive IPDR and SPACER implementation are compared against IPDR. The Speedup columns denote the percentage decrease from naive IPDR and SPACER to IPDR, gray cells indicate that IPDR slowed down. Marked * inputs hung in certain cases of SPACER, see Appendix D.*

7 Discussion

This section further interprets the results from Section 6 before drawing our final conclusions in Section 9.

In addition to the total runtimes, the implementation [13] records additional statistics to give insight into the performance of IPDR. Section 7.1 will explain these statistics and the remainder of this section will refer to them to discuss the experimental results in further detail. This discussion will display the most relevant statistics as graphs. Section A.1, Section A.2 and Section A.3 fully list all recorded statistics as graphs for completeness.

7.1 Statistics

The *runtime* measures the time between the beginning and end of Section 4.1 and Section 4.2, which includes the time spent copying information between iterations. The *incremental runtime* measures this time between the beginning and end of `ipdr-constrain` (Algorithm 6) and `ipdr-relax` (Algorithm 7).

The *handled-ctis* statistic counts the number of violations found on line 2 in Algorithm 1: states in F_k that reach a $\overline{P}$-state in one step. When displayed, it is paired with the total runtime of PDR.

The *handled-obligations* statistic keeps count of how many times PDR has fulfilled a proof-obligation in the minor iteration from Algorithm 3. It is paired with the total amount of time that has been spent handling such obligations: generalising cubes and finding predecessors to proof-obligations. This time is included in the total runtime statistic.

The SAT-*calls* statistic tracks the amount of queries that have been made to the SAT-solver over the entire algorithm, along with the time spent waiting for their results. Again, this amount of time is directly included in the runtime. It also contains the work of the SAT-queries done to find and generalise obligations.

Relaxing IPDR may not necessarily copy all learned clauses to a relaxing system. `relax` in Algorithm 7 copies a cube as far as possible. The *copied-levels* statistic is the percentage of how many "levels" were retained in the relaxed instance. If F_i is the highest candidate in which a cube is blocked, it has a level i. This percentage is paired with the time it took to copy these clauses.

As discussed in Section 5.3, C-IPDR may skip certain constraints if the a PDR iteration finds smaller trace. Since the seeds for PDR's SAT-solvers are changed in each experiment, this may cause certain experimental runs for the same input to skip different constraints. In practice, this mostly occurs for the initial PDR run: the constraint is set to the maximum possible value and the resulting trace is often much shorter. Afterwards, as the PDR runs become harder, the traces cardinality tends to match their constraint. In these cases, the statistics for a PDR run are recorded for each constraint it was run for. For example, in 10 experiments C-IPDR first finds a trace of cardinality 12 nine times and one time it finds a trace of cardinality 11. The statistics for constraint 12 will have nine entries, since one run skipped that constraint, but this work is included in the entry for constraint 11.

Due to a bug in the collection of copyrate, discovered late, the real percentages may be higher than reported. Clauses are reported when they can be pushed no further, but not when they are pushed to the furthest possible frame. In most runs this is barely noticeable, if at all, since clauses are rarely pushed this far. Only for simpler inputs does this become apparent. This is most notable with the *Peter*(2) experiment in Section 6.3, where IPDR fully saturates, copies all clauses fully and PDR terminates on initialisation.

7.2 Peterson's algorithm

Section 6.3 showed that relaxing IPDR consistently reduces the total runtimes of incrementally verifying Peterson's Algorithm for the tested instances. This section goes into more detail on the performance of IPDR.

7.2.1 Statistics

This section examines the statistics mentioned in Section 7.1. A consistent trend for all instances is that improvements are much lesser for 0, 1 and 2 interleavings, but become significant starting at 3 interleavings. For *Peter*(3), IPDR was actually slightly slower for 2 interleavings, before significantly improving for 3 interleavings.

2 Processes. Figure 7.1 reveals that runtimes and the number of initial ctis is consistently lower on every incremental step for R-IPDR. Starting at around 3 interleavings, each new instance with naive IPDR seems to not be significantly more difficult to solve than the previous. Around that same point, R-IPDR's runtimes significantly reduce and reach a point of saturation where no new ctis or obligations are

The same trend can be seen in the number of total obligations and SAT-queries in Figure A.2. Once there are no more ctis and obligations, only a minimal number of SAT-queries is performed in each iteration to perform the initialisation and phase and the copying of clauses to the relaxed instance.

3 Processes. When R-IPDR for *Peter*(3) reaches 3 interleavings, the total runtime is significantly decreased compared to naive IPDR, shown in Figure 7.2 along with a reduction in the number of obligations. Figure A.3a and Figure A.4b show that the same is true for the initial ctis and SAT-queries.

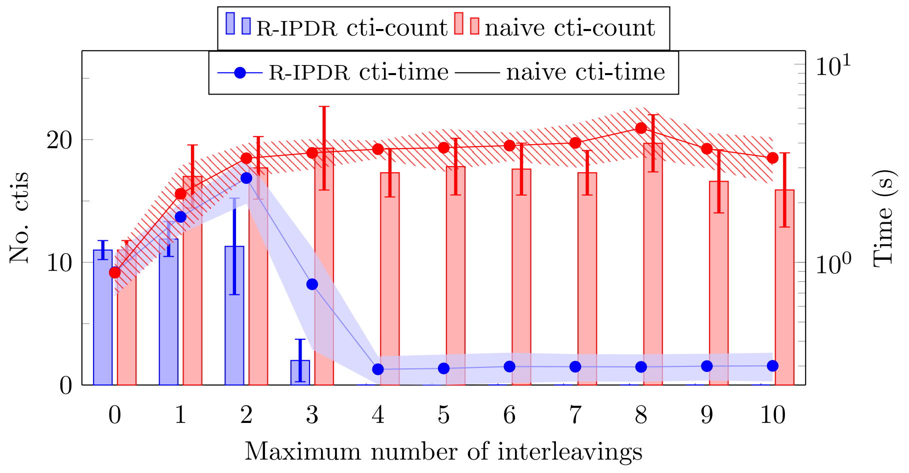


Figure 7.1: Statistics for R-IPDR *from* Peter(2) *(10 repetitions). Left axis: the number of counterexamples found in the major iteration (cti-count). Right axis: the total runtime (cti-time).*

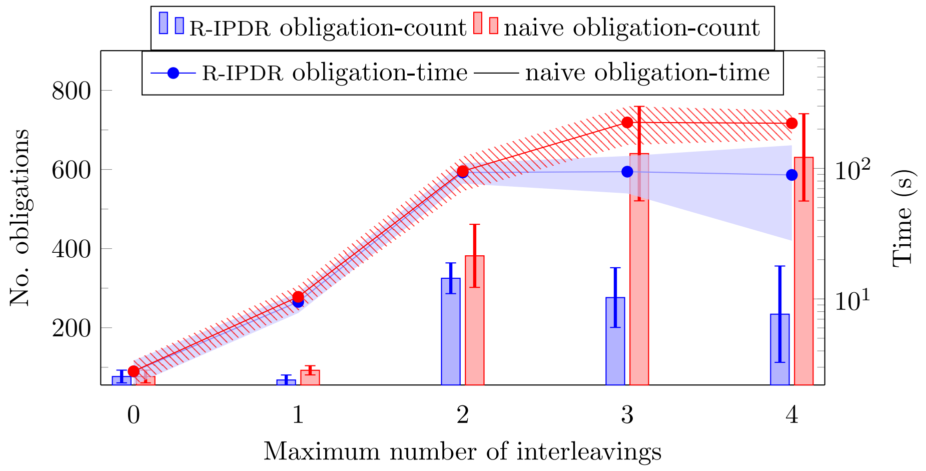


Figure 7.2: Statistics from R-IPDR *for* Peter(3) *(10 repetitions). Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time).*

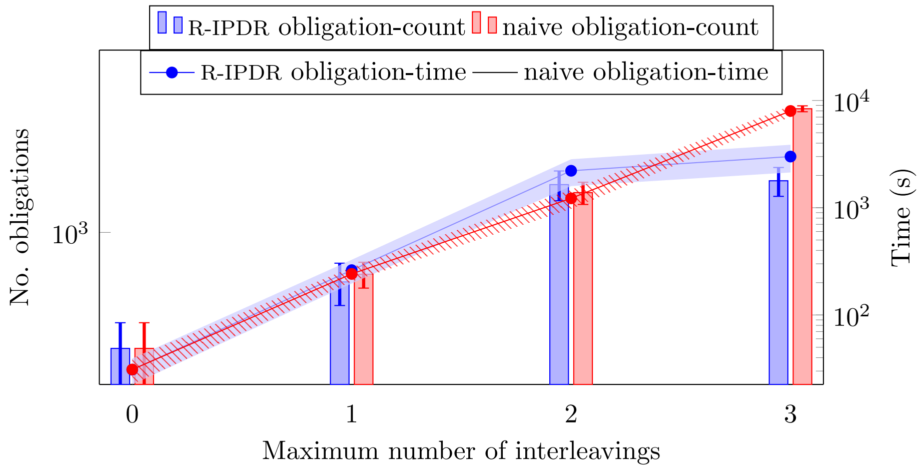


Figure 7.3: Statistics from R-IPDR *for Peter*(4) *(10 repetitions). Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time).*

4 Processes. Figure 7.3 shows a significant decrease in runtime when using R-IPDR, but only for 3 interleavings (the greatest constraint tested). For 1 and 2 interleavings, the average number of obligations is reduced, but there is a greater variance. Meanwhile, the runtime for 1 interleaving is slightly higher with R-IPDR and noticeably higher for 2 interleavings. This initially poor performance is compensated when relaxing to 3 interleavings: both the number of obligations and total runtime decrease significantly when using R-IPDR.

Figure A.5a and Figure A.6b show that the same holds true for the number of initial ctis and SAT-queries.

7.2.2 Saturation

Every tested input (Peterson's Algorithm for p processes) in Section 6.3 became easier to verify when using relaxing IPDR to incrementally increase the number of allowed interleavings. In each case, IPDR started showing the greatest improvements after 2 interleavings. This seems to indicate that the clauses learned during the instance with 2 interleavings contain valuable information for future instances, but the earlier instances do not. *Peter*(2) reaches a point of saturation at 4 interleavings, where PDR is initiated with all the information necessary to immediately terminate.

Just like *Peter*(2), *Peter*(3) and *Peter*(4) began showing its speedup at 3 interleavings. Because the latter two were tested for a lesser number of interleavings, it is unknown whether these would reach a similar point of saturation as *Peter*(2).

7.3 Reversible pebbling game

Section 6.4 showed that constraining IPDR consistently reduces the total runtimes of finding an optimal strategy for the Reversible Pebbling Game for the tested instances. Relaxing IPDR actually showed a consistent degradation. This section goes into more detail of the potential bad cases for IPDR.

7.3.1 Statistics

This section examines the statistics mentioned in Section 7.1. Some of the redundant or less relevant graphs for the discussion are fully displayed in Section A.2 and Section A.3.

Constraining ham7tc. Among the longest running inputs (100 seconds and longer), `ham7tc` had the greatest decrease in runtime of 56% when using C-IPDR over the naive implementation.

Figure 7.4 shows a consistent reduction in the number of obligations and corresponding runtime when using C-IPDR. This reduction is linear within the logarithmic graph and thus exponential in reality. Figure A.7a and Figure A.8b shows a corresponding reduction in the number of ctis and SAT-queries made.

Additionally, the table in Figure A.7a also reports a tighter standard deviation for C-IPDR than for the naive implementation, roughly in correspondence with the decrease in total runtime. Figure A.7b and this table show that while the time spent on the incremental steps in C-IPDR increases exponentially as the constraint is tightened, this is insignificant compared to the total runtime.

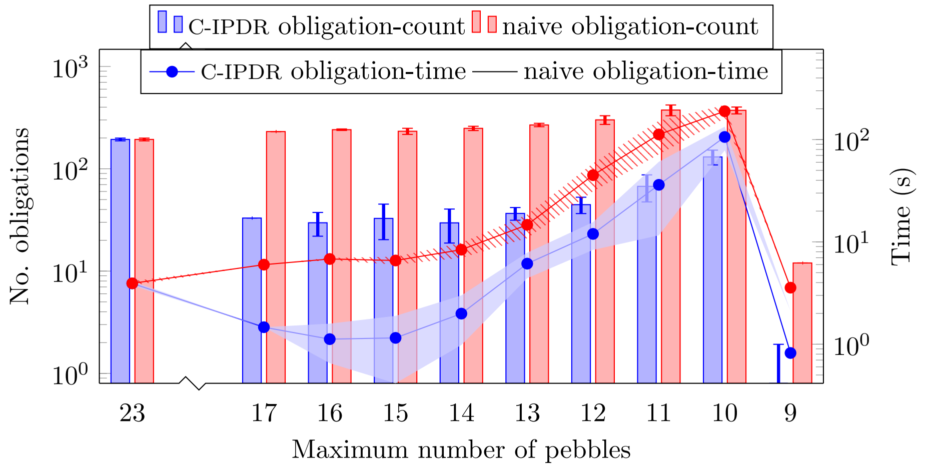


Figure 7.4: Statistics from C-IPDR for `ham7tc` (10 repetitions). Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time). On constraint 23, the algorithm found a trace with 18 pebbles, so it could continue with constraint 17 afterwards.

Constraining gf2^5mult_29_129. Of the three bad inputs for C-IPDR, the `gf2^5mult_29_129` circuit was the one with the greatest runtime. This should give more opportunity to observe the effects of IPDR.

Figure A.9a and Figure A.10a does show similar behaviour to `ham7tc`: where the number of ctis and *obligations* is consistently reduced when using C-IPDR. Only not with the same decrease in runtime.

Figure 7.5 reveals that there is not a corresponding trend in the work done by the SAT-solver. While the number of SAT-queries is initially reduced in C-IPDR, it increases starting at constraint 12 and stays greater than those made by the naive version until last iteration. This increase also corresponds to an increase in runtime, making C-IPDR slower overall.

