

國立臺灣大學電機資訊學院電子工程學研究所

碩士論文

Graduate Institute of Electronics Engineering  
College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis



抽取謂詞以改進性質導向可達性技術

Improving Property Directed Reachability by Predicate

Extraction

江俊毅

Chun-Yi Chiang

指導教授：黃鐘揚 博士

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

中華民國 109 年 7 月

July, 2020

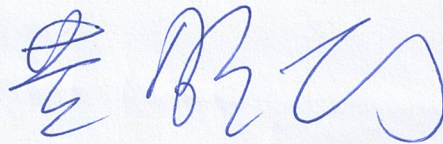
國立臺灣大學碩士學位論文  
口試委員會審定書

抽取謂詞以改進性質導向可達性技術

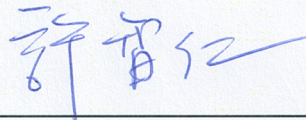
Improving Property Directed Reachability by Predicate Extraction

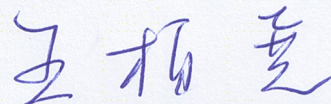
本論文係江俊毅君 (R07943110) 在國立臺灣大學電子工程學研究所完成之碩士學位論文，於民國 109 年 7 月 24 日承下列考試委員審查通過及口試及格，特此證明

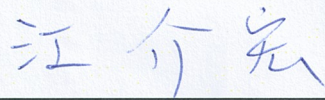
口試委員：



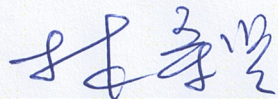
(指導教授)







系主任、所長





## 誌謝

終於到了求學生涯的一個斷點，雖然不知道之後會不會有機會再回歸校園生活，但目前來說也是該到外面去闖一闖了。

感謝我的指導教授黃鐘揚老師，有幸跟了你三年，能體會到如此開明的風氣，自由選擇研究的方向，也不會特地訂立強硬的期限或繁雜的規則，讓我們隨心所欲之餘也給予適時的提點，才不至於迷失方向，中間儘管遭遇些許失敗的挫折，最後也還是完成了這份論文。

感謝實驗室的各位，不管是已經畢業的學長姐或是還在實驗室的同學和助理，有問題的時候有人可以一起解決是很幸運的事情。

感謝我的朋友們，從高中到大學再到研究所，雖然沒有太多特別激情的回憶，只是重複著平淡的事情，但身邊有你們的陪伴的確讓人安心不少。

最後要感謝我的父母，不辭勞苦的把我養大，供我念書，離開家裡來念大學已經六年了，偶爾仍會感到非常思念，在我最低落的時候我總是相信還有你們會支持我，你們是我繼續堅持往下最大的動力。

畢業雖然是個結束，卻也是個新的開始，希望未來的我能活得精采快樂，不負現在的努力。



## 中文摘要

自從在 2011 年被提出後，性質導向可達性技術至今仍是最好的模型檢查演算法。但是仍有許多的案例是性質導向可達性技術難以解決的，因此，為了改進這個演算法，有許多的研究先後被發表。在這篇論文中，我們藉由獨立的檢查謂詞以輔助性質導向可達性演算法。此外，作為證明的例子，我們也提供了兩種能夠輕易從演算過程中得出的樣式。原本的性質導向可達性演算法與新方法皆被實作於 Ia2b 這個自製的模型檢查環境上。透過利用硬體模型檢查比賽的資料所做的實驗得知，我們所提出的方法能夠解出比原本的性質導向可達性技術更多的案例。

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、謂詞修正



# Abstract

Since proposed in 2011, PDR (IC3) has been the best model checking algorithm until now. However, there are still many cases that PDR struggles to solve. Hence, many works are presented to enhance the algorithm. In this thesis, we propose a general method to aid PDR by solving predicate separately. Furthermore, we provide two kinds of useful pattern easily recognized from the solving process as example. The original PDR algorithm and the new feature are implemented on a custom model checking environment called Ia2b. The experiment on HWMCC benchmarks shows that our method can solve more cases than the classic PDR.

**Keywords:** Formal Verification, Model Checking, Satisfiability Problem, Property Directed Reachability, Predicate Refinement



# Contents

誌謝	i
中文摘要	ii
<b>Abstract</b>	<b>iii</b>
<b>Contents</b>	<b>iv</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Contributions of the Thesis . . . . .	2
1.2 Organizations of the Thesis . . . . .	3
<b>2 Preliminaries</b>	<b>4</b>
2.1 Propositional Satisfiability . . . . .	4
2.2 Finite State Boolean Transition System . . . . .	5
2.3 Model Checking Problem . . . . .	5
2.4 Property Directed Reachability . . . . .	8
2.4.1 The Monotonicity of Frames . . . . .	10
2.4.2 Ternary Simulation . . . . .	13
2.4.3 Recursively Blocking Cubes . . . . .	14



2.4.4	Propagating Blocked Cubes . . . . .	15
2.4.5	Other Subroutines . . . . .	17
<b>3</b>	<b>Predicate Extraction</b>	<b>18</b>
3.1	Motivation . . . . .	18
3.2	General Method and Correctness . . . . .	21
3.3	Two Kinds of Predicate . . . . .	23
3.3.1	Blocking Clause . . . . .	23
3.3.2	Proof Obligation . . . . .	25
3.3.3	Recap for 6s288r . . . . .	28
<b>4</b>	<b>Implementation</b>	<b>32</b>
4.1	The Model Checking Environment . . . . .	32
4.2	Optimization and Other Things We Tried . . . . .	34
4.2.1	Subsumption . . . . .	34
4.2.2	Ternary Simulation . . . . .	35
4.2.3	SAT Query . . . . .	37
4.2.4	Storage of Proof Obligation . . . . .	39
4.2.5	Propagating Cubes . . . . .	41
4.2.6	Activity of State Variables . . . . .	41
4.2.7	Predicate Extraction . . . . .	42
<b>5</b>	<b>Experimental Results</b>	<b>45</b>
5.1	Overview . . . . .	45
5.2	Performance . . . . .	48
5.3	Detailed Analysis . . . . .	61
<b>6</b>	<b>Conclusion and Future Work</b>	<b>66</b>
	<b>Reference</b>	<b>68</b>

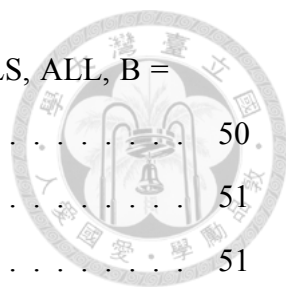


# List of Figures

2.1	The depiction of trace of frame. . . . .	9
2.2	The depiction of recBlockCubes. . . . .	14
3.1	36 blocked cubes out of the 889 ones in the inductive invariant of 6s288r. . . . .	19
3.2	The depiction of the similar clause problem. . . . .	20
3.3	The depiction of the reason for the similar clause problem. . . . .	21
3.4	An example for clause-based predicate. . . . .	24
3.5	The possible distribution of reachable and bad states for similar obligation problem. . . . .	25
3.6	An example for obligation-based predicate. . . . .	26
3.7	The proof obligations leading to the similar clauses. . . . .	28
3.8	The cone of 12141 !12142 12180. . . . .	29
3.9	The cone of the literal along with 12141 !12142 12180 in Figure 3.1. . . . .	29
3.10	Blocking procedure for the similar clauses (1). . . . .	30
3.11	Blocking procedure for the similar clauses (2). . . . .	30
3.12	Blocking procedure for the similar clauses (3). . . . .	31
4.1	The memory arrangement for cube of PDR. . . . .	34
5.1	The cumulative plot of PDR among different model checkers. . . . .	47
5.2	The cumulative plot of different limit of SAT query for CLS, INF, B = 20, M = 1 on pdr_vb. . . . .	49



5.3	The cumulative plot of different limit of SAT query for CLS, ALL, B = 10, M = 2 on pdr_vb. . . . .	50
5.4	The schematic diagram of the sweet spot prediction. . . . .	51
5.5	The depiction of sample points. . . . .	51
5.6	The cumulative plot of different ratio of Backtrack number and Match number for CLS, INF, L = 1000 on pdr_vb. . . . .	53
5.7	The cumulative plot of different ratio of Backtrack number and Match number for CLS, ALL, L = 1000 on pdr_vb. . . . .	54
5.8	The cumulative plot of two best ratios and the mixed version for CLS, L = 1000 on pdr_vb. . . . .	55
5.9	The cumulative plot of two best ratios and the mixed version for CLS, L = 300 on pdr_vb. . . . .	56
5.10	The cumulative plot of different limit of SAT query for OBL, T = 66 on pdr_vb. . . . .	58
5.11	The cumulative plot of different Obligation threshold for OBL, L = 300 on pdr_vb. . . . .	59
5.12	The cumulative plot of the best parameters for CLS, OBL and the combination of both on full. . . . .	60
5.13	The comparison between clause-based and obligation-based predicate. . .	63
5.14	The comparison between basic PDR and clause-based predicate (1). . . .	64
5.15	The comparison between basic PDR and clause-based predicate (2). . . .	65
5.16	The comparison between basic PDR and obligation-based predicate. . . .	65





## List of Tables

5.1	Comparison of PDR among different model checkers. . . . .	47
5.2	Comparison among different limit of SAT query for CLS, INF, $B = 20$ , $M = 1$ on pdr_vb. . . . .	49
5.3	Comparison among different limit of SAT query for CLS, ALL, $B = 10$ , $M = 2$ on pdr_vb. . . . .	50
5.4	Comparison among different ratio of Backtrack number and Match number for CLS, INF, $L = 1000$ on pdr_vb. . . . .	53
5.5	Comparison among different ratio of Backtrack number and Match number for CLS, ALL, $L = 1000$ on pdr_vb. . . . .	54
5.6	Comparison among two best ratios and the mixed version for CLS, $L = 1000$ on pdr_vb. . . . .	55
5.7	Comparison among two best ratios and the mixed version for CLS, $L = 300$ on pdr_vb. . . . .	56
5.8	Comparison among different limit of SAT query for OBL, $T = 66$ on pdr_vb. . . . .	58
5.9	Comparison among different Obligation threshold for OBL, $L = 300$ on pdr_vb. . . . .	59
5.10	Comparison among the best parameters for CLS, OBL and the combination of both on full. . . . .	60



# Chapter 1

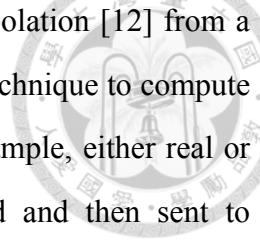
## Introduction

Model checking is the problem to check if a specification is violated under a transition system. This kind of problem is common in the area of formal verification. For hardware verification, as the design becomes more complicated, the gate count in a circuit grows dramatically, which makes the complexity of this problem becomes higher and higher. Hence, many innovative algorithms are proposed continually to tackle with the problem.

At early time, Binary Decision Diagram (BDD) [1] is used to compute the exact image by building the transition relation. The process continues until either a fixed point is reached or the current reachability intersects with the violation. However, BDD has poor scalability for large design since it often suffers from memory explosion.

As SAT algorithm gets tremendous efficiency improvement [2, 3, 4, 5, 6, 7], many researchers notice the practicality for SAT-based model checking method. Bounded Model Checking (BMC) [8] unfolds the circuit to check whether some states are reachable in this length. BMC itself can only answer whether the system is safe until a given length instead of proving the property completely. Afterwards, many algorithms based on BMC are proposed to enable the proof.

[9, 10] uses BMC as base case to support K-Induction (IND) for the proof. The SAT query of K-Induction also involves unrolling. Furthermore, the simple-path constraints should be added to make the algorithm complete.



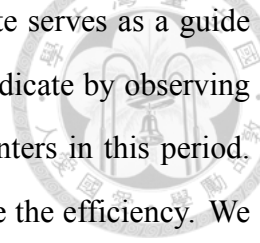
In [11], McMillan shows a procedure to construct Craig's interpolation [12] from a resolution refutation generated by modern SAT solver and use this technique to compute the over-approximated image. BMC is used for finding counter example, either real or spurious one. If BMC returns UNSAT, interpolant is computed and then sent to fixed-point checking. If the result is true, an inductive invariant set is found; otherwise, the interpolant is used for the next BMC solving. This method is called Interpolation-based Model Checking (IMC). On the other hand, Interpolation-Sequence Based Model Checking [13] works in similar way but different manner. There are also efforts to explore the property about interpolant [14, 15].

After several years, Bradley proposes a novel non-unrolling method [16] based on his previous work [17]. His implementation called IC3 won the third place in Hardware Model Checking Competition (HWMCC) 2010. Only two well-tuned integrated verification tools can beat it. Later, Eén presents a more efficient version of IC3 called Property Directed Reachability (PDR) [18]. From then on, PDR becomes the most important model checking algorithm and is commonly implemented in verification environment.

There are many works trying to modify PDR to improve it like [19, 20, 21, 22, 23]. Among them, [24] lets a blocked cube be a spurious proof obligation when it fails to be propagated to the next frame. This can be seen as over-approximating the property to let PDR refine the reachability more eagerly. However, this seems to be not so useful compared to its overhead. Hence, the authors need to include other techniques to demonstrate the potential of the method.

## 1.1 Contributions of the Thesis

In this thesis, we propose a general method to aid PDR by solving predicate separately. The meaning of this method is to force PDR solve for what we guess useful instead of only the original property. PDR will search for what is actually unreachable more eagerly



since we over-approximate the property by predicates. The predicate serves as a guide to smooth the whole process of solving. This is because we find predicate by observing the solving process locally and point out what obstacle PDR encounters in this period. By handling the obstacle separately in time, it is expected to improve the efficiency. We further provide two examples of how to produce such predicate by blocking clauses and proof obligations. Both of them can help PDR solve cases that are hardly proven by the original PDR among various implementations. Overall, PDR with predicate extraction performs better than the original one.

## **1.2 Organizations of the Thesis**

The rest of the thesis is organized as follows. Some background knowledge is provided in chapter 2 to define the problem and method clearly. In chapter 3, we explain our method for PDR based on predicate solving. The implementation details including the whole environment and optimization are listed in chapter 4. We show the experimental results in chapter 5 to support our method. In the last chapter, we make a conclusion and discuss about the future work.



# Chapter 2

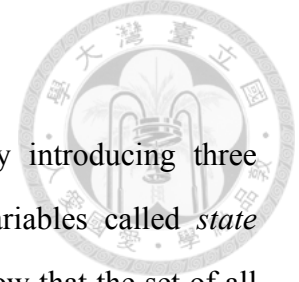
## Preliminaries

In this chapter, we give a series of basic knowledge for PDR step by step to help readers realize the background.

### 2.1 Propositional Satisfiability

One-dimensional Boolean space is a set containing only two elements  $\mathbb{B} = \{\mathbf{TRUE}, \mathbf{FALSE}\}$ . Multi-dimensional Boolean space is defined by the  $n$ -ary Cartesian product  $\mathbb{B}^n = \mathbb{B} \times \mathbb{B} \dots \times \mathbb{B}$ , where  $n$  is the number of dimension. A *propositional variable* is a variable whose value is defined on  $\mathbb{B}$ . A *literal* is a propositional variable with positive or negative polarity. A *cube* is a conjunction of literals, while a *clause* is a disjunction of literals. A *disjunctive normal form* (DNF, also called sum of product, SOP) is a disjunction of cubes, while a *conjunctive normal form* (CNF, also called product of sum, POS) is a conjunction of clauses. The *satisfiability* (SAT) problem is to answer if there exists an assignment such that the given propositional formula is evaluated to **TRUE**. If the answer is yes, we call it *satisfiable* (SAT); otherwise, it is *unsatisfiable* (UNSAT). We say that a clause is satisfied if at least one literal in it is evaluated to **TRUE**. The given formula is usually in the form of CNF, which means the instance is SAT iff all the clauses are satisfied under the assignment.

## 2.2 Finite State Boolean Transition System

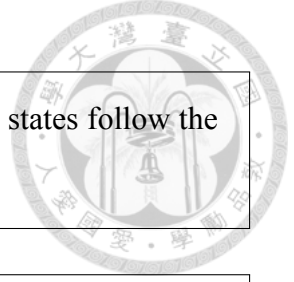


We can define the finite state Boolean transition system  $M$  by introducing three components  $\langle S, Init, Tr \rangle$ .  $S$  is a finite set of propositional variables called *state variable*, and a *state*  $s$  is an assignment on them. Therefore, we know that the set of all states is equal to  $\mathbb{B}^{|S|}$ , where the vertical bars means the cardinality of a set.  $Init(S)$  is a propositional formula to mark the initial state. That is, a state  $s$  belongs to the initial state iff  $Init(s)$  is **TRUE**.  $Tr(S, S')$  is also a propositional formula, and it is used to indicate the transition relationship of the system.  $S'$  is a copy from  $S$  to represent the next state after transition. Given two states  $s_1, s_2$ , either identical or different,  $s_1$  can transit to  $s_2$  iff  $Tr(s_1, s_2)$  is **TRUE**.

To correspond with the circuit representation that we encounter in real problem, we define an equivalent form as transition function. Consider that a state may transit to more than one state, we introduce another set of propositional variables  $I$  called *input variable*. Formally, the domain of the transition function is  $\mathbb{B}^{|S \cup I|}$  and the codomain is  $\mathbb{B}^{|S'|}$ . The system will go to  $Tf(s, i)$  under the current state  $s$  and input combination  $i$ . To convert the transition function to the transition relation, we just perform existential quantification on the input variables. That is,  $Tr(S, S') = \exists I. (S' = Tf(S, I))$ .

## 2.3 Model Checking Problem

A *property*  $p$  is a propositional formula to mark the set of states meeting the expectation. A *bad state* is a state  $s$  such that  $p(s)$  is **FALSE**, and we call the rest states as *good state*. Model checking can be divided to two main categories, including *safety* and *liveness* problem. Safety problem (**AG**  $p$ ) is to answer whether the system can go a bad state or not without any time limit. On the other hand, liveness problem (**AF**  $p$ ) checks if the system can eventually reach a good state along any branch. The algorithms introduced in this thesis focus on the safety problem. If **AG**  $p$  is true, we say the problem is safe (UNSAT); otherwise, it is violated (SAT).



**Definition 1.** A *trace* is a finite sequence of states that all adjacent states follow the transition relation.

**Definition 2.** A state is *reachable at k steps* from another state if there exists a trace of length k such that it is the last state and the first one is that state. A state is *reachable* from another state if it is reachable at any step; otherwise, it is *unreachable*. If not specified explicitly, it means reachable from the initial state.

The safety problem can then be described as whether all the bad states are unreachable. That is, it is to show that there exists no counterexample, where a *counterexample* means a trace to any bad state. In order to prove this problem, we need to introduce the following two terms.

**Definition 3.** A set of states characterized by a propositional formula  $Ind$  is *inductive* iff it contains the initial state and no state belonging to it can transit out of itself.

1.  $Init(S) \rightarrow Ind(S)$  (Base)
2.  $Ind(S) \wedge Tr(S, S') \rightarrow Ind(S')$  (Inductive)

**Definition 4.** A set of states characterized by a propositional formula  $Inv$  is *invariant* iff it holds the property. That is, all the states belonging to it are good.

$$Inv(S) \rightarrow p(S)$$

**Lemma 1.** The safety problem is true iff we can find an inductive and invariant set.

The proof is trivial to show that all the bad states are not reachable at any step by



induction. Then the rest problem is how to compute the inductive invariant set. A straightforward way is to compute the set of all the reachable states, which is actually the smallest inductive set. This scheme is called *exact reachability analysis*, which is adapted by BDD and some SAT-based techniques like [25]. A basic procedure for exact reachability analysis is to compute the image of the current set of states until fixpoint while starting from the initial state. The *image* means the set of states that can transit from any state of the current set. That is,  $\text{image}(R) = \{s' \mid \exists s \in R, \text{Tr}(s, s') = \mathbf{TRUE}\}$ . Furthermore, we say the set reaches a *fixpoint* if it becomes inductive and cannot transit to any other state.

