國立臺灣大學電機資訊學院電機工程學系

碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

自適應多時間框架的性質導向可達性演算法

An Adaptive Multi-Timeframe Property Directed Reachability Algorithm

陳韋旭

Wei-Hsu Chen

指導教授: 黃鐘揚 博士

Advisor: Chung-Yang (Ric) Huang Ph.D.

中華民國 114 年 8 月

August, 2025

doi:10.6342/NTU202503854

摘要

性質導向可達性（Property Directed Reachability, PDR）演算法為現今解決模型檢查（Model Checking）問題最有效之技術之一。然而，在一般典型的實現方法中，常因數量過多的回溯檢查（backtracking checks）而導致效率低落。雖然回溯檢查中大多數的證明義務（proof obligations）相對容易被阻擋，但少數困難的證明義務卻佔據了大比例的求解時間。為了解決此問題，我們提出一種多時間框架的性質導向可達性演算法，將動態時間框架擴展策略整合進多時間框架的轉移模型中，利用多時間框架電路之邏輯推論，以加速對困難證明義務之阻擋過程。

儘管這樣的整合在概念上看似簡單，先前的研究卻在正確性與完備性方面遭遇挑戰。本研究透過額外的精化機制克服這些問題，並提供形式化的正確性與完備性證明。

我們將所提出的演算法實作於兩個先進的性質導向可達性框架——ABC 與 rIC3——並以硬體模型驗證競賽（HWMCC）的基準測試集進行實驗評估。實驗結果顯示，與典型的單時間框架相比，我們的方法在效能上表現更為優異。

關鍵字：正規驗證、模型檢查、可滿足性問題、性質導向可達性技術、時間框架延展

doi:10.6342/NTU202503854

Abstract

Property Directed Reachability (PDR) is one of the most effective techniques for solving model checking problems. However, in its typical implementation, it often suffers from excessive backtracking. While most proof obligations are relatively easy to block, a small number of difficult ones dominate the solving time. To address this inefficiency, we propose a multi-timeframe PDR algorithm that integrates a dynamic timeframe expansion strategy into a multi-timeframe transition framework. This design leverages reasoning across multiple timeframes to accelerate the blocking of difficult obligations.

Although this integration appears straightforward, prior attempts have encountered challenges regarding soundness and completeness. Our approach resolves these issues through additional refinement mechanisms, supported by formal proofs of correctness and completeness.

We implement the proposed algorithm on two state-of-the-art PDR frameworks—ABC and rIC3—and evaluate it using benchmarks from the Hardware Model Checking

doi:10.6342/NTU202503854

Competition (HWMCC). Experimental results show that our approach consistently outperforms typical single-timeframe PDR in both frameworks.

Keywords: Formal Verification, Model Checking, Satisfiability Problem, Property Directed Reachability, Timeframe Expansion

doi:10.6342/NTU202503854

Contents

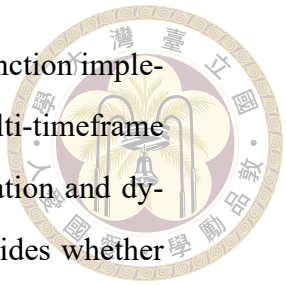
doi:10.6342/NTU202503854

doi:10.6342/NTU202503854

doi:10.6342/NTU202503854

List of Figures

doi:10.6342/NTU202503854



doi:10.6342/NTU202503854

doi:10.6342/NTU202503854

List of Tables

doi:10.6342/NTU202503854

Chapter 1 Introduction

This chapter introduces the background and motivation for improving Property Directed Reachability (*PDR*) in section 1.1, reviews related work in section 1.2, summarizes the main contributions in section 1.3, and outlines the structure of the thesis in section 1.4.

1.1 Background and Motivation

Formal verification plays a crucial role in the integrated circuit (IC) industry. Given a transition system and a safety property, model checking can either detect violations of the property (i.e., bugs) or formally prove that the design satisfies the property under all possible behaviors. Identifying bugs early in the design process significantly reduces downstream manufacturing costs and improves overall design quality. As modern hardware designs grow in scale and complexity, the efficiency of model checking algorithms becomes increasingly critical.

Among various model checking techniques—such as K-induction [8], interpolation-based model checking [7], and bounded model checking [1]—PDR [4], also known as IC3 [2], stands out as one of the most efficient methods for verifying both `SAFE` and `UNSAFE` designs. PDR maintains an over-approximation of the reachable states and tries to find an inductive invariant to prove safety. Specifically, it starts from a bad state violating the

doi:10.6342/NTU202503854

property and attempts to recursively find its predecessors until either a counterexample trace is found or all predecessor states are blocked. The algorithm terminates when it either reaches an initial state (i.e., finds a counterexample) or reaches a fixpoint (i.e., finds an inductive invariant).

Despite several enhancements to the search process, such as state generalization and three-valued simulation proposed in [4], typical PDR still suffers from performance bottlenecks due to its exhaustive and recursive exploration of predecessor states. In particular, it can generate a large number of proof obligations and experience significant backtracking, which degrades overall performance.

To address this issue, we propose an *adaptive multi-timeframe PDR algorithm*. Unlike typical PDR, which only considers single-step transitions during blocking, the proposed approach allows the use of multi-timeframe transitions when necessary. The intuition is that most proof obligations are eventually blocked (unless a counterexample is found), and allowing deeper unrolling within a single SAT query can introduce additional constraints that facilitate faster blocking. The detailed motivation and empirical analysis for this approach will be discussed in chapter 3.

Multi-timeframe reasoning within the PDR framework has been explored in prior work. In the next section, we briefly review two related studies.

1.2 Related Work on Multi-Timeframe PDR

Prior work has investigated extensions of PDR that incorporate multi-timeframe reasoning. These studies aim to address specific limitations of typical PDR by enabling deeper transition reasoning. [11] proposes improving PDR using two types of dynamic

doi:10.6342/NTU202503854

timeframe expansion: parallel timeframe expansion ($\texttt{PDR}_{\texttt{parallel}}$) and serial timeframe expansion ($\texttt{PDR}_{\texttt{serial}}$). Both approaches unroll the transition relation d times. The key difference is that the former solves d PDR problems with different initial states ($I = R_0, R_1, \cdots, R_{d-1}$), while the latter solves a single PDR problem with an initial state defined as $I = \bigvee_{i=0}^{d-1} R_i$. Their experimental results show that these multi-timeframe variants of PDR can solve more instances compared to the typical PDR approach. Although this approach improves performance, it may be more accurately viewed as a reformulation of the original model checking problem using an unrolled transition relation, rather than as a direct adaptation of the PDR algorithm.

[10] combines Bounded Model Checking (*BMC*) [1], a model checking technique for bug finding, with Complementary Approximate Reachability (*CAR*) [6]. CAR is a SAT-based model checking algorithm inspired by PDR. In [10], the authors adopt the *Backward-CAR* framework defined in [6], which maintains two sequences: the *O-sequence* and the *U-sequence*. The *O-sequence* represents an over-approximation in the backward direction, while the *U-sequence* is an under-approximation in the forward direction. CAR checks whether a state in the U-sequence can reach the O-sequence through a single-step transition.

This mechanism is similar in spirit to PDR, but in the reverse direction. In PDR, the algorithm checks whether a state (i.e., a proof obligation) that leads to a property violation can be reached from its predecessors in the over-approximation sequence known as the *R-sequence*. In [10], the authors observe that this search process could get ”stuck” in certain cases and address this issue by integrating BMC. By leveraging an unrolled transition relation, the algorithm can explore deeper paths and escape from such problematic regions. Although this integration improves the original CAR framework in terms of bug-finding

doi:10.6342/NTU202503854

capability, it introduces a completeness limitation. While the original CAR algorithm is complete (i.e., it can handle both `SAFE` and `UNSAFE` cases), the modified version combined with BMC is only applicable to `UNSAFE` instances. The authors acknowledge the challenge of extending their approach to achieve completeness.

1.3 Thesis Contributions

This thesis proposes a new approach to improve the efficiency of the PDR algorithm by introducing adaptive multi-timeframe reasoning. The main contributions of this work are summarized as follows:

- To the best of our knowledge, this is the first work to identify and analyze the challenges of integrating multi-timeframe transitions into the typical PDR algorithm.
- We propose an adaptive multi-timeframe PDR algorithm that selectively unrolls the transition relation during the blocking phase to improve the efficiency of blocking hard-to-block cubes. We also provide formal proofs of soundness and completeness.
- Experimental results show that our algorithm, implemented on top of two state-of-the-art PDR frameworks, ABC [3] and rIC3 [9], consistently outperforms their original counterparts. In both cases, the multi-timeframe approach achieves better performance, demonstrating the robustness and generality of the proposed algorithm.

doi:10.6342/NTU202503854

1.4 Thesis Organization

The remainder of this thesis is organized as follows. In chapter 2, we introduce the basic concepts, notation, and the typical PDR algorithm that serve as the foundation for this work. Chapter 3 presents an experimental evaluation where we identify specific performance bottlenecks in the typical PDR algorithm, which motivates the need for enhancing it with multi-timeframe reasoning. In chapter 4, we discuss the challenges of integrating multi-timeframe transitions into PDR, focusing on both correctness and efficiency aspects. Chapter 5 introduces our adaptive multi-timeframe PDR algorithm, including its pseudocode, an illustrative example, and the corresponding proofs of soundness and completeness. Chapter 6 presents a case study that demonstrates how the proposed algorithm improves upon typical PDR in a hardware model checking benchmark. Chapter 7 reports the experimental results, where we compare the proposed algorithm against typical implementations. Finally, in chapter 8, we conclude the thesis and outline potential directions for future work.

doi:10.6342/NTU202503854

Chapter 2 Preliminary

2.1 Propositional Logic Foundations for Property Directed Reachability

This section introduces basic concepts in propositional logic and Boolean satisfiability (*SAT*) solving that are fundamental to understanding the PDR algorithm. We define Boolean variables, literals, and cubes, describe conjunctive normal form (*CNF*) formulas, and present the notion of SAT queries.

Boolean Expressions and Cubes

A *Boolean variable* is a variable that takes a value in the Boolean domain $\mathbb{B} = \{0, 1\}$, representing *false* and *true*, respectively.

A *literal* is either a Boolean variable x (a *positive literal*) or its negation $\neg x$ (a *negative literal*).

A *cube* is a conjunction of literals, typically written as $l_1 \wedge l_2 \wedge \cdots \wedge l_k$, or more compactly as $l_1 l_2 \cdots l_k$. A cube specifies an assignment to a subset of the Boolean variables. In the context of transition systems, a cube may represent a symbolic state (if partial) or a

doi:10.6342/NTU202503854

concrete state (if complete).

Conjunctive Normal Form

A Boolean formula is in CNF if it is a conjunction of clauses, where each clause is a disjunction of literals. CNF is the standard input format for modern SAT solvers. In model checking, the transition relation of a finite state system is typically encoded as a CNF formula to enable efficient SAT-based reasoning.

Boolean satisfiability Queries

A SAT query asks whether a given Boolean formula is satisfiable—that is, whether there exists an assignment to the variables that makes the formula evaluate to true. In our context, SAT queries are always expressed in CNF form.

For example, a typical SAT query in PDR may be of the form:

$$\text{SAT?}\,[R_i(X) \wedge \neg c(X) \wedge T(X, X') \wedge c'(X')]\,,$$

where $R_i(X)$ is an over-approximation of the reachable states at frame i, $c(X)$ is the cube to be blocked, $T(X, X')$ is the transition relation, and $c'(X')$ is the corresponding cube over the next-state variables.

doi:10.6342/NTU202503854

2.2 Finite State Transition System

A *finite state transition system* S is a tuple

$$S = (X, I, T)$$

where:

- X is a finite set of Boolean variables representing the system state.
- $I(X)$ is a Boolean formula over X, representing the set of *initial states*.
- $T(X, X')$ is a Boolean formula over the current-state variables X and next-state variables X', representing the *transition relation*.

Each *state* of the system is an assignment $s \in \{0,1\}^{|X|}$ to the variables in X. The formula $I(X)$ characterizes the initial states, and $T(X, X')$ characterizes the valid transitions between states. A pair $(s, s') \in \{0,1\}^{|X|} \times \{0,1\}^{|X|}$ satisfies T if there exists a transition from state s to state s'.

2.3 Model Checking

Model checking is the process of verifying whether a transition system satisfies a given property.

Given a transition system $S = (X, I, T)$ and a safety property $P(X)$, the goal is to determine whether all reachable states satisfy P.

doi:10.6342/NTU202503854

The property is UNSAFE if there exists a finite sequence of states (called a *counterexample trace*) $\langle s_0, s_1, \ldots, s_n \rangle$ such that:

1. $I(s_0)$ holds (i.e., s_0 is an initial state),
2. $T(s_i, s_{i+1})$ holds $\forall i \in [0, n)$ (i.e., each step follows the transition relation),
3. $\neg P(s_n)$ holds (i.e., the last state violates the property).

If no such counterexample exists, then all reachable states satisfy the property P, and the system is said to be SAFE with respect to P. Formally, the safety condition can be expressed as:

$$\text{Reach}(S) \subseteq P,$$

where $\text{Reach}(S)$ denotes the set of all states reachable from the initial states I via the transition relation T.

2.4 Property Directed Reachability Algorithm

The PDR algorithm, also known as IC3, is a SAT-based approach to the model checking problem. This section provides a high-level overview; a more comprehensive description can be found in [4].