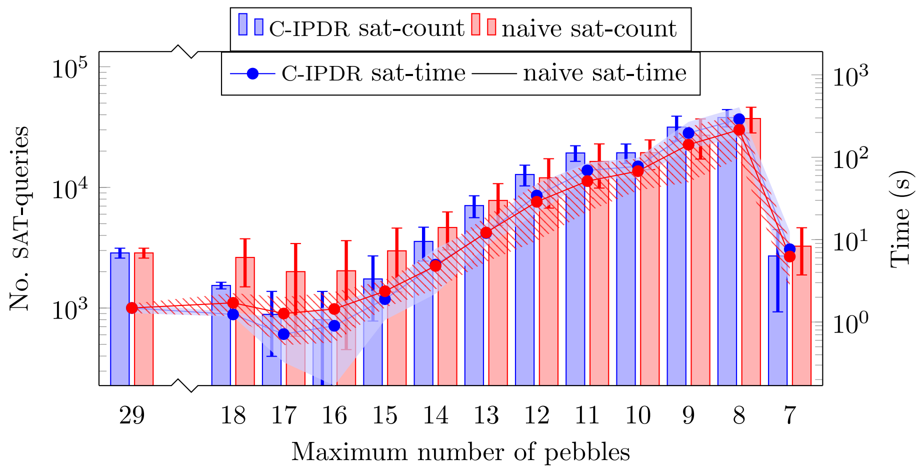


Figure 7.5: Statistics for the constraining `gf2^5mult_29_129` experiment (10 repetitions). Left axis: the total number of calls to the SAT-solver (sat-count). Right axis: the total time spent waiting on the SAT-solver (sat-time).

Relaxing 5bitadder. The `5bitadder` input is the one of the two larger inputs (100 seconds or more in R-IPDR) that still had a reduction in runtime when using R-IPDR.

In Figure A.11a, R-IPDR still needs to consistently considers fewer counterexamples when relaxing constraints. Only constraint 8 had a slight increase in runtime despite this.

Figure 7.6 shows a consistent decrease in the number of obligations, although to a lesser degree than typically found in C-IPDR. The figure also shows that the last two iterations need to handle considerably more obligations than the earlier two.

Figure A.12b shows that there is only a small gap between the amount of SAT-queries than there is between the obligations handled. Constraint 8 actually performs slightly more queries in R-IPDR, manifesting as a slightly higher runtime for that constraint.

Figure A.11b shows that at least 50% of cubes could be copied to a relaxed instance, while only a fraction of the total runtime is spent on doing so.

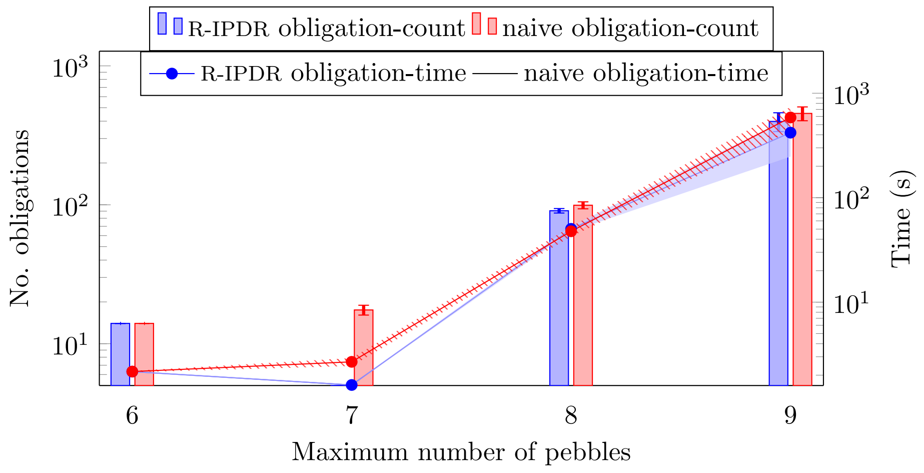


Figure 7.6: Statistics from R-IPDR for ***5bitadder*** *(10 repetitions). Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time).*

Relaxing ham7tc. The `ham7tc` input was one of the more promising results for C-IPDR: a decrease of 56%. For R-IPDR, there is a considerable increase in runtime instead.

Figure A.13a shows that the number of initial ctis are reduced, while runtime does not decrease. For the initial constraint 7, PDR does not survive initialisation and instantly solves the instance without considering a cti. From Figure A.13b, we see that most cubes are able to be copied over to constraints 9 and 10. For constraint 8, there were no obligations to copy in the first place.

Figure 7.7 shows that the vast majority of obligations are handled in constraint 10. Constraint 8 and 9 handle tens of obligations, while constraint 10 handles hundreds. It should be noted that this means that the total number of copied clauses to constraint 9 and 10 is relatively low, compared to the amount that will be learned in constraint 10. Meanwhile, in that largest iteration, R-IPDR has to fulfil more obligations than the naive implementation did.

Figure A.14b shows an increase in SAT-queries and SAT-time largely in correspondence with the increase in obligations performed for R-IPDR. The only exception is constraint 9, where there was a decrease in obligations, but R-IPDR performs more SAT-queries and takes more time doing so than the naive approach.

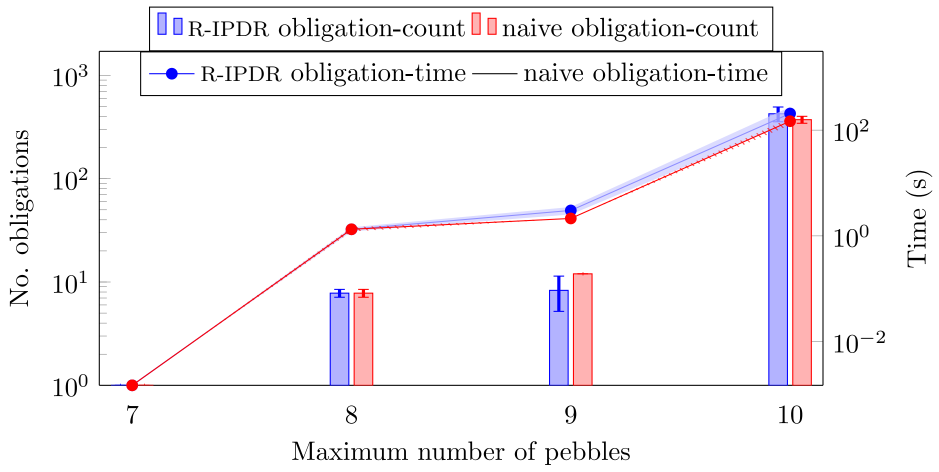


Figure 7.7: Statistics from R-IPDR for `ham7tc` (10 repetitions). Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time).

Relaxing hwb4tc. The `hwb4tc` input is a smaller one, but did have the greatest increase in runtime for R-IPDR relative to naive.

Figure A.15a and Figure 7.8 show an consistent decrease in the number of initial ctis and obligations. Except in the final iteration with constraint 10. That iteration is again the point where the bulk of the work is performed and handles roughly the same amount of obligations in R-IPDR as in the naive version.

Figure A.15b shows that most clauses could be copied to relaxing iterations. Although it must be noted that only the final iteration performed significant work, which could not be copied as R-IPDR terminated.

Figure A.16b shows that the amount of SAT-queries and the time spent on them remains virtually unchanged between the R-IPDR and naive versions. Only in the final iteration is there a noticeable difference, where R-IPDR performs worse.

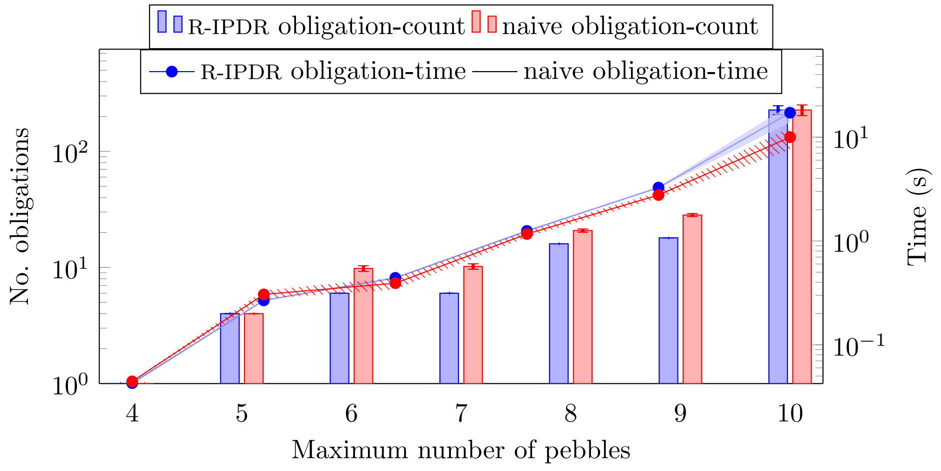


Figure 7.8: Statistics from R-IPDR for `hwb4tc` (10 repetitions). Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time). For constraint 4, no obligations were enqueued as PDR terminated in the initialization step. Its obligation-count value was set to 1 to accommodate the logarithmic scale and the obligation-time to the total runtime (which is a close approximation).

7.3.2 Workload distributions

The experiments with the Reversible Pebbling game in Section 6.4 showed that copying clauses with IPDR results in consistently shorter runtimes for constraining instances. Relaxing instances showed improvements for smaller inputs, but fell off for greater inputs (100 seconds or more). On the contrary, Peterson's Algorithm saw significant speedups when using relaxing IPDR. This seems to indicate that the effectiveness of IPDR and its relaxing and constraining variants may be dependent on the given problem.

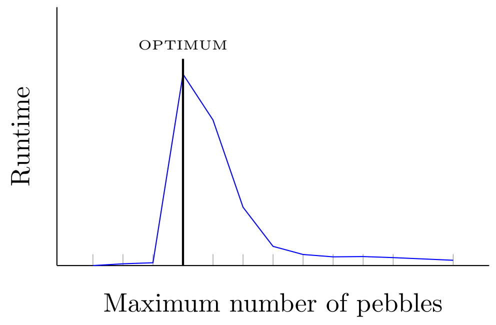
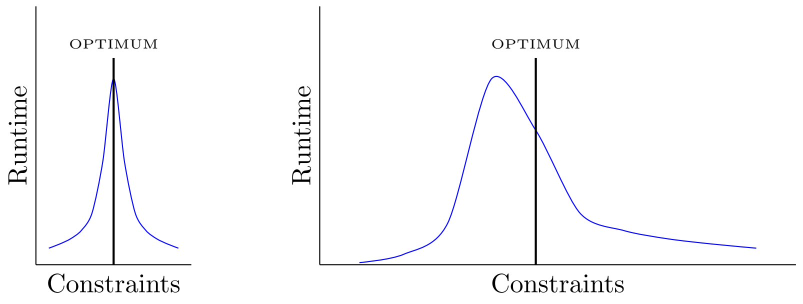


Figure 7.9: The (combined) runtimes of PDR *for different pebbling constraints with* `ham7tc`*. The data is the "naive cti-time" from Figure A.7a (constraint 11 through 23) and Figure A.13a (constraint 7 through 10).*



(a) Even workload. *(b) Uneven workload. Approaching the optimum from the relaxing side is much harder than the constraining side.*

Figure 7.10: Theoretical workload distributions for IPDR*.*

Closer examination of the pebbling inputs reveals a potential cause. The Pebbling Game has a upper and lower bound for the possible pebbling strategies: the number of nodes and output nodes respectively. The figures found at Section A.2 and Section A.3 show that the optimum pebbling strategy tends to lie closer to the lower bound than to the upper bound.

Figure 7.9 combines the total runtimes from both the relaxing and constraining statistics for `ham7tc` and shows which constraint was ultimately found as the optimum. The first of the few relaxing instances are trivial to solve compared to the others and instantly becomes very difficult when it approaches the optimum (a well-known phenomenon in SAT-solving [54, 55, 56]). In these early instances, there are fewer opportunities for PDR to learn clauses and thus for IPDR to carry those into the next instance. When the problems becomes difficult to solver, there are no clauses to benefit from and the clauses learned during that instance cannot be reused as it is the last or one of the last instances.

On the contrary, constraining instances are more numerous and increase in difficulty much more gradually. Thus, constraining PDR should have more opportunity to copy clauses and experimental results in Section 6.4.2 show a corresponding speedup. Despite these speedups, the low number of iterations when relaxing provides a lower total runtime than constraining with IPDR.

It then follows that the binary search algorithm depicts an average between the two: its total runtime is short than constraining IPDR due to the lower number of iterations required, it achieves greater speedups relative to the naive implementation than relaxing IPDR, but ultimately does not have a better runtime than relaxing IPDR.

In a practical sense, problems with an unbalanced workload distribution similar to Figure 7.9 seem to represent a bad case for IPDR. The low amount of relaxing iterations override the benefit of IPDR when constraining, while not providing a benefit when relaxing.

Two better cases would be those depicted in Figure 7.10. Either the relaxing and constraining workloads are balanced, making either option viable and providing a great opportunity for the binary search approach. Or, the side with the fewest iterations (relaxing in this example) is too hard when it approaches the optimum and approaching from the other (constraining) side is more viable. Peterson's algorithm actually represents an extreme case of the latter, where there is no optimum (or rather: it is ∞) and only the relaxing approach is possible at all.

8 Related work

IPDR is partially inspired by work on incremental SAT-solving to copy clauses between related instances of a SAT-solver [43] by Katalin Fazekas, Armin Biere and Christoph Scholl. IPDR firstly uses an incremental solver in its PDR algorithm, which benefits from this work, and uses a similar principle to copy information to new instances of PDR.

Dirk Beyer and Matthias Dangl created a baseline for research into PDR and its surrounding area [5]. They provide a PDR implementation to be used as reference and an experimental evaluation against adjacent tools.

CPAchecker (Configurable Program Analysis) [6] is an acclaimed software verification tool and framework by Dirk Beyer and M. Erkan Keremoglu. Building an infrastructure to evaluate the performance of verification algorithms is a considerable software engineering effort on top of the creation on new algorithms. The CPA formalism allows one to express program analyses and the corresponding CPAchecker tool allows the integration of any new verification algorithm that is accordingly expressed.

Alessandro Cimatti, Alberto Griggio, Sergio Mover and Stefano Tonetta [23] implemented a similar incremental PDR-based algorithm, extended to SMT formulae, to specifically solve the parameter synthesis problem. Their experiments show that their approach improves on similar SMT-based solutions.

Orna Grumberg, Flavio Lerda, Ofer Strichman and Michael Theobald present an efficient procedure to verify multi-processes systems [22]. Their procedure uses an increasing number of allowed interleavings, the same parameter used to relax Peterson's Algorithm in this thesis, to achieve a more aggressive partial-order-reduction.

The work of B. Bittner, M. Bozzano, A. Cimatti, M. Gario and A. Griggio in parameter synthesis that finds optimal optimal solutions in the Pareto-front [12] produces several algorithms based on PDR. One algorithm is also able to abuse monotonicity in the cost-functions to be optimised and incrementally reuses the invariants produced by PDR to refine the search-space.

Jianwen Li, Shufang Zhu, Yueling Zhang, Geguang Pu, Moshe Vardi presented an algorithm inspired by PDR: Complementary Approximate Reachability (CAR) [61]. Where PDR maintains a monotone over-approximation of the reachable states, CAR uses different techniques to maintain both an over- and under-approximation. Their experiments found CAR to be competitive with and complementary to PDR. The over-approximating nature of PDR allowed for easy copying of clauses with IPDR, but required explicit checking for relaxing IPDR. Maintaining over-approximations in both directions may allow IPDR to benefit the same way in both directions as well.