---

**Algorithm 1:** *safetyByExactReach*

---

**Input:** Transition system  $M$ , property  $p$   
**Output:** safe or violated

- 1  $R \leftarrow \text{Init}$
- 2 **while not**  $R(S) \wedge \text{Tr}(S, S') \rightarrow R(S')$
- 3      $R \leftarrow R \vee \text{image}(R)$
- 4 **if**  $R \wedge \neg p = \text{SAT}$
- 5     **return** violated
- 6 **else**
- 7     **return** safe

---

However, the computing effort is too expensive so that it is undesirable to perform on large design. Instead, we compute an approximation of the reachable states in more affordable way. In order to use the approximation for proof, we need to make sure the approximation maintains an ordering relative to the image, which means one must be a subset of the other. *Under-approximation* assures that the approximation is contained in the image; otherwise, *over-approximation* ensures the approximation being a superset of the image. The modern SAT-based algorithms like IMC or PDR are usually over-approximations. We can still complete the proof if the obtained inductive set is also invariant, but we cannot answer that the property is violated otherwise since there may exist a smaller inductive set that holds the property. Whenever we encounter a counterexample, we need to verify whether it is real or just spurious. Hence, the procedure for over-approximation usually combines image computation and spurious

counterexample refinement to form a whole picture. The refinement for IMC is just discarding the current set and restart the computation after unrolling one more timeframe. On the other hand, PDR uses blocking clause to exclude the spurious counterexample and form the image simultaneously.



## 2.4 Property Directed Reachability

In this section, we introduce the method of basic PDR to facilitate the explanation of our method in the next chapter. The content basically follows the implementation in [18] with proper rearrangement and renaming.

How PDR treats the reachability is to continuously refine the over-approximation whenever a spurious counterexample is found by adding clauses to refute it. Throughout the process, PDR maintains a trace of frames  $\langle F_0, F_1, \dots, F_n \rangle$ , where *frame* is a set of clauses over-approximating the states reachable at a certain step or less. The trace has the following 5 properties:

1.  $F_0 = Init$ .
2.  $\forall i > 0$ ,  $F_i$  is set of clauses and is represented by the intersection of them.
3.  $\forall i > 0$ ,  $F_{i+1}$  is a subset of  $F_i$  and all the clauses in  $F_i$  contain the initial state, which means  $\forall i \geq 0$ ,  $F_i \rightarrow F_{i+1}$ .
4.  $F_i(S) \wedge Tr(S, S') \rightarrow F_{i+1}(S')$ ,  $F_{i+1}$  over-approximates the image of  $F_i$ .
5.  $F_i \rightarrow p$ , the frame holds the property except for the tail of the trace.

The pseudo code of the main process is shown in Algorithm 2. In addition to that the first frame  $F_0$  is initial, we keep another special frame  $F_\infty$  that represents the current inductive set. We can always place  $F_\infty$  behind the tail of the trace without violating the above 5 rules. The length of trace keeps increasing over each iteration. For each new frame except for the first one, it starts by copying the clauses from  $F_\infty$ . Then we can try

to propagate the clause in the previous frame to the current one, which is performed in *propBlockedCubes*. The fixpoint checking is also included in this section. If there still exists a witness of violation in the last frame, we need to check if it is real or spurious. This is done by *recBlockCubes*. After the tail of the trace becomes invariant, we proceed to the next frame.

---

**Algorithm 2:** *safetyByPDR*

---

**Input:** Transition system  $M$ , property  $p$   
**Output:** safe or violated

```

1  $F_0 \leftarrow \text{Init}$ 
2  $F_\infty \leftarrow \text{Tautology}$ 
3 for  $curFrame = 0$  to  $\infty$ 
4   while TRUE
5      $badCube \leftarrow \text{getBadCube}(p, curFrame)$ 
6     if  $badCube \neq \text{NULL}$ 
7       if not  $\text{recBlockCubes}(badCube, curFrame)$ 
8         return violated
9     if  $\text{propBlockedCubes}(curFrame)$ 
10    return safe

```

---

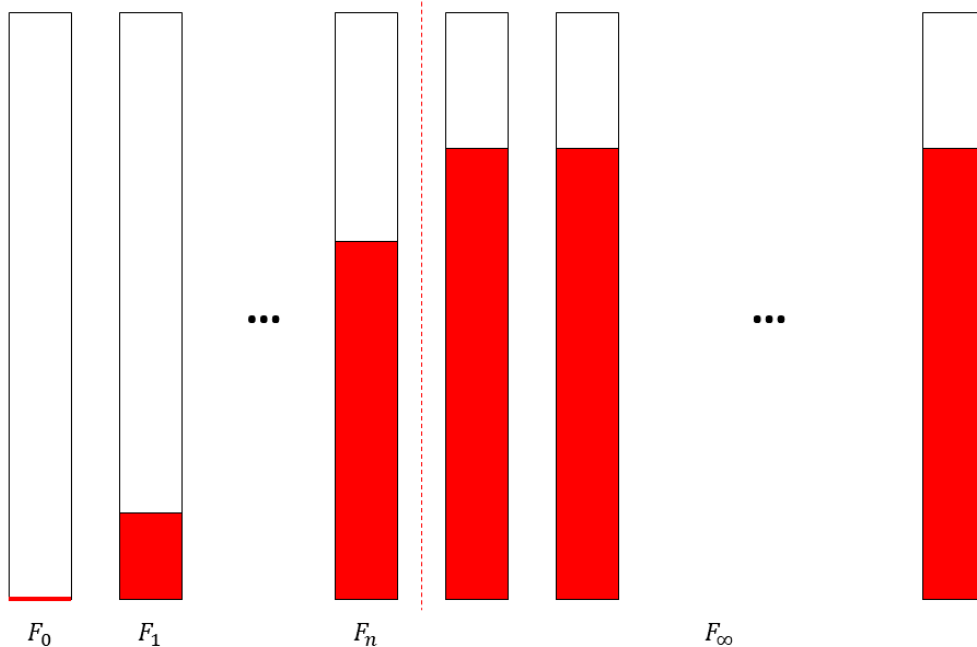
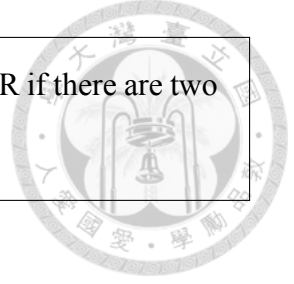


Figure 2.1: The depiction of trace of frame.  $F_\infty$  represents all the frames beyond the current frame. When newing a frame, just add one of them into consideration.



**Theorem 1.** At the end of each iteration, the safety is proved by PDR if there are two adjacent frames with the same size.

*Proof :*

We show that the set of states of the two frames are inductive and invariant.

1. The size of the two frames are the same.

By Property 3, the two frames have the same clauses. That is, the same set of states.

$$|F_i| = |F_{i+1}| \Rightarrow F_i = F_{i+1}$$

By Property 4, this set is inductive.

$$F_i(S) \wedge Tr(S, S') \rightarrow F_{i+1}(S') = F_i(S')$$

2. At the end of iteration, all the frames holds the property. It is invariant.

$$F_i \rightarrow p$$

Hence,  $F_i$  is both inductive and invariant. By Lemma 1, the safety is proved. ■

In the following subsections, we introduce each component of PDR to explain the algorithm and its correctness more clearly.

### 2.4.1 The Monotonicity of Frames

**Definition 5.** A set of states  $F$  is *inductive relative to* another one  $G$  iff it contains the initial state and no state belonging to both  $G$  and  $F$  can transit out of  $F$  [17].

1.  $Init(S) \rightarrow F(S)$  (Base)
2.  $F(S) \wedge G(S) \wedge Tr(S, S') \rightarrow F(S')$  (Inductive)

When a cube is examined whether it is reachable at a certain frame, only the previous frame is involved. We check if there exists any state in the previous frame that can transit into the cube. If the answer is not, we can add the negation of the cube as *blocking clause* to block it at the frame. We will use *blocking clause* and *blocked cube* interchangeably since they are just negation of each other. A special feature for PDR about the checking is to query whether  $F_{k-1}(S) \wedge \neg c(S) \wedge Tr(S, S') \rightarrow \neg c(S')$  holds, which means  $\neg c$  is inductive relative to  $F_{k-1}$ . It is the term  $\neg c(S)$  that enables the addition of blocking clause to all the frames before the target one.

**Theorem 2.** If the addition of blocking clause meets the following criteria, the trace of frames holds the Property 4.

$\neg c$  is added to  $F_1 \sim F_k$  if it contains the initial state and is inductive relative to  $F_{k-1}$ .

1.  $c \wedge Init = \text{UNSAT}$
2.  $F_{k-1}(S) \wedge \neg c(S) \wedge Tr(S, S') \wedge c(S') = \text{UNSAT}$

*Proof :*

For simplicity, we only keep two  $F_\infty$  from Figure 2.1 to preserve the inductive property of itself. For consistency of indices, we assign the two  $F_\infty$  to be  $F_{n+1}$  and  $F_{n+2}$ . In addition, we assume to add the clause to the first frame even if it actually does not since this is equivalent. Then it turns out that we need to show

$$\forall i \in [0, n+1], F'_i(S) \wedge Tr(S, S') \rightarrow F'_{i+1}(S'),$$

$$\text{where } F'_i = \begin{cases} F_i \wedge \neg c, & \forall i \in [0, k] \\ F_i, & \forall i \in [k+1, n+2] \end{cases}$$

1. For  $i > k$ , nothing changes.



2. For  $i = k$

$$\begin{aligned}
 F'_i(S) \wedge Tr(S, S') &= F_i(S) \wedge \neg c(S) \wedge Tr(S, S') \\
 &\rightarrow F_i(S) \wedge Tr(S, S') \\
 &\rightarrow F_{i+1}(S') = F'_{i+1}(S')
 \end{aligned}$$

3. For  $i < k$

(i)  $\neg c$  is inductive relative to  $F_i$

$$\begin{aligned}
 F'_i(S) \wedge Tr(S, S') &= F_i(S) \wedge \neg c(S) \wedge Tr(S, S') \\
 &\rightarrow F_{k-1}(S) \wedge \neg c(S) \wedge Tr(S, S') \\
 &\rightarrow \neg c(S')
 \end{aligned}$$

(ii) Only adding  $\neg c$  to  $F_i$  holds the inductive property

$$\begin{aligned}
 F'_i(S) \wedge Tr(S, S') &= F_i(S) \wedge \neg c(S) \wedge Tr(S, S') \\
 &\rightarrow F_i(S) \wedge Tr(S, S') \\
 &\rightarrow F_{i+1}(S')
 \end{aligned}$$

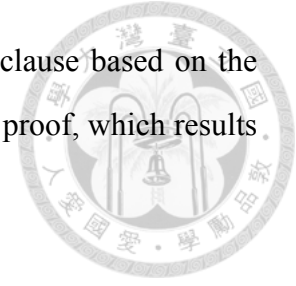
(iii) By composition of the above two formula

$$F'_i(S) \wedge Tr(S, S') \rightarrow F_{i+1}(S') \wedge \neg c(S') = F'_{i+1}(S')$$

Combine the three cases, we know that Property 4 holds. ■

The result is that the former frame must be a subset of the latter one (Property 3). That is, the set of clauses monotonically decreases along the trace. Hence, in the implementation, we just record the difference of adjacent frames to prevent duplication. In order to reuse the SAT solver, we use assumption to activate the frames and mark the

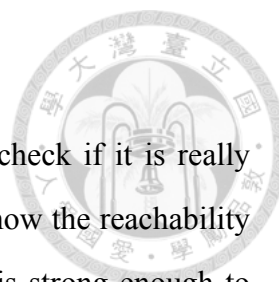
cube. For modern SAT solver, it is able to derive a final conflict clause based on the assumption. We can remove the literals that do not participate in the proof, which results in a higher frame and a larger cube.



## 2.4.2 Ternary Simulation

From the witness of violation in the last frame, if a cube cannot be blocked at a certain frame, we can find a state in the previous frame that cause the reachability and recursively do the blocking. We can use ternary simulation to collect more states with the same property to form a larger cube. The transition for ternary logic on AND gate includes  $\{0 \wedge X = 0, 1 \wedge X = X, X \wedge X = X \text{ and } \neg X = X\}$  besides the basic Boolean operation. We first define *target* as a set of signals with the value we want. It can also be described as a cube since all the values should hold simultaneously, but is not restricted to only state variable. In *getBadCube*, the target is the violation of property  $\neg p$ . In *checkReach*, the target is the given cube  $c$ . Starting with a normal simulation, the value of the given state can transit to the target. Now we force one variable in the state to be  $X$  and perform the ternary simulation on the circuit. If there is no signal in the target becoming  $X$ , the changed variable is actually don't-care so it can be set to  $X$ . Otherwise, we need to reset the variable to its original value since it is crucial for the transition. The iteration can be performed for every state variable to find as much don't-care as we can. We can call the obtained cube as *proof obligation* since the whole cube should be blocked at every frame to prove the safety. Note that this statement makes sense since the obtained cube of a proof obligation is still a proof obligation.

**Lemma 2.** All the states in the cube obtained by ternary simulation can transit to its target. (Only consider the variables in the target.)



### 2.4.3 Recursively Blocking Cubes

Starting from a witness of violation in the last frame, we need to check if it is really reachable from the initial state. From the previous subsection, we know the reachability checking only involves the previous frame. If the previous frame is strong enough to ensure the unreachability, we just add blocking clause to every corresponding frame. However, if it is too weak to refute, we still do not know whether the proof obligation is reachable or not. If we want to block the proof obligation in this frame, we first need to block the state in the previous frame that can transit to it. Hence, we extract the assignment from the SAT solver to form a new proof obligation at the previous frame. If the blocking of the new proof obligation succeeds, we can go back to the original one and try to block it again. But if it fails, just repeat the same process at more previous frames recursively. The extreme case is reaching  $F_0$ , which means no refinement can be done anymore and shows the presence of a counterexample. Otherwise, at the end of the subroutine, all the frames are refined to be strong enough to block the first proof obligation in the last frame.

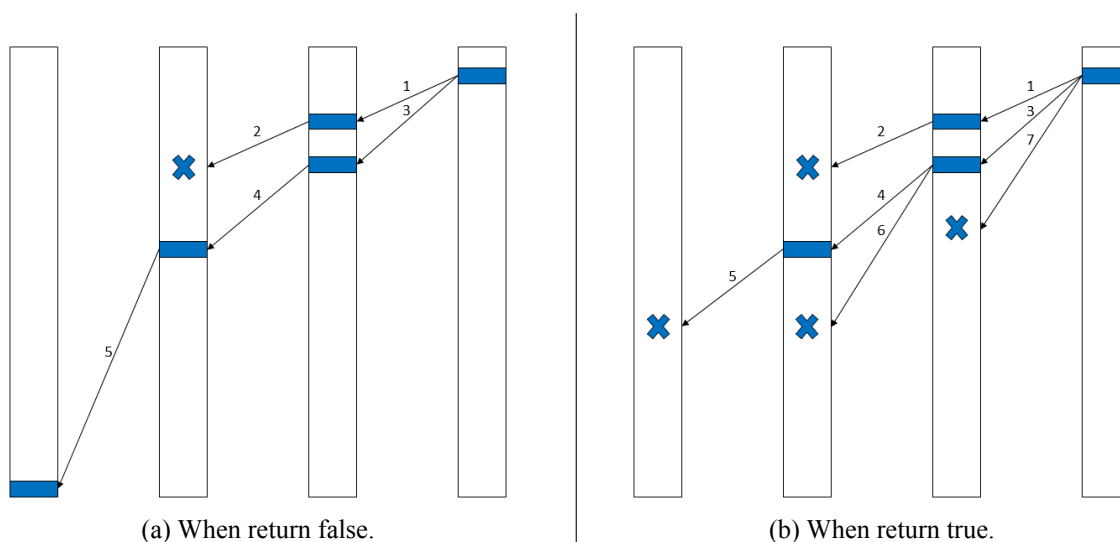
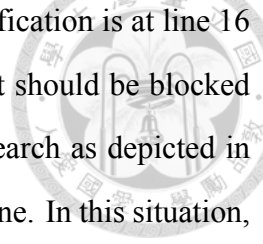


Figure 2.2: The depiction of `recBlockCubes`. The blue rectangle means proof obligation. The X means unreachable from the previous frame. The arrow means the reachability checking. The number associated with each arrow means the order to perform.





The process is illustrated by Algorithm 3. The most special modification is at line 16 where we push the blocked proof obligation to further frame since it should be blocked everywhere. Without this setting, the process acts as a depth first search as depicted in Figure 2.2 where the depth is between the current frame and the last one. In this situation, a stack is enough to record the trace of proof obligations. However, with this setting, there will be multiple proof obligations waiting at a frame to be blocked so that we should arrange more space to record. In addition, an integer *blockFrame* is introduced to indicate where the blocking takes place now. If a proof obligation should be blocked at the first frame, we know a counterexample is found and return false. We always try to block the proof obligation with the lowest frame throughout the process. When we focus on a frame, we can pick any of the proof obligation in that frame to block. First we check if it is already blocked at that frame by simply subsumption or even SAT query. We then check the reachability relative to the previous frame if the answer is not. If it is reachable from the previous frame, go to deeper frame recursively. Otherwise, add blocking clause properly after performing generalization on it. After emptying the task for a frame, go back to the shallower one. If all the proof obligations are blocked without reaching the first frame, the first one is really unreachable and true is returned. Last, we introduce the term *bad depth* to represent the distance of proof obligation between the last one. The bad depth of the last proof obligation is 0 since it is exactly the violation of property, while the bad depth of the rest proof obligations is one more than that of the one producing it.

#### 2.4.4 Propagating Blocked Cubes

The process to propagate cube is illustrated in Algorithm 4. It is very simple that checking the reachability at the next frame for every blocking clause in every frame. For the detail of implementation, we only check the difference between two adjacent frames since the storage is exactly like that. To check whether the two frames are equivalent, we can just check if the number of difference between two frames is zero.



---

**Algorithm 3: *recBlockCubes***

---

**Input:** Cube *badCube*, the frame to block *f*

**Output:** TRUE or FALSE

**Data:** An infinite array of set of cubes *badArr*

```
1 badArr[f].add(badCube)
2 blockFrame  $\leftarrow$  f
3 while blockFrame  $\leq$  f
4   if blockFrame = 0
5     return FALSE
6   targetCube  $\leftarrow$  badArr[blockFrame].pop()
7   if not isBlocked(targetCube, blockFrame)
8     preCube  $\leftarrow$  checkReach(targetCube, blockFrame)
9     if preCube  $\neq$  NULL
10      badArr[blockFrame].add(targetCube)
11      blockFrame  $\leftarrow$  blockFrame - 1
12      badArr[blockFrame].add(preCube)
13   else
14     addBlockedCube(generalize(targetCube, blockFrame))
15     if blockFrame < f
16       badArr[blockFrame + 1].add(targetCube)
17     if badArr[blockFrame].empty()
18       blockFrame  $\leftarrow$  blockFrame + 1
19 return TRUE
```

---

---

**Algorithm 4: *propBlockedCubes***

---

**Input:** The currently maximum frame *curFrame*

**Output:** TRUE or FALSE

```
1 for k = 1 to curFrame
2   for all cubes c  $\in$  Fk
3     if checkReach(c, k + 1) = NULL
4       addBlockedCube(c, k + 1)
5   if Fk = Fk+1
6     return TRUE
7 return FALSE
```

---

## 2.4.5 Other Subroutines

Generalization in PDR means to gain more information from a proof obligation as we can. The information here is to block larger cube in higher frame, which means to get closer to the exact reachability. The generalization process roughly contains three phases.