Given a transition system $S = (X, I, T)$ and a safety property $P(X)$, the PDR algorithm returns SAFE along with an inductive invariant if the property holds. Otherwise, it returns UNSAFE together with a counterexample trace.

PDR maintains a sequence of formulas $\langle R_0, R_1, \ldots, R_N \rangle$, where each R_i is a set of clauses representing an over-approximation of the set of states reachable from the initial

doi:10.6342/NTU202503854

states within at most i transitions. In particular, R_0 corresponds to the initial condition.

In the implementation of the PDR algorithm [4], the data structure used is F_k, which represents the set of cubes last blocked at frame k. The over-approximate reachable states at frame k can be computed as:

$$R_k = \bigwedge_{i=k}^{n} F_i.$$

The algorithm maintains the following invariants throughout execution:

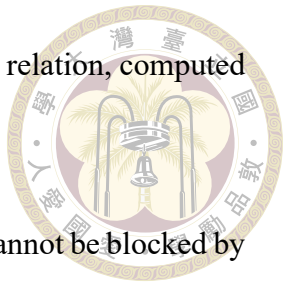
1. $R_0 = I$
2. R_i is a set of clauses, for all $i \geq 1$
3. $R_i \subseteq R_{i+1}$, i.e., $R_i \rightarrow R_{i+1}$, for all $i \geq 1$
4. $R_i \rightarrow P$, for all $0 \leq i < N$, where N is the current length of the trace
5. *image*$(R_i) \rightarrow R_{i+1}$, i.e., R_{i+1} over-approximates the image of R_i

The main function, `PDR Main Loop()`, shown in algorithm 1, serves as the top-level control loop of the PDR algorithm. If there is a proof obligation that can lead to the violation of property P, it attempts to block this cube recursively via the function `recBlockCube()`, shown in algorithm 2. The core procedure to determine whether a cube can be blocked is performed by the function `generalize()`, shown in algorithm 3.

The function `generalize()` issues a SAT query and returns a Boolean value indicating whether the cube can be blocked. Two additional results are passed via reference arguments: `unsat_core` and `pred`. If the cube is blocked, `unsat_core` is assigned a generalized cube that causes the SAT query to return `UNSAT`. Otherwise, `pred` is assigned

doi:10.6342/NTU202503854

a predecessor cube that can reach the current cube under the transition relation, computed using ternary simulation.



The PDR algorithm terminates in one of two cases: (1) if a cube cannot be blocked by `recBlockCube()`, the algorithm returns `UNSAFE` with a counterexample trace that violates the property P; or (2) if `propagateBlockedCubes()` returns `TRUE`, indicating that an inductive invariant proving P has been found.

The function `propagateBlockedCubes()`, shown in algorithm 4, attempts to propagate each cube $c \in F_k$ to F_{k+1}. If a cube c can be propagated from frame k to frame $k+1$, it is removed from F_k and added to F_{k+1}. If, at any frame k, the set F_k becomes empty, meaning all cubes have propagated to F_{k+1}, it implies $R_k = R_{k+1}$, and thus an inductive invariant has been found. In this case, the algorithm terminates with `SAFE`.

Algorithm 1 PDR Main Loop

Input: Transition system T, initial state I, and property P
Output: `SAFE` or `UNSAFE`

```
1:  for n ← 1 to ∞ do
2:      while true do
3:          c ← getBadCube(n)
4:          if c == None then
5:              break
6:          end if
7:          if !recBlockCube(c, n) then
8:              return UNSAFE
9:          end if
10:     end while
11:     newFrame()
12:     if propagateBlockedCubes(n) then
13:         return SAFE
14:     end if
15: end for
```


doi:10.6342/NTU202503854

Algorithm 2 `recBlockCube`

Input: Cube `c`, frame `n`
Output: `TRUE` or `FALSE`

```
1:  PriorityQueue<ProofObligation> q
2:  po ← {cube : c, frame : n}
3:  q.push(po)
4:  while !q.empty() do
5:    po ← q.top()
6:    q.pop()
7:    if po.frame == 0 then
8:      return FALSE
9:    end if
10:   blocked ← generalize(po.cube, po.frame, pred, unsat_core)
11:   if blocked then
12:     addBlockedCube(unsat_core, po.frame)
13:     if po.frame < n then
14:       next_po ← {cube : po.cube, frame : po.frame + 1}
15:       q.push(next_po)
16:     end if
17:   else
18:     q.push(po)
19:     pred_po ← {cube : pred, frame : po.frame - 1}
20:     q.push(pred_po)
21:   end if
22: end while
23: return TRUE
```

Algorithm 3 `generalize`

Input: Cube c, frame f, predecessor `pred`, unsat_core `unsat_core`
Output: `TRUE` or `FALSE`

```
1: if SAT? [R_{f-1}(X) ∧ ¬c(X) ∧ T(X, X') ∧ c'(X')] then
2:   pred ← ternarySim(c, c')
3:   unsat_core ← None
4:   return FALSE
5: else
6:   pred ← None
7:   unsat_core ← get_UNSAT_core()
8:   return TRUE
9: end if
```


doi:10.6342/NTU202503854

Algorithm 4 `propagateBlockedCubes`

Input: Current maximum timeframe n
Output: `TRUE` or `FALSE`

1: **for** $i \leftarrow 1$ to n **do**
2: **for** $c \in F_i$ **do**
3: **if** `generalize`(c, i+1, _, _) **then**
4: `addClause`($\neg c, i+1$)
5: $F_i \leftarrow F_i - c$
6: $F_{i+1} \leftarrow F_{i+1} \cup c$
7: **end if**
8: **end for**
9: **if** $|F_i| = 0$ **then**
10: *// Find inductive invariant* R_i
11: **return** `TRUE`
12: **end if**
13: **end for**
14: **return** `FALSE`

doi:10.6342/NTU202503854

Chapter 3 Motivation

In this chapter, we analyze the behavior of the typical PDR algorithm implemented in ABC [3] and identify key inefficiencies that motivate the development of our enhanced approach. In section 3.1, we observe that while most proof obligations are easy to block, a small number of difficult cases dominate the overall solving time. Furthermore, as shown in section 3.2, these difficult cases typically require only shallow backtracking. Based on these findings, in section 3.3 we derive the motivation for integrating a multi-timeframe framework into the typical PDR algorithm, which we formally propose and develop in chapter 5.

To facilitate the analyses in this chapter, we define the following terms:

- **Critical timeframe**: the timeframe that incurs the highest total solving time across all frames.
- **Proof obligation tree**: a depth-first-search tree representing the blocking process of the root proof obligation. Each node corresponds to a proof obligation, and child nodes represent its predecessors. The root node represents a cube that may lead to a violation of the safety property P in one transition T (the bad cube returned by `getBadCube()` in algorithm 1), and a leaf node represents a cube successfully blocked by the function `generalize()` in algorithm 3.

doi:10.6342/NTU202503854

- **Critical proof obligation tree**: the most time-consuming proof obligation tree among all trees rooted at the critical timeframe.

3.1 Bottleneck Analysis of Typical Property Directed Reachability Implementations

We evaluate the typical PDR algorithm using its implementation in ABC [3] on benchmarks from the Hardware Model Checking Competition (HWMCC). The time limit for each case is set to 3600 seconds. The goal of this evaluation is to identify performance bottlenecks in timeout cases. Since we focus on the most time-consuming parts of the solving process, we analyze the time required to block proof obligation trees at the critical timeframe.

We begin by analyzing the distribution of the solving time of the **critical proof obligation trees**. The analysis includes two categories: solved cases and timeout cases. As shown in figure 3.1, the distributions between these two categories differ significantly. In solved cases, even the most difficult proof obligations at the critical timeframe are relatively quick to block. In contrast, in timeout cases, the most difficult proof obligations require substantially more solving time. This difference in blocking difficulty at the critical timeframe distinguishes timeout cases from solved ones and reveals a key bottleneck.

To further investigate the root cause of this bottleneck, we examine the timeout cases in more detail. For each timeout case, we compare two values in the critical timeframe: the **maximum** solving time (i.e., the solving time of the critical proof obligation tree) and the **90%-rank** solving time. The 90%-rank refers to the value at the 90th percentile in a sorted list of blocking times—that is, 90% of the proof obligations in the critical timeframe

doi:10.6342/NTU202503854

have blocking times less than or equal to this value.

As shown in figure 3.2, most proof obligations, as reflected by the 90%-rank values, are relatively easy to block. On the other hand, a small number of proof obligations, indicated by the max values, are significantly more difficult and dominate the total solving time.

This leads to our first key observation:

Observation 1. *In timeout cases, the overall solving cost is dominated by a small subset of hard-to-block proof obligations.*

To better understand the nature of these difficult cases, we next analyze the structure of the critical proof obligation trees, focusing on the depth of their leaves.

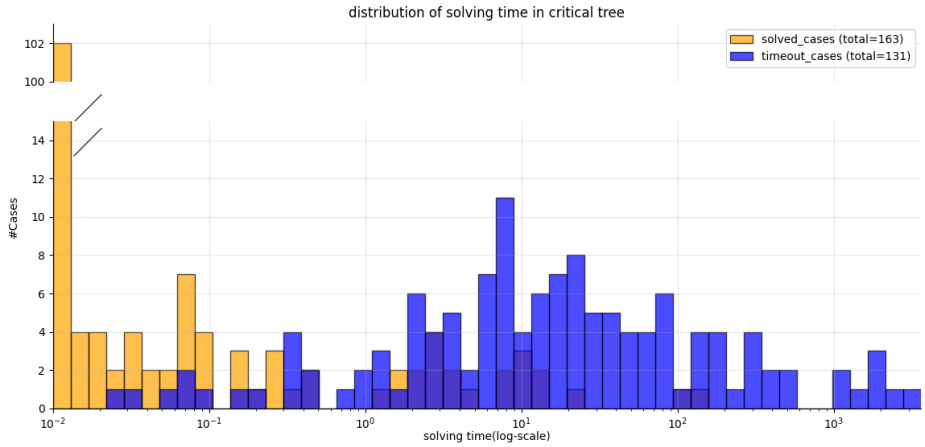


Figure 3.1: Distribution of the maximum blocking time at the critical timeframe for solved and timeout cases.

3.2 Backtracking Depths in Difficult Proof Obligation Trees

To better understand the structure of difficult proof obligations in timeout cases, we analyze the depth of leaves in their corresponding proof obligation trees. In particular, we

doi:10.6342/NTU202503854

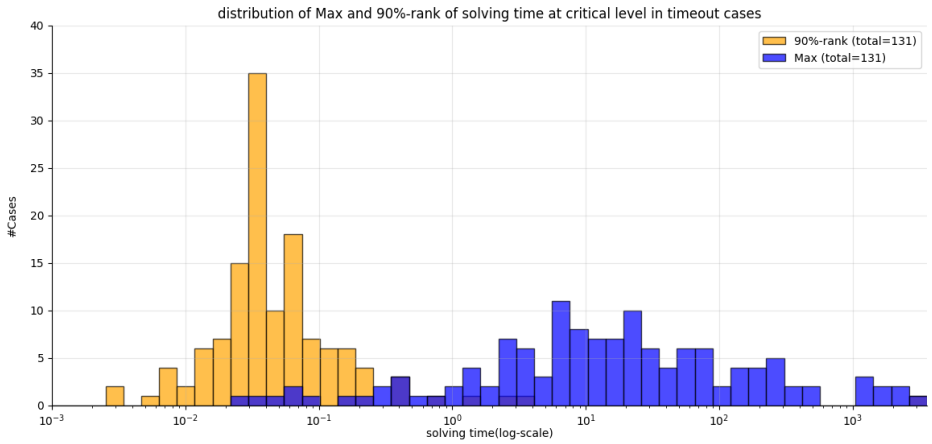


Figure 3.2: Distribution of maximum and 90%-rank blocking times at the critical timeframe in timeout cases.

focus on the **critical proof obligation trees**.

For each timeout case, we measure the depth of each leaf in the critical proof obligation tree, where depth is defined as the distance from the root to the leaf. Since each critical tree typically contains many leaves—and thus many depth values—we use the **median** of these depths to represent the backtracking behavior of the tree. To account for differences in the critical timeframe across cases, we normalize the depth by dividing it by the value of the critical timeframe.

Figure 3.3 shows the histogram of these normalized median values across all timeout cases. The results reveal that, in most cases, the leaves in the critical proof obligation trees are not very deep. In other words, although the overall proof obligation tree is difficult to block, the majority of its branches are resolved after only a few levels of backtracking.

This leads to our second key observation:

Observation 2. *In difficult proof obligation trees, most leaves are located at shallow depths.*

This insight suggests that the information required to block the root proof obligation

doi:10.6342/NTU202503854

is often available within a narrow temporal window.