A competitor to PDR (and thus IPDR) is CEGAR, or Counterexample-guided Abstraction Refinement [27] by Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu and Helmut Veith. CEGAR begins with an abstraction of the model-in-question that may discover counterexamples that are not valid under the actual model. When discovered, these counterexamples are used to refine the abstraction and continue. Johannes Birgmeier, Aaron R. Bradley and Georg Weissenbacher in turn created CTIGAR [11], which combined concepts from both approaches to form a new competitive tool.

The Vienna Verification Tool [50] by Henning Günther, Alfons Laarman and Georg Weissenbacher implement CTIGAR [11] to completely verify parallel software by using a dynamic version of Lipton's reduction [51, 64]. They provide an open-source tool on the VVT website [49].

Aman Goel and Karem Sakallah extend IPDR to symmetric incremental induction [47], which proves safety properties for all possible instances of symmetric protocols. The algorithm produces quantified inductive invariants that hold for all of the protocol's parameters.

The Reversible Pebbling Game, used in this thesis as a reference (inspired by Giulia Meuli et al. [68]), is often used to study space/time trade-offs in quantum computing. The Spooky Pebbling Game is a variant used to study also study trade-offs between classical- and quantum-space/trade-offs. Arend-Jan Quist and Alfons Laarman [77] designed and implemented a solver for the Spooky Pebbling Game based on SAT-solving. Sebastiaan Brand, Tim Coopmans and Alfons Laarman give an efficient CNF encoding for quantum logic gates [19] that they use to synthesise target graph states. Their CNF encoding allows them to use SAT-based methods, namely BMC, in their implementation.

9 Conclusions

We formulated constraining and relaxing systems, a group of monotonically related systems, and introduced the IPDR algorithm which uses similarities between such systems to copy information from one instance of the PDR algorithm to the next. Our experiments in Section 6 that copying clauses to a constraining or relaxing instance consistently reduces the proof-obligations handled by PDR, compared to when solely performing separate PDR runs, which often results in a reduced runtime when solving a problem that can be formulated as a series of constraining or relaxing systems. This series need not be solely constraining of relaxing, IPDR can switch between the two at will without additional overhead costs.

The PDR implementation used for IPDR is not fully optimised. It does include various features from related work [17, 39, 52] to enhance performance, but further optimisations fell outside the scope of the thesis. As a result, the PDR implementation is not competitive with the SPACER engine [36, 57, 59] that is used as a reference. In smaller experiments, IPDR seems to compensate for this gap in performance, but this falls off for the larger inputs. Thus, this thesis does show the positive effects IPDR, but does not necessarily provide a model checker capable of defeating the state-of-the-art out of the box.

The extent of the benefits of IPDR seem problem-dependent to a degree. Experiments show a difference in how the two different problems benefit from IPDR, although both show improvements over performing a non-IPDR incremental algorithm. Verifying Peterson's Algorithm consistently improves total runtimes when incrementing using IPDR over the naively incrementing algorithm and can even outperform a conventional, singular PDR model checking approach. Constraining the Reversible Pebbling Game gives consistent speedups, but relaxing does not. Section 7.3.2 discusses how pebbling is a potential bad case for IPDR, specifically relaxing as it provides little opportunities to copy clauses. The binary search implementation enjoys the benefits of constraining, but suffers the problems of relaxing. For now, the binary search algorithm illustrates a bi-directional potential of IPDR, but does not provide consistently better approach.

Overall, IPDR seems a worthwhile addition to PDR. Copying clauses is an optimisation on the table that can be exploited with relatively little computational cost. But additional input problems will need to be explored and PDR should be integrated into state-of-the-art tolls before it can truly show its merits.

10 Future work

An important step to further IPDR will be optimising the PDR implementation [13] underlying it or integrating IPDR within existing state-of-the-art tools. Much work has been done to adapt or improve IPDR within the Model Checking community, and still is. It fell outside of the scope of this thesis to exhaustively implement such improvements while enabling the functions of IPDR. This work is not in competition with such efforts, but is potentially one of these many to tune or enhance PDR in certain use-cases.

The Reversible Pebbling Problem encountered a single type of bad input when constraining. The `gf2^3mult_11_47`, `gf2^4mult_19_83` and `gf2^5mult_29_129` inputs, which compute the same Galois Field Multiplication function, showed an atypical bad performance when constraining. We have not uncovered a specific cause for this. Uncovering which type of inputs are such a particularly bad case will be important to fully explore IPDR.

The two problems studied in Section 6 show that the effectiveness, and which type of IPDR is effective, is problem dependent. Section 7.3.2 theorises that the curve describing runtime of the Reversible Pebbling Problem for each constraint is ill-suited to IPDR. It also suggests two curves that could potentially be better cases for IPDR. Finding and exploring problems that adhere to such a curve could potentially show an especially good niche for IPDR to fill. Specifically, problems with a more symmetrical curve could be more suited to a binary search algorithm. Finding such a problem and testing this algorithm may reveal the practicality of the bi-directional capabilities of IPDR, which is at the moment only a nifty feature.

References

[1] Aaron Bradley (arbrad). *IC3Ref*. Accessed: 2022-12-30. URL: https://github.com/arbrad/IC3ref/blob/master/IC3.cpp.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN: 026202649X.

[3] Ramón Béjar, César Fernàndez, and Francesc Guitart. "Encoding Basic Arithmetic Operations for SAT-Solvers". In: vol. 220. Jan. 2010.

[4] Charles H. Bennett. "Time/Space Trade-Offs for Reversible Computation". In: *SIAM Journal on Computing* 18.4 (1989), pp. 766–776. DOI: 10.1137/0218053.

[5] Dirk Beyer and Matthias Dangl. "Software Verification with PDR: An Implementation of the State of the Art". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Armin Biere and David Parker. Cham: Springer International Publishing, 2020, pp. 3–21. ISBN: 978-3-030-45190-5. DOI: 10.1007/978-3-030-45190-5_1.

[6] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 184–190. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_16.

[7] Armin Biere and Toni Jussila, eds. *Hard-ware model checking competition 2007 (HWMCC07)*. Vol. 10867. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2007.

[8] Armin Biere et al. "Bounded model checking". In: *Handbook of satisfiability* 185.99 (2009), pp. 457–481.

[9] Armin Biere et al., eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. ISBN: 978-1-58603-929-5.

[10] Armin Biere et al. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by W. Rance Cleaveland. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207. ISBN: 978-3-540-49059-3.

[11] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. "Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR)". In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 831–848. ISBN: 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_55.

[12] B. Bittner et al. "Towards Pareto-optimal parameter synthesis for monotonic cost functions". In: *2014 Formal Methods in Computer-Aided Design (FMCAD)*. 2014, pp. 23–30. DOI: 10.1109/FMCAD.2014.6987591.

[13] Max Blankestijn. *Incremental PDR*. Accessed: 2023-09-08. URL: https://https://github.com/Majeux/ipdr.

[14] Max Blankestijn and Alfons Laarman. "Incremental Property Directed Reachability". In: *Formal Methods and Software Engineering*. Cham: Springer International Publishing (accepted for publication), 2023.

[15] Max Blankestijn and Alfons Laarman. *Incremental Property Directed Reachability*. 2023. arXiv: 2308.12162 [cs.SC].

[16] George Boole. *An investigation of the laws of thought on which are founded the mathematical theories of logic and probabilities*. London, Walton and Maberly, 1854, p. 446. DOI: 10.5962/bhl.title.29413.

[17] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87. ISBN: 978-3-642-18275-4. DOI: 10.1007/978-3-642-18275-4_7.

[18] Aaron R. Bradley and Zohar Manna. "Checking Safety by Inductive Generalization of Counterexamples to Induction". In: *Proceedings of the Formal Methods in Computer Aided Design*. FMCAD '07. USA: IEEE Computer Society, 2007, pp. 173–180. ISBN: 0769530230. DOI: 10.1109/FAMCAD.2007.15.

[19] Sebastiaan Brand, Tim Coopmans, and Alfons Laarman. *Quantum Graph-State Synthesis with SAT*. 2023. arXiv: 2309.03593 [quant-ph].

[20] Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[21] J.R. Burch et al. "Symbolic model checking: 10/sup 20/ states and beyond". In: *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*. 1990, pp. 428–439. DOI: 10.1109/LICS.1990.113767.

[22] Prantik Chatterjee et al. "Proof-Guided Underapproximation Widening for Bounded Model Checking". In: *Computer Aided Verification*. Ed. by Sharon Shoham and Yakir Vizel. Cham: Springer International Publishing, 2022, pp. 304–324. ISBN: 978-3-031-13185-1. DOI: 10.1007/978-3-031-13185-1_15.

[23] Alessandro Cimatti et al. "Parameter synthesis with IC3". In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 165–168. DOI: 10.1109/FMCAD.2013.6679406.

[24] E.M. Clarke et al. *Handbook of model checking*. May 2018, pp. 1–1210. DOI: 10.1007/978-3-319-10575-8.

[25] Edmund Clarke et al. "Completeness and Complexity of Bounded Model Checking". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Bernhard Steffen and Giorgio Levi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 85–96. ISBN: 978-3-540-24622-0. DOI: 10.1007/978-3-540-24622-0_9.

[26] Edmund Clarke et al. "Computational challenges in bounded model checking". In: *International Journal on Software Tools for Technology Transfer* 7.2 (Apr. 2005), pp. 174–183. ISSN: 1433-2787. DOI: 10.1007/s10009-004-0182-5.

[27] Edmund Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-45047-4. DOI: 10.1007/10722167_15.

[28] Edmund M. Clarke and E. Allen Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Logics of Programs*. Ed. by Dexter Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71. ISBN: 978-3-540-39047-3. DOI: 10.1007/BFb0025774.

[29] Edmund M. Clarke et al. “Model Checking and the State Explosion Problem”. In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30. ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1.

[30] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. Ed. by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman. ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047.

[31] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047.

[32] O. Coudert and J.C. Madre. “A unified framework for the formal verification of sequential circuits”. In: *1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*. 1990, pp. 126–129. DOI: 10.1109/ICCAD.1990.129859.

[33] Martin Davis, George Logemann, and Donald Loveland. “A Machine Program for Theorem-Proving”. In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557.

[34] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory”. In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034.

[35] J.W. de Bakker and L.G.L.T. Meertens. “On the completeness of the inductive assertion method”. In: *Journal of Computer and System Sciences* 11.3 (1975), pp. 323–357. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(75)80056-0.

[36] Leonardo De Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS’08/ETAPS’08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3540787992.

[37] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. “A Scalable Algorithm for Minimal Unsatisfiable Core Extraction”. In: *Theory and Applications of Satisfiability Testing - SAT 2006*. Ed. by Armin Biere and Carla P. Gomes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 36–41. ISBN: 978-3-540-37207-3. DOI: 10.1007/11814948_5.

[38] E. W. Dijkstra. "Solution of a Problem in Concurrent Programming Control". In: *Commun. ACM* 8.9 (Sept. 1965), p. 569. ISSN: 0001-0782. DOI: 10.1145/365559.365617.

[39] Niklas Een, Alan Mishchenko, and Robert Brayton. "Efficient Implementation of Property Directed Reachability". In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD '11. Austin, Texas: FMCAD Inc, 2011, pp. 125–134. ISBN: 9780983567813.

[40] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518. ISBN: 978-3-540-24605-3. DOI: 10.1007/978-3-540-24605-3_37.

[41] Niklas Eén and Niklas Sörensson. "Temporal Induction by Incremental SAT Solving". In: *Electronic Notes in Theoretical Computer Science* 89.4 (2003). BMC'2003, First International Workshop on Bounded Model Checking, pp. 543–560. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(05)82542-3.

[42] Eitan Farchi, Yarden Nir, and Shmuel Ur. "Concurrent Bug Patterns and How to Test Them". In: *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IPDPS '03. USA: IEEE Computer Society, 2003, p. 286.2. ISBN: 0769519261.

[43] Katalin Fazekas, Armin Biere, and Christoph Scholl. "Incremental Inprocessing in SAT Solving". In: *Theory and Applications of Satisfiability Testing – SAT 2019*. Ed. by Mikoláš Janota and Inês Lynce. Cham: Springer International Publishing, 2019, pp. 136–154. ISBN: 978-3-030-24258-9. DOI: 10.1007/978-3-030-24258-9_9.

[44] Grigory Fedyukovich et al. "Solving Constrained Horn Clauses Using Syntax and Data". In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. 2018, pp. 1–9. DOI: 10.23919/FMCAD.2018.8603011.

[45] Robert W. Floyd. "Assigning Meanings to Programs". In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. Dordrecht: Springer Netherlands, 1993, pp. 65–81. ISBN: 978-94-011-1793-7. DOI: 10.1007/978-94-011-1793-7_4.

[46] Roman Gershman, Maya Koifman, and Ofer Strichman. "Deriving Small Unsatisfiable Cores with Dominators". In: *Proceedings of the 18th International Conference on Computer Aided Verification*. CAV'06. Seattle, WA: Springer-Verlag, 2006, pp. 109–122. ISBN: 354037406X. DOI: 10.1007/11817963_13.

[47] Aman Goel and Karem Sakallah. "On Symmetry and Quantification: A New Approach to Verify Distributed Protocols". In: *NASA Formal Methods: 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 131–150. ISBN: 978-3-030-76383-1. DOI: 10.1007/978-3-030-76384-8_9.

[48] E. Pascal Gribomont. "Atomicity refinement and trace reduction theorems". In: *Computer Aided Verification*. Ed. by Rajeev Alur and Thomas

A. Henzinger. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 311–322. ISBN: 978-3-540-68599-9. DOI: 10.1007/3-540-61474-5_79.

[49] Henning Günther. *The Vienna Verification Tool website*. http://vvt.forsyte.at/. Last accessed: 2019-05-29.

[50] Henning Günther, Alfons Laarman, and Georg Weissenbacher. "Vienna Verification Tool: IC3 for Parallel Software". In: vol. 9636. 2016.

[51] Henning Günther et al. "Dynamic reductions for model checking concurrent software". In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer. 2017, pp. 246–265.

[52] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. "Better generalization in IC3". In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 157–164. DOI: 10.1109/FMCAD.2013.6679405.

[53] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123973375. DOI: 10.5555/2385452.

[54] Marijn Heule. "Schur Number Five". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (Apr. 2018). DOI: 10.1609/aaai.v32i1.12209.

[55] Marijn J. H. Heule and Oliver Kullmann. "The Science of Brute Force". In: *Commun. ACM* 60.8 (July 2017), pp. 70–79. ISSN: 0001-0782. DOI: 10.1145/3107239.