1. Use the final conflict clause produced by SAT solver to remove the literals not related to the UNSAT proof, which leads to a larger cube and a higher frame. This procedure is performed in every *checkReach*, which means it is also activated by the rest two steps.
2. More eagerly, iteratively check the reachability by ignoring one literal at a time. If the resulting cube is still unreachable, the literal will be actually removed. Note that the cube cannot intersect with the initial state, or the criterion of correctness fails.
3. By fixing the cube, check if it is unreachable at further frame until the pushing fails.

**Definition 6.** A cube  $c_1$  subsumes another one  $c_2$  iff all the literals in  $c_1$  are also in  $c_2$ .

$$\{l \mid l \in c_1\} \subseteq \{l \mid l \in c_2\}$$

Subsumption is another basic part in PDR. It is performed at two places.

1. To check a proof obligation is blocked at the frame or not. If it is subsumed by any of the blocking clause, it has already been blocked.
2. Before adding a blocking clause to the corresponding frame, all the clauses subsumed by it in that frame are removed to prevent redundancy.



## Chapter 3

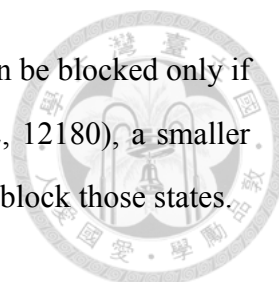
# Predicate Extraction

In this chapter, we explain our method, property directed reachability with predicate extraction. We first describe the problem we observe and how it inspires us to come up with the solution. Then a simple general method by predicate solving is proposed with the proof of its correctness. Finally, we give two examples of predicate to demonstrate the feasibility and efficiency.

### 3.1 Motivation

Among various implementations of PDR, the final inductive and invariant sets are all distinct due to the don't-cares. Although not directly related to the runtime, the number of clauses somewhat illustrates how efficiently the engine solves the problem. There may exist many possible inductive invariant sets, so the problem is how to find a proper one from them with elegant representation of clauses. Hence we collect the inductive invariant set with the form of clauses for those safe benchmarks. By studying these cases, we observe that there often exist similarities among the clauses.

6s288r is a classic example depicted in Figure 3.1. For most of the clauses in the figure, they all share a common part (enclosed in the red box) and only differ by one literal. Intuitively, if the common part is added during the process, we may save the effort to prove those clauses. However, at first glance, this kind of pattern seems to tell



us that it is a reachable state that hinders the blocking since a cube can be blocked only if all the states in it are unreachable. That is, in cube (12141, !12142, 12180), a smaller cube or even a minterm may be reachable so that PDR must avoid to block those states.

```

(706) Size = 2, Lit = 12140 12171
(707) Size = 2, Lit = 12140 12173
(708) Size = 2, Lit = 12140 12175
(709) Size = 4, Lit = 12141 !12142 12180 12199
(710) Size = 4, Lit = 12141 !12142 12180 12452
(711) Size = 4, Lit = 12141 !12142 12180 12556
(712) Size = 4, Lit = 12141 !12142 12180 12560
(713) Size = 4, Lit = 12141 !12142 12180 12704
(714) Size = 4, Lit = 12141 !12142 12180 12706
(715) Size = 4, Lit = 12141 !12142 12180 12716
(716) Size = 4, Lit = 12141 !12142 12180 12725
(717) Size = 4, Lit = 12141 !12142 12180 12730
(718) Size = 4, Lit = 12141 !12142 12180 12732
(719) Size = 4, Lit = 12141 !12142 12180 12741
(720) Size = 4, Lit = 12141 !12142 12180 12743
(721) Size = 4, Lit = 12141 !12142 12180 12751
(722) Size = 4, Lit = 12141 !12142 12180 12753
(723) Size = 4, Lit = 12141 !12142 12180 12761
(724) Size = 4, Lit = 12141 !12142 12180 12764
(725) Size = 4, Lit = 12141 !12142 12180 12766
(726) Size = 4, Lit = 12141 !12142 12180 12768
(727) Size = 4, Lit = 12141 !12142 12180 12773
(728) Size = 4, Lit = 12141 !12142 12180 12775
(729) Size = 4, Lit = 12141 !12142 12180 12777
(730) Size = 4, Lit = 12141 !12142 12180 12784
(731) Size = 4, Lit = 12141 !12142 12180 12786
(732) Size = 4, Lit = 12141 !12142 12180 12787
(733) Size = 4, Lit = 12141 !12142 12180 12792
(734) Size = 4, Lit = 12141 !12142 12180 12802
(735) Size = 4, Lit = 12141 !12142 12180 12807
(736) Size = 4, Lit = 12141 !12142 12180 14200
(737) Size = 4, Lit = 12141 !12142 12180 14201
(738) Size = 4, Lit = 12141 !12142 12180 14202
(739) Size = 4, Lit = 12141 !12142 12180 14293
(740) Size = 4, Lit = 12141 !12142 12180 14319
(741) Size = 5, Lit = 12141 !12142 12205 !12223 12554

```

Figure 3.1: 36 blocked cubes out of the 889 ones in the inductive invariant set of 6s288r. The number in the parentheses means the index. The numbers after "Lit" are the IDs of latch variable. Exclamation mark means the literal is complement.

Surprisingly, by directly set this cube to be the property, we easily prove that the whole cube is unreachable by PDR itself. It only costs five clauses for the inductive invariant set within tens of milliseconds. Now the question is why PDR struggles to block the bad states spreading around the cube by so many clauses instead of only one. (For convenience, we extend the term *bad* to describe any state that leads to the violation eventually since the result is equivalent.) Here we restate the seemingly reachable state as *hard-to-block* state and provide a possible reason in the next paragraph.

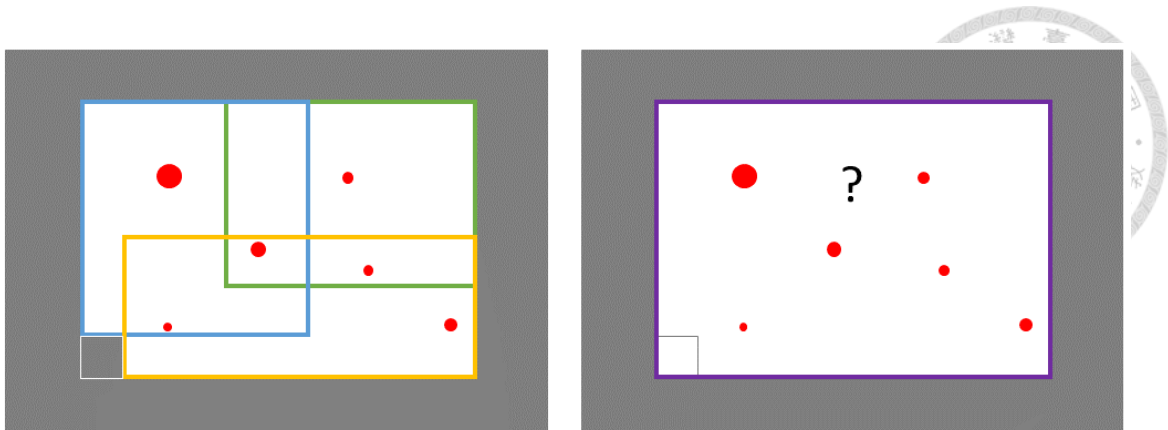


Figure 3.2: The depiction of the similar clause problem. The largest rectangle is the whole Boolean space. The middle rectangle means the cube as common part. There are many proof obligations (red spot) spreading around the cube. The gray area denotes that the state is reachable while the unreachable state in the white area has been blocked. The left subgraph illustrates what we observe. In order not to touch the lower-left cube, many blocking clauses are needed to tackle with the blocking. However, there is actually no such a state, as in the right subgraph. Why not just use a larger clause to represent it?

From the details introduced in the previous chapter, we know that every blocking clause is generated from a proof obligation. When excluding a proof obligation, we also block some other states by removing some literals of it in generalization process. A don't-care state that is neither reachable nor bad is included in the final result if there is no proof obligation generalized to block it. Hence, the reachability depends on the appearance of proof obligation to a high degree. Corresponding to the name, property directed reachability, the reachability is computed by continuously eliminating the violation we want to prove. This scheme makes sense since we can just focus on the point of the problem. However, it sometimes prevents us to make better use of the don't-care state. A state can be proven to be unreachable only if all the predecessors of it have been proven earlier. The hard-to-block state must be don't-care since it will eventually be blocked if it is bad. In addition, we know that all the predecessors of it are don't-care as well. If any of the predecessors is difficult to be blocked with the guide of the obtained proof obligation during the procedure, it probably remains reachable until the end. Therefore, PDR gives up for proving the unreachability of the hard-to-block state immediately without digging into the proof recursively just because it is not a bad

state. Consequently, PDR tends to block around the hard-to-block state since it cannot assure that it is unreachable, which leads to that many similar clauses.

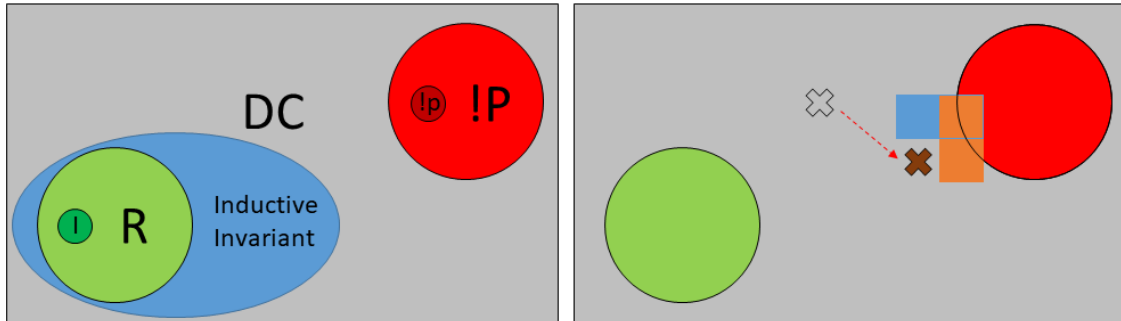


Figure 3.3: The depiction of the reason for the similar clause problem.  $I$ ,  $R$ ,  $DC$ ,  $!p$ ,  $!P$  means the initial state, the reachable states, the don't-cares, the violation of property and the bad states, respectively. As the left subgraph, an inductive invariant set must include  $R$  and exclude  $!P$ . For the right subgraph, in order to block the bad states, two blocking clauses are involved (blue and orange). During generalization of them, they all want to include the brown states. However, the predecessor of the brown state has little connectivity to the property. Hence, PDR fails to prove their unreachability continually.

The inspiration for us is straightforward. If we can force the procedure to prove for some property more eagerly, we can make better use of the don't-care states to find a more efficient and elegant clause representation. Hence we let the procedure solve the candidate separately if they find something weird, just as what we human beings do. If it really helps, the obtained information can be used to help the original main procedure. Now, two phases are needed, including how to extract some patterns as predicate to solve and how to get information from it.

### 3.2 General Method and Correctness

As shown in Algorithm 5, the procedure is very simple. Given a predicate as any form to represent an assumed property, we start another PDR procedure with resource limit to work on it. Note that the procedure is activated without predicate solving to ensure the completeness. The resource limit is determined by the number of SAT query to indicate the effort PDR has spent. We can modify  $L$  in accordance with how much feedback we

expect to get from it. The procedure terminates if the answer is proved or the resource limit is used up. If the result is safe, we can collect the inductive invariant set. Otherwise, we can still get  $F_\infty$  from it. For both case, we obtain an inductive set containing only clauses. Then we directly merge it back to the  $F_\infty$  of the original main procedure. The merge is trivial for two CNF since they basically do nothing except for some detailed checking like subsumption. After finishing the solving, PDR turns back to what it does previously and just go on without preparing anything for the correctness.

---

**Algorithm 5:** *solvePredicate*

---

**Input:** Predicate  $p'$  and the limit for SAT query  $L$

**Output:** None

```

1 Set SAT query limit to  $L$ 
2  $result \leftarrow pdrMain(M, p')$ 
3 if  $result = \text{safe}$ 
4    $ind \leftarrow collectIndInv()$ 
5 else // violated or unknown
6    $ind \leftarrow getInfFrame()$ 
7  $F_\infty \leftarrow F_\infty \wedge ind$ 

```

---

**Theorem 3.** Property directed reachability with predicate solving is sound.

*Proof :*

1. Property 1 and 5 are not related to this modification and hold trivially.
2. The merged set  $ind$  is composed of clauses and does not exclude the initial state so that Property 2 and 3 still hold.
3. By following the proof for Theorem 2, we show that Property 4 holds.

$ind$  is inductive itself so that inductive relative to  $F_\infty$

$$F_\infty(S) \wedge ind(S) \wedge Tr(S, S') \rightarrow ind(S) \wedge Tr(S, S') \rightarrow ind(S')$$

Then it just means to add a set of clauses instead of only one.

The 5 properties still hold with the aid of predicate solving. The algorithm is sound. ■





### 3.3 Two Kinds of Predicate

We have introduced how to extract information from the predicate. It can be performed on predicate of general form. The rest problem is how to find an effective predicate. More precisely, we want to find the predicate within reasonable time compared to how much it gains for us. However, there should not be too much expectation on a single predicate, the improvement is probably established on a wide search. It may be good to spend less time for each checking to find a series of candidates no matter how useful they are.

In the following subsections, we provide two examples that can be easily identified during the solving process. Both of them are expressed by cube since the calculation only involves the essential components of PDR without doing further modification. It is worth mentioning that PDR can be seen as breaking the entire problem into small pieces without handling the whole timeframe at a time. What we do is to observe that whether PDR encounters obstacles during the last period of time. If it solves for a similar problem for a while, we can just make an assumption on that part to help PDR conquer it. Then we expect an improvement by smoothing the solving process with the aid of these local information.

#### 3.3.1 Blocking Clause

As being the problem we mentioned, the first example is to eliminate the similar clauses. Algorithm 6 illustrates the way to identify the candidates. We introduce two parameters to guide how to find the candidates, including Backtrack number ( $B$ ) and Match number ( $M$ ). Generally, each frame can be associated with different value of these two numbers. Backtrack number means how many clauses in the corresponding frame is backtracked to be considered from the last one. If the size of the frame is less than  $B$ , just take the whole frame into consideration. *findCommon* can be composed of any rule to find the common part to solve. Here we just handle the case that there is only one difference of literal between two clauses just as what we encountered. The number of occurrence of each

common part is recorded in a structure. Match number serves as a threshold. The candidate is picked only when it occurs more than or equal to  $M$  times. Note that the returned set can contains no or more than one cube. In our implementation, this subroutine is placed after every *addBlockedCube* then *solvePredicate* is activated for every candidate.

---

**Algorithm 6:** *findClsPredicate*

---

**Input:** A blocked cube  $c$  and the frame  $k$  to block it  
**Output:** A set of cubes  
**Data:** A map *candMap* from cube to unsigned number

```

1  $size \leftarrow |F_k|$ 
2  $B, M \leftarrow getParam(k)$ 
3  $n \leftarrow \min(size, B)$ 
4 for  $c'$  in the last  $n$  cubes  $\in F_k$ 
5    $common \leftarrow findCommon(c, c')$ 
6   if  $common \neq \mathbf{NULL}$ 
7     // By default, every cube maps to 0
8      $candMap[common] \leftarrow candMap[common] + 1$ 
9 return  $\{c' \mid candMap[c'] \geq M\}$ 

```

---

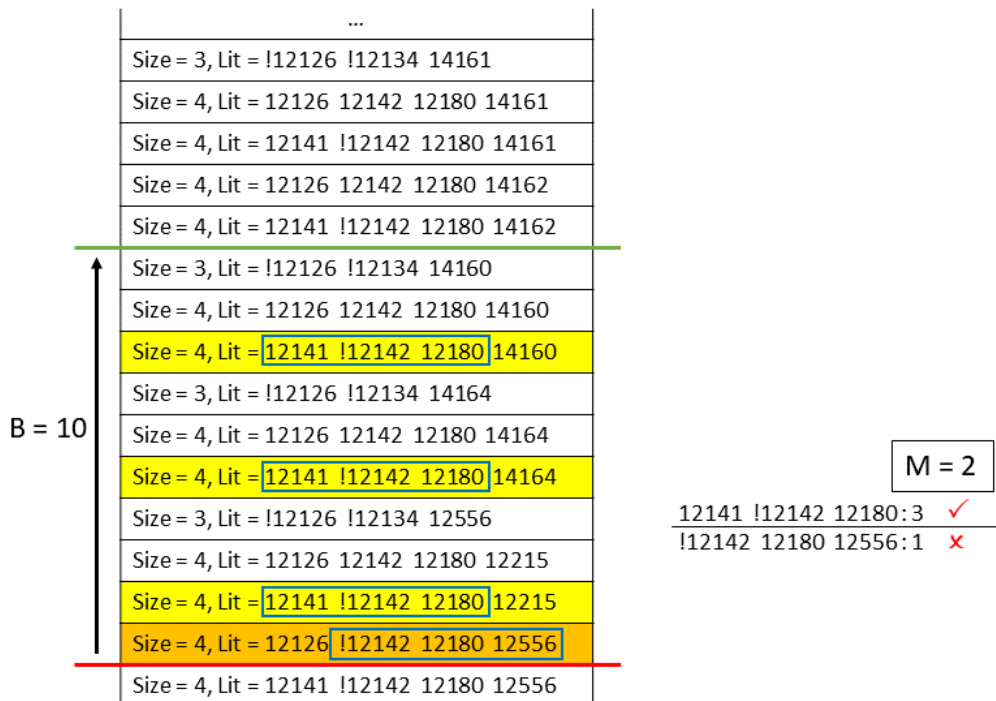
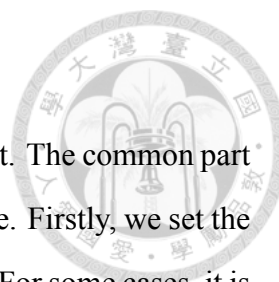


Figure 3.4: An example for clause-based predicate. After adding the cube to its corresponding frame, only the last several cubes are considered to find the common part. There may be multiple common parts, but only the ones occurring more than the threshold are picked.



### 3.3.2 Proof Obligation

For some cases, all the proof obligations for them share a common part. The common part here means the intersection of literals and we collect them to be a cube. Firstly, we set the negation of the cube as the initial state and keep the original property. For some cases, it is still safe, which means all of the bad states really locate in that cube. However, if we keep the initial state and set the cube as the property. They are easily proved to be violated, that is, not all of the reachable states locate in the negation of that cube. In Figure 3.5, we show the possible distribution of reachable and bad states.

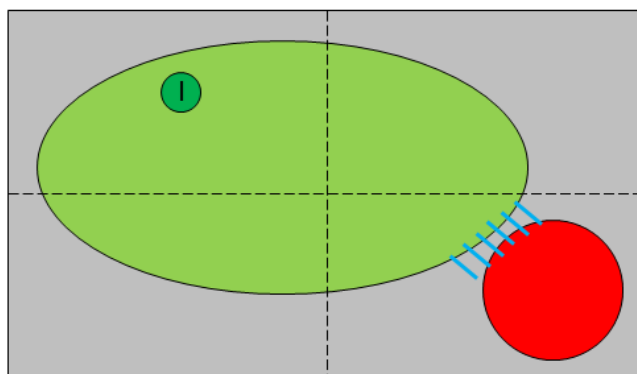


Figure 3.5: The possible distribution of reachable and bad states for similar obligation problem. The lower-right rectangle means the cube of shared literals. The difficulty of this kind of problem is probably to show that the reachable states do not intersect with the bad states in this cube.