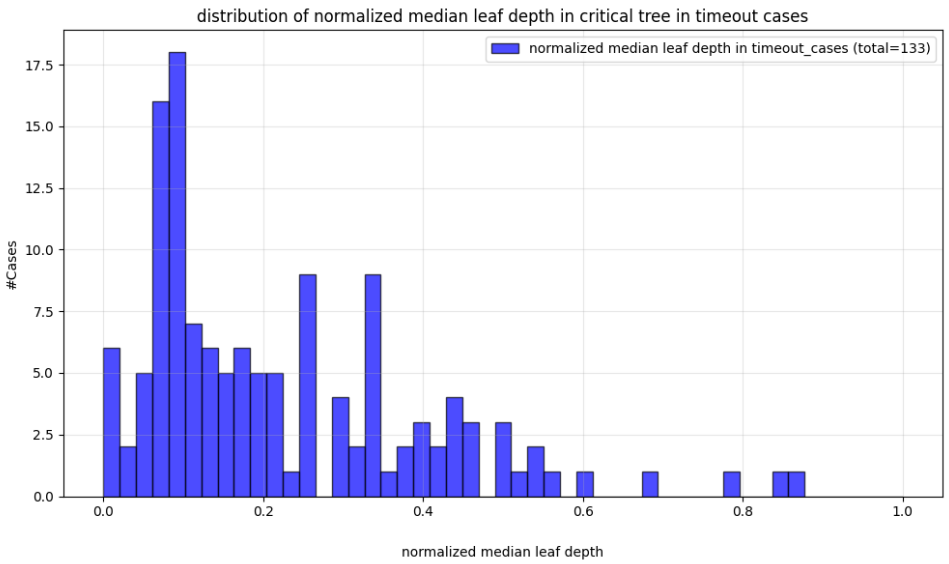


Figure 3.3: Distribution of normalized median leaf depths in critical proof obligation trees across all timeout cases.

3.3 Motivation for an Adaptive Multi-Timeframe Property Directed Reachability Algorithm

Based on the previous analyses, we summarize the following key points:

- **Observation 1:** Critical proof obligation trees are the primary bottleneck in timeout cases.
- **Observation 2:** Although these trees are difficult to block, most of their leaves lie at shallow depths, indicating that a narrow temporal window often suffices to block the root proof obligation.
- **Empirical fact:** Most proof obligations are eventually blocked with `UNSAT` results. Otherwise, the algorithm identifies a counterexample and terminates with an `UNSAFE` result.

doi:10.6342/NTU202503854

These points together suggest that the blocking of difficult proof obligations can potentially be improved by replacing recursive backtracking with a more direct approach. Specifically, we propose to adapt a **multi-timeframe framework** into the typical PDR algorithm, allowing the SAT solver to reason over multiple timeframes simultaneously. Compared to the typical single-timeframe PDR, the use of multi-timeframe transitions introduces more constraints into the SAT queries, which can help conclude `UNSAT` results more efficiently.

To illustrate the intuition behind applying multi-timeframe transitions, consider the example shown in figure 3.4. When the root proof obligation (a) is difficult to block through typical recursive backtracking, the algorithm must explore many cubes (e.g., b, m) before eventually blocking the proof obligation a. If we instead allow reasoning across multiple timeframes, it may become possible to skip intermediate cubes, thereby reducing the number of recursive calls and improving overall blocking efficiency.

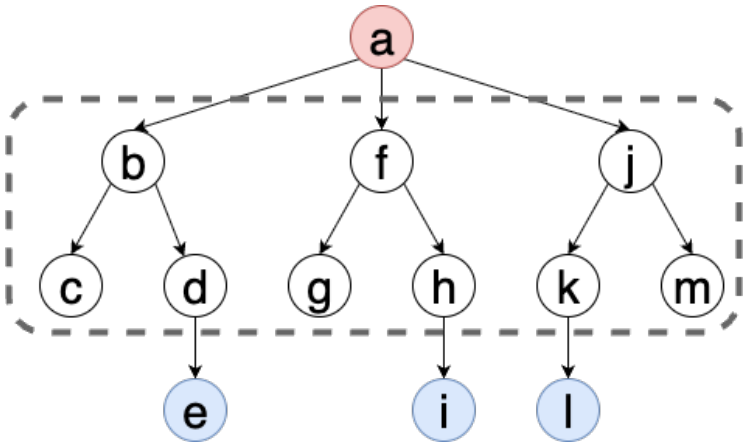


Figure 3.4: Illustration of a difficult proof obligation tree rooted at cube a. In typical PDR, the algorithm recursively explores intermediate cubes (white nodes) to reach leaf cubes (blue), which are eventually blocked. With multi-timeframe transitions, the solver can leverage information across multiple timeframes to bypass intermediate steps and block the root cube more efficiently.

doi:10.6342/NTU202503854

Chapter 4 The Challenges of Multi-Timeframe PDR

In this chapter, we examine the challenges involved in extending PDR to support multi-timeframe expansion. While the idea of applying unrolling within the PDR framework may initially seem straightforward, prior attempts—such as [10]—have reported incompleteness in their approaches. These limitations highlight the subtle but critical difficulties in preserving correctness, completeness, and efficiency when going beyond single-step transitions.

To make these challenges concrete, we begin in section 4.1 by introducing the necessary notation and presenting a naïve version of multi-timeframe PDR that closely mirrors typical PDR. We then analyze, in section 4.2, how this small change can break correctness and completeness. Section 4.3 discusses the efficiency-related challenges that arise even when correctness is preserved. Finally, section 4.4 summarizes the key insights of this chapter and sets the stage for the refined algorithm proposed in chapter 5.

doi:10.6342/NTU202503854

4.1 Notations and a Naïve Algorithm for Multi-Timeframe PDR

This section presents the notations used throughout this chapter and introduces a naïve attempt to extend PDR with multi-timeframe expansion. We begin by reviewing typical PDR notations and then introduce the additional notations required to describe timeframe-unrolled transitions. These extended notations will also serve as the foundation for the proposed multi-timeframe PDR algorithm in chapter 5. We then present a pseudocode listing of a multi-timeframe PDR algorithm that closely resembles typical PDR, differing only in how it formulates generalization queries using multiple steps.

4.1.1 Notations

We begin by introducing the notations used to describe both the typical and multi-timeframe variants of PDR. Table 4.1 lists the symbols used in typical PDR, while Table 4.2 defines the additional notation required to express timeframe-unrolled transitions. These extended notations play a central role in both the naïve and the proposed multi-timeframe PDR algorithms.

Table 4.1: Typical Notations Used in PDR

Symbol	Meaning
I	Initial state, represented as a cube (a conjunction of literals)
P	Safety property to be verified
T	Transition relation
R_k	Over-approximation of the states reachable from I within at most k steps
$\langle s, i \rangle$	Proof obligation at frame i, where s is a cube (state) that may lead to violation of the safety property P
s'	Cube s expressed over next-state variables

doi:10.6342/NTU202503854

Table 4.2: Additional Notations for Multi-Timeframe PDR

Symbol	Meaning
$s^{(j)}$	Cube s expressed over variables at the j-th next state; $s^{(0)} \equiv s, s^{(1)} \equiv s'$
$R_k^{(j)}$	Over-approximation of the states reachable from I within at most k steps, expressed over variables at the j-th next state; $R_k^{(0)} \equiv R_k$

4.1.2 A Naïve Multi-Timeframe Property Directed Reachability Algorithm

This subsection presents a naïve extension of the typical PDR algorithm—`PDR`$_{\texttt{multi}}$`-naïve()` in algorithm 5. As shown in the pseudocode, this version closely follows the structure of typical PDR in algorithm 1, with the only difference lying in the recursive blocking function `recBlockCube`$_{\texttt{multi}}$`-naïve()` in algorithm 6, which adapts generalization using multi-timeframe expansion performed by `generalize`$_{\texttt{multi}}$`()`, as shown in algorithm 7.

The function `generalize`$_{\texttt{multi}}$`()` is conceptually aligned with its typical counterpart `generalize()` in algorithm 3, with the primary difference lying in the SAT query formulation. Specifically, `generalize`$_{\texttt{multi}}$`()` checks:

$$\text{SAT?}\left[\neg c(X) \wedge \left(\bigwedge_{i=0}^{\ell-1} R^{(i)}_{f-\ell+i}(X^{(i)}) \wedge T(X^{(i)}, X^{(i+1)})\right) \wedge c^{(\ell)}(X^{(\ell)})\right],$$

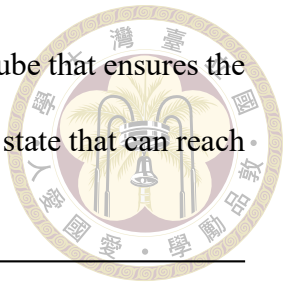
where f is the current frame, ℓ is the length of timeframe expansion, and c is the cube to be blocked. In contrast, the typical `generalize()` checks:

$$\text{SAT?}\left[R_{f-1}(X) \wedge \neg c(X) \wedge T(X, X') \wedge c'(X')\right].$$

Both functions return a Boolean indicating whether the cube can be blocked. The associated results are returned via two reference arguments: `unsat_core` and `pred`. If

doi:10.6342/NTU202503854

the cube is blocked, `unsat_core` is assigned a generalized blocking cube that ensures the above SAT query is `UNSAT`. Otherwise, `pred` represents a predecessor state that can reach the current cube under the transition relation.



Algorithm 5 $\text{PDR}_{\text{multi}}$-naïve Main Loop

Input: Transition system T, initial state I, and property P
Output: `SAFE` or `UNSAFE`

```
1:  for n ← 1 to ∞ do
2:    while true do
3:      c ← getBadCube(n)
4:      if c == None then
5:        break
6:      end if
7:      if !recBlockCube_multi-naïve(c, n) then
8:        return UNSAFE
9:      end if
10:   end while
11:   newFrame()
12:   if propagateBlockedCubes(n) then
13:     return SAFE
14:   end if
15: end for
```

4.2 Challenges on Correctness and Completeness

In this section, we show that the naïve multi-timeframe PDR algorithm may incorrectly block states that are actually reachable. Specifically, we focus on the case where the call to $\texttt{generalize()}_{\texttt{multi}}$ (line 11 in algorithm 6) returns `TRUE`, indicating that the algorithm proceeds to the blocking stage. We then examine whether it is valid to add the resulting blocking cube to the reachability sets R. For clarity, this analysis is divided into two parts: in section 4.2.1, we consider the case where the blocking cube is the original proof obligation c, and show that this is a valid blocking step; in section 4.2.2, we analyze the practical case where the blocking cube is the generalized cube c_{gen}, which corresponds to the `unsat_core` returned by $\texttt{generalize}_{\texttt{multi}}\texttt{()}$ in algorithm 6, and demonstrate that

doi:10.6342/NTU202503854

Algorithm 6 recBlockCube$_{\text{multi}}$-naïve

Input: Cube c, frame n
Output: TRUE or FALSE

```
1: PriorityQueue<ProofObligation> q
2: ProofObligation po ← {cube :c, frame :n}
3: q.push(po)
4: while q.empty() do
5:   po ← q.top()
6:   q.pop()
7:   if po.frame == 0 then
8:     return FALSE
9:   end if
10:  timeframe_expansion ← decide_timeframe_expansion()
11:  blocked ← generalize_multi(po.cube, po.frame, timeframe_expansion,
     pred, unsat_core)
12:  if blocked then
13:    addBlockedCube(unsat_core, po.frame)
14:    if po.frame < n then
15:      ProofObligation next_po ← {cube: po.cube, frame: po.frame + 1}
16:      q.push(next_po)
17:    end if
18:  else
19:    q.push(po)
20:    ProofObligation pred_po ← {cube: pred, frame: po.frame -
       timeframe_expansion}
21:    q.push(pred_po)
22:  end if
23: end while
24: return TRUE
```

Algorithm 7 generalize$_{\text{multi}}$

Input: Cube c, frame f, timeframe_expansion l, predecessor pred, unsat_core unsat_core
Output: TRUE or FALSE

```
1: if SAT? [¬c(X) ∧ (⋀_{i=0}^{ℓ-1} R^(i)_{f-ℓ+i}(X^(i)) ∧ T(X^(i), X^(i+1))) ∧ c^(ℓ)(X^(ℓ))] then
2:   pred ← ternarySim(c, c^(l))
3:   unsat_core ← None
4:   return FALSE
5: else
6:   pred ← None
7:   unsat_core ← get_UNSAT_core()
8:   return TRUE
9: end if
```


doi:10.6342/NTU202503854

this blocking step can lead to unsoundness.

4.2.1 Valid Blocking When the Blocking Cube is an Original Proof Obligation

This subsection considers a simplified case in which the blocking cube is the original proof obligation c. We show that in this case, adding $\neg c$ as a blocking cube to the reachable frame sets R_k for $k \leq f$ is valid and does not eliminate any state that is actually reachable from the initial state.

To justify this, we analyze the SAT query used in multi-timeframe generalization. Suppose the following SAT query is `UNSAT`:

$$\text{SAT?} \left[\neg c(X) \wedge \left(\bigwedge_{i=0}^{\ell-1} R^{(i)}_{f-\ell+i}(X^{(i)}) \wedge T(X^{(i)}, X^{(i+1)}) \right) \wedge c^{(\ell)}(X^{(\ell)}) \right],$$

we now analyze whether it is sound to add $\neg c$ to R_k for each $k \leq f$.

We separate the analysis into two cases based on the value of k:

Case 1: $k \geq \ell$**.** Since $R_i \subseteq R_j$ for all $i < j$ (a PDR invariant), and $k \leq f$, we have:

$$R_{k-\ell+i} \subseteq R_{f-\ell+i} \quad \text{for all } i \in [0, \ell-1]$$

Thus, the formula

$$\text{SAT?} \left[\neg c(X) \wedge \left(\bigwedge_{i=0}^{\ell-1} R^{(i)}_{k-\ell+i}(X^{(i)}) \wedge T(X^{(i)}, X^{(i+1)}) \right) \wedge c^{(\ell)}(X^{(\ell)}) \right],$$

doi:10.6342/NTU202503854

is also UNSAT. Therefore, $\neg c$ is a valid blocking cube at frame k, and it does not exclude any reachable states.