[56] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. "Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer". In: *Theory and Applications of Satisfiability Testing – SAT 2016*. Ed. by Nadia Creignou and Daniel Le Berre. Cham: Springer International Publishing, 2016, pp. 228–245. ISBN: 978-3-319-40970-2. DOI: 10.1007/978-3-319-40970-2_15.

[57] Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura. "μZ– An Efficient Engine for Fixed Points with Constraints". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 457–462. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_36.

[58] Jinbo Huang. "MUP: A Minimal Unsatisfiability Prover". In: *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*. ASP-DAC '05. Shanghai, China: Association for Computing Machinery, 2005, pp. 432–437. ISBN: 0780387376. DOI: 10.1145/1120725.1120907.

[59] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. "SMT-Based Model Checking for Recursive Programs". In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 17–34. ISBN: 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_2.

[60] Leonid A. Levin. "Universal Sequential Search Problems". In: *Problems of Information Transmission* 9.3 (1973).

[61] Jianwen Li et al. "Safety Model Checking with Complementary Approximations". In: *Proceedings of the 36th International Conference on Computer-*

Aided Design. ICCAD '17. Irvine, California: IEEE Press, 2017, pp. 95–100.

[62] Mark H. Liffiton and Karem A. Sakallah. "On Finding All Minimally Unsatisfiable Subformulas". In: *Theory and Applications of Satisfiability Testing*. Ed. by Fahiem Bacchus and Toby Walsh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 173–186. ISBN: 978-3-540-31679-4. DOI: 10.1007/11499107_13.

[63] Andrzej Lingas. "A PSPACE complete problem related to a pebble game". In: *Automata, Languages and Programming*. Ed. by Giorgio Ausiello and Corrado Böhm. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 300–321. ISBN: 978-3-540-35807-7.

[64] Richard J. Lipton. "Reduction: A Method of Proving Properties of Parallel Programs". In: *Commun. ACM* 18.12 (Dec. 1975), pp. 717–721. ISSN: 0001-0782. DOI: 10.1145/361227.361234.

[65] Inês Lynce and João Silva. "On Computing Minimum Unsatisfiable Cores". In: May 2004.

[66] Dmitri Maslov. *Reversible Logic Synthesis Benchmarks Page*. https://reversiblebenchmarks.github.io/. Accessed: 2021-07-24.

[67] Kenneth Lauchlin McMillan. "Symbolic Model Checking: An Approach to the State Explosion Problem". UMI Order No. GAX92-24209. PhD thesis. USA, 1992.

[68] Giulia Meuli et al. *Reversible Pebbling Game for Quantum Memory Management*. 2019. arXiv: 1904.02121 [quant-ph].

[69] Maher Mneimneh et al. "A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas". In: *Theory and Applications of Satisfiability Testing*. Ed. by Fahiem Bacchus and Toby Walsh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 467–474. ISBN: 978-3-540-31679-4. DOI: 10.1007/11499107_40.

[70] Matthew W. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of the 38th Annual Design Automation Conference*. DAC '01. Las Vegas, Nevada, USA: Association for Computing Machinery, 2001, pp. 530–535. ISBN: 1581132972. DOI: 10.1145/378239.379017.

[71] Alexander Nadel and Vadim Ryvchin. "Efficient SAT Solving under Assumptions". In: *Theory and Applications of Satisfiability Testing – SAT 2012*. Ed. by Alessandro Cimatti and Roberto Sebastiani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 242–255. ISBN: 978-3-642-31612-8. DOI: 10.1007/978-3-642-31612-8_19.

[72] Yoonna Oh et al. "AMUSE: A Minimally-Unsatisfiable Subformula Extractor". In: *Proceedings of the 41st Annual Design Automation Conference*. DAC '04. San Diego, CA, USA: Association for Computing Machinery, 2004, pp. 518–523. ISBN: 1581138288. DOI: 10.1145/996566.996710.

[73] G.L. Peterson. "Myths about the mutual exclusion problem". In: *Information Processing Letters* 12.3 (1981), pp. 115–116. ISSN: 0020-0190. DOI: 10.1016/0020-0190(81)90106-X.

[74] David A. Plaisted and Steven Greenbaum. “A Structure-preserving Clause Form Translation”. In: *Journal of Symbolic Computation* 2.3 (1986), pp. 293–304. ISSN: 0747-7171. DOI: 10.1016/S0747-7171(86)80028-1.

[75] Amir Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[76] J. P. Queille and J. Sifakis. “Specification and verification of concurrent systems in CESAR”. In: *International Symposium on Programming*. Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 337–351. ISBN: 978-3-540-39184-5.

[77] Arend-Jan Quist and Alfons Laarman. “Optimizing Quantum Space Using Spooky Pebble Games”. In: *International Conference on Reversible Computation*. Ed. by Martin Kutrib and Uwe Meyer. Cham: Springer Nature Switzerland, 2023, pp. 134–149. ISBN: 978-3-031-38100-3. DOI: 10.1007/978-3-031-38100-3_10.

[78] Ishai Rabinovitz and Orna Grumberg. “Bounded Model Checking of Concurrent Programs”. In: *Computer Aided Verification*. Ed. by Kousha Etessami and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 82–97. ISBN: 978-3-540-31686-2. DOI: 10.1007/11513988_9.

[79] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Global value numbers and redundant computations”. In: *ACM-SIGACT Symposium on Principles of Programming Languages*. 1988. DOI: 10.1145/73560.73562.

[80] Ofer Shtrichman. “Pruning Techniques for the SAT-Based Bounded Model Checking Problem”. In: *Correct Hardware Design and Verification Methods*. Ed. by Tiziana Margaria and Tom Melham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 58–70. ISBN: 978-3-540-44798-6. DOI: 10.1007/3-540-44798-9_4.

[81] G. S. Tseitin. “On the Complexity of Derivation in Propositional Calculus”. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28.

[82] John Venn. “On the Diagrammatic and Mechanical Representation of Propositions and Reasonings”. In: *Philosophical Magazine* 9.59 (1880), pp. 1–18.

[83] Siert Wieringa. “On Incremental Satisfiability and Bounded Model Checking”. In: *CEUR Workshop Proceedings* 832 (Jan. 2011), pp. 13–21.

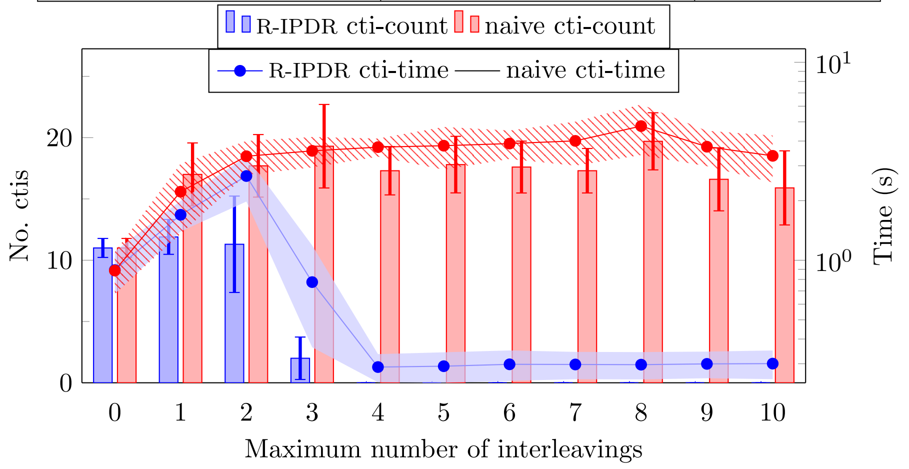
[84] Jianmin Zhang, Sikun Li, and Shengyu Shen. “Extracting Minimum Unsatisfiable Cores with a Greedy Genetic Algorithm”. In: *Proceedings of the 19th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*. AI’06. Hobart, Australia: Springer-Verlag, 2006, pp. 847–856. ISBN: 3540497870. DOI: 10.1007/11941439_89.

A Statistics

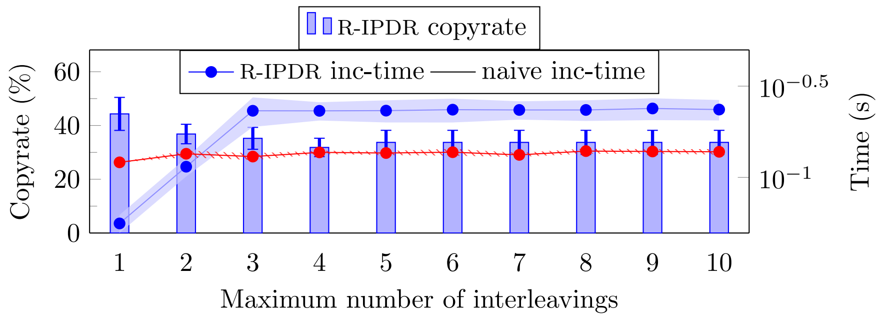
The figures in this section list the statistics, described in Section 7.1, that were collected during the experiments from Section 6.3 and Section 6.4. They include the figures displayed in Section 7.3 and Section 7.2, in addition to other statistics that were less relevant to the discussion or redundant.

A.1 Relaxing Peterson statistics

	R-IPDR	Naive R-IPDR	Improvement
avg. time	10.144 s	38.654 s	73.76 %
std. dev. time	1.371 s	2.729 s	
avg. incremental time	2.043 s		
std. dev. incremental time	0.252 s		
mutual exclusion holds	yes	yes	

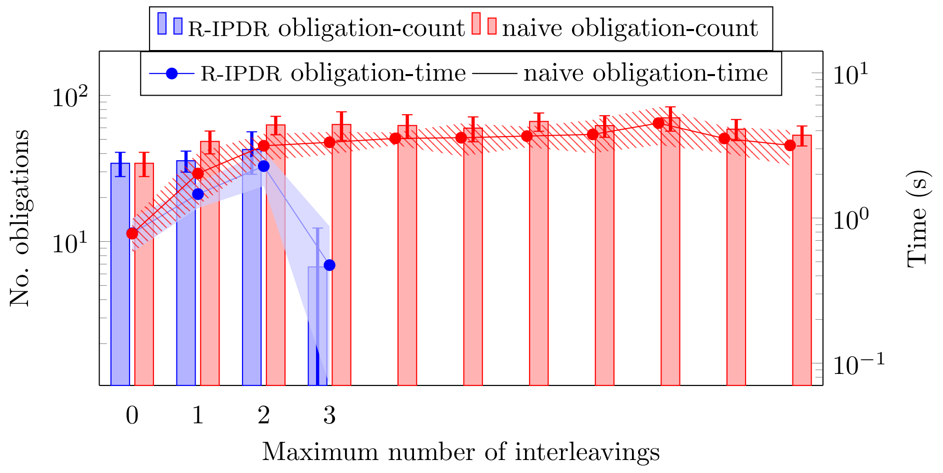


(a) Left axis: the number of counterexamples found in the major iteration (cti-count). Right axis: the total runtime (cti-time).

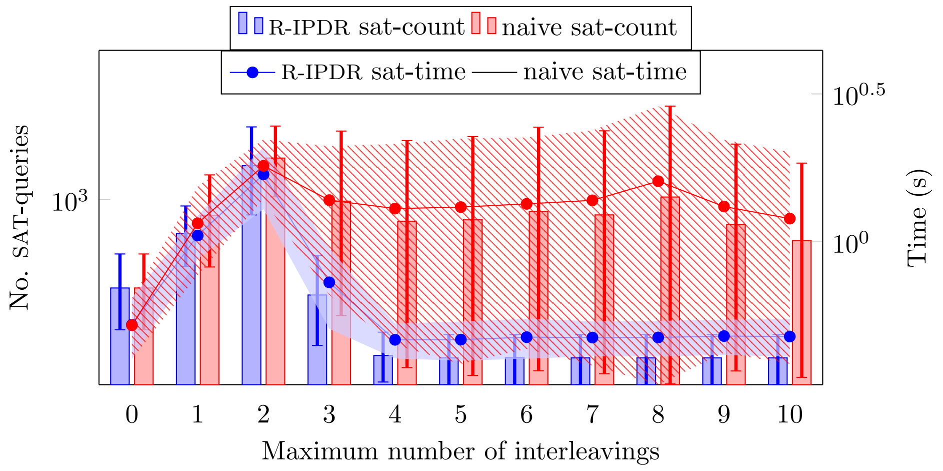


(b) Left axis: the percentage of clauses that could be copied to the next instance (listed on the x-axis). Right axis: the runtime was spent on copying clauses to the next instance. This number is already included in the times of the upper graph. Due to the bug mentioned in Section 7.1, the copyrates starting at 4 interleavings are likely under-reported.

Figure A.1: Statistics for the relaxing Peter(2) experiment, averaged over 10 repetitions.



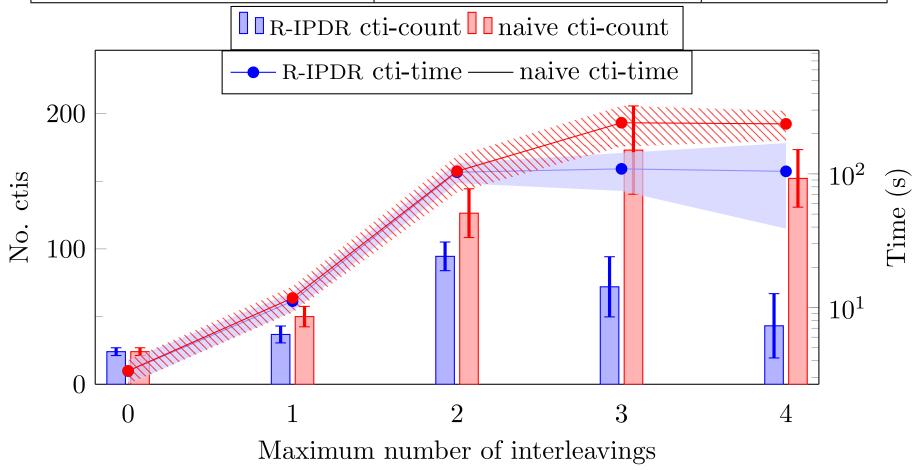
(a) Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time).



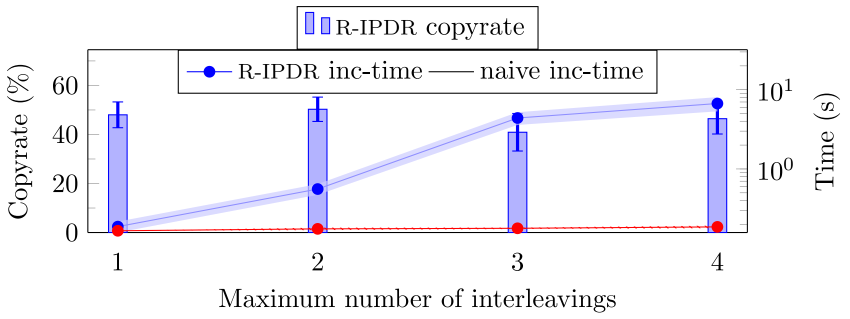
(b) Left axis: the total number of calls to the SAT-solver (sat-count). Right axis: the total time spent waiting on the SAT-solver (sat-time).