The global constraint is too loose to represent a useful information. It turns out that local information makes sense again. We present the procedure to collect the common part in Algorithm 7. A set of cube and a counter are kept globally since the result becomes available after a period of accumulation. At each time of collection, the counter increases by one to indicate that one more proof obligation is considered. Furthermore, we continuously compute the common part by the given obligation. However, we do not return predicate in this subroutine. This is not like the case for blocking clause that the candidate can be produced at every query to find predicate. Algorithm 8 demonstrate when and how to use obligation as predicate. The collection is performed when the proof obligation is shown to be not blocked by this frame. After checking the reachability, we

check the availability of predicate only if the obligation can be reached from the previous frame. We introduce a parameter called Obligation threshold ( $T$ ) here. If the cumulative number of obligations exceeds  $T$ , we collect the common part as a cube to be predicate if it is not empty. After checking the predicate, we reset the cumulative number and continue to block cube recursively.

---

**Algorithm 7:** *findOblPredicate*

---

**Input:** A proof obligation  $o$

**Output:** None

**Data:** A set of literals  $common$  and an unsigned number  $numObl$

(Keep globally. By default,  $common = \emptyset, numObl = 0$ )

```

1  $numObl \leftarrow numObl + 1$ 
2 if  $numObl = 1$ 
3    $common \leftarrow \{l \mid l \in o\}$ 
4 else
5    $common \leftarrow common \cap \{l \mid l \in o\}$ 

```

---

Obligation	Common Part	# Obligation	Predicate	
$a b' c f$	$a b' c f$	1	X	← Check (T = 3)
$b' c d e' f$	$b' c f$	2	NULL	← Check (T = 3)
$a b' c d$	$b' c$	3	X	← Check (T = 3)
$b' c d e$	$b' c$	4	$b' c$	← Check (T = 3)
$a b c d' e$	$a b c d' e$	1	X	← Check (T = 3)
$a' c d' f g$	$c d'$	2	X	← Check (T = 3)
$a' d e' f$	None	3	NULL	← Check (T = 3)

Figure 3.6: An example for obligation-based predicate. At each beginning, all the literals are collected. During the process, intersection is performed to pick up the sharing literals. At the time to check, if the cumulative number of obligations is still smaller than the threshold or there is no common part during this period, NULL is returned to disable the predicate solving; otherwise, a cube is returned as a predicate by gathering the rest literals.



---

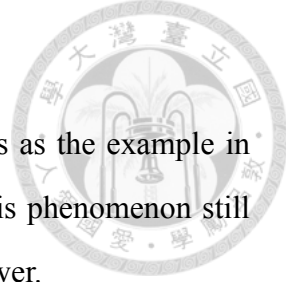
**Algorithm 8:** *recBlockCubes<sub>obl</sub>*

---

**Input:** Cube *badCube*, the frame to block *f*  
**Output:** TRUE or FALSE  
**Data:** An infinite array of set of cubes *badArr*

```
1 badArr[f].add(badCube)
2 blockFrame ← f
3 while blockFrame ≤ f
4   if blockFrame = 0
5     return FALSE
6   targetCube ← badArr[blockFrame].pop()
7   if not isBlocked(targetCube, blockFrame)
8     findOblPredicate(targetCube)
9     preCube ← checkReach(targetCube, blockFrame)
10    if preCube ≠ NULL
11      badArr[blockFrame].add(targetCube)
12      blockFrame ← blockFrame − 1
13      badArr[blockFrame].add(preCube)
14      if numObl ≥ T
15        if not common.empty()
16          solvePredicate(toCube(common), L)
17          numObl ← 0
18      else
19        addBlockedCube(generalize(targetCube, blockFrame))
20      if blockFrame < f
21        badArr[blockFrame + 1].add(targetCube)
22      if badArr[blockFrame].empty()
23        blockFrame ← blockFrame + 1
24 return TRUE
```

---



### 3.3.3 Recap for 6s288r

In this subsection, we provide more details about 6s288r that serves as the example in Section 3.1 by the following figures. Although the real cause of this phenomenon still needs to be verified, these facts can be seen as clues for the final answer.

```

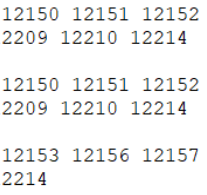
Check: frame = 3, BadDepth = 8, Size = 178, Lit = !12126 12128 12129 !12130 !12131 12132 12134
!12135 12137 !12138 12139 !12140 12141 !12142 !12160 !12161 !12162 12164 !12166 !12168 !12170
!12172 !12174 !12179 12180 !12184 12185 !12186 !12189 12190 !12192 !12194 12195 !12197 !12199
!12200 !12201 12203 !12205 !12206 !12207 !12208 12215 12316 12320 12324 12328 12332 !12336
!12337 !12338 !12339 !12340 !12341 !12342 !12343 !12344 !12345 !12346 !12347 !12348 !12349
!12350 !12351 !12352 !12353 !12354 !12355 !12356 !12357 !12358 !12359 !12360 !12361 !12362
!12363 !12364 !12365 !12366 !12367 !12368 !12369 !12370 !12371 !12372 12445 !12452 12453
!12454 !12455 12458 !12530 12537 12545 12550 !12553 12555 !12556 !12558 !12677 !12678 12711
!12726 12742 12756 12769 12778 !12788 !12793 12807 !13098 !13101 !13102 !13104 !13106 !13107
!13109 !13110 !13111 !13112 !13113 !13114 !13115 !13116 !13117 !13791 13799 !13809 13819 13829
13839 13849 !13859 13869 !13879 !13889 13909 13941 !13951 !13961 13971 13981 !13991 !14001
14011 14021 !14031 14041 14051 !14060 14091 !14101 14111 14121 !14131 14141 14151 14161 14171
14181 14201 !14211 !14240 !14241 !14242 !14299 !14300 !14301 !14302 !14303 !14304 !14305
!14307 !14308 !14309 !14310 !14311 !14312
-----
UNSAT generalization: frame = 3, Size = 4, Lit = 12141 !12142 12180 14201

Check: frame = 2, BadDepth = 8, Size = 92, Lit = !12126 12128 12129 !12130 !12131 12132 12134
!12135 12137 !12138 12139 !12140 12141 !12142 !12160 !12161 !12162 !12164 !12166 !12168 !12170
!12172 12174 !12179 12180 !12184 12185 !12186 !12189 12190 12192 !12194 12195 !12197 !12199
!12200 !12201 !12203 !12205 !12207 !12208 12215 12316 12320 12324 12328 12332 !12336 !12446
!12447 !12448 !12449 !12450 !12451 !12452 12453 !12454 !12455 !12530 12537 12545 12548 !12556
!12677 !12678 !12681 !12683 !12697 12708 12716 12732 !12743 12751 12753 !12775 !12786 !12789
!12802 !12807 !13098 !13101 !13102 !13106 !13107 !13108 !13115 !13117 !13791 !14240 !14241
!14242 !14299
-----
UNSAT generalization: frame = 2, Size = 4, Lit = 12141 !12142 12180 12753

Check: frame = 3, BadDepth = 8, Size = 123, Lit = !12126 12128 12129 !12130 !12131 12132 12134
!12135 12137 !12138 12139 !12140 12141 !12142 !12160 !12161 !12162 12164 !12166 !12168 !12170
!12172 !12174 !12179 12180 !12184 12185 !12189 12190 12192 !12194 12195 !12197 !12199 !12200
!12201 !12203 !12205 !12207 !12208 12212 !12215 12222 12226 12230 12234 12237 12240 12243
12246 12250 12253 12256 12259 12262 12265 12268 12271 12274 12277 12280 12283 12286 12289
12292 12295 12298 12301 12304 12307 12310 12313 12317 12321 12325 12329 12333 !12336 !12408
!12446 12447 12448 12449 12450 !12451 !12452 12453 !12454 !12455 !12530 12537 12545 12548
!12556 !12677 !12678 !12681 !12683 !12697 !12706 !12708 12730 12743 !12751 !12753 !12770
!12780 !12786 !12802 !12807 !13098 !13101 !13102 !13104 !13106 !13107 !13115 !13117 !13791
!14240 !14241 !14242 !14299
-----
UNSAT generalization: frame = Inf, Size = 4, Lit = 12141 !12142 12180 12743

```

Figure 3.7: The proof obligations leading to the similar clauses. "Check" means to block proof obligation in *recBlockCubes* and the number after "frame" is the frame to block it. The blocked cube after "UNSAT generalization" with the frame to add it is generated after successfully blocking a proof obligation. Note that the three examples are not under computation in a row, we just collect them together to present. Most of these proof obligations have bad depth equal to 8, and the rest small portion with 14 or 16. We know that these proof obligations locate in the 8th steps from !p. Furthermore, they all share a common part but different for the rest literals and even the length. We then guess that these proof obligations describe a similar scheme of possible violation. Limited by the reachability, PDR has difficulty blocking them all at a time.



```

12141: 12126 12132 12140 12141 12142 12143 12144 12145 12146 12147 12148 12149 12150 12151 12152
12153 12156 12157 12158 12176 12181 12186 12191 12192 12196 12197 12202 12203 12209 12210 12214

12142: 12126 12132 12140 12141 12142 12143 12144 12145 12146 12147 12148 12149 12150 12151 12152
12153 12156 12157 12158 12176 12181 12186 12191 12192 12196 12197 12202 12203 12209 12210 12214

12180: 12126 12132 12134 12140 12141 12142 12147 12148 12149 12150 12151 12152 12153 12156 12157
12158 12176 12180 12181 12186 12191 12192 12196 12197 12202 12203 12209 12210 12214

```

Figure 3.8: The cone of 12141 !12142 12180. The variables (latch ID) after the colons are the respective cone of these three latches. A variable is in the cone iff the function of the latch is dependent of this variable. We can see that the cone of these three latches are similar, and the distribution of variables locates in a range smaller than 100 (12126 ~ 12214).

```

12452: 12126 12132 12134 12140 12141 12142 12147 12148 12149 12150 12151 12152 12153 12154 12155
12156 12157 12158 12162 12164 12166 12168 12170 12172 12174 12176 12181 12186 12191 12192 12196
12197 12202 12203 12209 12210 12214 12452

12556: 12128 12129 12205 12556

12560: 12128 12129 12134 12138 12160 12194 12199 12200 12205 12206 12207 12213 12223 12550 12551
12552 12553 12554 12560

12786: 12126 12128 12129 12130 12131 12132 12134 12581 12786

12787: 12128 12129 12134 12205 12550 12551 12553 12555 12556 12557 12558 12559 12560 12561 12562
12563 12564 12698 12787 12788

12792: 12128 12129 12134 12205 12550 12551 12553 12555 12556 12557 12558 12559 12560 12561 12562
12563 12564 12698 12792 12793 12794

```

Figure 3.9: The cone of the literal along with 12141 !12142 12180 in Figure 3.1. We just take 6 out of them to present. These latches are not necessarily related to 12141, 12142 and 12180, but they all depend on itself.

We have not found a strong evidence to show the cause of the similar clause problem. We just make a guess based on these observations. In the following three figures, the reason that the clauses with 12141 !12142 12180 can be added is by the help of the clauses added before them. This is because the proof obligation generates the others so that we know it is not able to be blocked initially. For the three situations, the first added clause cannot be enlarged anymore. (Become Tautology, 12142 12180, !12126 !12134) We then guess that it is this limit that prevents PDR from removing the literal during generalization.

```

Check: frame = 1, BadDepth = 8, Size = 96, Lit = !12126 12128 12129 !12130 !12131 12132 12134
!12135 12137 !12138 12139 !12140 12141 !12142 !12160 !12161 !12162 !12164 !12166 !12168 !12170
!12172 12174 !12179 12180 !12184 12185 !12186 !12189 12190 12192 !12194 12195 !12199 !12200
!12201 !12205 !12207 !12208 12215 12221 12316 12320 12324 12328 12332 12337 12410 !12447
!12448 !12449 !12450 !12452 12453 !12454 !12455 12458 12494 12531 12537 12545 12548 !12556
!12677 !12678 !12681 !12697 12708 12716 !12730 !12732 !12743 !12751 !12753 !12761 !12764
!12766 !12773 !12775 !12784 !12786 !12802 !12807 !13098 !13101 !13102 !13104 !13106 !13107
!13115 !13117 !13791 !14240 !14241 !14242 !14299
-----
UNSAT generalization: frame = 1, Size = 1, Lit = 12716
-----
Check: frame = 2, BadDepth = 8, Size = 96, Lit = !12126 12128 12129 !12130 !12131 12132 12134
!12135 12137 !12138 12139 !12140 12141 !12142 !12160 !12161 !12162 !12164 !12166 !12168 !12170
!12172 12174 !12179 12180 !12184 12185 !12186 !12189 12190 12192 !12194 12195 !12199 !12200
!12201 !12205 !12207 !12208 12215 12221 12316 12320 12324 12328 12332 12337 12410 !12447
!12448 !12449 !12450 !12452 12453 !12454 !12455 12458 12494 12531 12537 12545 12548 !12556
!12677 !12678 !12681 !12697 12708 12716 !12730 !12732 !12743 !12751 !12753 !12761 !12764
!12766 !12773 !12775 !12784 !12786 !12802 !12807 !13098 !13101 !13102 !13104 !13106 !13107
!13115 !13117 !13791 !14240 !14241 !14242 !14299
-----
UNSAT generalization: frame = 2, Size = 4, Lit = 12141 !12142 12180 12716

```

Figure 3.10: Blocking procedure for the similar clauses (1). The actions of blocking in these three figures take place in a row. Although in the presented situations, the last literals are the same (12716, 12766, 12743), actually there exists the case that does not follow this regularity. In this figure, the two blocking clauses are generated by one proof obligation at different frames.

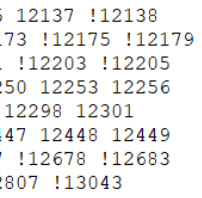
```

Check: frame = 2, BadDepth = 9, Size = 105, Lit = 12126 12128 12129 !12134 12135 12137 !12138
12139 !12140 !12141 12142 !12160 !12161 !12163 !12165 !12167 !12169 !12171 !12173 12175 !12179
12180 !12184 12185 !12186 !12189 12190 12192 !12194 12195 !12197 !12199 !12200 !12201 !12203
!12205 !12207 !12208 12215 !12336 !12446 !12447 !12448 !12449 !12450 !12451 !12452 12453
!12454 !12455 !12530 12537 12545 12549 !12556 !12677 !12678 !12697 12708 12716 12732 !12743
12751 12766 !12773 !12775 !12784 !12786 !12789 !12802 !12807 !12822 !13043 !13098 !13101
!13102 !13106 !13107 !13108 !13109 !13110 !13111 !13112 !13113 !13114 !13115 !13116 !13117
!13791 !14240 !14241 !14242 !14299 !14300 !14301 !14302 !14303 !14304 !14305 !14307 !14308
!14309 !14310 !14311 !14312
-----
UNSAT generalization: frame = 2, Size = 3, Lit = 12142 12180 12766
-----
Check: frame = 3, BadDepth = 9, Size = 105, Lit = 12126 12128 12129 !12134 12135 12137 !12138
12139 !12140 !12141 12142 !12160 !12161 !12163 !12165 !12167 !12169 !12171 !12173 12175 !12179
12180 !12184 12185 !12186 !12189 12190 12192 !12194 12195 !12197 !12199 !12200 !12201 !12203
!12205 !12207 !12208 12215 !12336 !12446 !12447 !12448 !12449 !12450 !12451 !12452 12453
!12454 !12455 !12530 12537 12545 12549 !12556 !12677 !12678 !12697 12708 12716 12732 !12743
12751 12766 !12773 !12775 !12784 !12786 !12789 !12802 !12807 !12822 !13043 !13098 !13101
!13102 !13106 !13107 !13108 !13109 !13110 !13111 !13112 !13113 !13114 !13115 !13116 !13117
!13791 !14240 !14241 !14242 !14299 !14300 !14301 !14302 !14303 !14304 !14305 !14307 !14308
!14309 !14310 !14311 !14312
-----
UNSAT generalization: frame = 3, Size = 4, Lit = 12126 12142 12180 12766
-----
Check: frame = 3, BadDepth = 8, Size = 93, Lit = !12126 12128 12129 !12130 !12131 12132 12134
!12135 12137 !12138 12139 !12140 12141 !12142 !12160 !12161 !12162 !12164 !12166 !12168 !12170
!12172 12174 !12179 12180 !12184 12185 !12186 !12189 12190 12192 !12194 12195 !12197 !12199
!12200 !12201 !12203 !12205 !12207 !12208 12215 12316 12320 12324 12328 12332 !12336 !12446
!12447 !12448 !12449 !12450 !12451 !12452 12453 !12454 !12455 !12530 12537 12545 12548 !12556
!12677 !12678 !12681 !12697 12708 12716 12732 !12743 12751 12766 !12773 !12775 !12784 !12786
!12789 !12802 !12807 !13098 !13101 !13102 !13106 !13107 !13108 !13109 !13110 !13111 !13112 !13113 !13114 !13115 !13116 !13117
!14241 !14242 !14299
-----
UNSAT generalization: frame = 4, Size = 4, Lit = 12141 !12142 12180 12766

```

Figure 3.11: Blocking procedure for the similar clauses (2). In this figure, the above two proof obligations are the same, and the third is the one generating it.





```

Check: frame = 2, BadDepth = 10, Size = 119, Lit = !12126 !12128 !12134 !12135 12137 !12138
12139 !12140 !12141 12142 !12160 !12161 !12163 12165 !12167 !12169 !12171 !12173 !12175 !12179
12180 !12184 12185 !12189 12190 12192 !12194 12195 !12197 !12199 !12200 !12201 !12203 !12205
!12207 !12208 12212 !12215 12222 12226 12230 12234 12237 12240 12243 12246 12250 12253 12256
12259 12262 12265 12268 12271 12274 12277 12280 12283 12286 12289 12292 12295 12298 12301
12304 12307 12310 12313 12317 12321 12325 12329 12333 !12336 !12408 !12446 12447 12448 12449
12450 !12451 !12452 12453 !12454 !12455 !12530 12537 12545 12549 !12556 !12677 !12678 !12683
!12697 !12706 !12708 12730 12743 !12751 !12753 !12770 !12780 !12786 !12802 !12807 !13043
!13098 !13099 !13101 !13102 !13105 !13106 !13107 !13115 !13117 !13791 !14240 !14241 !14242
-----
UNSAT generalization: frame = Inf, Size = 3, Lit = !12126 !12134 12743
-----
Check: frame = 3, BadDepth = 9, Size = 120, Lit = 12126 12128 12129 !12134 12135 12137 !12138
12139 !12140 !12141 12142 !12160 !12161 !12163 12165 !12167 !12169 !12171 !12173 !12175 !12179
12180 !12184 12185 !12189 12190 12192 !12194 12195 !12197 !12199 !12200 !12201 !12203 !12205
!12207 !12208 12212 !12215 12222 12226 12230 12234 12237 12240 12243 12246 12250 12253 12256
12259 12262 12265 12268 12271 12274 12277 12280 12283 12286 12289 12292 12295 12298 12301
12304 12307 12310 12313 12317 12321 12325 12329 12333 !12336 !12408 !12446 12447 12448 12449
12450 !12451 !12452 12453 !12454 !12455 !12530 12537 12545 12549 !12556 !12677 !12678 !12683
!12697 !12706 !12708 12730 12743 !12751 !12753 !12770 !12780 !12786 !12802 !12807 !13043
!13098 !13099 !13101 !13102 !13105 !13106 !13107 !13115 !13117 !13791 !14240 !14241 !14242
-----
UNSAT generalization: frame = Inf, Size = 4, Lit = 12126 12142 12180 12743
-----
Check: frame = 3, BadDepth = 8, Size = 123, Lit = !12126 12128 12129 !12130 !12131 12132 12134
!12135 12137 !12138 12139 !12140 12141 !12142 !12160 !12161 !12162 12164 !12166 !12168 !12170
!12172 !12174 !12179 12180 !12184 12185 !12189 12190 12192 !12194 12195 !12197 !12199 !12200
!12201 !12203 !12205 !12207 !12208 12212 !12215 12222 12226 12230 12234 12237 12240 12243
12246 12250 12253 12256 12259 12262 12265 12268 12271 12274 12277 12280 12283 12286 12289
12292 12295 12298 12301 12304 12307 12310 12313 12317 12321 12325 12329 12333 !12336 !12408
!12446 12447 12448 12449 12450 !12451 !12452 12453 !12454 !12455 !12530 12537 12545 12548
!12556 !12677 !12678 !12681 !12683 !12697 !12706 !12708 12730 12743 !12751 !12753 !12770
!12780 !12786 !12802 !12807 !13098 !13101 !13102 !13104 !13106 !13107 !13115 !13117 !13791
!14240 !14241 !14242 !14299
-----
UNSAT generalization: frame = Inf, Size = 4, Lit = 12141 !12142 12180 12743

```

Figure 3.12: Blocking procedure for the similar clauses (3). In this figure, the upper proof obligation is generated by the lower one. That is, the second generates the first and the third generates the second.