Case 2: $k < \ell$**.** In this case, some indices $k - \ell + i$ would be negative, making the above SAT query undefined. Instead, we reason based on the semantics of PDR. If cube c cannot be blocked in R_k, then there exists a path from the initial state I to a state in c in exactly k steps. Since $k < \ell \leq f$, and f is the frame in which c was first observed, this implies that there exists a counterexample path shorter than the current depth, contradicting the assumption that all shorter paths have already been blocked. Hence, c must be blockable at frame k, and adding $\neg c$ to R_k is valid.

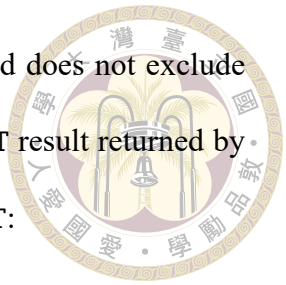
In both cases, blocking c preserves soundness and does not eliminate any state reachable from the initial state within the first f frames.

4.2.2 Invalid Blocking When the Blocking Cube is a Generalized Cube

This subsection examines the practical case in which the blocking cube is the generalized cube c_{gen}, returned as the `unsat_core` by the function $\texttt{generalize}_{\texttt{multi}}\texttt{()}$ at line 11 in algorithm 6. As in typical PDR, the actual blocking cube added to the reachability sets is c_{gen}. We show that it is unsound to add $\neg c_{\text{gen}}$ to the reachability sets R_k for $k \leq f$. In particular, such blocking may exclude states that are reachable from the initial state, violating the definition of R_k as an over-approximation of the states reachable within k steps.

We again divide the analysis into two cases: one where $k \geq \ell$, and another where $k < \ell$, where ℓ is the length of the timeframe expansion.

doi:10.6342/NTU202503854



Case 1: $k \geq \ell$**.** In this case, blocking $\neg c_{\text{gen}}$ at frame k is valid and does not exclude any reachable state. The justification follows directly from the `UNSAT` result returned by $\texttt{generalize}_{\texttt{multi}}\texttt{()}$, which verifies the following SAT query is `UNSAT`:

$$\text{SAT?}\left[\neg c_{gen}(X) \wedge \left(\bigwedge_{i=0}^{\ell-1} R_{f-\ell+i}^{(i)}(X^{(i)}) \wedge T(X^{(i)}, X^{(i+1)})\right) \wedge c_{gen}^{(\ell)}(X^{(\ell)})\right].$$

Given the PDR invariant that $R_{k-\ell+i} \subseteq R_{f-\ell+i}$ for $k \leq f$, we can substitute $R_{k-\ell+i}$ for the original frame references and conclude the following SAT query is also `UNSAT`:

$$\text{SAT?}\left[\neg c_{gen}(X) \wedge \left(\bigwedge_{i=0}^{\ell-1} R_{k-\ell+i}^{(i)}(X^{(i)}) \wedge T(X^{(i)}, X^{(i+1)})\right) \wedge c_{gen}^{(\ell)}(X^{(\ell)})\right].$$

Thus, $\neg c_{\text{gen}}$ is a valid blocking cube at frames $k \geq \ell$.

Case 2: $k < \ell$**.** This case reveals a fundamental flaw in the naïve algorithm. Note that the blocking cube c_{gen} is a generalization of the original proof obligation c, such that $c \subseteq c_{\text{gen}}$. Therefore, c_{gen} may cover states unrelated to the original cube c. Specifically, there may exist a state $c_{\text{reach}} \subseteq c_{\text{gen}}$ such that $c_{\text{reach}} \wedge c = \emptyset$, and c_{reach} may still be reachable from the initial state within k steps. As a result, we cannot block c_{reach} from R_k.

Because $c_{\text{reach}} \subseteq c_{\text{gen}}$, it would be excluded by adding $\neg c_{\text{gen}}$ to R_k. This incorrectly eliminates a reachable state, violating the PDR invariant that each R_k must over-approximate the states reachable from the initial condition within k steps. In this way, the algorithm breaks soundness by blocking states that should not be excluded at that frame.

A concrete example is illustrated in Figure 4.1. Let the original proof obligation cube be $c = ab$, and let the generalized blocking cube be $c_{\text{gen}} = a$, returned by the function $\texttt{generalize}_{\texttt{multi}}\texttt{()}$. Here, c_{gen} subsumes a state $c_{\text{reach}} = a\neg b$ that is not part of

doi:10.6342/NTU202503854

the original obligation and does not lead to a violation of the property. However, c_{reach} is reachable from the initial state and would be incorrectly removed from R_k if $\neg c_{\text{gen}}$ is blocked. This exemplifies how over-generalization leads to unsound blocking in the naïve multi-timeframe PDR algorithm.

It is instructive to compare this behavior with that of typical PDR. The typical PDR algorithm can be seen as a special case of multi-timeframe PDR in which the timeframe expansion ℓ is fixed to 1. Under this condition, all blocking operations fall into the case where $k \geq \ell$, making it always sound to block the generalized cube c_{gen}. As a result, the issue of over-blocking due to generalization does not arise in the typical single-timeframe PDR.

To address this soundness issue in the naïve multi-timeframe PDR, we will present a refined algorithm in chapter 5. The proposed multi-timeframe PDR algorithm incorporates an additional refinement mechanism to ensure that only unreachable states are blocked, thereby preserving the correctness and soundness of the reachability sets.

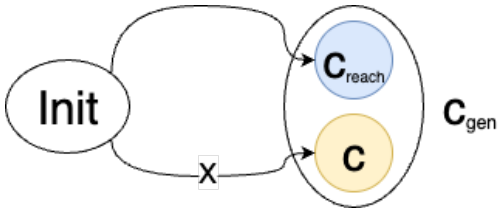


Figure 4.1: Illustration of unsound blocking in the naïve algorithm. Generalizing $c = ab$ to $c_{\text{gen}} = a$ unintentionally blocks reachable state $c_{\text{reach}} = a\neg b$. Note that cube c is unreachable from the initial state.

4.3 Challenges on Efficiency

This section focuses exclusively on the efficiency challenges associated with multi-timeframe PDR and does not address issues of correctness or soundness, which are dis-

doi:10.6342/NTU202503854

cussed separately in section 4.2.

A key factor that significantly affects the performance of multi-timeframe PDR is the selection of the timeframe expansion length ℓ. Depending on the choice of ℓ, the algorithm can exhibit very different behaviors, potentially leading to inefficiency or ineffectiveness.

To illustrate this, consider two extreme cases:

- When the expansion length $\ell = 1$, the algorithm essentially reduces to typical PDR. As discussed in chapter 3, typical PDR struggles with certain classes of problems due to excessive solving time on a single hard-to-block proof obligation.
- Conversely, when $\ell = f$, where f is the current frame, the algorithm behaves similarly to bounded model checking (BMC), which unrolls the transition relation all the way back to the initial state. While this approach may be helpful for finding counterexamples, it becomes ineffective in proving safety, where BMC typically performs poorly.

In conclusion, a moderate and dynamically selected expansion length ℓ, adapted to the current proof obligation and solving context, is essential for improving efficiency. Our proposed algorithm incorporates such a mechanism, which is presented and discussed in detail in section 5.3.

4.4 Summary

This chapter has examined the fundamental challenges in extending PDR to support multi-timeframe transition. While the idea of integrating timeframe expansion into

doi:10.6342/NTU202503854

the PDR framework may initially seem straightforward, we have shown that a naïve approach introduces significant issues. Specifically, we demonstrated that multi-timeframe generalization can break the soundness of the algorithm by blocking reachable states, and that improper selection of timeframe expansion length can severely impact efficiency.

These findings highlight that the integration of multi-timeframe reasoning into PDR is non-trivial. A carefully designed algorithm must address both the correctness and efficiency challenges identified in this chapter. In chapter 5, we present our refined multi-timeframe PDR algorithm, which incorporates a dedicated refinement mechanism and dynamic timeframe selection strategy to overcome these limitations.

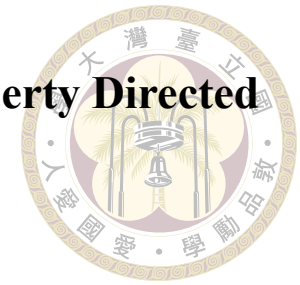
doi:10.6342/NTU202503854

Chapter 5 The Proposed Multi-Timeframe Property Directed Reachability Algorithm

In this chapter, we introduce the proposed multi-timeframe PDR algorithm, which adapts multi-timeframe transition to the PDR framework and addresses the correctness and efficiency challenges identified in chapter 4. Section 5.1 describes the algorithm in detail, including its core procedures and an illustrative example. Section 5.2 proves the correctness and completeness of the algorithm, directly addressing the over-blocking issues discussed in section 4.2. Section 5.3 focuses on improving efficiency through dynamic timeframe expansion, specifically targeting the issues outlined in section 4.3 regarding the selection of appropriate timeframes for different cases.

doi:10.6342/NTU202503854

5.1 Overview of the Multi-Timeframe Property Directed Reachability Algorithm



In this section, we present the proposed multi-timeframe PDR algorithm. Section 5.1.1 provides a detailed explanation of the algorithm, while section 5.1.2 illustrates its behavior through a concrete example.

5.1.1 Algorithm Description

This section presents the multi-timeframe PDR algorithm. Our approach extends the typical PDR framework by introducing dynamic timeframe expansion and supporting utility functions to ensure correctness and completeness.

We present the main function, $\texttt{PDR}_{\texttt{multi}}()$, and the recursive cube blocking mechanism, $\texttt{recBlockCube}_{\texttt{multi}}()$, as shown in algorithms 8 and 9. These are largely consistent with their counterparts in the typical PDR framework, `PDR()` (algorithm 1) and `recBlockCube()` (algorithm 2), with key differences highlighted to support the multi-timeframe PDR.

We also introduce two specialized components, $\texttt{blockCube}_{\text{multi}}()$ and `recRefine()` in algorithms 10 and 11, which support the adaptation of the multi-timeframe framework.

To better illustrate the structure and interaction of these components, we provide a brief description of each function below.

- $\texttt{PDR}_{\texttt{multi}}()$ (see algorithm 8): This is the top-level control loop of the multi-timeframe PDR algorithm. It follows the typical PDR workflow by iteratively attempting to

doi:10.6342/NTU202503854

block proof obligations. In our multi-timeframe framework, this task is delegated to `recBlockCube`$_{\texttt{multi}}$`()`, replacing the use of `recBlockCube()` in the typical PDR.

- **`recBlockCube`$_{\texttt{multi}}$`()`** (see algorithm 9): This function is responsible for recursively blocking a given proof obligation and closely follows the structure of the typical `recBlockCube()` function in algorithm 2. Its control flow is illustrated in figure 5.1, where key differences from typical PDR are highlighted in color. The adaptive control logic integrates timeframe expansion and multi-timeframe reasoning: Line 4 initializes the `timeframe_expansion` variable, and lines 11 – 17 implement logic to determine when an expansion is needed. If progress is stalled, the function clears pending proof obligations and restarts with an increased expansion value (corresponding to the yellow region in figure 5.1). Lines 18–21 invoke `blockCube`$_{\texttt{multi}}$`()` to handle the actual multi-timeframe blocking (blue region in figure 5.1).

- **`blockCube`$_{\texttt{multi}}$`()`** (see algorithm 10): This function handles the core logic of multi-timeframe cube blocking. It begins by calling `generalize`$_{\texttt{multi}}$`()`, as shown in algorithm 7, to determine whether the given cube can be blocked. This function is the same as the one introduced in section 4.1.2, and is conceptually aligned with its typical counterpart `generalize()`, with the key difference lying in the formulation of the SAT query. If blocking fails (lines 7–9), the function behaves similarly to typical PDR: both the original `cube` and its predecessor `pred` are pushed into the proof obligation queue. However, unlike in typical PDR, the predecessor's frame index is set to `frame - timeframe_expansion` rather than `frame - 1`, to account for multi-timeframe reasoning. If blocking succeeds, the function applies a refinement step (lines 3–4) before adding the blocking cube. This refinement is essential for

doi:10.6342/NTU202503854

preserving correctness in the multi-timeframe setting. The key refinement step is done by the function `recRefine()`.

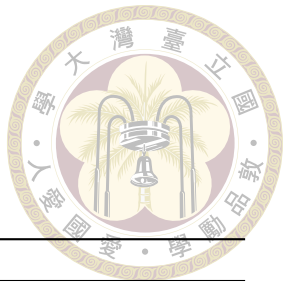
- **`recRefine()`** (see algorithm 11): This function performs the refinement step. It recursively refines the reachability set R to ensure that the target cube is unreachable from $R_{\texttt{frame}}$. This is done by repeatedly calling the typical `generalize()` function, which attempts to block the cube using single-timeframe transitions. The function returns a generalized cube that is guaranteed to be unreachable from $R_{\texttt{frame}}$. The final blocking cube is then computed as the conjunction of two components: the `unsat_core` obtained from $\texttt{generalize}_{\texttt{multi}}\texttt{()}$ and the `unsat_core_refine` returned by `recRefine()`. We go through the full procedure here to clarify its behavior; in the subsequent correctness proof in section 5.2.2, we demonstrate that this refinement step is essential to preserving the soundness of the multi-timeframe PDR algorithm.