Figure A.2: Statistics for the relaxing Peter(2) experiment, averaged over 10 repetitions.

	R-IPDR	Naive R-IPDR	Improvement
avg. time	342.747 s	598.523 s	42.73 %
std. dev. time	78.325 s	100.434 s	
avg. incremental time	11.836 s		
std. dev. incremental time	1.916 s		
mutual exclusion holds	yes	yes	

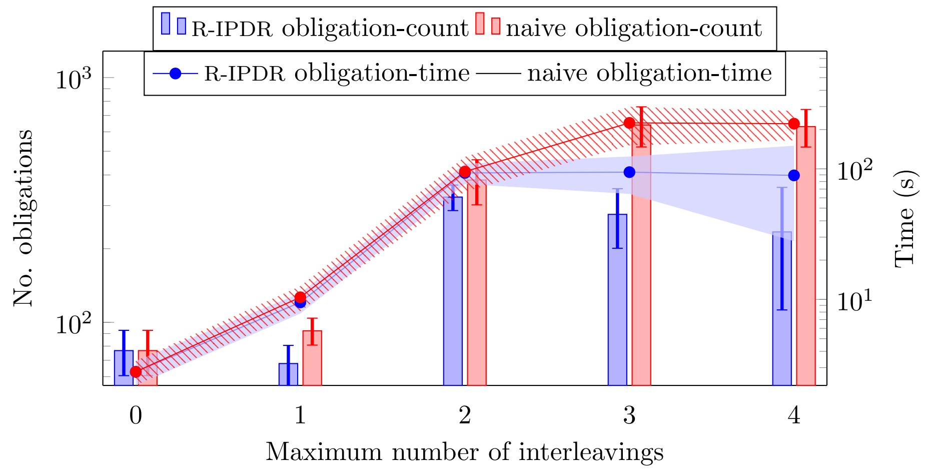


(a) Left axis: the number of counterexamples found in the major iteration (cti-count). Right axis: the total runtime (cti-time).

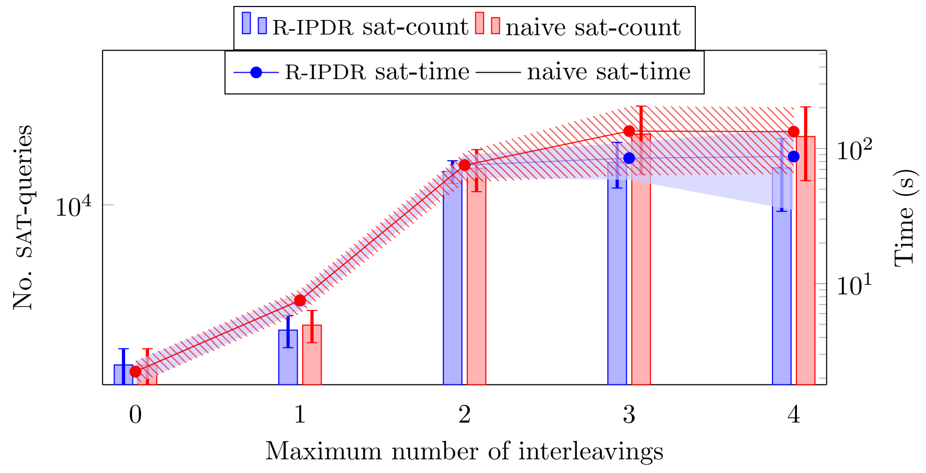


(b) Left axis: the percentage of clauses that could be copied to the next instance (listed on the x-axis). Right axis: the runtime was spent on copying clauses to the next instance. This number is already included in the times of the upper graph.

Figure A.3: Statistics for the relaxing Peter(3) experiment, averaged over 10 repetitions.



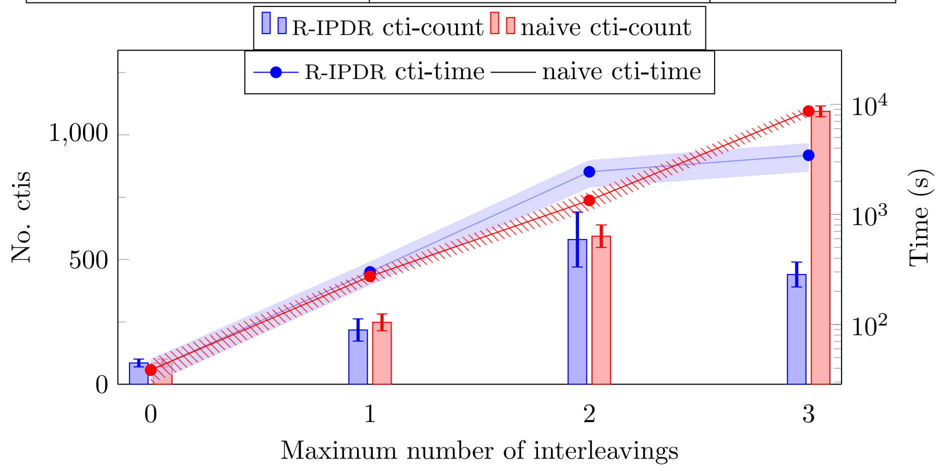
(a) Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time).



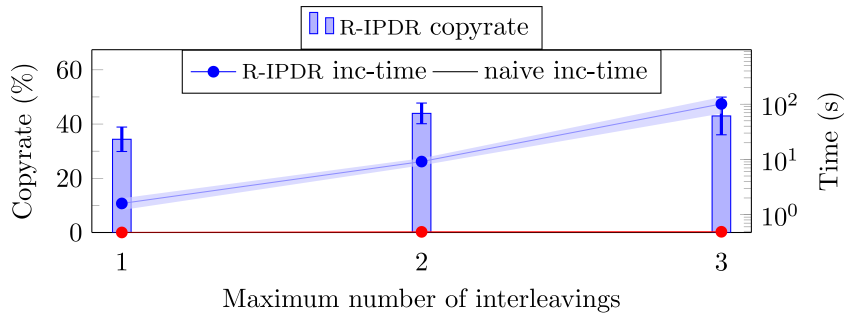
(b) Left axis: the total number of calls to the SAT-solver (sat-count). Right axis: the total time spent waiting on the SAT-solver (sat-time).

Figure A.4: Statistics of the relaxing Peter(3) experiment, averaged over 10 repetitions.

	R-IPDR	Naive R-IPDR	Improvement
avg. time	6328.118 s	10321.160 s	38.69 %
std. dev. time	1788.829 s	1279.712 s	
avg. incremental time	111.919 s		
std. dev. incremental time	34.766 s		
mutual exclusion holds	yes	yes	

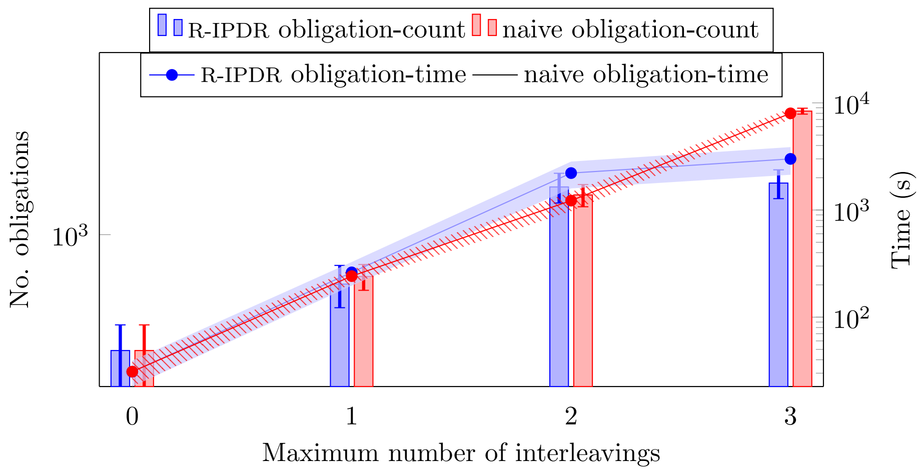


(a) Left axis: the number of counterexamples found in the major iteration (cti-count). Right axis: the total runtime (cti-time).

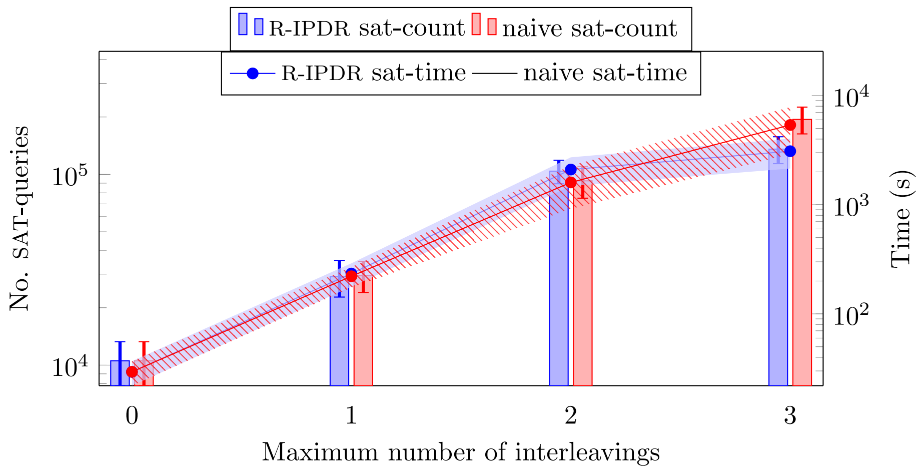


(b) Left axis: the percentage of clauses that could be copied to the next instance (listed on the x-axis). Right axis: the runtime was spent on copying clauses to the next instance. This number is already included in the times of the upper graph.

Figure A.5: Statistics for the relaxing Peter(4) experiment, averaged over 3 repetitions.



(a) Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time).

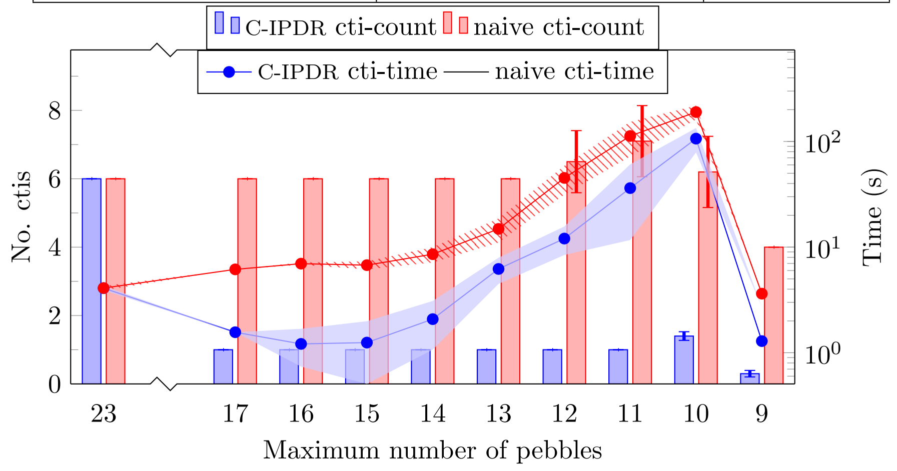


(b) Left axis: the total number of calls to the SAT-solver (sat-count). Right axis: the total time spent waiting on the SAT-solver (sat-time).

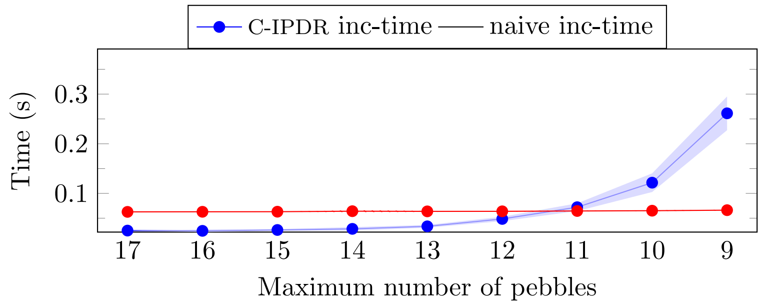
Figure A.6: Statistics for the relaxing Peter(4) experiment, averaged over 3 repetitions.

A.2 Constraining Pebbling statistics

	C-IPDR	Naive C-IPDR	Improvement
avg. time	170.244 s	386.719 s	55.98 %
std. dev. time	33.414 s	62.920 s	
avg. incremental time	0.594 s		
std. dev. incremental time	0.044 s		
max. invariant: constraint	9	9	
min. $F_1 \dots F_k$: length	18	9	0.00 %
min. strategy: pebbled	10	10	
min. strategy: length	25	26	3.85 %

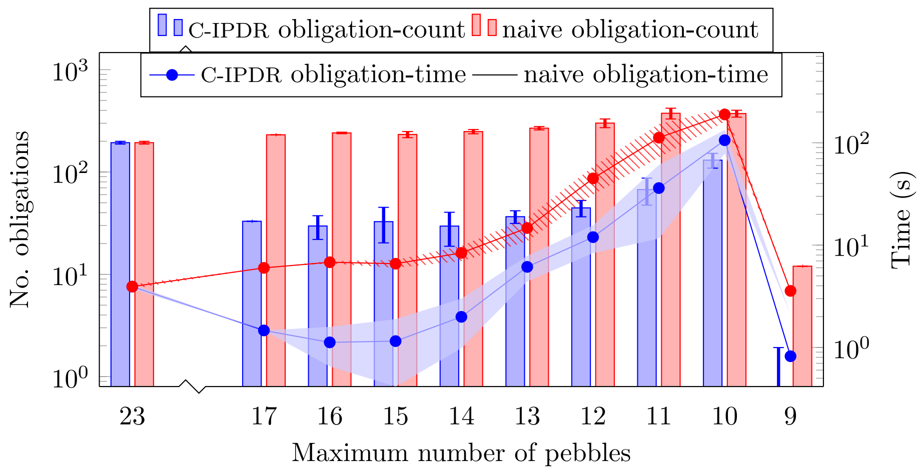


(a) Left axis: the number of counterexamples found in the major iteration (cti-count). Right axis: the total runtime (cti-time).