We further make an informal conclusion here. For the case of 6s288r in this subsection, we know that there also exists regularity for the proof obligations. However, since the proof obligation related to this phenomenon does not appear in a row, we cannot get the common part by our checking method. On the other hand, after the blocking clause is added to the frame, we can observe the similarity from the clauses. Hence, sometimes these two kinds of predicate may be complementary for each other, but it still needs more modification to achieve a balance of cooperation.



## Chapter 4

# Implementation

In this chapter, we describe our model checking environment called "Ia2b" briefly. We also introduce what optimizations are applied in the engine to make it more efficient.

### 4.1 The Model Checking Environment

For the research of this thesis, we implement a model checking environment called "Ia2b". It contains compilation environment, command line interface, AIG network structure, simple simplification methods and various model checking algorithms.

The compilation is very trivial that every sub-directory under the source directory serves a module. Every compilation unit in each module is compiled to an object file, then all the object files are gathered to form a static library. The dependency simply follows the hierarchy mentioned above.

A basic command line interface is provided to facilitate the use of model checking engine. It is not as powerful as the mature shells like Bash at all, but is enough for the experiment. In addition to the fundamental string manipulation, the auto completion for command token, option and name of files is available. We support variadic length of tokens for command. Furthermore, for each token, the corresponding utility will be invoked as long as the mandatory part of it is satisfied. That is, we do not need to type full string for the command. Moreover, signal handling is also supported. For example,

the command line will be arranged to the settings at the time the process is paused when restoring, or the position of the characters will be changed to follow the window size.

We only support And Inverter Graph (AIG) for the network structure to follow the benchmarks in HWMCC [26, 27]. That is, there are only five gate types in the network, including constant zero (Const0), primary input (PI), latch (Latch), primary output (PO), and AND gate (And). In addition, some basic logic minimization methods like [28, 29, 30] are included though they are not activated in the experiment.

In addition to PDR, another engine like BMC, IND and IMC are also implemented, but we will not focus on them here. The implementation of our PDR basically follows the description in [18]. We simply introduce some main data structures we use and the optimization is presented in the next section. Figure 4.1 depicts the memory arrangement of cube for PDR. This structure is used by both proof obligation and blocking clause. All the information is collected to store in a chunk of memory that is continuous. Bad depth and the previous proof obligation are only meaningful for proof obligation and are wastes for blocking clause. Signature for subsumption is the method proposed by [31]. There are two Boolean flags for miscellaneous purpose. Reference count is to record how many places hold the address of this chunk of memory. Once the counter decreases to be zero, the occupied memory should be released since no one can access it anymore. Number of literals is kept in order to define the boundary of the literals after it. This array-like memory allocation for a series of literals comes from MiniSAT [6]. However, we do not explicitly employ memory management to allocate all the cubes together. Instead, we just use the built-in allocation utility from C++ to request and return the memory. Besides, the structure for the frames is simply vector of vector of cube.  $F_{\infty}$  is always kept to be the last element. This linear storage is convenient for iteration and is actually not so inefficient for deletion of subsumed clauses.

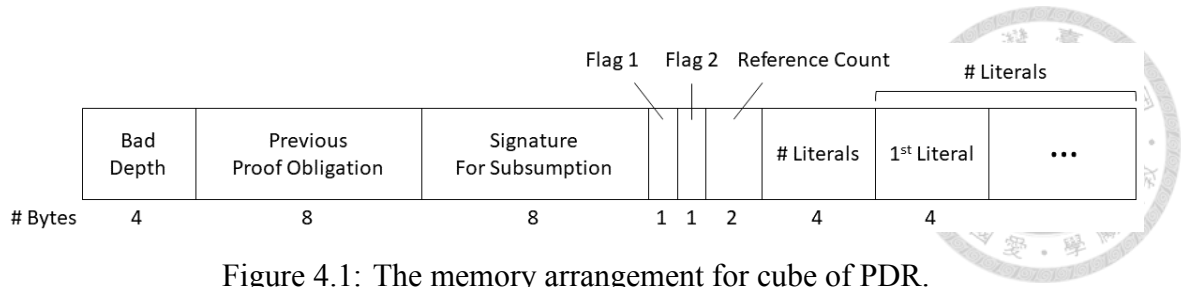


Figure 4.1: The memory arrangement for cube of PDR.

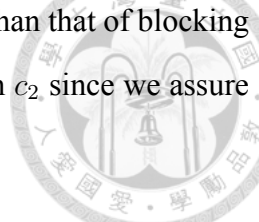
## 4.2 Optimization and Other Things We Tried

In this section, we introduce different implementation techniques for every part of PDR and discuss about their efficiency.

### 4.2.1 Subsumption

Although subsumption checking is a very simple task, but it is performed for a great number of times throughout the process. We need to check subsumption for every blocking clause before either blocking a proof obligation or adding a new blocking clause. That is, as the number of blocking clause grows, the overhead of subsumption increases dramatically. The following are a series of methods to check if  $c_1$  subsumes  $c_2$ .

1. The most trivial method is to iterate over  $c_2$  for every literal in  $c_1$  to check if it is in  $c_2$ . The complexity is  $|c_1| \times |c_2|$ .
2. For the rest methods, we use some rules to do early break on the checking. The use of *signature* is proposed by [31] to record the existence of a literal to a specific bit in a number. Before doing the detailed checking, we first check the sizes and the signatures to filter the apparently failed cases. For the cubes, we maintain that they are sorted. If we find that a literal in  $c_1$  is at a position of  $c_2$ , the literals of  $c_1$  behind that must locate after that position in  $c_2$  if the subsumption holds. This results in a linear checking procedure since we just need to iterate over  $c_2$  for one time without going back again. The complexity is  $|c_1| + |c_2|$  and we adopt this method in our implementation.

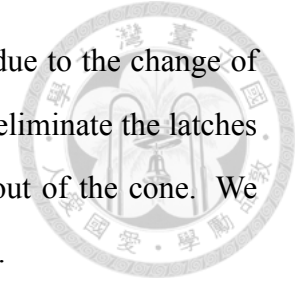
- 
3. Consider that the size of proof obligation is often much larger than that of blocking clause. We can use binary search to check if a literal in  $c_1$  is in  $c_2$  since we assure the cube is sorted. The complexity becomes  $|c_1| \times \log|c_2|$ .
  4. As in [31], the subsumption checking can be generalized to be a self-subsumption one. However, this introduces an overhead that cannot be ignored. This is because there are more literals being processed by self-subsumption checking due to its looser criteria. When the subsumption is sure to fail, they are still possible to be self-subsumption. Since we just need pure subsumption here, so the general process for self-subsumption is discarded.

#### 4.2.2 Ternary Simulation

Ternary simulation plays an important role for efficiency of PDR because of not only the effect to maximize the proof obligation but also the runtime taken by it. Hence, it is important to reduce the overhead of it without sacrificing the quality. In the following, we introduce several kinds of method and the last two have not been verified by implementation yet.

1. The ternary value is encoded by two bits and the transition of an AND gate follows the method in MiniSAT 2.2.0 [32]. We encode the result in a magic number in advance and use bit shift to get the answer based on the two inputs. This is actually equivalent to a  $3 \times 3$  look-up table. For the most straightforward method, we do the normal simulation in the topological order. We collect the cone related to the target first and the latches not in the cone are trivially don't-care. Then we do constant propagation based on the value of primary inputs since we will not modify these values during the process. By doing these two steps, we can reduce the number of gates to simulate massively.
2. Since we only modify the value of one latch at a time, the influence may not transit to a wide range. Event-driven simulation here is a good choice to only simulate the

gate in need. An event is that we need to re-simulate a gate due to the change of any of its inputs. We still need to collect the cone of target to eliminate the latches trivially don't-care and to prevent the event from spreading out of the cone. We adopt this scheme thanks to the balance of runtime and quality.



3. Backward simulation is very fast since its complexity is only linear to the size of the cone. First we assign the value of every gate in the cone by performing simulation once or extracting from the SAT solver. Then we backward mark the set of don't-care based on the following rules from the target.

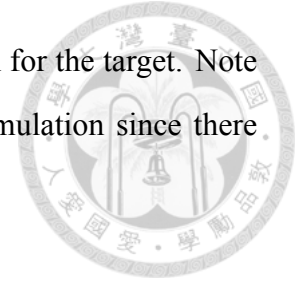
- (a) If the output is 1, both of the inputs must be also 1s and should be kept.
- (b) If the output is 0 and the input combinations is either (0, 1) or (1, 0), we can keep the value 0 and mark the other input with value 1 to be don't-care.
- (c) If the output is 0 and both of the inputs are also 0s, we can pick one of the inputs to keep the value and mark the other to be don't-care.

Note that the mark of don't-care is dominated if the value is mandatory for another part of circuit. After one traversal, we can directly remove the latches that are really don't-care.

4. The cube in the original PDR can only contain state variables. This scheme sometimes increases the number of blocking clause like representing XOR gate. An idea is to use the internal signal of the circuit in the cube to represent the state. This does not mean to find a cut for representation, but to use different set of gates in different cubes. Note that the used gate cannot be related to any primary input, that is, the cone of it can only contain latches. If it does contain any primary input, the semantics for state representation is broken.

5. The last method uses SAT instead of ternary simulation. First we make an assumption on the negation of the target. We then make assumptions on the assignment of primary inputs and latches. The query must be UNSAT so that we

can pick the literals in the final conflict clause that are enough for the target. Note that this method cannot be applied before normal ternary simulation since there may exist don't-care at the target after this checking.



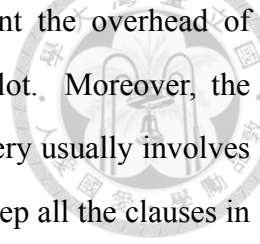
### 4.2.3 SAT Query

SAT query takes most of the runtime in PDR. Although one query is simple compared to that of the other model checking algorithm, there are plenty of SAT queries in one run of PDR. We focus on how to reduce the runtime of SAT in each query instead of the number of queries since the latter one is too hard to predict.

SAT solver itself is used as a black box in our implementation since we choose to use powerful open-source libraries instead of customization. We try a series of similar solvers including MiniSAT 1.14 [32], MiniSAT 2.2.0 [32] and Glucose 3.0 [33], where the latter is an improved version of the former. It is not surprising that Glucose performs the best thanks to its ability to solve SAT so that we adopt it in our implementation.

How to convert the constraints of circuit to CNF that is recognized by SAT solver is another problem that has a great effect on the performance. The basic procedure follows [34] to introduce a new variable for each internal signal. It is simple and fast to convert but too fundamental to be efficient for SAT solving. On the other hand, there are various techniques aiming at reducing the size of CNF [35, 36, 37, 38]. Although the size of CNF is not directly related to how difficult an SAT instance is, the improvement from these techniques are supported by the experiment in their papers. For simplicity, we just adopt Tseitin transformation in our implementation.

Another important part is to reduce the overhead stemming from the old information accumulated during the process. We heavily rely on the incremental feature of SAT solver to reuse one solver for multiple queries. If we do not make the clause database more clear and simple intentionally, there are more and more redundant constraints kept in the solver. As indicated by [39], ABC [40] uses a smart conversion scheme to only add the part of circuit related to the current solving. Since the SAT query often involves

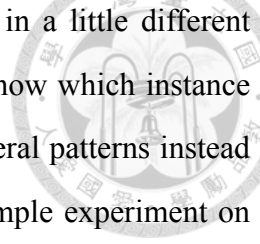


a small portion of latches in each round, this scheme can prevent the overhead of Boolean constraint propagation (BCP) from the unused clauses a lot. Moreover, the concept is also applicable to the clauses in each frame. The SAT query usually involves only the frames close to the last one, so there is no need to always keep all the clauses in the solver. Actually the smart conversion scheme works by the help of solver recycling. We recycle the solver for a number of queries to reset the data and state in it. This action is originally designed to reduce the overhead from the unused activation variables [18] since we frequently add new variable to enable a temporary clause. Hence, we initially set the number of unused variables to be the threshold to recycle. However, with the above conversion scheme, it turns out that the improvement mainly comes from the concise clause database. Therefore, we modify that using the exact number of SAT queries to be the threshold.

As suggested by [39], we can apply approximate SAT solving for PDR. From the experiment, the cost of satisfiable case is often more expensive than the unsatisfiable one [18]. In addition, for most of the time, only the unsatisfiable answer is meaningful for us to drive the proof. Hence, we can set a resource limit on one SAT query before solving and just see the aborted case as a satisfiable one. In [39], the authors use a static number 100 to limit the number of decisions and achieve an improvement. Our method is to record the maximum number of decisions for all the unsatisfiable calls so far and use  $1.5\times$  of this value as the bound. We also observe an improvement in the experiment but does not include this technique in the presented result. Note that the approximate solving cannot be applied at the call that we really need the assignment from the satisfiable call like *checkReach*.

Similar to the case of PDR, it also takes many SAT queries to construct a Functionally Reduced AIG (FRAIG) [28]. A special feature for FRAIG is to use the assignment from previous satisfiable call for simulation to distinguish if the instance is satisfiable before actually solving. In detail, right when getting an assignment, we are able to simulate based on it since we have already known all the possible SAT calls in the





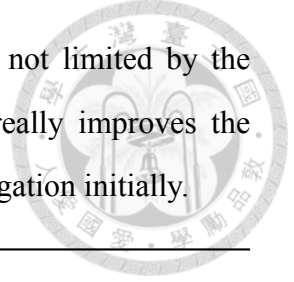
future. On the other hand, this technique can be applied to PDR in a little different manner. We should record all the patterns so far since we do not know which instance will be under solving. As a tradeoff, we can just record the last several patterns instead without losing too many opportunities of recognition. From our simple experiment on 6s288r, the coverage relative to the case that all the patterns are stored can be over 80% if we use 5 numbers with 64 bits to record. In the current implementation, we have not adopted this technique yet.

#### 4.2.4 Storage of Proof Obligation

Originally in [18], the authors use a priority queue in *recBlockCubes* to get the proof obligation with minimum frame. In addition, keeping the obligations in one frame act like a stack is thought to be beneficial by them. In Algorithm 3, we suggest to maintain an infinite array of set to store all the proof obligation by their corresponding frame [41]. By this structure, we can easily know which obligation has the minimum frame and keep track with the arrangement in each frame. Besides the stack-like behavior, we also try the queue-like one by implementing the frame as deque structure. Furthermore, we have tried to tackle with the obligation with maximum bad depth first. In our implementation, we adopt the classic stack-like scheme.

On the other hand, as indicated in Subsection 2.4.3, the blocked proof obligation is reused to be blocked at further frame. In the original algorithm, the reuse is not thorough so that we present the procedure again in Algorithm 9 to discuss about the problem. First, the return value of *isBlocked* is no longer a simple Boolean. It is now an integer to represent the highest frame that can block it. 0 is used as a special number to indicate the failure of blocking since we should not block any obligation at the first frame. Instead of simply discarding the blocked obligation, we preserve and push it to the frame that it has not been blocked to prevent redundant checking. In addition, we use the frame returned by *generalize* to know the destination the blocking clause is added. We then push the obligation to that frame instead of just to the next one. We only adopt this basic

modification in the experiment. Furthermore, the pushing is also not limited by the currently maximum frame anymore. Combing these techniques really improves the efficiency, just like the tendency introduced by reusing the proof obligation initially.




---

**Algorithm 9:** *recBlockCubes<sub>reuse</sub>*

---

**Input:** Cube *badCube*, the frame to block *f*  
**Output:** TRUE or FALSE  
**Data:** An infinite array of set of cubes *badArr*

```

1 badArr[f].add(badCube)
2 blockFrame ← f
3 while blockFrame ≤ f
4   if blockFrame = 0
5     return FALSE
6   targetCube ← badArr[blockFrame].pop()
7   Fblock ← isBlocked(targetCube, blockFrame)
8   if Fblock = 0
9     preCube ← checkReach(targetCube, blockFrame)
10    if preCube ≠ NULL
11      badArr[blockFrame].add(targetCube)
12      blockFrame ← blockFrame − 1
13      badArr[blockFrame].add(preCube)
14    else
15      newCube, Fblock ← generalize(targetCube, blockFrame)
16      addBlockedCube(newCube, Fblock)
17      if Fblock ≠ ∞
18        badArr[Fblock + 1].add(targetCube)
19      if badArr[blockFrame].empty()
20        blockFrame ← blockFrame + 1
21    else
22      if Fblock ≠ ∞
23        badArr[Fblock + 1].add(targetCube)
24 return TRUE

```

---

Because of pushing the proof obligation to further frame, we can find a counterexample longer than the currently maximum frame. Furthermore, the found counterexample is no longer guaranteed to be the shortest one. In order to find the counterexample, we store the previous one in every proof obligation. When reaching the first frame, we can iterate through this record to the first proof obligation that violates the property. Then we can get the input combinations by examining this sequence.



### 4.2.5 Propagating Cubes

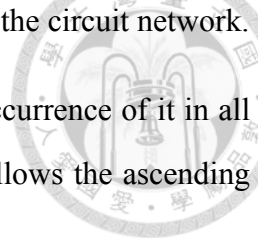
The runtime taken by propagating cubes is usually not dominating, but we can still carry out some idea to enhance it.

[24] propagates the blocked cubes out of the boundary of currently maximum frame. This scheme gives more opportunities for us to recognize an inductive subset of clauses when observing no difference between any two adjacent frames. Though it may be not invariant, we can always add them to  $F_\infty$ . However, no obvious improvement can be noticed after activating this scheme. We guess that the reason is that these subset of clauses can be propagated to any further frame since the set is inductive itself. Hence, the reachability does not change at all if we just explicitly add them to  $F_\infty$ .

As stated in Subsection 4.2.3, we should not let the past information be the burden of the current process. A lazy propagation scheme is introduced here to propagate the cube in the frame that seems to be changed during this iteration. Since we are not capable of detecting the change exactly, we observe whether any proof obligation is under blocking in this frame instead. For many cases, the first several frames will not be involved in the blocking anymore. Hence, trying to propagate the clauses in those frames all the time may not be a good choice so that we try to skip it. We can achieve a small improvement from that since it is seldom the bottleneck. On the other hand, when successfully propagating a cube to further frame without enlarging it, we can skip the subsumption checking for those frame before its original one since it is performed at its previous adding.