Algorithm 8 $\text{PDR}_{\text{multi}}$ Main Loop

Input: Transition system T, initial state I, and property P
Output: `SAFE` or `UNSAFE`

```
1:  for n ← 1 to ∞ do
2:    while true do
3:      c ← getBadCube(n)
4:      if c == None then
5:        break
6:      end if
7:      if !recBlockCube_multi(c, n) then
8:        return UNSAFE
9:      end if
10:   end while
11:   newFrame()
12:   if propagateBlockedCubes() then
13:     return SAFE
14:   end if
15: end for
```


doi:10.6342/NTU202503854



Algorithm 9 `recBlockCube`$_{\texttt{multi}}$`(c, n)`

Input: Cube c, frame n
Output: `TRUE` or `FALSE`

```
1: PriorityQueue<ProofObligation> q
2: ProofObligation po ← {cube : c, frame : n}
3: q.push(po)
4: timeframe_expansion ← None
5: while !q.empty() do
6:    po ← q.top()
7:    q.pop()
8:    if po.frame == 0 then
9:       return FALSE
10:   end if
11:   if is_time_to_expand() then
12:      q.clear()
13:      po ← {cube : c, frame : n}
14:      q.push(po)
15:      timeframe_expansion ← expand(timeframe_expansion, n)
16:      continue
17:   end if
18:   if timeframe_expansion ≠ None and po.frame == n then
19:      blockCube_multi(po, timeframe_expansion, q)
20:      continue
21:   end if
22:   blocked ← generalize(po.cube, po.frame, pred, unsat_core)
23:   if blocked then
24:      addBlockedCube(unsat_core, po.frame)
25:      if po.frame < n then
26:         ProofObligation next_po ← {cube : po, frame : po.frame + 1}
27:         q.push(next_po)
28:      end if
29:   else
30:      q.push(po)
31:      ProofObligation pred_po ← {cube : pred, frame : po.frame − 1}
32:      q.push(pred_po)
33:   end if
34: end while
35: return TRUE
```


doi:10.6342/NTU202503854

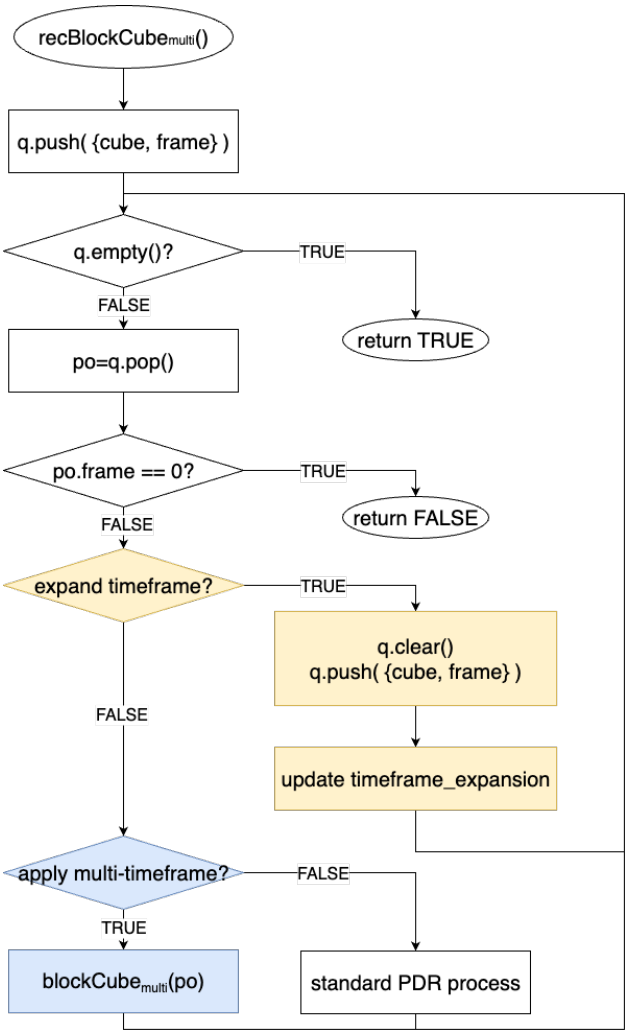


Figure 5.1: Flowchart of `recBlockCube`$_{\texttt{multi}}$`()` in algorithm 9. This function implements the core adaptive control logic in our proposed multi-timeframe PDR algorithm. The yellow region represents the initialization and dynamic timeframe expansion logic, where the algorithm decides whether to reset and grow the timeframe window. The blue region indicates the proposed multi-timeframe blocking process, applied selectively based on current conditions. The remaining structure follows the typical control flow in the typical PDR. Together, these colored regions highlight the key enhancements over typical single-timeframe PDR.

doi:10.6342/NTU202503854

Algorithm 10 $\texttt{blockCube}_{\texttt{multi}}$

Input: po, timeframe_expansion, q

```
1: blocked ← generalize_multi(po.cube, po.frame, timeframe_expansion,
   pred, unsat_core)
2: if blocked then
3:    unsat_core_refine ← recRefine(po.cube, timeframe_expansion - 1)
4:    blocking_cube ← unsat_core_refine ∧ unsat_core
5:    addBlockedCube(blocking_cube, po.frame)
6: else
7:    q.push(po)
8:    ProofObligation pred_po ← { cube: pred, frame: po.frame -
      timeframe_expansion }
9:    q.push(pred_po)
10: end if
```

Algorithm 11 recRefine

Input: Cube cube, frame frame
Output: A generalized cube that does not intersect with $R_{\texttt{frame}}$

```
1: assert(frame ≥ 1)
2: while !generalize(cube, frame, pred, unsat_core) do
3:    recRefine(pred, frame - 1)
4: end while
5: addBlockedCube(unsat_core, frame)
6: return unsat_core
```


doi:10.6342/NTU202503854

5.1.2 Illustrative Example

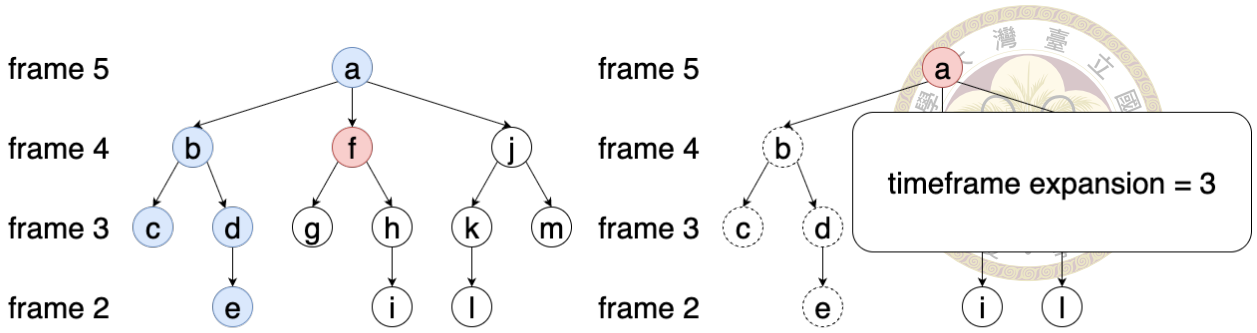
To aid understanding of the multi-timeframe PDR algorithm, this section presents a concrete example. We focus on the initialization process and the outcome of the multi-timeframe cube-blocking procedure. The example illustrates the core intuition behind how our multi-timeframe framework improves the efficiency of blocking proof obligations that may be difficult to resolve using typical single-timeframe propagation.

Figure 5.2a shows a proof obligation tree. The root node a is a cube that, via one transition, leads to a violation of the safety property P. The traversal of the tree follows alphabetical order. In this scenario, assume that in the function $\texttt{recBlockCube}_{\texttt{multi}}()$, nodes b through e have already been visited and successfully blocked.

However, when visiting node f, the algorithm detects that the associated proof obligation tree rooted at a is difficult to block, based on the solving time. As a result, it triggers the initialization procedure (highlighted in yellow in figure 5.1). After this reinitialization, the algorithm updates the timeframe expansion parameter; assume that `timeframe_expansion` is set to 3. In the next iteration, the control flow moves into the multi-timeframe blocking logic (blue region in figure 5.1).

Figure 5.2b illustrates the updated proof obligation tree structure after applying the multi-timeframe framework. Notably, previously blocked nodes b through e are not revisited. Moreover, in the subsequent recursive blocking process from node a, only nodes i and l are visited due to the expanded timeframe. Compared with the original single-timeframe PDR, the adoption of the multi-timeframe framework reduces the number of visited proof obligations, thereby improving the efficiency of the blocking process.

doi:10.6342/NTU202503854



(a) Typical PDR proof obligation tree

(b) Modified tree after applying multi-timeframe blocking

Figure 5.2: Comparison of proof obligation trees before and after multi-timeframe expansion

5.2 Correctness and Completeness

In this section, we first state the invariants maintained by the multi-timeframe PDR algorithm in section 5.2.1. Then, we prove the correctness of the algorithm in section 5.2.2, followed by the proof of completeness in section 5.2.3.

5.2.1 Invariant Property of Multi-Timeframe Property Directed Reachability

In this section, we state the invariant properties maintained by the multi-timeframe PDR algorithm. These invariants are similar to those used in the typical PDR framework [4]. We list the invariant properties below:

1. $R_0 = I$
2. R_i is a set of clauses, for all $i \geq 1$
3. $R_i \subseteq R_{i+1}$, i.e., $R_i \rightarrow R_{i+1}$, for all $i \geq 1$

doi:10.6342/NTU202503854

4. $R_i \rightarrow P$, for all $i < N$, where N is the current length of the trace

5. R_i is an over-approximation of the exact reachable states from the initial state within i steps. Formally,

$$R_{i,\text{exact}} := \bigcup_{j=0}^{i} \text{image}^j(I), \quad \text{and} \quad R_{i,\text{exact}} \rightarrow R_i$$

Note that the fifth invariant is a relaxed version of the one used in typical PDR. In typical PDR, the invariant requires that R_{i+1} over-approximates the image of R_i, i.e., $\text{image}(R_i) \rightarrow R_{i+1}$. In contrast, our formulation only requires that R_i over-approximates all reachable states within i steps.

In the remainder of this section, we show that the above invariants are preserved by the multi-timeframe PDR algorithm. Specifically, we demonstrate that if the invariants hold prior to one iteration of the main loop in algorithm 8, they continue to hold afterward.

After one iteration of the main loop in algorithm 8, it is straightforward to see that Invariants 1, 2, and 4 continue to hold. For Invariant 3, whenever a blocking cube is added to R_k, it is also added to all prior frames R_i for $i < k$. This ensures that $R_i \subseteq R_{i+1}$ is maintained. We now focus on the maintenance of **Invariant 5**.

Invariant 5: To show that each R_k remains an over-approximation of the exact reachable states within k steps, we examine the two situations in which a clause $\neg c$ is added to some R_k:

1. **Case 1:** `propagateBlockedCubes()` adds a clause to R_k if cube c can be propagated forward from frame $k-1$. Specifically, this occurs when

$$\texttt{generalize}(c, k, \texttt{pred}, \texttt{unsat\_core}) = \texttt{TRUE},$$

doi:10.6342/NTU202503854

which implies that c is unreachable from R_{k-1}. Hence,

$$R_{k,\text{exact}} \cap c = \emptyset,$$

and the addition of $\neg c$ to R_k preserves Invariant 5.

2. **Case 2:** The function $\texttt{generalize}_{\texttt{multi}}\texttt{(cube, f, l, pred, unsat\_core)}$ returns `TRUE` in algorithm 10, triggering a blocking clause addition to frame k, where $k \leq f$. We analyze this by separating into two subcases:

   - **Subcase 2.1 ($1 \leq k < l$):** In this range, we consider the cube `unsat_core_refine` returned by `recRefine(cube, l-1)`. Upon returning, the function ensures that

     $$R_{l-1} \cap \texttt{unsat\_core\_refine} = \emptyset.$$

     Since $k < l$, by Invariant 3, $R_k \subseteq R_{l-1}$, which implies

     $$R_k \cap \texttt{unsat\_core\_refine} = \emptyset.$$

     By Invariant 5, $R_{k,\text{exact}} \subseteq R_k$, so

     $$R_{k,\text{exact}} \cap \texttt{unsat\_core\_refine} = \emptyset.$$

     The final blocking cube is computed as

     $$\texttt{blocking\_cube} = \texttt{unsat\_core\_refine} \wedge \texttt{unsat\_core},$$

     which gives

     $$R_{k,\text{exact}} \cap \texttt{blocking\_cube} = \emptyset.$$

doi:10.6342/NTU202503854

Note that the refinement process performed by the function `recRefine()` (see algorithm 11) is essential to ensure soundness in Subcase 2.1. It guarantees that the blocking cube excludes all reachable states from R_{l-1}, and consequently from any R_k for $k < l$. This step ensures that Invariant 5 is upheld even when multi-timeframe blocking is applied.

- **Subcase 2.2 ($l \leq k \leq f$):** Given that $\texttt{generalize}_{\texttt{multi}}()$ returns `TRUE`, the following SAT query is unsatisfiable:

$$\text{SAT?} \left[\neg \texttt{unsat\_core} \wedge \left(\bigwedge_{i=0}^{l-1} R_{f-l+i}^{(i)} \wedge T^{(i)} \right) \wedge \texttt{unsat\_core}^{(l)} \right] \text{ is UNSAT.}$$

By Invariant 3, $R_{k-l+i} \subseteq R_{f-l+i}$, and thus the following query is also unsatisfiable:

$$\text{SAT?} \left[\neg \texttt{unsat\_core} \wedge \left(\bigwedge_{i=0}^{l-1} R_{k-l+i}^{(i)} \wedge T^{(i)} \right) \wedge \texttt{unsat\_core}^{(l)} \right] \text{ is UNSAT.}$$

Therefore,

$$R_{k,\text{exact}} \cap \texttt{unsat\_core} = \emptyset.$$

The final blocking cube is computed as

$$\texttt{blocking\_cube} = \texttt{unsat\_core\_refine} \wedge \texttt{unsat\_core},$$

which gives

$$R_{k,\text{exact}} \cap \texttt{blocking\_cube} = \emptyset.$$

In both cases, the blocking cube c added to frame k does not intersect with the exact

doi:10.6342/NTU202503854

reachable states at frame k, i.e.,

$$R_{k,\text{exact}} \cap c = \emptyset.$$

Therefore, **Invariant 5** continues to hold.