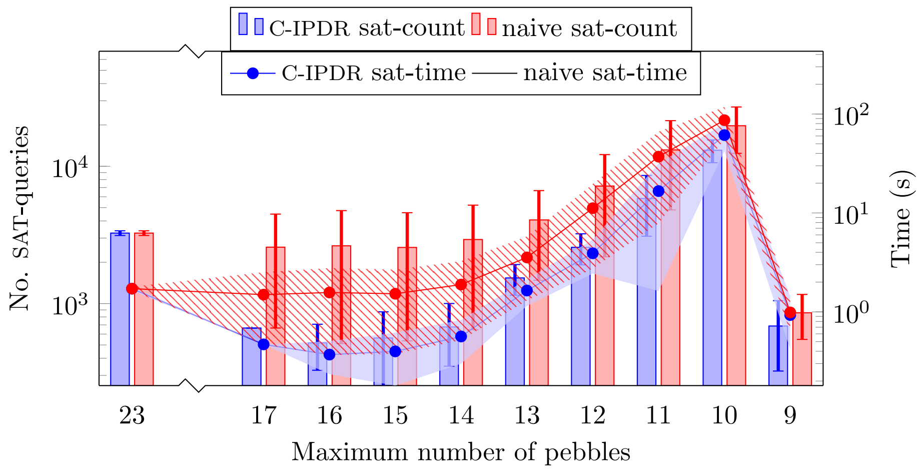


(b) The runtime spent on copying clauses to the next instance (listed on the x-axis). This number is already included in the runtimes of the upper graph.

Figure A.7: Statistics for the constraining `ham7tc` experiment, averaged over 10 repetitions. On constraint 23, the algorithm found a trace with 18 pebbles, so it could continue with constraint 17 afterwards.



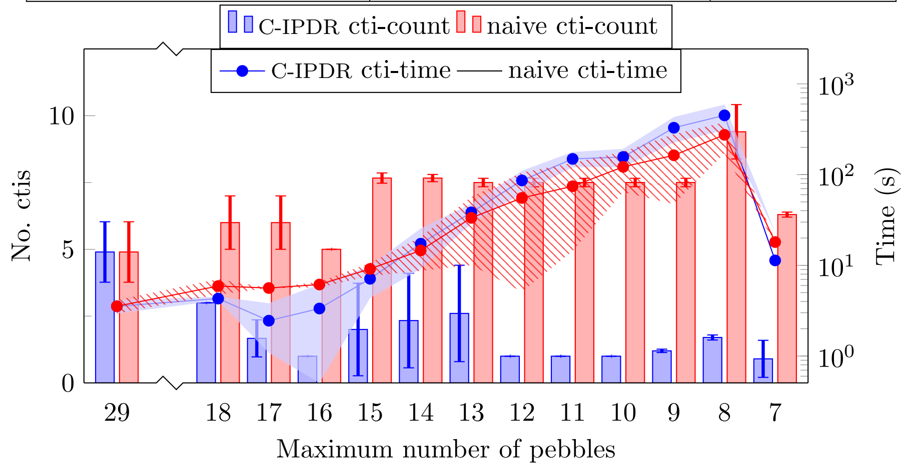
(a) Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time).



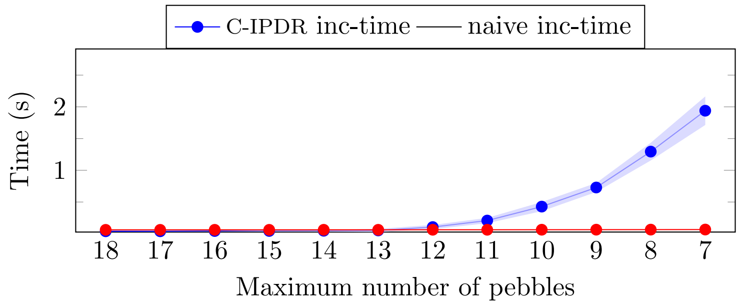
(b) Left axis: the total number of calls to the SAT-solver (sat-count). Right axis: the total time spent waiting on the SAT-solver (sat-time).

Figure A.8: Statistics for the constraining `ham7tc` *experiment, averaged over 10 repetitions. On constraint 23, the algorithm found a trace with 18 pebbles, so it could continue with 17 afterwards.*

	C-IPDR	Naive C-IPDR	Improvement
avg. time	1243.961 s	760.006 s	-63.68 %
std. dev. time	148.944 s	351.693 s	
avg. incremental time	4.834 s		
std. dev. incremental time	0.552 s		
max. invariant: constraint	7	7	
min. F length	19	7	0.00 %
min. strategy: pebbled	8	8	
min. strategy: length	47	49	4.08 %

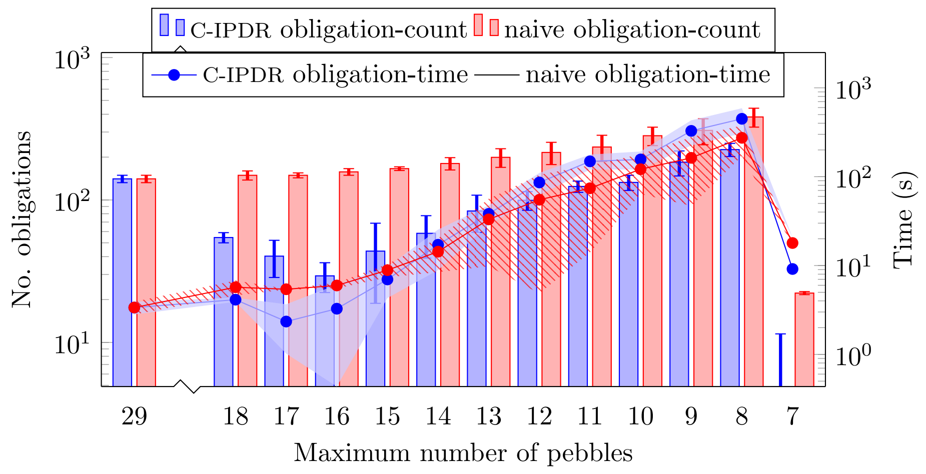


(a) Left axis: the number of counterexamples found in the major iteration (cti-count). Right axis: the total runtime (cti-time).

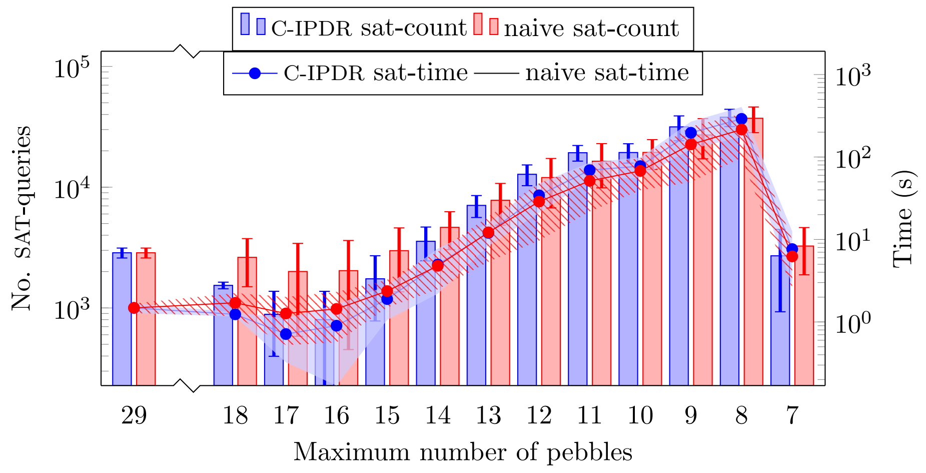


(b) The runtime spent on copying clauses to the next instance (listed on the x-axis). This number is already included in the runtimes of the upper graph.

Figure A.9: Statistics for the constraining `gf2^5mult_29_129` experiment, averaged over each repetition. On constraint 29, the algorithm found a trace with 19 pebbles, so it could continue with constraint 18 afterwards.



(a) Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time).

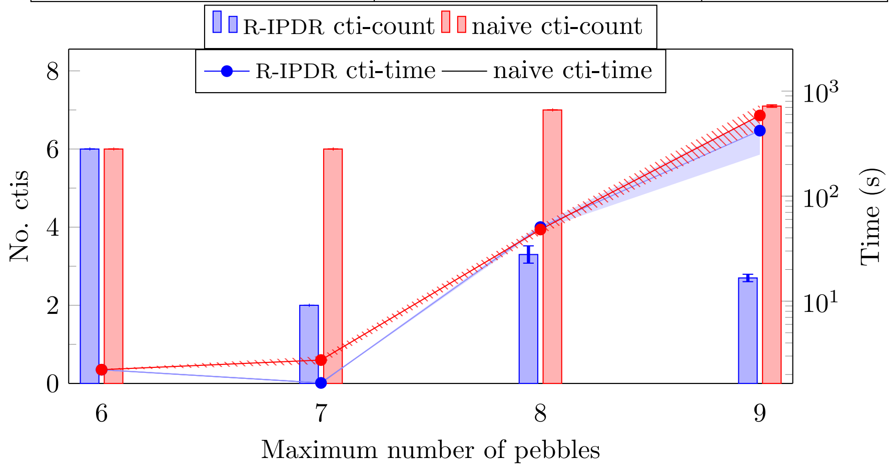


(b) Left axis: the total number of calls to the SAT-solver (sat-count). Right axis: the total time spent waiting on the SAT-solver (sat-time).

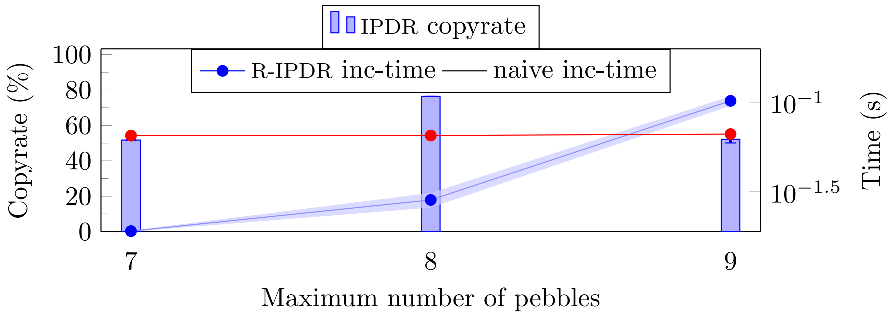
Figure A.10: Statistics for the constraining `gf2^5mult_29_129` experiment, averaged over 10 repetitions. On constraint 29, the algorithm found a trace with 19 pebbles, so it could continue with 18 afterwards.

A.3 Relaxing Pebbling statistics

	R-IPDR	Naive R-IPDR	Improvement
avg. time	474.993 s	639.886 s	25.77 %
std. dev. time	171.428 s	150.129 s	
avg. incremental time	0.149 s		
std. dev. incremental time	0.006 s		
max. invariant: constraint	8	8	
min. strategy: pebbled	9	9	
min. strategy: length	38	36	-5.56 %

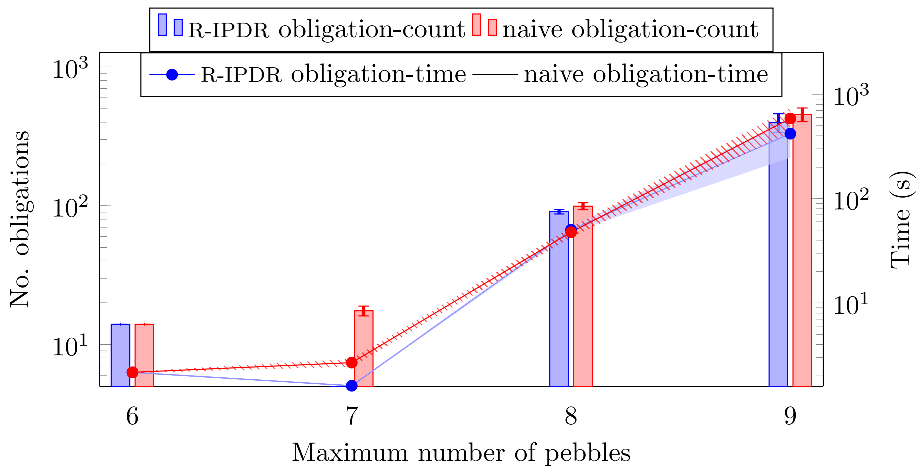


(a) Left axis: the number of counterexamples found in the major iteration (cti-count). Right axis: the total runtime (cti-time).

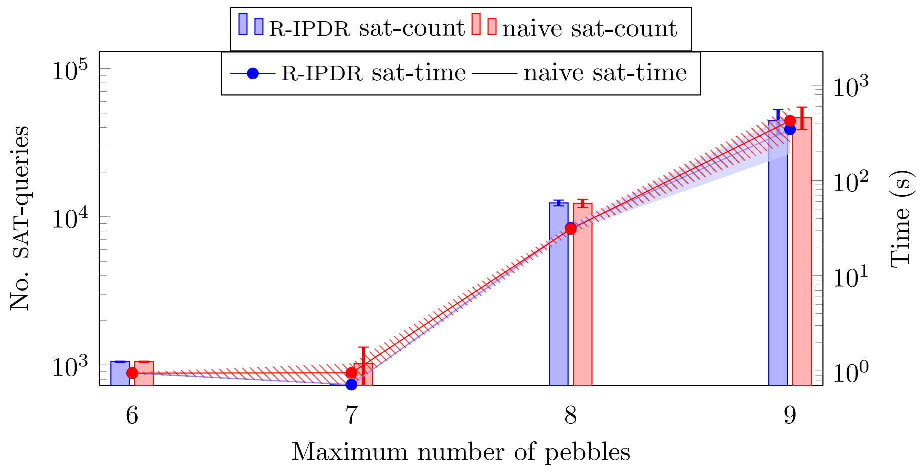


(b) Left axis: the percentage of clauses that could be copied to the next instance (listed on the x-axis). Right axis: the runtime was spent on copying clauses to the next instance. This number is already included in the times of the upper graph.

Figure A.11: Statistics for the relaxing `5bitadder` *experiment, averaged over 10 repetitions.*



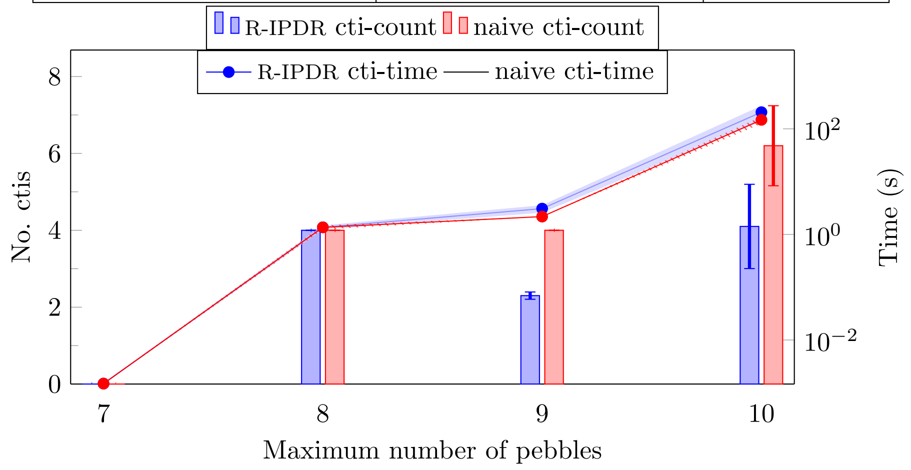
(a) Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time).