### 4.2.6 Activity of State Variables

We can consider the order of latch variable when performing ternary simulation and generalization. The goal of removing literals is to reach a local optimal that there is no literal removable in the cube. However, this may not be a global optimum with minimum number of literal or the most suitable form for the PDR process. The methods we have tried are as the following.

- 
1. Use a static order that just follow the index of latch variable in the circuit network.
  2. Allocate counters for each variable to record the number of occurrence of it in all the blocking cubes. We dynamically change the order that follows the ascending order of the counters or reversely.
  3. In order to focus on the current information, the counter can be decayed periodically just like VSIDS [4].

We do not observe obvious difference for the above methods so that the basic static order is picked for the experiment.

#### 4.2.7 Predicate Extraction

In this subsection, we discuss about the things we tried for predicate extraction in addition to the final version introduced in Chapter 3.

A main feature of our method is the flexibility before and after solving the predicate. As long as we maintain the 5 properties, there are various operations can be tried. Start with the case before the predicate solving.

1. Reuse nothing. As the version we present, we start a new PDR procedure to solve the predicate.
2. Reuse  $F_\infty$ . Since  $F_\infty$  is itself an inductive set, we can add it to a new PDR procedure to constraint the search space. The result is similar to that of reusing nothing.
3. Reuse all the clauses. In this scheme, we set the predicate to be the property of the original process temporarily instead of creating a new one. Then we directly solve the predicate on the original process with all the clauses in it. However, the result is poor compared to the above two. A possible reason is that the convergence related to the predicate is interrupted by the original clauses. In a separate PDR, the inductive invariant set may be easily found. But in the same PDR, the solving for predicate is difficult or even pollutes the original learned information.

Then we explain how to merge the obtained information back to the original process. The merging only applies for the first two reusing schemes since the last one just works on the same PDR.

1. Merge the inductive set. As the version we present, we only merge the obtained inductive set back to  $F_\infty$ . In addition to check if any clause is subsumed by the added ones, we need to check if any added clause is subsumed by the original one in  $F_\infty$ .
2. Merge all the clauses. This has not been verified by implementation. The concept is that we can merge all the clauses in the separate PDR back to its corresponding frame. A simple proof for Property 4 is as follows.

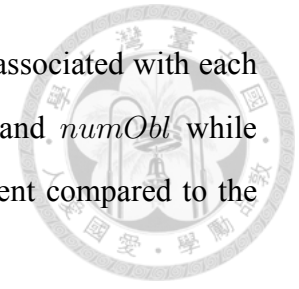
$$\begin{aligned}
 F_{i-1}(S) \wedge Tr(S, S') &= F_{i-1,1}(S) \wedge F_{i-1,2}(S) \wedge Tr(S, S') \\
 &= F_{i-1,1}(S) \wedge Tr(S, S') \wedge F_{i-1,2}(S) \wedge Tr(S, S') \\
 &\rightarrow F_{i,1}(S') \wedge F_{i,2}(S') = F_i(S')
 \end{aligned}$$

Since we only record the difference between adjacent frames, the clauses in a frame does not actually represent the set of states of it. The similar clauses may not be detected because we observe one frame at a time. Another method is to allocate a fixed length of space to store the last several added clauses and find the candidate from them. In this scheme, we do not simply consider the frame that the clause is blocked at. The result has no improvement and the possible reason is that the accuracy decreases after we mix all the clauses together without classification.

Another scheme is to find the candidate after the last frame becomes invariant, that is, before propagating the cubes. This aims at tackling with the predicates at a time without interrupting the blocking phase. The result is not good so that dealing with the obstacle in time seems to be more reasonable.

For predicate of obligation, we present a version that gather all the obligations to find the common part. This version does not classify the obligations by their own

characteristics. We have tried to distinguish them by the bad depth associated with each obligation. The set for each depth maintains its own *common* and *numObl* while returning separate predicate. The experiment shows the improvement compared to the original PDR, but does not outperform the version we propose.





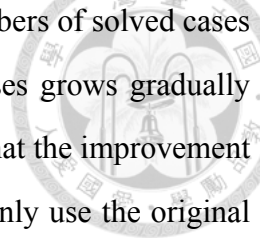
## Chapter 5

# Experimental Results

In this chapter, we provide a series of experimental results. The experiment is conducted on a machine with Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz and 125G memory. The operating system is Ubuntu 16.04.6 LTS and the compiler is g++ 7.5.0. The set of benchmarks is collected from HWMCC 12, 13, 14, 15 and 17. The case with property not related to any latch is removed and there are total 1020 cases left. The runtime limit is set to 3600 seconds in the following data.

### 5.1 Overview

We first compare our basic PDR with the implementation in two well-known model checkers including V3 [41] and ABC [42]. Both V3 and ABC are activated by their default options without any other preprocess command. That is, only ABC solves with the strashed circuit in that it naturally maintains the strashed form for every network. Either Ia2b or V3 does not modify the circuit structure before solving the property. In figure 5.1, Our engine outperforms V3 a lot and is close to ABC. But we also see that there is an obvious gap between the average runtime of Ia2b and ABC. Since the total solved number of cases are similar, we know that ABC still outperforms us for many cases. The main difference between Ia2b and ABC is the runtime of SAT query. The remarkable runtime of SAT query in ABC probably results from its sophisticated CNF



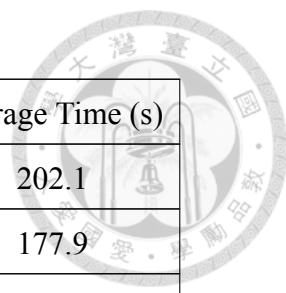
conversion technique [38]. But at the time close to the limit, the numbers of solved cases are almost the same. This is probably because the impact of clauses grows gradually along the process due to the accumulated quantity in each frame so that the improvement from CNF of circuit decreases. For the following experiment, we only use the original PDR in Ia2b (called "baseline") for comparison to reveal the effect of our method.

In this round, actually we let the three model checkers run 7200 seconds totally to test the limit of original PDR. (We still present the version with 3600 seconds time limit here.) We further collect all the cases that any of the three engines can solve within 7200 seconds for a preliminary test of our method. In the following sections, we call this set as "pdr\_vb" and the original one is referred as "full".

For the rest experiment, there are two main results we want to show. First of all, we need to assure that our method at least outperforms the basic PDR engine. In order to do this, we place the curve of basic PDR in every figure to show the difference. For those parameters that gets the efficiency worse, we provide explanation. Secondly, we want to show the trend for a "sweet spot" for predicate solving. After we find a predicate, we need to explicitly solve it to get useful information. If we put too much effort on them, too much overhead is added to the main procedure compared to the benefit. The effort here means the runtime to solve predicate rather than finding it. Otherwise, if we explore too less about the predicates, there is no much difference compared to the original method. Hence, we predict a balance between the solving effort and the achieved feedback. Because there are too many possible combinations of parameters, we do not aim at finding a best one among them. Actually the best value of parameter is probably case dependent. Instead, we just use a series of comparison to show the trend.

There is not an obvious dependency among all the parameters in our method. Hence, we decide to just vary one parameter at a time while fixing all the other ones and to compare which value is more suitable. Or we say it is closer to the sweet spot under this combination of parameters.





Checker	Total	SAT	UNSAT	Unique Solve	Average Time (s)
v3	487	141	346	3	202.1
abc	544	148	396	25	177.9
ia2b_baseline	543	149	394	20	195.1

Table 5.1: Comparison of PDR among different model checkers. Unique solve means only this variant can solve among all the ones in the same table. Average time involves only solved cases, so the comparison on this column should be directed for the variants with the same or at least similar number of solved cases.

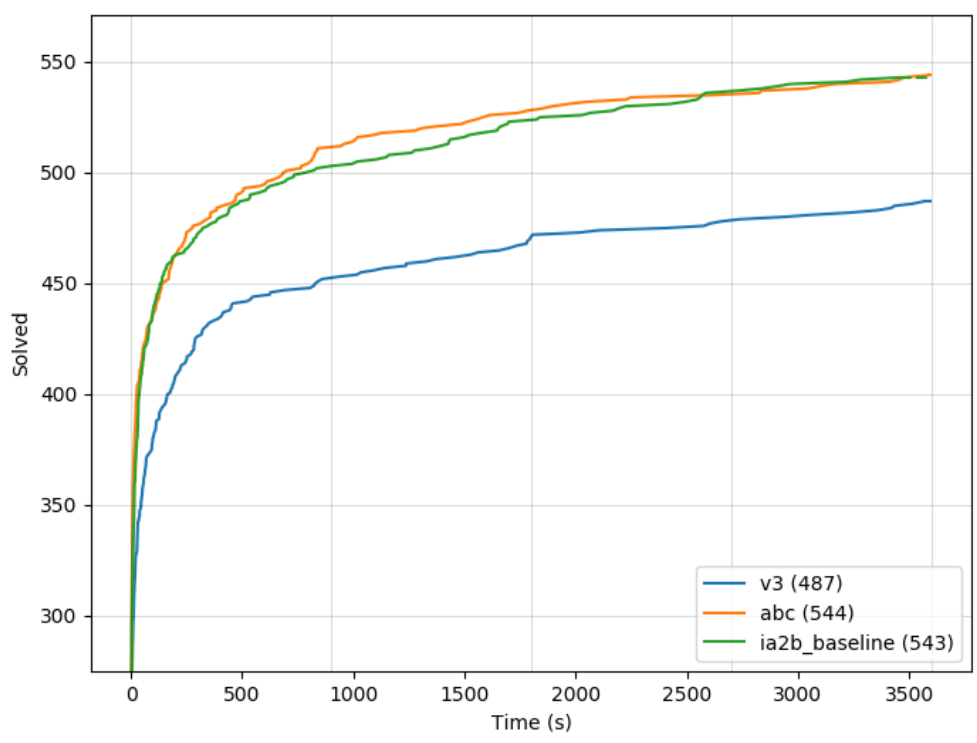
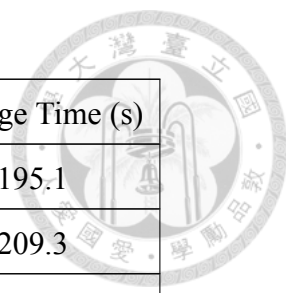


Figure 5.1: The cumulative plot for PDR among different model checkers. The X-axis means the runtime in second while the Y-axis means the number of cases. The labels in the lower-right corner indicate the representative of each curve with different colors. The number enclosed by the parentheses means the total number of solved cases, which is also indicated by the horizontal dashed line at the tail of curve.

## 5.2 Performance

In this section, we present the result for clause-based (CLS), obligation-based (OBL) and combined predicate in order. The first two kinds are conducted on "pdr\_vb" for a preliminary examination, while the last one is on "full" to demonstrate the final result. Note that either in the table or in the figure, all the variants are sorted by the solving effort on predicate in the ascending order. The first one is the basic PDR that is the special case with no effort spent. For both CLS and OBL, we use a small SAT query limit ( $L = 300$ ) for the experiment to find a relatively proper value combination based on this  $L$ . Then we use this value combination back to test different  $L$ . Last, the best  $L$  found in the previous round is used to test the trend discovered at the first.

For CLS, each frame can be assigned by different Backtrack number ( $B$ ) and Match Number ( $M$ ). For simplicity, we introduce only two schemes including INF and ALL. INF is to observe only  $F_\infty$ , that is, the Backtrack number of the other frames are all zeros. ALL is to treat all the frames equally with the same  $B$  and  $M$ . The value we choose to test different  $L$  is  $(B, M) = (20, 1)$  and  $(10, 2)$  for INF and ALL respectively. The result is shown by Figure 5.2 and 5.3. If fixing the other parameters, we assume the number of predicates are the same. Hence, the effort is proportional to the magnitude of  $L$ . The number of solved cases of all are larger than the one of baseline except for the case with no limit ( $L = \infty$ ). This case is intentionally added to show the extreme case of predicate solving. Apparently, it loses the direction to reach the final answer. Furthermore, the trend is obvious that the best one locates in the middle and the result degrades to the both sides.



SAT Limit	Total	SAT	UNSAT	Unique Solve	Average Time (s)
baseline	543	149	394	0	195.1
300	553	150	403	3	209.3
1000	554	149	405	3	213.3
5000	550	147	403	0	235.6
no limit	495	130	365	1	219.8

Table 5.2: Comparison among different limit of SAT query for CLS, INF, B = 20, M = 1 on pdr\_vb.

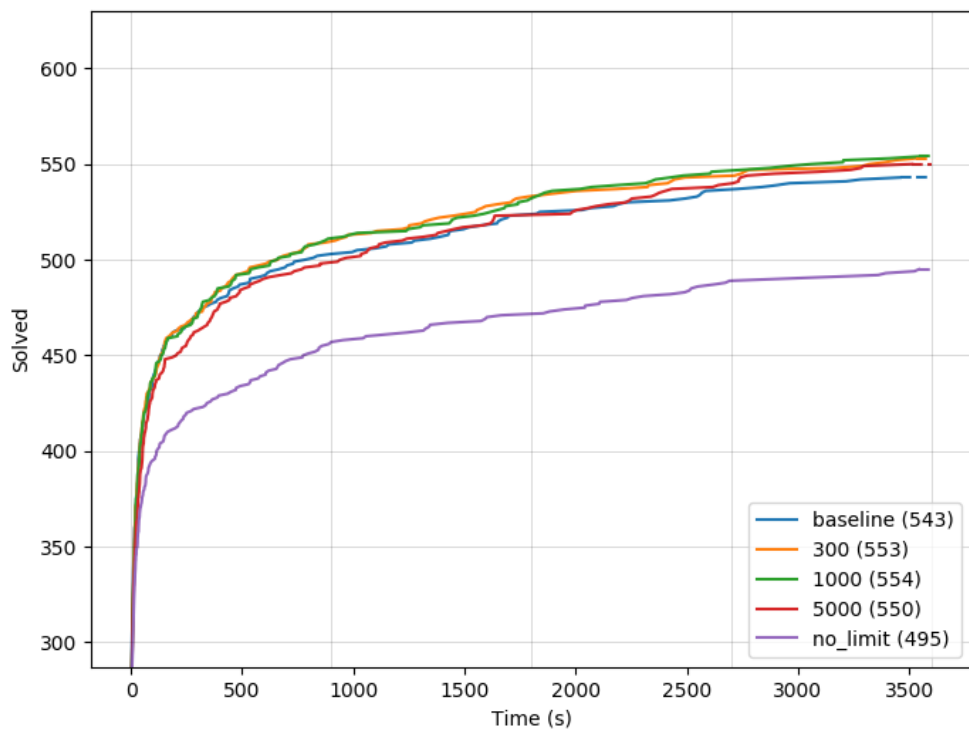
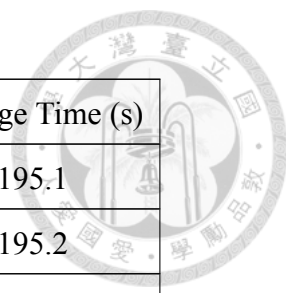


Figure 5.2: The cumulative plot of different limit of SAT query for CLS, INF, B = 20, M = 1 on pdr\_vb.



SAT Limit	Total	SAT	UNSAT	Unique Solve	Average Time (s)
baseline	543	149	394	2	195.1
300	554	149	405	1	195.2
1000	558	151	407	4	211.5
5000	551	149	402	1	240.4
no limit	464	124	340	1	191.7

Table 5.3: Comparison among different limit of SAT query for CLS, ALL, B = 10, M = 2 on pdr\_vb.

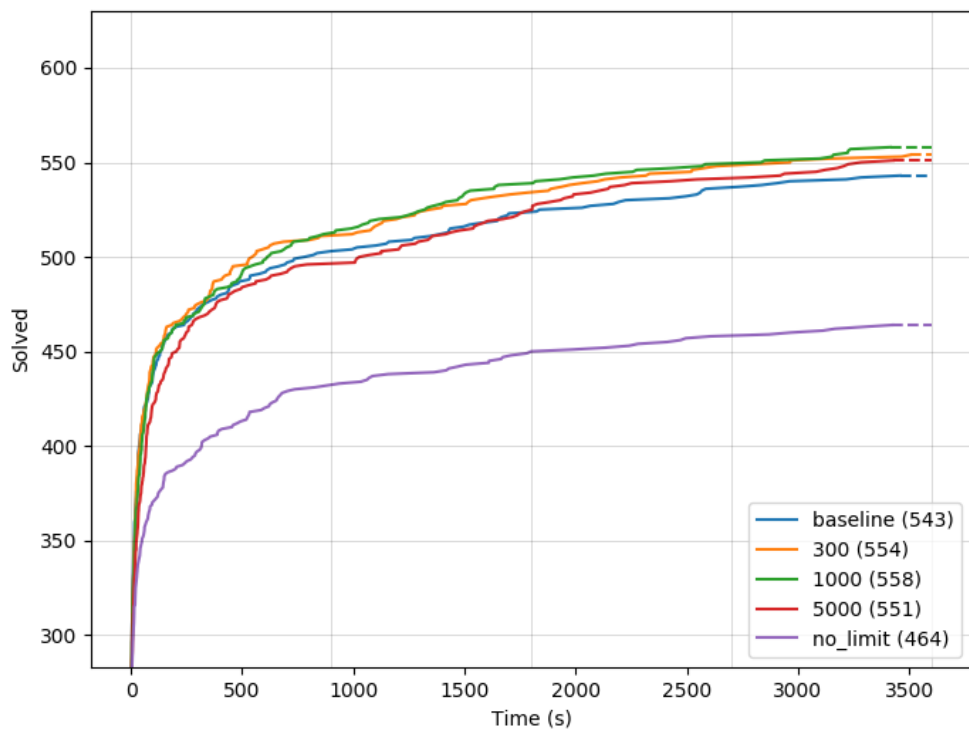
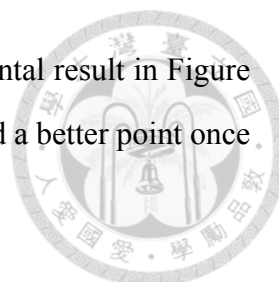


Figure 5.3: The cumulative plot of different limit of SAT query for CLS, ALL, B = 10, M = 2 on pdr\_vb.



We then present the concept of sweet spot again by the experimental result in Figure 5.4. In addition, since the sample points are loose, we can further find a better point once the trend is confirmed. The concept is demonstrated in Figure 5.5.

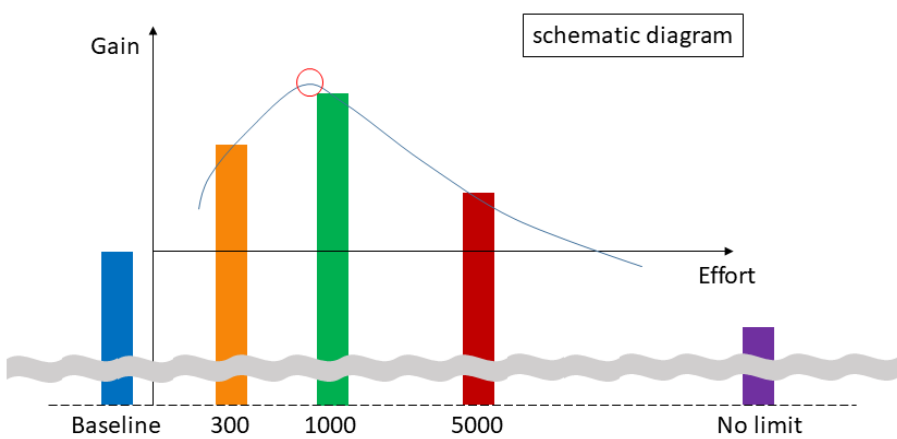


Figure 5.4: The schematic diagram of the sweet spot prediction and illustration by the result of CLS, ALL,  $B = 10$ ,  $M = 2$  on  $pdr\_vb$ . X-axis means the effort and Y-axis means the gain. Note that the gain can be negative, which means the overall performance is worse. The bars mean the number of solved cases in Figure 5.3 and they are not in exact scale. In this figure, we use difference among these numbers to represent the gain.