In conclusion, all five invariant properties are maintained throughout the execution of the multi-timeframe PDR algorithm.

5.2.2 Proof of Correctness

To prove the correctness of the multi-timeframe PDR algorithm, we need to show the following:

1. If the algorithm returns `SAFE`, the property P holds for all states reachable from the initial state I.
2. If the algorithm returns `UNSAFE` with a counterexample, the counterexample trace is a valid sequence of states from I to a bad state that violates P.

Case 1: Algorithm Returns `SAFE`. The algorithm returns `SAFE` only if the function `propagateBlockedCubes()` returns `TRUE` in algorithm 8. This indicates that there exists a frame index i such that all clauses in R_i can be propagated to R_{i+1}, and moreover, $R_i = R_{i+1}$.

From the propagation condition, we have:

$$R_i \wedge T \rightarrow R'_{i+1}.$$

doi:10.6342/NTU202503854

Given that $R_i = R_{i+1}$, this simplifies to:

$$R_i \wedge T \rightarrow R_i'.$$

In addition, by Invariant 1, Invariant 3, and Invariant 4, we know:

$$I \subseteq R_i \quad \text{and} \quad R_i \rightarrow P.$$

Together, these three conditions:

$$I \subseteq R_i, \quad R_i \wedge T \rightarrow R_i', \quad R_i \rightarrow P$$

show that R_i is an inductive invariant that proves the safety property P. Thus, the algorithm correctly returns `SAFE`.

Case 2: Algorithm Returns `UNSAFE`. From **Invariant 5** in section 5.2.1, we know that each R_i is an over-approximation of the exact reachable states within i steps:

$$R_{i,\text{exact}} \subseteq R_i.$$

Because the multi-timeframe PDR algorithm never blocks any reachable state, if a counterexample trace is discovered, it must be composed of valid transitions from I that lead to a state violating P. This trace is therefore a valid counterexample.

Conclusion. The algorithm only returns `SAFE` when it finds an inductive invariant proving P, and only returns `UNSAFE` when it constructs a valid counterexample trace. Therefore, the correctness of the algorithm is guaranteed.

doi:10.6342/NTU202503854

5.2.3 Proof of Completeness

In this section, we prove the completeness of the multi-timeframe PDR algorithm. Specifically, we show that if the safety property P is violated in the system, the algorithm will eventually return `UNSAFE`. Otherwise, it will return `SAFE`.

From **Invariant 3** in section 5.2.1, we know that the sequence of over-approximations $R_0, R_1, \ldots$ is monotonic with respect to inclusion, i.e., $R_i \subseteq R_{i+1}$ for all $i \geq 0$. Since the state space is finite, the total number of distinct clause sets over transition system S is finite. Therefore, the algorithm cannot produce an infinite strictly increasing sequence of R_i sets.

As a result, the algorithm must eventually reach one of the following outcomes:

- It finds a fixpoint such that $R_i = R_{i+1}$. In this case, as discussed in section 5.2.2, R_i serves as an inductive invariant proving the safety property, and the algorithm returns `SAFE`.
- It encounters a reachable bad state that violates P. Since the algorithm never blocks any reachable state due to **Invariant 5**, the bad state must be reachable from I, and the algorithm returns `UNSAFE` with a valid counterexample trace.

Therefore, the multi-timeframe PDR algorithm is guaranteed to terminate with either a valid inductive invariant or a valid counterexample, ensuring its completeness.

doi:10.6342/NTU202503854

5.3 Dynamic Timeframe Expansion for Efficiency

To improve the efficiency of the multi-timeframe PDR algorithm, we adopt a dynamic timeframe expansion strategy. As discussed in section 4.3, using a fixed expansion depth for all proof obligations may either struggle to resolve difficult cases or fail to provide sufficient reachability information.

This dynamic expansion mechanism is implemented in the function `recBlockCube_multi()`, at line 11 of algorithm 9 to determine when to expand the timeframe.

As highlighted in the motivation chapter (chapter 3), most proof obligation trees are relatively easy to solve, while only a few require deeper reasoning or longer timeframes. Our strategy focuses on selectively addressing these hard cases without compromising the efficiency of the overall process.

The key idea is to dynamically increase the expansion depth only for those proof obligation trees identified as difficult. Specifically, we classify a tree as difficult if its solving time is more than twice the median solving time among all trees and also exceeds a predefined time threshold. When such a case is detected, we double the timeframe expansion until the transition path reaches the initial frame. Although doubling may appear aggressive, it significantly improves efficiency by accelerating the blocking of difficult proof obligations.

doi:10.6342/NTU202503854

Chapter 6 Case Study

In this chapter, we present two representative benchmark cases—one **UNSAFE** and one **SAFE**—that are uniquely solved by our adaptive multi-timeframe PDR algorithm. These cases demonstrate the effectiveness and improvement of our approach over existing methods. For a fair comparison, all algorithms are implemented within the same verification framework ABC [3], which provides a state-of-the-art implementation of the typical PDR and BMC algorithms. Our adaptive multi-timeframe PDR is implemented on top of the same framework to ensure consistency.

Section 6.1 analyzes the UNSAFE case *6s218b2950.aig*, where we additionally compare against BMC. Section 6.2 examines the SAFE case *6s310r.aig*.

6.1 UNSAFE Case: *6s218b2950.aig*

This section illustrates the effectiveness of the adaptive multi-timeframe PDR algorithm through an **UNSAFE** benchmark: *6s218b2950.aig*. We compare our approach against the typical PDR and the BMC algorithm, the latter being known as one of the most effective methods for finding counterexamples in UNSAFE designs. All algorithms are implemented within the same verification framework introduced earlier to ensure a consistent and fair comparison.

doi:10.6342/NTU202503854

Figure 6.1 visualizes the accumulated solving time at each timeframe. The typical PDR (`ABC_PDR`) experiences a sharp increase in solving time at Frame 26, where it becomes stuck attempting to block a difficult proof obligation and ultimately times out (3600 seconds). In contrast, the adaptive multi-timeframe PDR (`ABC_PDR-multi`) detects this difficulty and performs timeframe expansion up to Frame 26, enabling it to efficiently construct a counterexample. Once `timeframe_expansion` reaches 26, the algorithm effectively behaves like a variant of the BMC approach. As a result, the total solving time is comparable to that of the BMC implementation (`ABC_BMC`). This case highlights how adaptive expansion enables earlier counterexample discovery in frames where typical PDR fails to make progress.

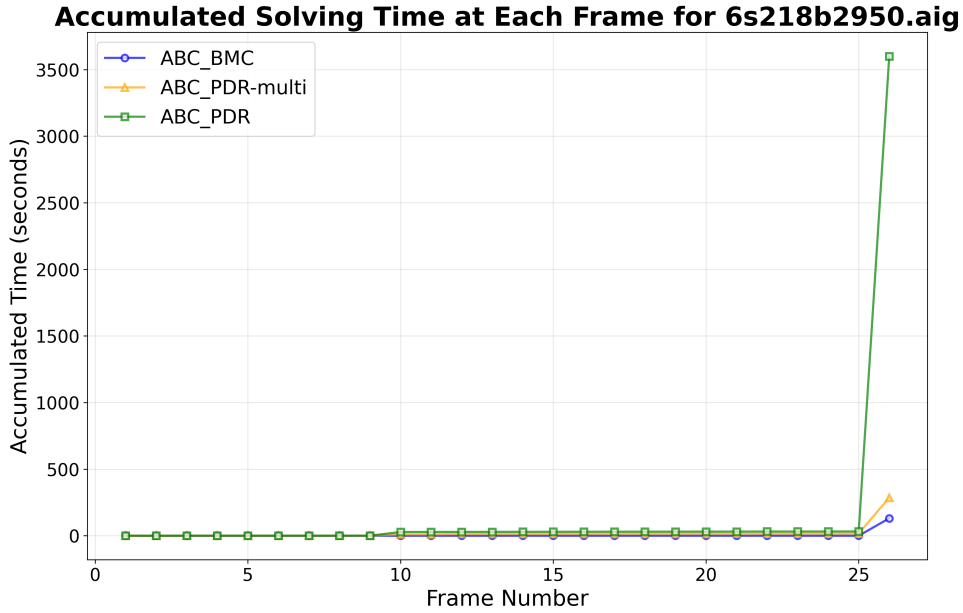
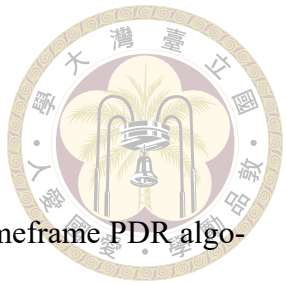


Figure 6.1: Accumulated solving time at each timeframe for *6s218b2950.aig*. The typical PDR (`ABC_PDR`) experiences a sharp spike in solving time at Frame 26 and fails to find a counterexample. The adaptive multi-timeframe PDR (`ABC_PDR-multi`) performs timeframe expansion and successfully finds a counterexample at Frame 26. After expanding to the initial state, the multi-timeframe PDR algorithm behaves similarly to the BMC (`ABC_BMC`) approach and achieves a comparable solving time.

doi:10.6342/NTU202503854

6.2 SAFE Case: *6s310r.aig*



This section illustrates the effectiveness of the adaptive multi-timeframe PDR algorithm through a **SAFE** case: *6s310r.aig*. In this case, the typical PDR fails to prove the property within the time limit (3600 seconds), while our algorithm successfully verifies the design. Both implementations are built on the same verification framework described earlier to ensure a fair comparison.

Figure 6.2 visualizes the accumulated solving time at each timeframe for the *6s310r.aig* case. The typical PDR (`ABC_PDR`) exhibits a sharp increase in solving time at Frame 15 and ultimately fails to make further progress, indicating that it becomes stuck while attempting to block a hard proof obligation. In contrast, the adaptive multi-timeframe PDR (`ABC_PDR-multi`) identifies the difficulty and performs timeframe expansion. This enables it to block the challenging obligation more efficiently and continue solving, eventually yielding a `SAFE` result at a deeper timeframe.

This outcome aligns with **Observation 1** from chapter 3, which states that in timeout cases, the overall solving cost is often dominated by a small subset of hard-to-block proof obligations. This behavior demonstrates how dynamic timeframe expansion allows the algorithm to adapt to bottlenecks caused by difficult proof obligations, effectively overcoming scenarios in which typical PDR fails to make progress.

To further illustrate the adaptive expansion strategy employed by our algorithm, figure 6.3 presents a bar chart showing the distribution of timeframe expansion counts across selected frames. Each group of bars corresponds to a single frame, and the bars within a group represent different expansion values (e.g., 2, 4, 8, and so on), with their heights

doi:10.6342/NTU202503854

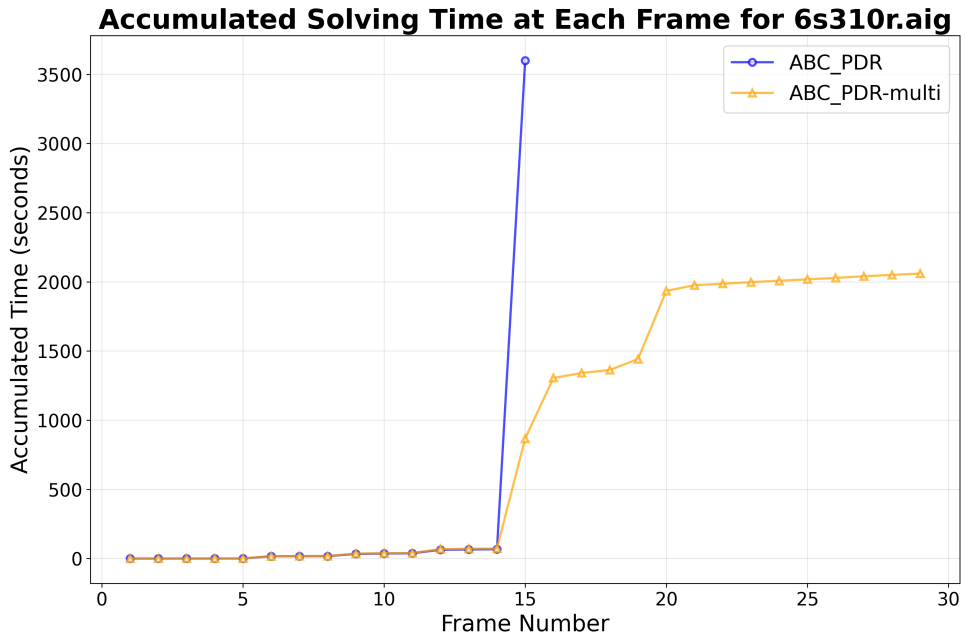


Figure 6.2: Accumulated solving time at each timeframe for *6s310r.aig*. The typical PDR (`ABC_PDR`) stalls at Frame 15, while the adaptive multi-timeframe PDR (`ABC_PDR-multi`) performs timeframe expansion, continues solving efficiently, and concludes with a `SAFE` result at Frame 29.