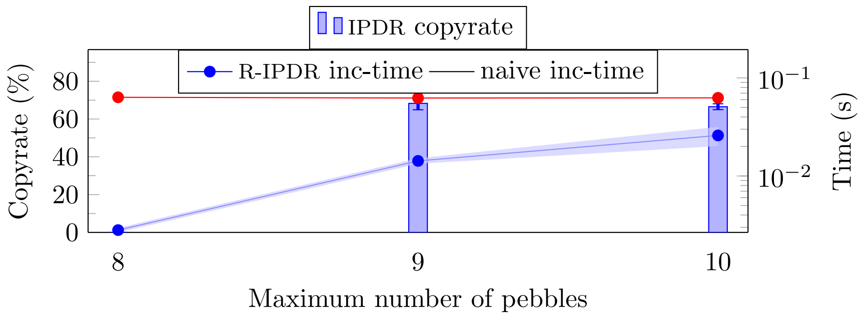
(b) Left axis: the total number of calls to the SAT-solver (sat-count). Right axis: the total time spent waiting on the SAT-solver (sat-time).

Figure A.12: Statistics for the relaxing `5bitadder` experiment, averaged over 10 repetitions.

	R-IPDR	Naive R-IPDR	Improvement
avg. time	211.390 s	151.867 s	-39.19 %
std. dev. time	67.712 s	22.490 s	
avg. incremental time	0.043 s		
std. dev. incremental time	0.006 s		
max. invariant: constraint	9	9	
min. strategy: pebbled	10	10	
min. strategy: length	27	26	-3.85 %

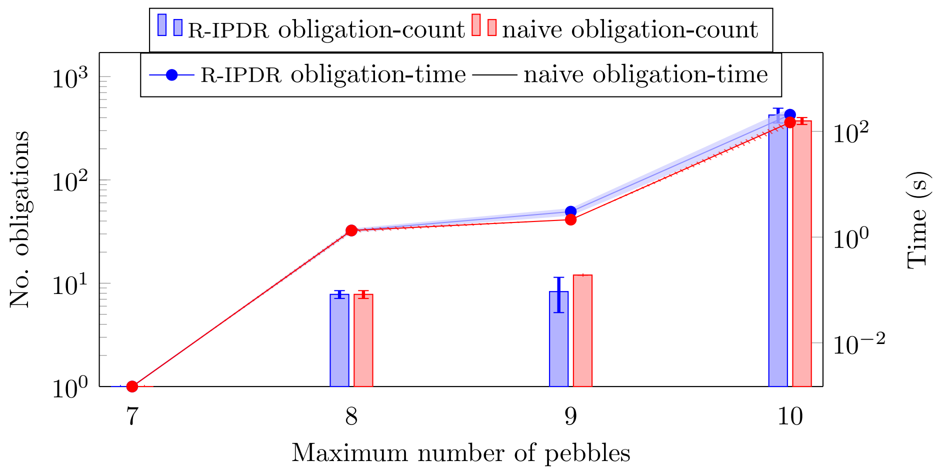


(a) Left axis: the number of counterexamples found in the major iteration (cti-count). Right axis: the total runtime (cti-time).

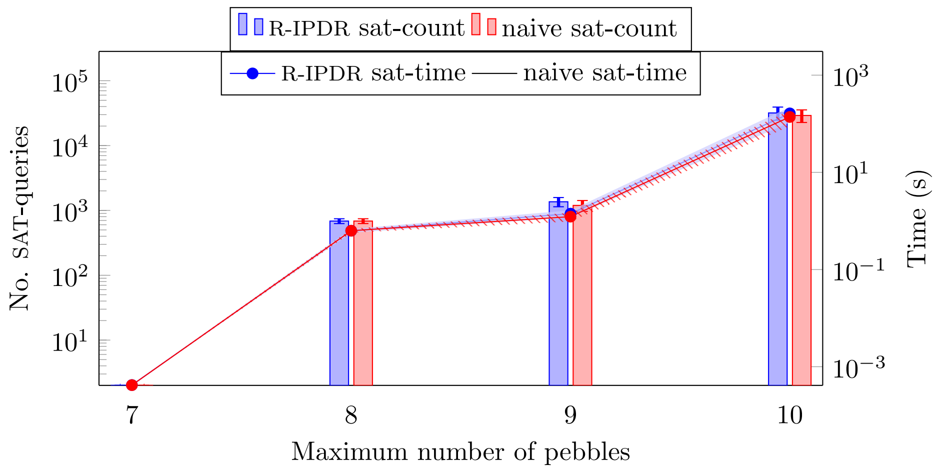


(b) Left axis: the percentage of clauses that could be copied to the next instance (listed on the x-axis). Right axis: the runtime was spent on copying clauses to the next instance. This number is already included in the times of the upper graph.

Figure A.13: Statistics for the relaxing `ham7tc` *experiment, averaged over 10 repetitions.*



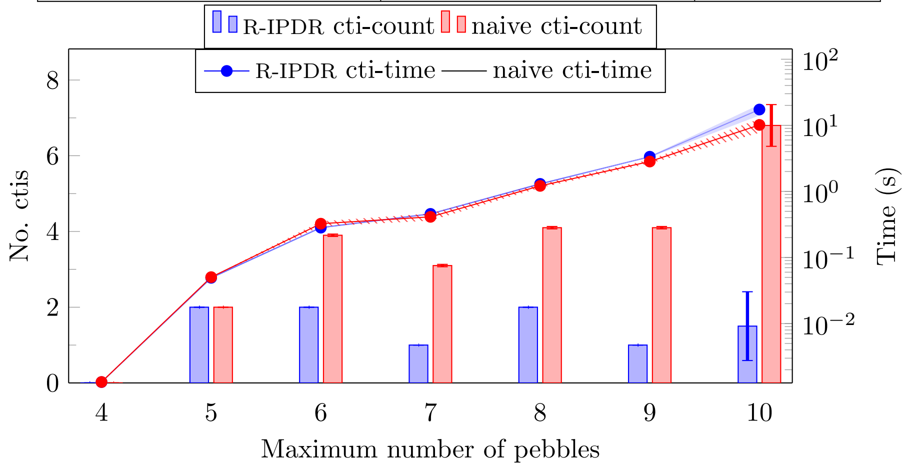
(a) Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time). For constraint 7, no obligations were enqueued as PDR *terminated in the initialization step. Its obligation-count value was set to 1 to accommodate the logarithmic scale and the obligation-time to the total runtime (which is a close approximation).*



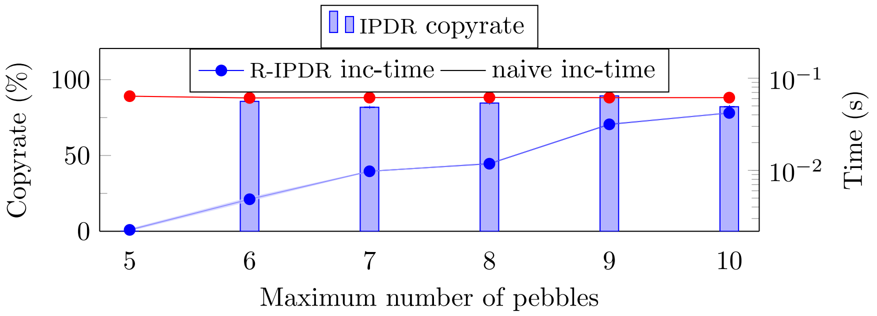
(b) Left axis: the total number of calls to the SAT*-solver (sat-count). Right axis: the total time spent waiting on the* SAT*-solver (sat-time).*

Figure A.14: Statistics for the relaxing `ham7tc` *experiment, averaged over 10 repetitions.*

	R-IPDR	Naive R-IPDR	Improvement
avg. time	22.853 s	15.325 s	-49.12 %
std. dev. time	3.234 s	2.167 s	
avg. incremental time	0.102 s		
std. dev. incremental time	0.001 s		
max. invariant: constraint	9	9	
min. F length	9	9	0.00 %
min. strategy: pebbled	10	10	
min. strategy: length	23	23	0.00 %

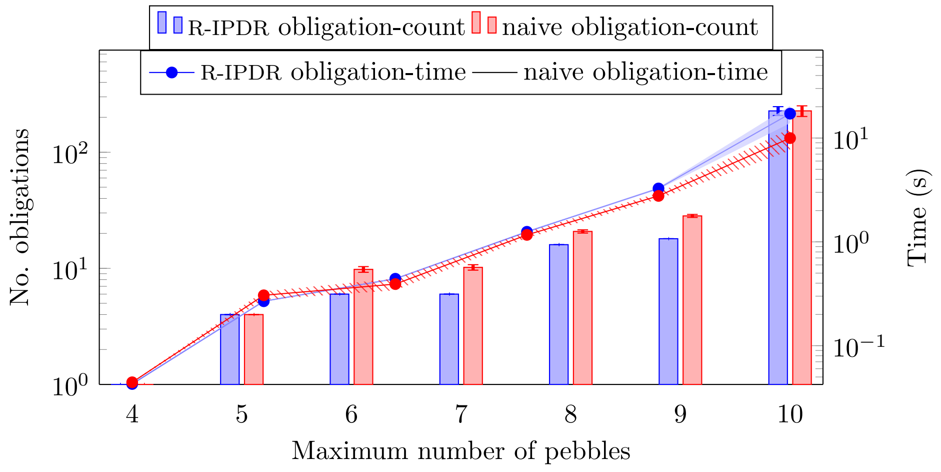


(a) Left axis: the number of counterexamples found in the major iteration (cti-count). Right axis: the total runtime (cti-time).

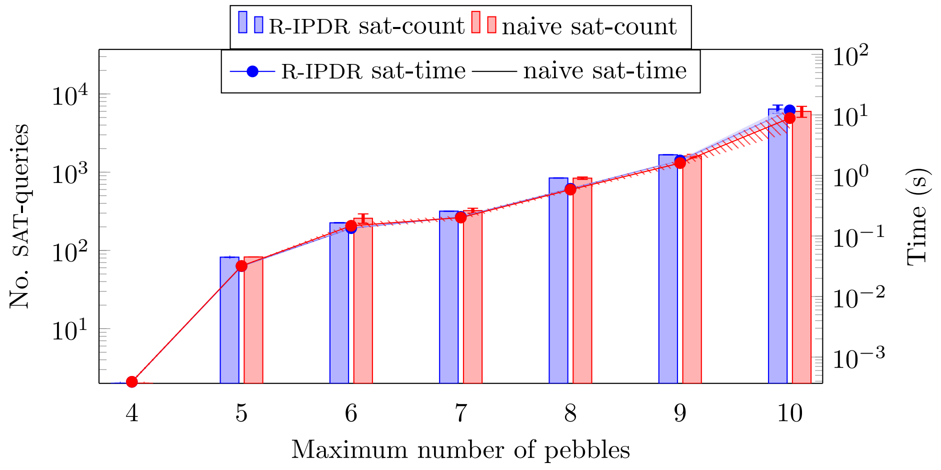


(b) Left axis: the percentage of clauses that could be copied to the next instance (listed on the x-axis). Right axis: the runtime was spent on copying clauses to the next instance. This number is already included in the times of the upper graph.

Figure A.15: Statistics for the relaxing `hwb4tc` *experiment, averaged over 10 repetitions.*



(a) Left axis: total number of obligations (obligation-count) handled by the minor iteration. Right axis: the time spent on them (obligation-time). For constraint 4, no obligations were enqueued as PDR *terminated in the initialization step. Its obligation-count value was set to 1 to accommodate the logarithmic scale and the obligation-time to the total runtime (which is a close approximation).*



(b) Left axis: the total number of calls to the SAT*-solver (sat-count). Right axis: the total time spent waiting on the* SAT*-solver (sat-time).*

Figure A.16: Statistics for the relaxing `hwb4tc` *experiment, averaged over 10 repetitions.*

B PDR in set and SAT

PDR uses propositional logic to encode a symbolic transition system and a SAT-solver to reason over it. Section 2.3 showed how this propositional logic relates to the intuitive set-representation of the system. This section gives the explicit propositional logic encodings used by the PDR algorithm in Section 3 and IPDR in Section 4, derived from the definitions in Section 2.3.

B.1 Properties

The set-theoretic properties of the PDR state from Definition 3.6 can be represented with propositional logic.

Candidates. The properties of each candidate in the sequence F, with propositional formula $\mathcal{F}$, from Definition 3.3 can be represented as per the alternative Definition B.1.

Definition B.1. Given symbolic transition system $M = (X, \mathcal{I}, \mathcal{T})$ and a propositional formula $\mathcal{P}$. The *sequence of candidate inductive invariants* is defined as $\mathcal{F} = \{\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots\}$. It has the following properties Φ [17], which are maintained throughout the PDR algorithm:

$$\mathcal{F}_0 = \mathcal{I}, \tag{Φ_0}$$

$$\forall 0 \leq i \leq k \colon \mathcal{F}_i \Rightarrow \mathcal{P}, \tag{Φ_1}$$

$$\forall 0 \leq i < k \colon \mathcal{F}_i \Rightarrow \mathcal{F}_{i+1}, \tag{Φ_2}$$

$$\forall 0 \leq i < k \colon \mathcal{F}_i \wedge \mathcal{T} \Rightarrow \mathcal{F}'_{i+1} \tag{Φ_3}$$

Obligations. The properties of the queue of obligations O, defined in Definition 3.5, can be alternatively defined by Definition B.2.

Definition B.2. $O \subseteq \mathbb{B}^X \times \mathbb{N}$ is a set of *proof obligations* adhering to the following properties:

$$\forall (s, i) \in O \colon 0 < i \leq k, \tag{Ω_1}$$

$$\forall (s, i) \in O \colon \neg s \textit{ is inductive relative to } \mathcal{F}_{i-1} \colon \mathcal{F}_{i-1} \wedge \neg s \wedge \mathcal{T} \Rightarrow \neg s', \tag{Ω_2}$$

$$\forall (s, i) \in O \colon \exists \pi \in \mathit{Paths}(TS), \mathcal{B} \Rightarrow \neg \mathcal{P} \colon \pi = s, \pi_1, \pi_2, \ldots \mathcal{B} \tag{Ω_3}$$

B.2 SAT queries used by PDR

PDR performs several SAT-queries that are intuitively represented using set-theory. The queries in this section are, sometimes repeatedly, used throughout PDR.