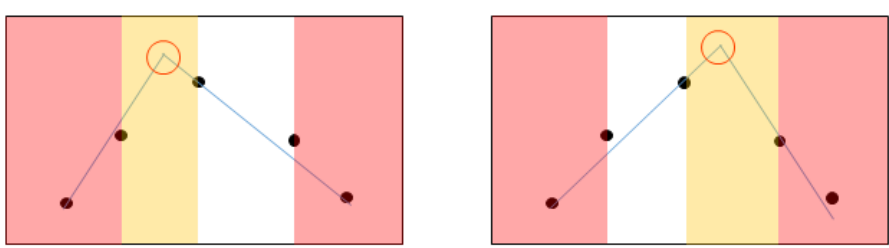
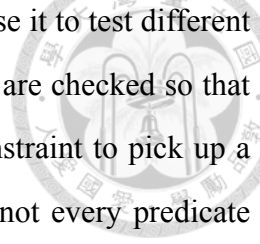


Figure 5.5: The depiction of sample points. X-axis means the effort and Y-axis means the gain. If we obtain the five points by experiment and basically confirm that the trend exists, we can exclude the outer range (red area) since the best point is not possible to be at there. Then we can apply denser sample points in the yellow area to find a better one.



Since  $L = 1000$  is the best choice of the above two figures, we use it to test different  $B$  and  $M$  for both INF and ALL. The higher  $B$  is, the more clauses are checked so that the number of predicates increases. The lower  $M$  is, the looser constraint to pick up a common part as candidate, which leads to less predicate. Though not every predicate uses up all the SAT query limit, we assume they consumes the same effort. Hence, the effort is roughly proportional to the ratio  $B/M$ . The result is shown in Figure 5.6 and 5.7. The best choices are 20\_1 and 10\_2 for INF and ALL respectively. There are two possible reasons. The first one is that the number of frame under observation differs. This reason corresponds to the hypothesis of sweet spot we made above. In INF, we only consider  $F_\infty$  so that the constraint should be looser to create more chance. However, in ALL, all the frames are under consideration so that we need to apply a tighter criterion. The second is to state the different usefulness among the frames. All the clauses in  $F_\infty$  forms an inductive set so that they are more meaningful and should be applied by a looser constraint. On the other hand, we should not emphasize too much on the other frames since they seem to be no so useful. An interesting phenomenon is that the performance of 10\_1 and 10\_2 for ALL are poor. Nonetheless, this does not happen for the case with  $L = 300$ . This is probably an evidence that we put too much effort on predicate solving.

If the reason is actually the second one, there should be possible to combine the benefit from the two different sources. Hence, we introduce another scheme MIX that uses  $(B, M) = (20, 1)$  and  $(10, 2)$  for  $F_\infty$  and the other frames respectively and simultaneously. In Figure 5.8, MIX does not perform well and it is even the worst one. However, as the case with  $L = 300$  depicted by Figure 5.9, MIX is the best one instead. We then make a conclusion that the reason is closer to the first one. MIX has a higher possibility to find predicate than both INF and ALL if we simply consider the ratio  $B/M$  for every frame. For the case with  $L = 1000$ , INF\_20\_1 and ALL\_10\_2 are relatively better choices, so changing to MIX does not improve. But for  $L = 300$ , INF\_20\_1 and ALL\_10\_2 have not reached the sweet spot, so changing to MIX is better.

B_M	Total	SAT	UNSAT	Unique Solve	Average Time (s)
baseline	543	149	394	3	195.1
10_2	546	148	398	3	199.2
10_1	548	147	401	3	196.4
20_1	554	149	405	3	213.3

Table 5.4: Comparison among different ratio of Backtrack number and Match number for CLS, INF, L = 1000 on pdr\_vb.

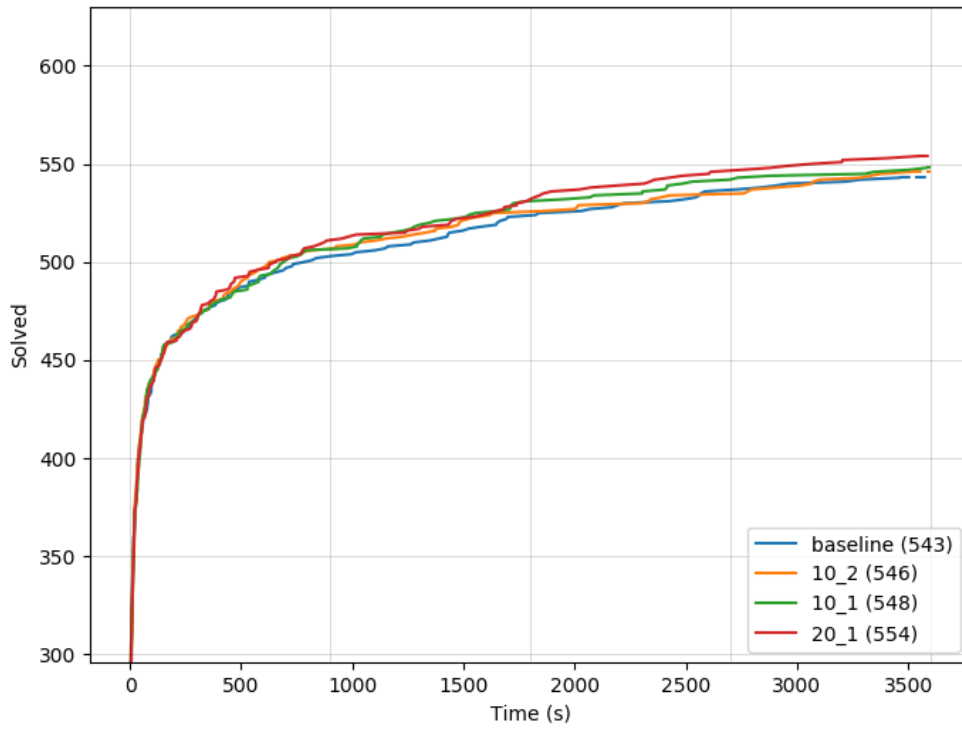


Figure 5.6: The cumulative plot of different ratio of Backtrack number and Match number for CLS, INF, L = 1000 on pdr\_vb.

B_M	Total	SAT	UNSAT	Unique Solve	Average Time (s)
baseline	543	149	394	5	195.1
10_2	558	151	407	10	211.5
10_1	535	145	390	2	228.5
20_1	532	143	389	1	224.0

Table 5.5: Comparison among different ratio of Backtrack number and Match number for CLS, ALL, L = 1000 on pdr\_vb.

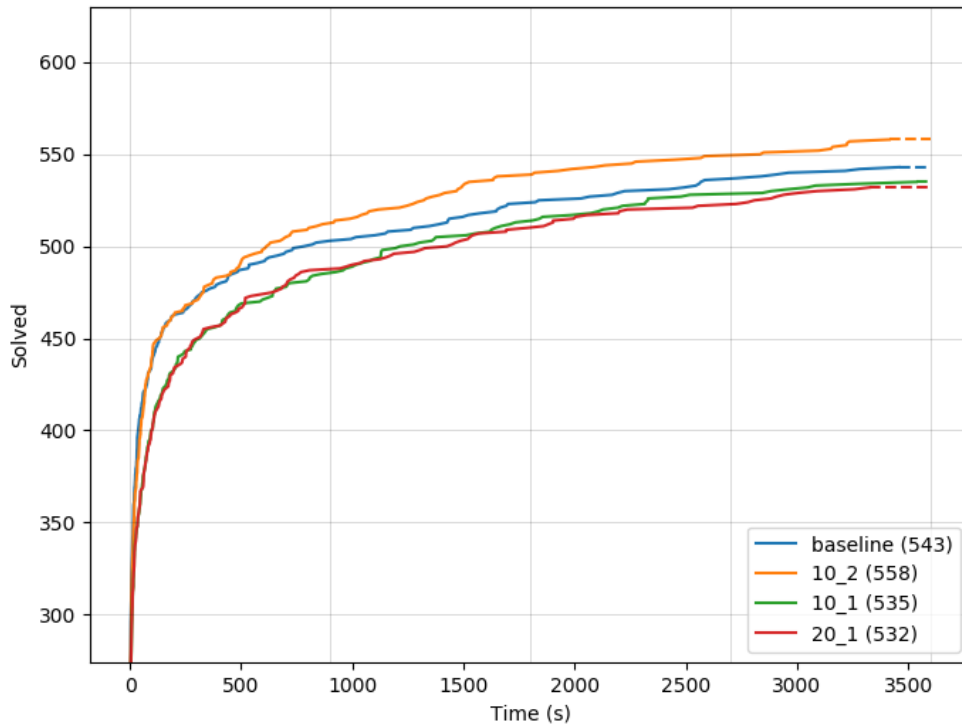
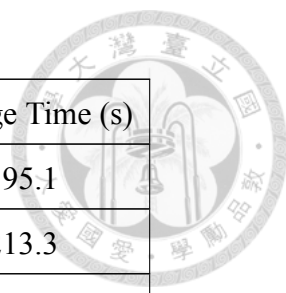


Figure 5.7: The cumulative plot of different ratio of Backtrack number and Match number for CLS, ALL, L = 1000 on pdr\_vb.





Type	Total	SAT	UNSAT	Unique Solve	Average Time (s)
baseline	543	149	394	3	195.1
inf_20_1	554	149	405	2	213.3
all_10_2	558	151	407	3	211.5
mix	553	148	405	0	211.0

Table 5.6: Comparison among two best ratios and the mixed version for CLS,  $L = 1000$  on pdr\_vb.

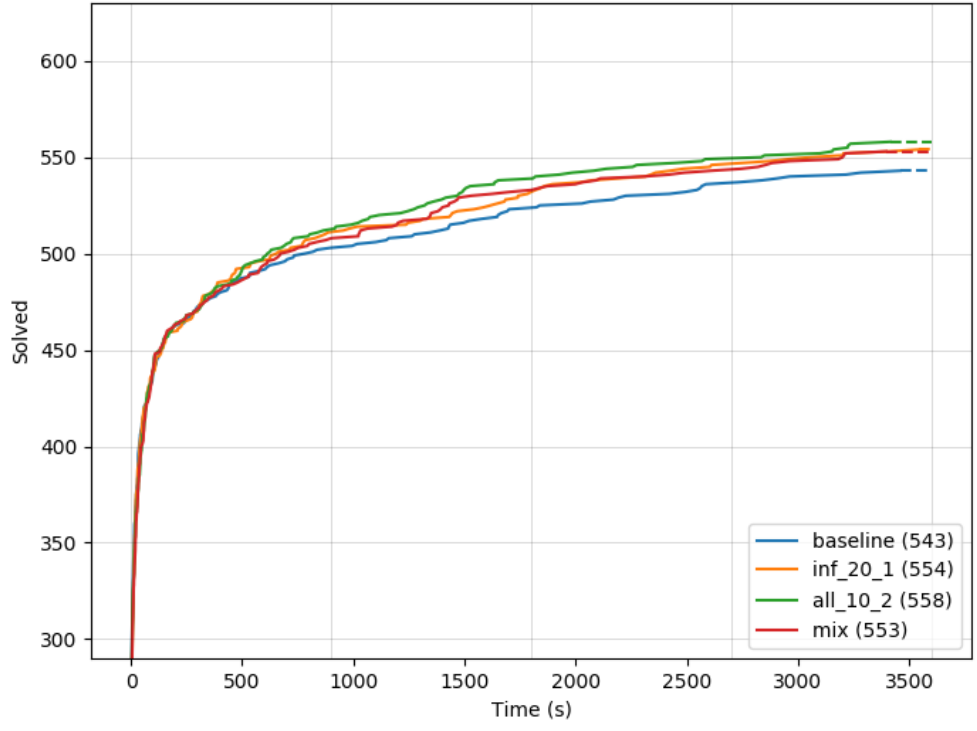
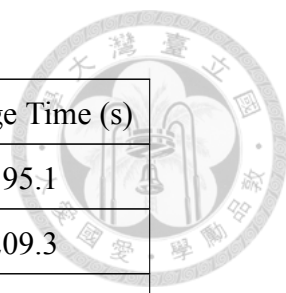


Figure 5.8: The cumulative plot of two best ratios and the mixed version for CLS,  $L = 1000$  on pdr\_vb.



Type	Total	SAT	UNSAT	Unique Solve	Average Time (s)
baseline	543	149	394	1	195.1
inf_20_1	553	150	403	0	209.3
all_10_2	554	149	405	1	195.2
mix	558	152	406	2	203.9

Table 5.7: Comparison among two best ratios and the mixed version for CLS,  $L = 300$  on pdr\_vb.

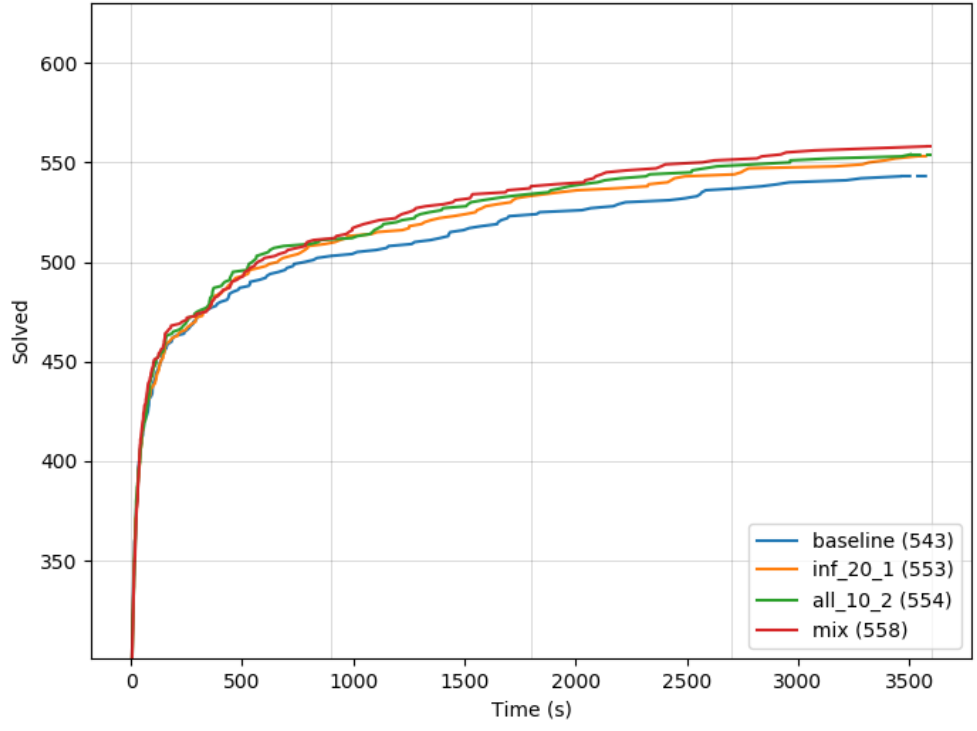
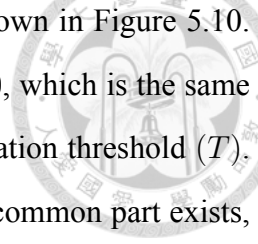


Figure 5.9: The cumulative plot of two best ratios and the mixed version for CLS,  $L = 300$  on pdr\_vb.



For OBL, we use  $T = 66$  to test different  $L$  and the result is shown in Figure 5.10. Again, the trend is obvious so that we choose the case with  $L = 300$ , which is the same as the preliminary test. We then use  $L = 300$  to test different Obligation threshold ( $T$ ). The higher  $T$  is, the less chance to check and the less possibility of common part exists, which leads to less predicate. Hence, the effort is inversely proportional to  $T$ . We observe the trend in Figure 5.11 once again and the best value is  $T = 66$ . It seems like that the hypothesis is also applicable to OBL.

Finally, we test all the cases in "full" for both CLS and OBL to examine the ability of them to handle the hard problem for the original PDR. The values of parameter are  $(B, M) = (10, 2)$  for CLS, ALL and  $T = 66$  for OBL. In addition, we also try to activate the two kinds of predicate simultaneously. The result is shown in Figure 5.12. As indicated by the result of MIX, we know that we cannot ensure the improvement by simply solving more and more predicates. This is because we do not gain the information right when we find the predicate and we need to solve it practically. On the other hand, we know that both CLS and OBL works well. They outperform the basic PDR by 29 and 18 cases respectively.

SAT Limit	Total	SAT	UNSAT	Unique Solve	Average Time (s)
baseline	543	149	394	5	195.1
300	551	148	403	4	193.8
1000	543	144	399	3	183.3
5000	540	144	396	2	188.0

Table 5.8: Comparison among different limit of SAT query for OBL, T = 66 on pdr\_vb.

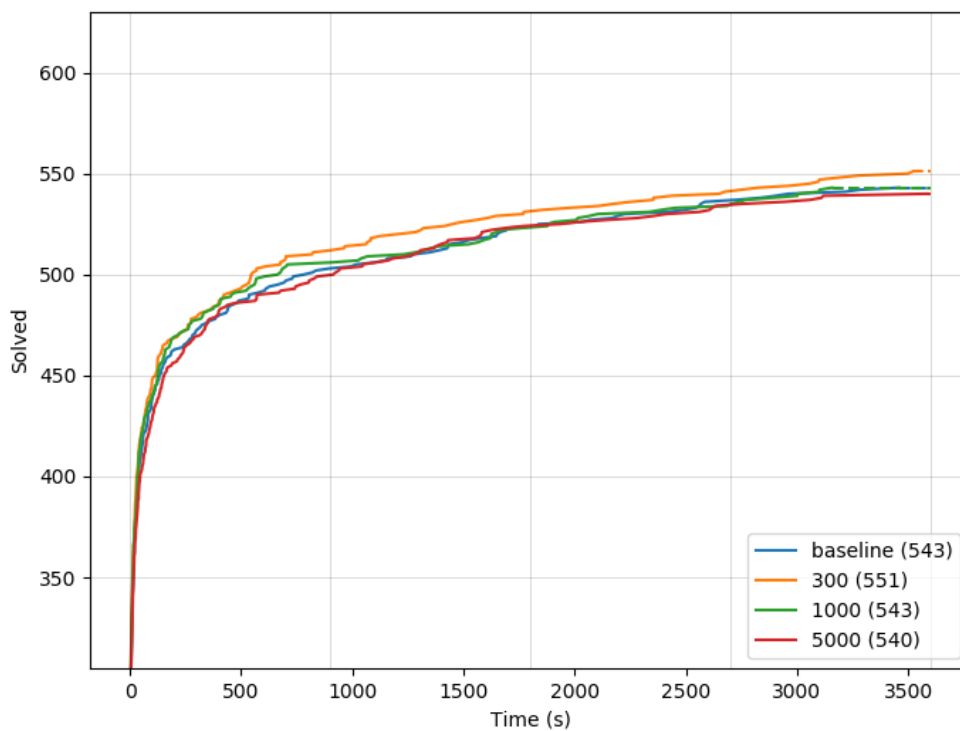


Figure 5.10: The cumulative plot of different limit of SAT query for OBL, T = 66 on pdr\_vb.

Threshold	Total	SAT	UNSAT	Unique Solve	Average Time (s)
baseline	543	149	394	3	195.1
100	543	147	396	1	191.4
66	551	148	403	5	193.8
36	540	145	395	1	170.1

Table 5.9: Comparison among different Obligation threshold for OBL, L = 300 on pdr\_vb.