indicating how frequently each expansion was applied. As indicated by the solving time spikes in figure 6.2, aggressive expansion is concentrated in the most challenging frames —particularly Frames 15, 16, 19, and 20. In some instances, the expansion reaches the frame index itself, effectively unrolling to the initial state to overcome hard-to-block obligations. In contrast, most other frames exhibit no expansion activity and are omitted from the chart, indicating that standard PDR was already sufficient and our algorithm correctly refrains from unnecessary adaptation.

doi:10.6342/NTU202503854

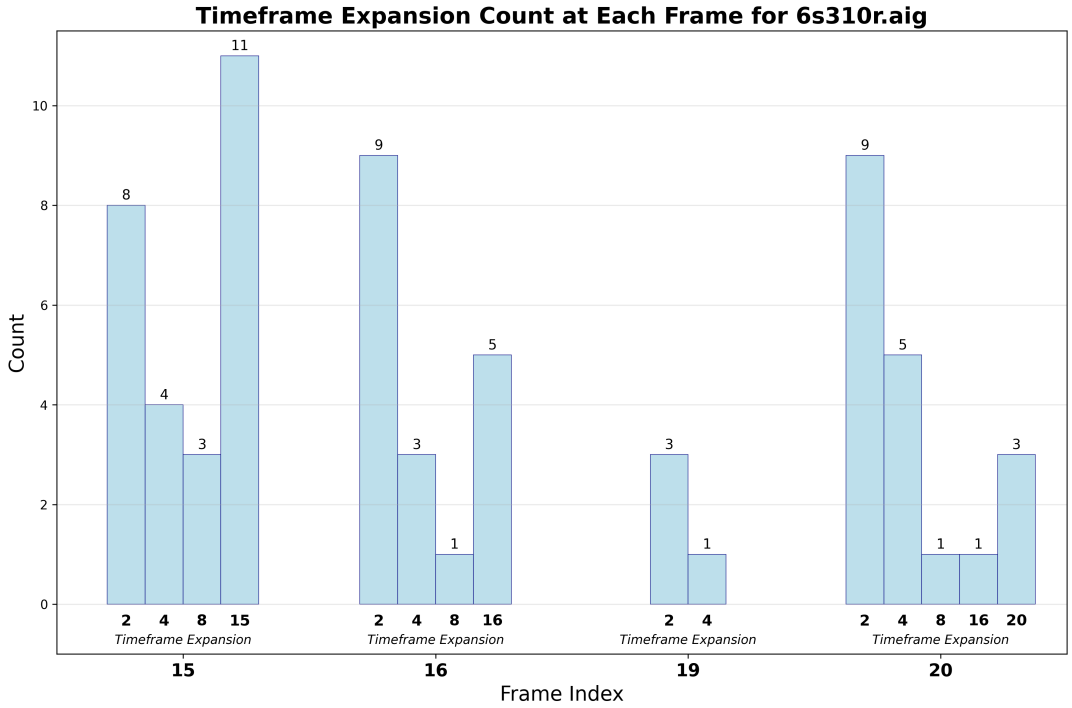


Figure 6.3: Distribution of timeframe expansion counts at selected frames for *6s310r.aig*. Each group of bars corresponds to a frame where expansion occurred, and each bar within the group represents a specific expansion value (e.g., 2, 4, 8, etc.) along with its application count. Frames not shown did not trigger any timeframe expansion.

doi:10.6342/NTU202503854

Chapter 7 Experimental Results

To evaluate the effectiveness of the proposed multi-timeframe PDR algorithm, we implement it on top of two state-of-the-art model checking frameworks: ABC [3] and rIC3 [9], resulting in two implementations referred to as `ABC_PDR-multi` and `rIC3-multi`, respectively.

For a fair comparison, we evaluate each implementation against its corresponding baseline. `ABC_PDR-multi` is compared to the typical PDR implementation in ABC (`ABC_PDR`), and `rIC3-multi` is compared to the original rIC3 (`rIC3`). This setup ensures that any observed improvements can be attributed specifically to the proposed expansion mechanism, rather than to differences between the underlying PDR frameworks.

7.1 Experimental Environment and Benchmarks

All experiments were conducted on a machine equipped with a `13th Gen Intel(R) Core(TM) i9-13900K @ 5.80GHz` processor and 128 GB of RAM, running `Ubuntu 22.04.3 LTS`. Each benchmark instance was executed with a timeout of 3600 seconds and a memory limit of 128 GB. All runs were performed in single-threaded mode to ensure consistent timing measurements.

doi:10.6342/NTU202503854

We evaluate two implementations of the multi-timeframe PDR algorithm based on ABC and rIC3. The time threshold for triggering timeframe expansion is set to 10 seconds in `ABC_PDR-multi` and 5 seconds in `rIC3-multi`.

The evaluation is conducted on the single safety property track of the HWMCC 2017 benchmark suite [5], which includes a diverse set of AIGER-encoded hardware verification problems. In total, we tested 300 benchmark instances using both the ABC-based and rIC3-based implementations.

7.2 Results on HWMCC Benchmarks

In this section, we present the experimental results comparing our multi-timeframe PDR implementations to their respective baselines. We begin with an overall runtime comparison across all solvers, followed by detailed analyses for each framework.

Figure 7.1 shows a cactus plot of runtime across all solved instances, sorted in increasing order. Each line represents one of the four implementations. The plot demonstrates that both `ABC_PDR-multi` and `rIC3-multi` generally outperform their baselines, `ABC_PDR` and `rIC3`, respectively.

Table 7.1 presents the comparison results between the multi-timeframe PDR implemented on ABC (`ABC_PDR-multi`) and the typical PDR in ABC (`ABC_PDR`). For each solver, we report the total number of solved cases, the number of `UNSAFE` and `SAFE` results, the `PAR-2` score,[1] and the number of uniquely solved cases. The results show that the multi-timeframe framework solves more benchmarks, achieves a lower `PAR-2` score, and uniquely solves several instances that the baseline fails to handle.

[1] `PAR-2` is the penalized average runtime, where unsolved instances are assigned twice the timeout value.

doi:10.6342/NTU202503854

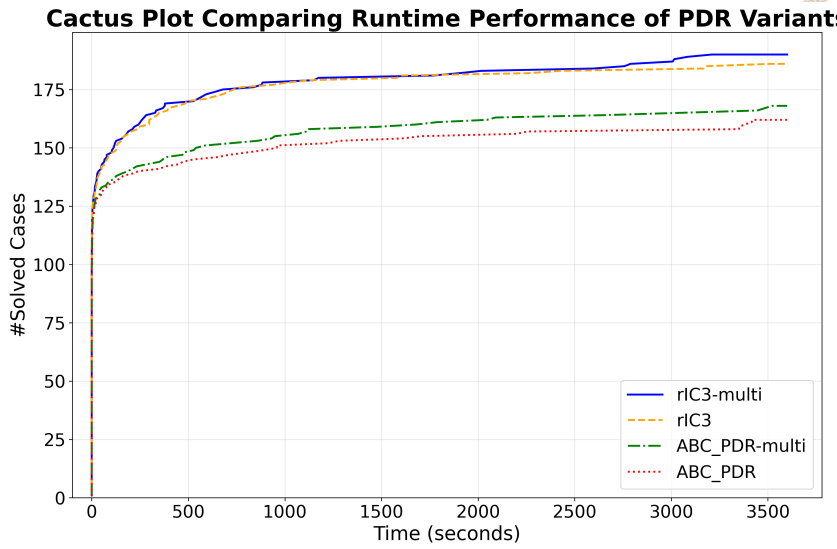


Figure 7.1: Cactus plot comparing runtime performance of typical and multi-timeframe PDR implementations.

Figure 7.2 further illustrates the number of solved instances using a bar chart. It highlights the total count for both solvers, with uniquely solved cases explicitly marked. This visualization clearly illustrates the additional coverage provided by `ABC_PDR-multi`.

Table 7.1: Comparison between ABC and `ABC_PDR-multi`.

Solver	Total Solved	UNSAFE	SAFE	PAR-2 (s)	Unique Solved
ABC_PDR	162	42	120	3431.66	3
ABC_PDR-multi	168	46	122	3293.44	9

Table 7.2 presents the comparison results between the multi-timeframe PDR implemented on rIC3 (`rIC3-multi`) and the original rIC3. Similar to the ABC case, the multi-timeframe version solves more benchmarks, achieves a lower `PAR-2` score, and uniquely solves several instances that the baseline fails to handle.

Figure 7.3 shows the number of solved instances using a bar chart. The total counts for both solvers are shown, with uniquely solved cases explicitly marked. This visualization clearly illustrates the additional coverage provided by `rIC3-multi`.

Overall, the improvements observed in both `ABC_PDR-multi` and `rIC3-multi` demon-

doi:10.6342/NTU202503854

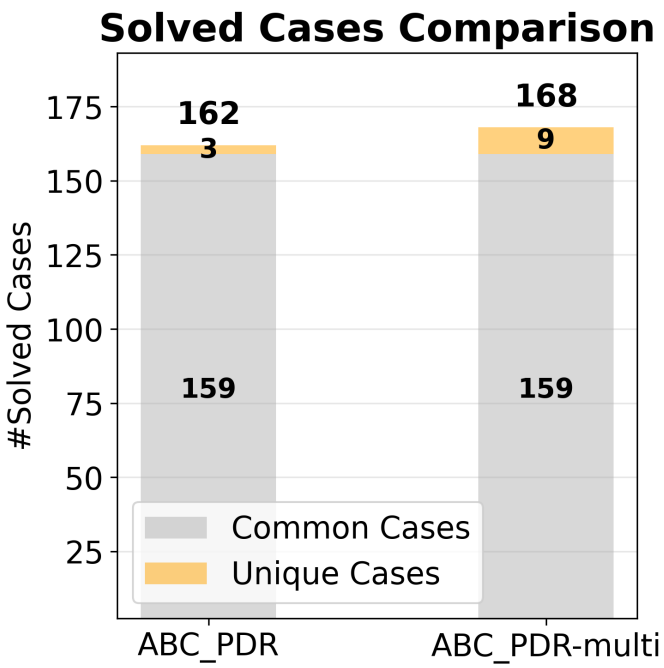


Figure 7.2: Solved cases for ABC and `ABC_PDR-multi`, with uniquely solved instances marked.

Table 7.2: Comparison between rIC3 and `rIC3-multi`.

Solver	Total Solved	UNSAFE	SAFE	PAR-2 (s)	Unique Solved
rIC3	186	45	141	2842.33	1
rIC3-multi	190	46	144	2774.30	5

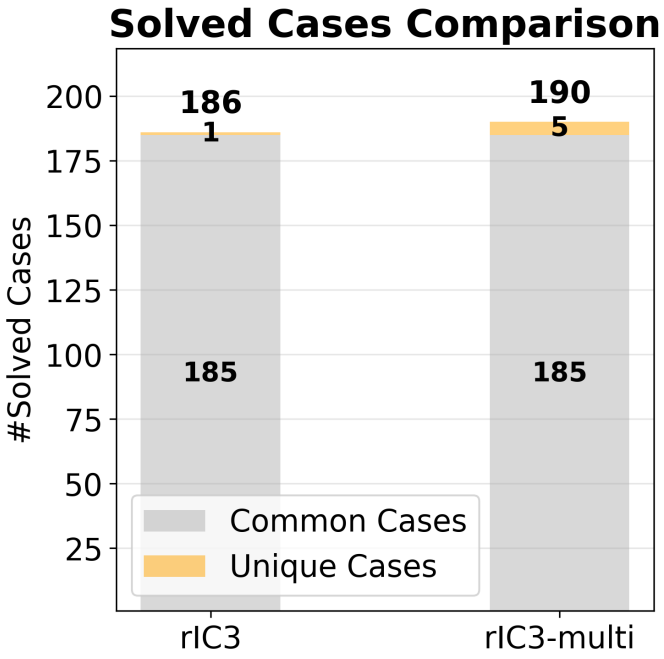


Figure 7.3: Solved cases for rIC3 and `rIC3-multi`, with uniquely solved instances marked.

doi:10.6342/NTU202503854

strate the effectiveness and robustness of the proposed multi-timeframe PDR algorithm. The consistent performance gains across two distinct PDR frameworks indicate that the proposed approach generalizes well and is not tied to a specific implementation.

7.3 Comparison with Pre-Expanded Timeframe Strategy

In this section, we compare our proposed multi-timeframe PDR algorithm with an alternative strategy that performs timeframe expansion prior to PDR execution. Instead of expanding timeframes dynamically during the solving process, this technique flattens the circuit structure by unrolling the transition relation ahead of PDR solving.

Since the pre-expansion is implemented using ABC's built-in `frames -F <expansion>` command, this set of experiments focuses solely on the implementation based on the ABC framework to ensure compatibility and consistent circuit handling.

To generate the pre-expanded circuit, we use the `frames -F <expansion>` command, which unfolds the sequential circuit for a fixed number of frames. In our experiments, we evaluate two expansion depths: 4 and 8.

Section 7.3.1 provides an overview comparison of all evaluated variants, including:

- Original PDR without pre-expansion (`ABC_PDR`)
- Multi-timeframe PDR without pre-expansion (`ABC_PDR-multi`)
- Original PDR with pre-expansion (`ABC_PDR-frames-4`, `ABC_PDR-frames-8`)
- Multi-timeframe PDR with pre-expansion (`ABC_PDR-multi-frames-4`, `ABC_PDR-multi-frames-8`)

doi:10.6342/NTU202503854

This comparison highlights how pre-expansion interacts with both the original and multi-timeframe variants of PDR.