Initiation. When a PDR state is initiated, 0- and 1-step reachability is quickly verified. Within `pdr` (Algorithm 1) this is abstracted away within the PDR state, although traditional PDR will explicitly check this. The IPDR functions `ipdr-constrain` (Algorithm 6) and `ipdr-relax` (Algorithm 7) perform them explicitly.

0-step reachability is verified with the query

$$I \not\subseteq P$$

This is represented by the SAT-query

$$\mathsf{SAT}(\mathcal{I} \wedge \neg\mathcal{P})$$

If the result is 1, then one of the initial states violates P.

1-step reachability is verified by

$$I.\Delta \not\subseteq P$$

This is represented by the SAT-query

$$\mathsf{SAT}(I \wedge \mathcal{T} \wedge \neg\mathcal{P}')$$

If the result is 1, then an initial state violates P in one step.

Querying a Predecessor. PDR often queries a direct predecessor from one or more states. The `pdr` function (Algorithm 1) queries a predecessor to $\overline{P}$ from F_k, denoted as the query

$$\exists(o, p) \in F_k.\Delta.\overline{P}$$

This is represented by the SAT-query

$$\mathsf{SAT}(\mathcal{F}_k \wedge \mathcal{T} \wedge \neg\mathcal{P}')$$

If this query returns 1 then there is a witness from which the cubes $\mathcal{o}$ and $\mathcal{p}'$ can be extracted, representing the transition (o, p) that leads to a violation of P.

The `block` function (Algorithm 3) queries a predecessor to a specific state from O with the query

$$\exists p \in F_i : \Delta(p, s)$$

This is represented by the SAT-query

$$\mathsf{SAT}(\mathcal{F}_n \wedge \Delta T \wedge \mathcal{s}')$$

If the result is 1, then there is a witness containing the cubes $\mathcal{p}$ and $\mathcal{s}'$, revealing the predecessor p.

Relative inductiveness. The relative inductiveness property is essential to PDR and is asserted in `remove-cube` (Algorithm 4) in order to push an unreachable state s as far as possible. This is done by performing the query

$$s \notin (F_i \setminus s).\Delta$$

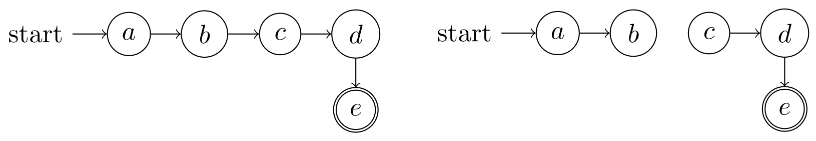
This is represented by the SAT-query

$$\mathsf{SAT}(\mathcal{F}_i \wedge \mathcal{T} \wedge \neg s \wedge s')$$

If the result is 1, then there is a transition in F_i to s and $\overline{s}$ is not inductive relative to F_i.

C Examples

C.1 Constraining IPDR



(a) Simple *(b) Simple*$^{\downarrow}$

Figure C.1: Two instances of a Simple system.

Consider the *Simple* system from Figure C.1. The system *Simple*$^{\downarrow}$ is a constrained instance of *Simple*, as defined by Definition 4.1, obtained by removing the transition $b \to c$.

The sections below will show the execution of PDR, as shown in Section 3, for these systems step-by-step, and show where the IPDR algorithm from Section 4 is able to avoid repeating work.

For the purposes of this example the transition system (X, I, Δ) is defined as follows:

- $X = \{ a, b, c, d, e \}$
- $I = \{ a \}$
- $\Delta = \{ (a, b), (b, c), (c, d), (d, e) \}$ and $\Delta^{\downarrow} = \{ (a, b), (c, d), (d, e) \}$
- With a property $P = \{ a, b, c, d \}$ and $\overline{P} = \{ e \}$, which searches for a trace to the e state.

This example will omit some functionality of PDR to improve readability, namely the propagation, clause pushing and generalisation mechanisms. This does not alter the correctness of the algorithm, it would only impact its performance.

C.1.1 PDR for the simple system

Initialisation. PDR first asserts that $I = \{ a \} \subseteq P$, which eliminates the 0-step trace, and that $I.\Delta = \{ b \} \subseteq P$, which eliminates the 1-step trace.

This allows PDR to be initialised with $F = \{ I, P \}$:

	F (candidates)	B (blocked)
0	$\{ a \}$	-
1	$\{ a, b, c, d \}$	$\{ \}$

Iteration k = 1. PDR finds the potential counterexample $d \in F_1$ which reaches e in a single step, and calls the `block(`d`, 0)` to verify if it is reachable from F_0.

PDR immediately finds that there is no state in F_0 that leads to d in one step, so d can be removed from F_1 and $(d, 0)$ is eliminated as an obligation.

With all obligations eliminated, PDR moves onto the next iteration with $F_0 = \{ a, F_1 \}$:

	F (candidates)	B (blocked)
0	$\{ a \}$	-
1	$\{ a, b, c \}$	$\{ d \}$
2	$\{ a, b, c, d \}$	$\{ \}$

Iteration k = 2. Again, PDR finds the counterexample $d \in F_2$ and attempts to `block(`d`, 1)`.

PDR now finds the state $c \in F_1$, which leads to d and attempts to `block(`c`, 0)`. This second execution of `block` then finds that there is no state in F_0 that leads to c, so c may be blocked in F_1. This eliminates the final predecessor of d in F_1, and allows d to be blocked in F_1 and F_2.

PDR moves onto the next iteration with $F_0 = \{ a, F_1, F_2 \}$:

	F (candidates)	B (blocked)
0	$\{ a \}$	-
1	$\{ a, b \}$	$\{ c, d \}$
2	$\{ a, b, c \}$	$\{ d \}$
3	$\{ a, b, c, d \}$	$\{ \}$

Iteration k = 3. The previous pattern repeats again, enqueueing the obligations $(d, 2)$, $(c, 1)$ and $(b, 0)$. PDR then finds that b is in fact reachable from F_0 and PDR terminates with the trace $a \to b \to c \to d \to e$.

C.1.2 PDR for the constrained simple system

PDR executes the same for the constrained instance of *Simple* as it did for the first instance for the initialisation and second iteration.

The first iteration ends when it finds that no F_0 state leads to d, eliminating it from F_1, and the second finds that there is no predecessor to c in F_0, eliminating c from F_1 and F_2.

We enter the third iteration with :

	F (candidates)	B (blocked)
0	$\{a\}$	-
1	$\{a,b\}$	$\{c,d\}$
2	$\{a,b,c\}$	$\{d\}$
3	$\{a,b,c,d\}$	$\{\}$

Iteration k = 3. PDR finds obligations $(d,2)$ and $(c,1)$ again with `block`. Within `block(`c`, 1)`, since the transition (b,c) has been removed in *Simple*$^{\downarrow}$, PDR finds that there is no predecessor to c in F_1. This eliminates c from F_1 and F_2, and with the obligation $(c,1)$ fulfilled d is eliminated F_1, F_2 and F_3.

Afterwards, PDR detects that $F_1 = F_2$, revealing an inductive invariant. Which can be seen easily as it includes the initial state, one cannot move out of $\{a,b\}$ and this set does not contain a $\overline{P}$-state.

	F (candidates)	B (blocked)
0	$\{a\}$	-
1	$\{a,b\}$	$\{c,d\}$
2	$\{a,b\}$	$\{c,d\}$
3	$\{a,b,c,d\}$	$\{d\}$

C.1.3 IPDR for the constrained simple system

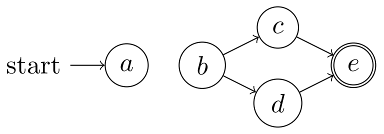
Note that every state that has been blocked after the second iteration of the first instance remains unreachable in the second: c and d cannot be reached in one step and d cannot be reached in two steps. In fact, likely owning to the simplicity of *Simple*, the executions of the two instances are exactly the same up until that point.

This intuitively seems like some extra work that could be avoided, and IPDR does just that. Following the description in Algorithm 6, all the clauses in B are copied over between instances, and the constrained instance starts at iteration $k = 3$. This then allows the constrained instance to execute **only** the third iteration, after which it is able to terminate.

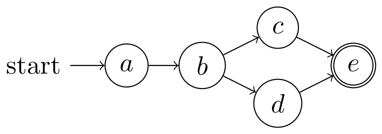
C.2 Relaxing IPDR

Consider the *Simple* system from Figure C.2. The system *Simple*$^{\downarrow}$ is a constrained instance of *Simple*, as defined by Definition 4.1, obtained by removing the transition $b \rightarrow c$.

The sections below will show the execution of PDR, as shown in Section 3, for



(a) Simple



(b) Simple$^{\uparrow}$

Figure C.2: Two instances of a Simple system.

these systems step-by-step, and show where the IPDR algorithm from Algorithm 6 is able to avoid repeating work.

For the purposes of this example the transition system (X, I, Δ) is defined as follows:

- $X = \{ a, b, c, d, e \}$
- $I = \{ a \}$
- $\Delta = \{ (b, c), (b, d), (c, e), (d, e) \}$
  and $\Delta^{\uparrow} = \{ (a, b), (b, c), (b, d), (c, e), (d, e) \}$
- With a property $P = \{ a, b, c, d \}$ and $\overline{P} = \{ e \}$, which searches for a trace to the e state.

This example will omit some functionality of PDR to improve readability, namely the propagation, clause pushing and generalisation mechanisms. This does not alter the correctness of the algorithm, it would only impact its performance.

C.2.1 PDR for the simple system

Initialisation. PDR first asserts that $I = \{ a \} \subseteq P$, which eliminates the 0-step trace, and that $I.\Delta = \{ b \} \subseteq P$, which eliminates the 1-step trace.

This allows PDR to be initialised with $F = \{ I, P \}$:

	F (candidates)	B (blocked)
0	$\{ a \}$	-
1	$\{ a, b, c, d \}$	$\{ \}$

Iteration k = 1. PDR finds the potential counterexample $d \in F_1$ which reaches e in a single step, and calls the `block(`d`, 0)` to verify if it is reachable from F_0.

PDR immediately finds that there is no state in F_0 that leads to d in one step, so d can be removed from F_1 and $(d, 0)$ is eliminated as an obligation.

Next, PDR finds the potential counterexample $c \in F_1$ and calls `block(`c`, 0)`. This also finds no F_0 state that leads to c, so c is removed from F_1 and $(c, 0)$ is eliminated.

With all obligations eliminated, PDR moves onto the next iteration with $F_0 = \{ a, F_1 \}$:

	F (candidates)	B (blocked)
0	$\{ a \}$	-
1	$\{ a, b \}$	$\{ c, d \}$
2	$\{ a, b, c, d \}$	$\{ \}$

Iteration k = 2. Again, PDR finds the counterexample $d \in F_2$ and attempts to `block(`d`, 1)`.

PDR now finds the state $b \in F_1$, which leads to d and attempts to `block(`c`, 0)`. This second execution of `block` then finds that there is no state in F_0 that leads to b, so b may be blocked in F_1. This eliminates the final predecessor of d in F_1, and allows d to be blocked in F_1 and F_2.

The state $c \in F_2$ is then encountered, calling `block(`c`, 1)`. Since b has already been eliminated by the obligation $(d, 1)$, c has no more predecessors in F_1 and may be blocked in F_1 and F_2.

PDR moves onto the next iteration with $F_0 = \{ a, F_1, F_2 \}$:

	F (candidates)	B (blocked)
0	$\{ a \}$	-
1	$\{ a \}$	$\{ b, c, d \}$
2	$\{ a, b \}$	$\{ c, d \}$
3	$\{ a, b, c, d \}$	$\{ \}$

Iteration k = 3. The previous pattern repeats again, enqueueing the obligations $(d, 2)$ and $(c, 2)$. PDR then finds that $b \in F_2$ can reach either of them, so the obligation $(b, 1)$ is enqueued. Since there is no $a \to b$ transition, b may be removed from F_1 and F_2. Now with $(b, 1)$ eliminated, $(d, 2)$ and $(c, 2)$ can also be eliminated, removing them from $F_1 \ldots F_3$.

	F (candidates)	B (blocked)
0	$\{ a \}$	-
1	$\{ a \}$	$\{ b, c, d \}$
2	$\{ a \}$	$\{ b, c, d \}$
3	$\{ a, b \}$	$\{ c, d \}$
4	$\{ a, b, c, d \}$	$\{ \}$

Now that $F_1 = F_2$, PDR terminates with an inductive invariant.

C.2.2 PDR for the relaxed simple system

Up until iteration $k = 2$, the relaxed instance goes through the exact same steps as the first instance.

Following those same steps, we enter the second iteration with:

	F (candidates)	B (blocked)
0	$\{a\}$	-
1	$\{a, b\}$	$\{c, d\}$
2	$\{a, b, c, d\}$	$\{\}$

Iteration k = 2. Obligations $(d, 1)$ and $(b, 0)$ are enqueued as before. While considering $(b, 0)$, PDR now finds that $a \rightarrow b$, causing the algorithm to terminate with the trace $a \rightarrow b \rightarrow d \rightarrow e$.

C.2.3 IPDR for the relaxed simple system

By initialising the system as per Algorithm 7, we get the following sequence and can begin iteration $k = 1$:

	F (candidates)	B (blocked)
0	$\{a\}$	-
1	$\{a, b\}$	$\{c, d\}$

Iteration k = 1. All predecessors to e are already blocked in F_1, so IPDR immediately moves on to iteration $k = 2$.

	F (candidates)	B (blocked)
0	$\{a\}$	-
1	$\{a, b\}$	$\{c, d\}$
2	$\{a, b, c, d\}$	$\{\}$

Iteration k = 2. At this point, F is exactly the same as it was for PDR in the relaxed instance, so it now only encounter the obligations $(d, 1)$ and $(b, 0)$ before terminating with a trace.

D Hanging experimental runs

During the experimental evaluation from Section 6.4, Z3's SPACER engine solves each input graph ten times using different random seeds for its fixed point engine: 1962830626, 1034687201, 209198572, 104875891, 399683891, 903448323, 1277237697, 844529350, 249979600, 554136435. However, SPACER would consistently hang when solving certain inputs with SPACER initialised to specific random seeds. The exact combinations of random seeds and graph that caused this denoted in Table D.1. The runs with these combinations of seeds and input graph where omitted from the experimental results.

Graph	SPACER constraining	SPACER relaxing	SPACER binary
`nth_prime4_inc_d1`	554136435		
`4b15g_1`		399683891	903448323
`5mod5tc`			554136435
`rd73d2`	1277237697	1962830626	
`mod5adders`		1962830626	1034687201, 209198572

Table D.1: Hanging experimental runs for the SPACER engine.