Next, section 7.3.2 compares `ABC_PDR-frames-4` with our dynamic `ABC_PDR-multi` to examine the effect of when expansion is applied. Specifically, the pre-expansion strategy unrolls the circuit before PDR begins, whereas the multi-timeframe PDR algorithm expands timeframes on demand during solving.

Finally, we compare `ABC_PDR-frames-4` with `ABC_PDR-multi-frames-4` in section 7.3.3, where both solvers operate on the same pre-expanded circuit. This comparison demonstrates that our multi-timeframe approach delivers further performance gains over standard PDR, even when starting from the same unrolled input.

7.3.1 Overall Comparison

Table 7.3 presents a comprehensive comparison of original and multi-timeframe PDR solvers, with and without pre-expansion. The results indicate that pre-expansion generally improves performance for both variants, particularly at a moderate expansion depth of 4. Notably, using a larger pre-expansion depth does not necessarily lead to better performance. In our experiments, a pre-expansion value of 4 strikes a sweet spot, yielding the most significant improvement for both the original PDR (`ABC_PDR_frames-4`) and the multi-timeframe PDR (`ABC_PDR-multi_frames-4`).

7.3.2 Pre-Expansion Before PDR vs. Dynamic Expansion During PDR

This subsection compares two different strategies for applying timeframe expansion:

doi:10.6342/NTU202503854

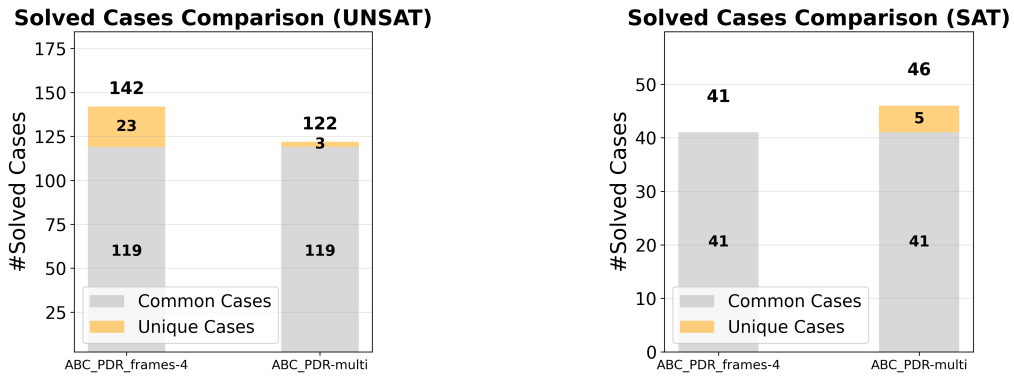
Table 7.3: Comparison of original and multi-timeframe PDR variants, with and without pre-expansion.

Solver	Total Solved	UNSAFE	SAFE	PAR-2 (s)
ABC_PDR	162	42	120	3431.66
ABC_PDR_frames-4	183	41	142	2978.48
ABC_PDR_frames-8	172	44	128	3267.47
ABC_PDR-multi	168	46	122	3293.44
ABC_PDR-multi_frames-4	185	45	140	2951.38
ABC_PDR-multi_frames-8	160	44	116	3532.53

- **Pre-expansion before PDR:** The circuit is statically unrolled to 4 frames before solving begins (`ABC_PDR-frames-4`).
- **Dynamic expansion during PDR:** The multi-timeframe PDR solver adaptively expands timeframes during the solving process (`ABC_PDR-multi`).

The total number of solved cases for each solver is summarized in table 7.3. To further highlight their differences, figure 7.4a and figure 7.4b present the number of uniquely solved instances in the `SAFE` and `UNSAFE` categories, respectively.

Overall, both strategies outperform the baseline `ABC_PDR`. `ABC_PDR-frames-4` demonstrates more improvement in solving `SAFE` cases, while `ABC_PDR-multi` shows a slight advantage in handling `UNSAFE` cases.



(a) Uniquely solved `SAFE` cases

(b) Uniquely solved `UNSAFE` cases

Figure 7.4: Number of uniquely solved cases by `ABC_PDR-frames-4` and `ABC_PDR-multi`, categorized by `SAFE` and `UNSAFE` outcomes.

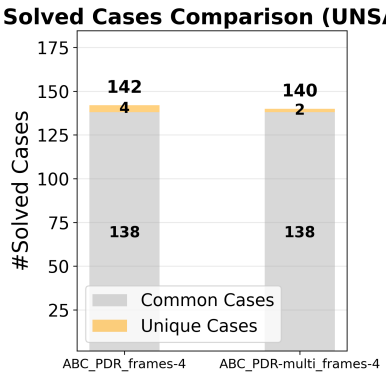
doi:10.6342/NTU202503854

7.3.3 Multi-Timeframe PDR on Pre-Expanded Circuits

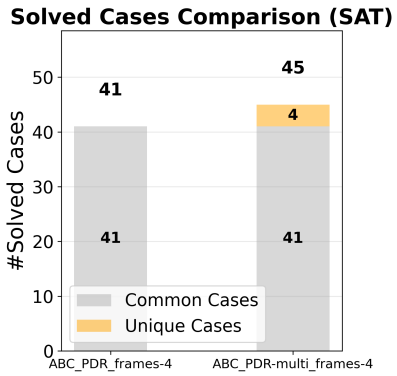
In this subsection, we investigate whether our multi-timeframe PDR algorithm continues to provide benefits when operating on circuits that have already been statically unrolled.

Specifically, we compare `ABC_PDR-frames-4` and `ABC_PDR-multi-frames-4`, where both solvers operate on the same pre-expanded circuit with four unfolded frames. This setup ensures that any difference in performance can be directly attributed to the multi-timeframe reasoning mechanism, as the input circuits are structurally identical.

As shown in table 7.3, `ABC_PDR-multi-frames-4` solves more benchmarks in general. This confirms that the multi-timeframe approach consistently achieves additional improvements even when applied to pre-expanded circuits. For a more detailed comparison, figure 7.5a and figure 7.5b show the number of uniquely solved instances in the `SAFE` and `UNSAFE` categories, respectively. `ABC_PDR-frames-4` solves more `SAFE` cases, while `ABC_PDR-multi-frames-4` performs better in the `UNSAFE` category.



(a) Uniquely solved `SAFE` cases.



(b) Uniquely solved `UNSAFE` cases.

Figure 7.5: Comparison of uniquely solved cases between `ABC_PDR-frames-4` and `ABC_PDR-multi-frames-4`.

doi:10.6342/NTU202503854

7.4 Effect of Multi-Timeframe PDR After Sequential Optimization

In this section, we examine whether our proposed multi-timeframe PDR algorithm continues to provide improvements after applying sequential optimization. Specifically, we apply ABC's `scorr` command, which performs K-step induction-based sequential sweeping to simplify the circuit before running PDR.

As in section 7.3, we restrict this evaluation to the ABC framework due to compatibility with the `scorr` transformation.

We evaluate four configurations:

- Original PDR without sequential optimization (`ABC_PDR`)
- Multi-timeframe PDR without sequential optimization (`ABC_PDR-multi`)
- Original PDR with sequential optimization (`ABC_PDR-scorr`)
- Multi-timeframe PDR with sequential optimization (`ABC_PDR-multi-scorr`)

Table 7.4 summarizes the results. Overall, sequential optimization improves performance across both solvers. Notably, `ABC_PDR-multi-scorr` achieves the best results in terms of total solved cases and `PAR-2` score, showing that our multi-timeframe strategy remains effective even after circuit simplification.

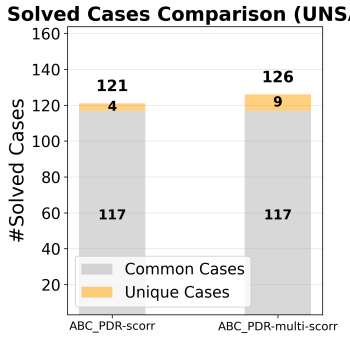
To better highlight the improvement introduced by multi-timeframe reasoning, we directly compare the two solvers that both apply sequential sweeping: `ABC_PDR-scorr` and `ABC_PDR-multi-scorr`. As shown in table 7.4, `ABC_PDR-multi-scorr` solves more

doi:10.6342/NTU202503854

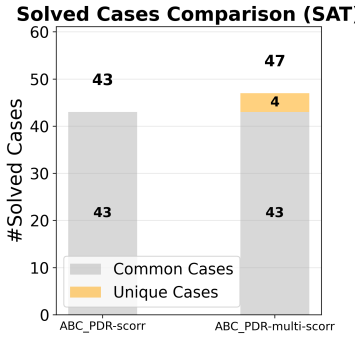
Table 7.4: Comparison of PDR solvers with and without sequential optimization (`scorr`).

Solver	Total Solved	UNSAFE	SAFE	PAR-2 (s)
ABC_PDR	162	42	120	3431.66
ABC_PDR-multi	168	46	122	3293.44
ABC_PDR-scorr	164	43	121	3312.90
ABC_PDR-multi-scorr	173	47	126	3127.04

instances and achieves a lower average runtime. Figure 7.6a and figure 7.6b show the number of uniquely solved instances in the `SAFE` and `UNSAFE` categories, respectively. These results confirm that our dynamic expansion strategy continues to offer gains even after simplification by `scorr`.



(a) SAFE category



(b) UNSAFE category

Figure 7.6: Uniquely solved cases after sequential optimization, comparing `ABC_PDR-scorr` and `ABC_PDR-multi-scorr`.

doi:10.6342/NTU202503854

Chapter 8 Conclusions and Future Work

8.1 Conclusions

This thesis presents an adaptive multi-timeframe extension to the PDR algorithm, addressing key inefficiencies in the typical single-timeframe approach. We observed that while most proof obligations are relatively easy to resolve, typical PDR often spends excessive time attempting to block a few particularly difficult obligations. To alleviate this issue, we proposed a dynamic timeframe expansion strategy within a multi-timeframe framework, enabling the algorithm to accelerate progress by operating over multiple transitions when necessary.We also provided formal proofs of correctness and completeness for the proposed algorithm. Experimental results demonstrate that the multi-timeframe PDR algorithm is effective, robust, and general. It consistently outperforms the original PDR in two separate implementations built on top of state-of-the-art PDR frameworks. Moreover, the algorithm continues to show improvements even when combined with pre-expanded timeframes and sequential optimizations, indicating that its benefits persist alongside other enhancement techniques.

doi:10.6342/NTU202503854

8.2 Future Work

One promising direction for future work is the deep integration of the multi-timeframe PDR algorithm with BMC. In industrial or competition settings such as HWMCC, multiple model checkers are often executed in parallel, as different solvers are suited to different types of designs. However, these checkers typically operate independently, without information sharing. With the full integration of our multi-timeframe extension into typical PDR, it may be possible to more deeply couple PDR and BMC. Specifically, `UNSAT` results from BMC could be reused by the PDR solver; after refinement via our algorithm, these results may yield valid blocking cubes that accelerate the PDR solving process.

Another potential improvement lies in refining the timeframe expansion strategy. For instance, we could evaluate proof obligations based on historical difficulty indicators such as solving time and depth. These metrics could inform heuristics for determining whether an obligation is hard to block and, in turn, guide the degree of timeframe expansion or adjust the triggering thresholds dynamically.

We believe the multi-timeframe framework expands the diversity and dimensionality of the typical PDR algorithm. This adaptation enhances the flexibility of PDR and builds a conceptual bridge between PDR and other unrolling-based model checking techniques such as BMC.

doi:10.6342/NTU202503854

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In W. R. Cleaveland, editor, Tools and Algorithms for the Construction and Analysis of Systems, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[2] A. R. Bradley. Sat-based model checking without unrolling. In R. Jhala and D. Schmidt, editors, Verification, Model Checking, and Abstract Interpretation, pages 70–87, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] R. Brayton and A. Mishchenko. Abc: An academic industrial-strength verification tool. In T. Touili, B. Cook, and P. Jackson, editors, Computer Aided Verification, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[4] N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In 2011 Formal Methods in Computer-Aided Design (FMCAD), pages 125–134, 2011.

[5] Hardware Model Checking Competition. HWMCC 2017 Benchmarks. https://hwmcc.github.io/, 2017. Accessed: 2025-07-18.

[6] J. Li, R. Dureja, G. Pu, K. Y. Rozier, and M. Y. Vardi. Simplecar: An efficient bug-finding tool based on approximate reachability. In H. Chockler and

doi:10.6342/NTU202503854

G. Weissenbacher, editors, Computer Aided Verification, pages 37–44, Cham, 2018. Springer International Publishing.

[7] K. L. McMillan. Interpolation and sat-based model checking. In W. A. Hunt and F. Somenzi, editors, Computer Aided Verification, pages 1–13, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[8] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In W. A. Hunt and S. D. Johnson, editors, Formal Methods in Computer-Aided Design, pages 127–144, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[9] Y. Su, Q. Yang, Y. Ci, T. Bu, and Z. Huang. The ric3 hardware model checker, 2025.

[10] X. Zhang, S. Xiao, J. Li, G. Pu, and O. Strichman. Combining bmc and complementary approximate reachability to accelerate bug-finding. In 2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pages 1–9, 2022.

[11] 楊承翰. 利用動態時間框架延展增進性質導向可達性技術. Master's thesis, 國立臺灣大學, Jan 2017.

doi:10.6342/NTU202503854