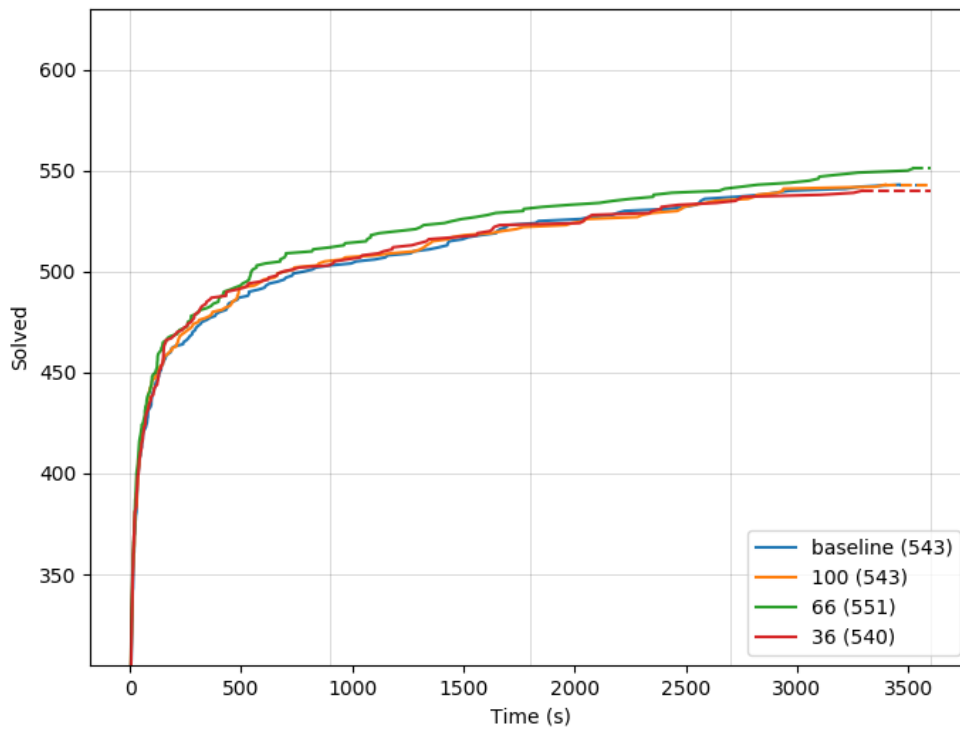


Figure 5.11: The cumulative plot of different Obligation threshold for OBL, L = 300 on pdr\_vb.

Type	Total	SAT	UNSAT	Unique Solve	Average Time (s)
baseline	543	149	394	2	195.1
cls	572	151	421	11	244.8
obl	561	150	411	2	232.7
cls + obl	567	151	416	9	234.5

Table 5.10: Comparison among the best parameters for CLS, OBL and the combination of both on full.

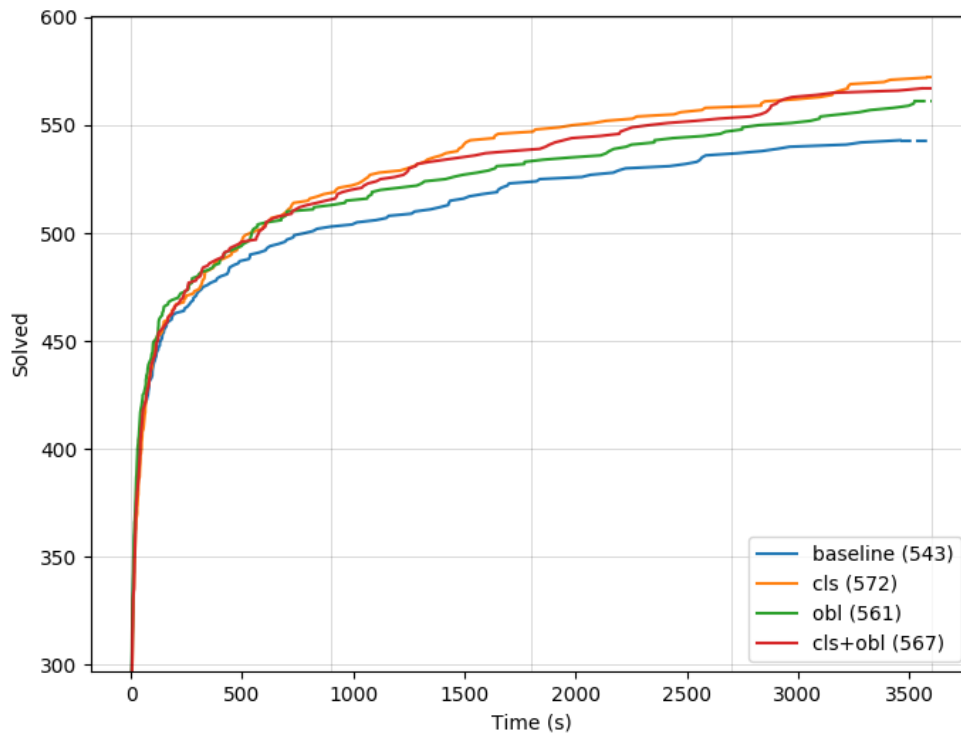


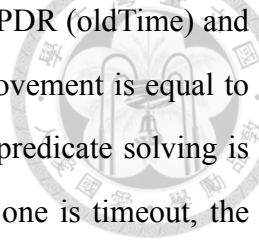
Figure 5.12: The cumulative plot of the best parameters for CLS, OBL and the combination of both on full.

### 5.3 Detailed Analysis

To further understand the effect of our method, we perform a series of analysis in order to find the relationship between the characteristic of predicate solving and the improvement. There are two kinds of comparison. The first one is to compare the basic PDR with the one activating one of the two predicates. The second one is to compare between the two predicates. We first introduce the information extracted from the process.

1. Total\_Pre: The number of predicates that is ever tried to prove.
2. PASS: The number of predicates successfully proved to be unreachable (safe).
3. Total\_Pre\_Inf: Total number of clauses in the inductive set produced by all the separate PDRs to solve predicate.
4. Added: Total number of clauses added back to the original PDR. After terminating the separate PDR, only the clause that is not subsumed by  $F_\infty$  can be added back to the original one.
5. Total\_Inf: Total number of clauses added to  $F_\infty$  of the original PDR throughout the process. This number includes the clauses counted in Added.
6. Cls\_Length: Average number of literals for the clauses counted in Total\_Inf.
7. Cls\_Length\_Pre: Average number of literals for the clauses counted in Added.
8. Runtime: Runtime for this configuration of PDR.
9. SAT\_Query\_Pre: The number of SAT query called by all the separate PDRs.

Note that the above numbers are extracted from the PDR with predicate solving since we want to find a better criterion to choose predicate. We only pick the runtime of basic PDR to show the improvement. Then we introduce the combination of the above numbers that we think meaningful.

- 
1. Improvement: The comparison between the Runtime of basic PDR (oldTime) and that of the PDR with predicate solving (newTime). The improvement is equal to  $(oldTime - newTime) / \text{Max}(oldTime, newTime)$ . PDR with predicate solving is more efficient iff the value is positive. In addition, if either one is timeout, the absolute value will be one. Last, in order to prevent noise, we do not consider the case that both of the configurations take less than five seconds. On the other hand, we only consider that one is faster than the other when comparing clause-based and obligation-based predicate.
  2. Added / Total\_Inf: The proportion of clauses in  $F_\infty$  added from the separate PDR. It is to show that how important the separate PDR plays the role for the original one.
  3. Added / (Total\_Inf \* SAT\_Query\_Pre): The proportion of clauses in  $F_\infty$  added from the separate PDR per SAT query. It is to show that how important one SAT query for separate PDR can contribute to the original one.
  4. PASS / Total\_Pre: The proportion of successful predicate.
  5. Added / Total\_Pre\_Inf: The proportion of clauses produced by the separate PDR that can be added back. It is to show the ability that the separate PDR can explore different kinds of clause.
  6. Added / SAT\_query\_Pre: The number of clauses added back to the original PDR per SAT query. It is to show that how efficient the separate PDR can generate useful clause.
  7. Cls\_Length\_Pre / Cls\_Length: The ratio of average length of clauses produced by the original and separate PDR. For a single clause, we think it excludes more states if it has shorter length and we call it stronger. It is to show which kinds of clause is stronger.

For the comparison of basic PDR and the one with predicate solving, there is no obvious target to predict the improvement. The even distribution takes place for both of



the predicates. Although we have difficulty identifying good predicate to maximize the benefits, we can still find some property about it. On the other hand, when comparing the two kinds of predicate, we can roughly distinguish the characteristic of efficiency for some cases, which provides us a way to choose between the predicates. In the following, we display some of the figures to demonstrate the analysis. In all the figures, each spot corresponds to a case. Furthermore, the experiment in this section only contains pdr\_vb.

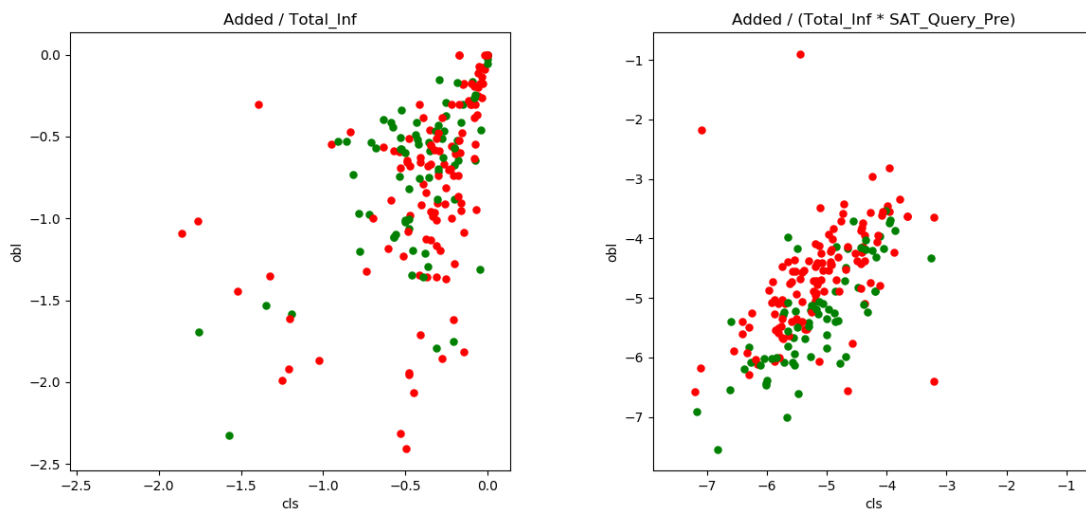


Figure 5.13: The comparison between clause-based and obligation-based predicate. X-axis is for clause and Y-axis is for obligation. The data are Added / Total\_Inf and Added / (Total\_Inf \* SAT\_Query\_Pre) and both of them are in log scale since the range of value is too wide to use linear scale. The figure is designed to be square to highlight the comparison. Green spot means clause-based predicate is more efficient and red one for obligation. There is no obvious boundary in the left subfigure. However, for the right subfigure, we can roughly find that the predicate is more efficient if the value is larger. That is, if we divide the square into two triangles by the line  $y = x$ , green spots mainly concentrate at the lower-right triangle, and vice versa.

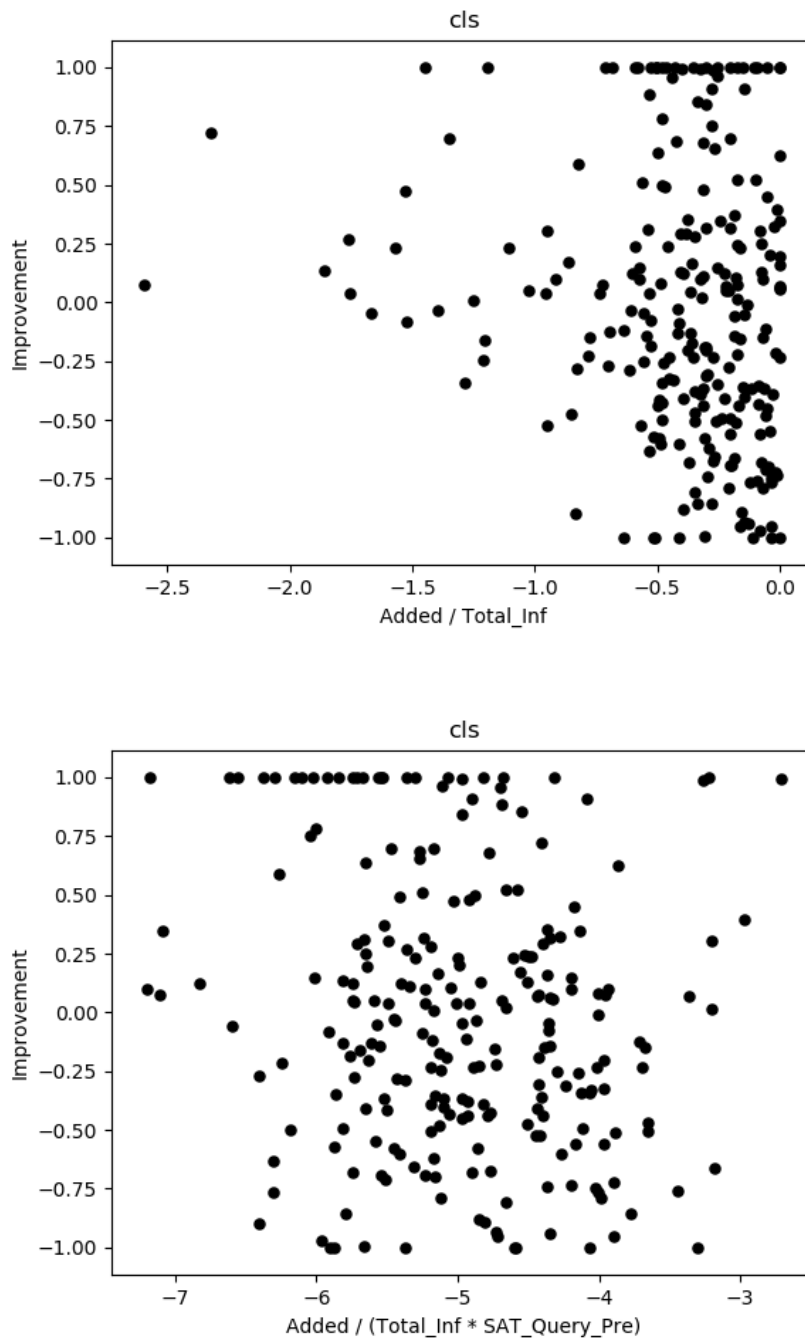


Figure 5.14: The comparison between basic PDR and clause-based predicate (1). Y-axis is the improvement and X-axis denotes the data including  $\text{Added} / \text{Total\_Inf}$  and  $\text{Added} / (\text{Total\_Inf} * \text{SAT\_Query\_Pre})$ . Both of the data are in log scale. There is no obvious target since almost for every value in X-axis, there exist positive and negative values for Y-axis. In other words, we cannot distinguish which range of value can lead to improvement.

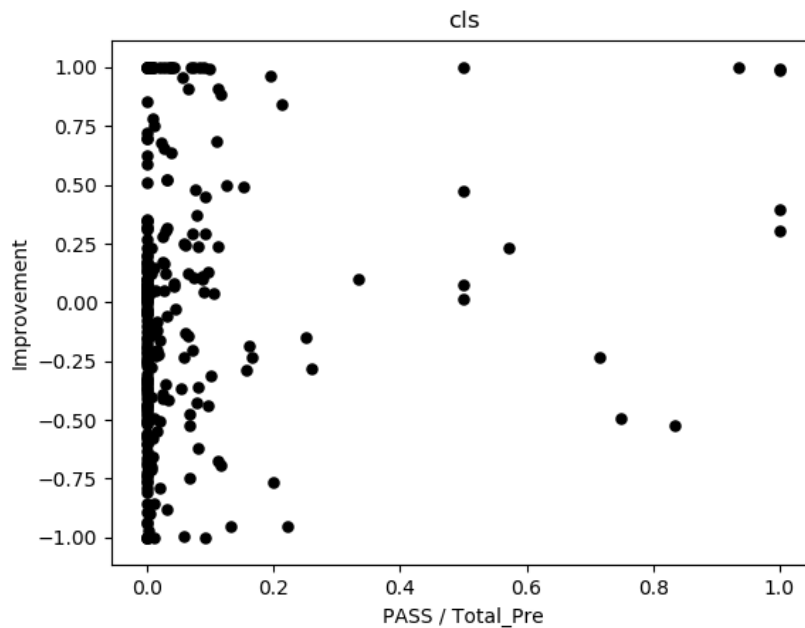


Figure 5.15: The comparison between basic PDR and clause-based predicate (2). Y-axis is the improvement and X-axis is PASS / Total\_Pre. There is no obvious target. Note that the proportion of PASS is actually very low, so the improvement can also result from the predicate that fails or aborts.

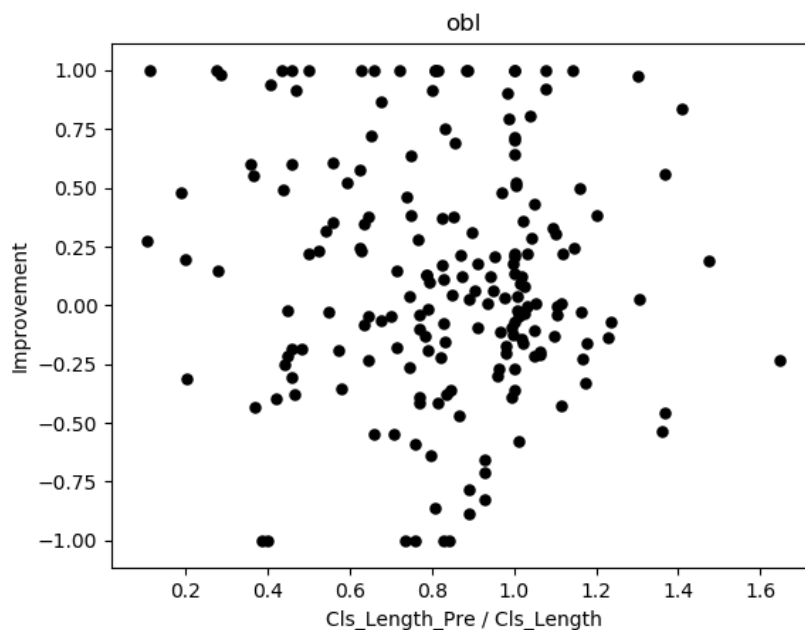


Figure 5.16: The comparison between basic PDR and obligation-based predicate. Y-axis is the improvement and X-axis is Cls\_Length\_Pre / Cls\_Length. There is no obvious target. Note that the average length of clauses for separate PDR is often shorter than that of all the clauses (ratio < 1).



## Chapter 6

# Conclusion and Future Work

In this thesis, we propose a flexible method to solve predicate separately. In addition, we provide two examples of useful predicate identified from blocking clauses and proof obligations. We show by the experiment that this method is efficient compared to the original PDR. Although we achieve a good result at the first step, there are still many works to do.

We have shown that there exists a trend for predicate solving. An important job is to find the absolutely best value for all the parameters. However, the values may be case-dependent and vary among different sets of benchmarks. We then modify the goal to find a fast and effective way to identify the best combination under a given set of circuits. On the other hand, since the case-dependency, another choice is to find a good heuristic to change the value dynamically. This again rely on the local information to do the modification.

On top of the two kinds of relatively general predicate we present, there may exists many other possibilities due to the variety of the circuit characteristics. By creating more and more patterns as predicate, we expect to answer more unsolved cases by the support of different one. Then the rest problem is to combine them automatically. As the experiment, simply activating all the predicates is not a guarantee of improvement. We provide two possible solutions here. The first one is to use multi-threads in parallel.

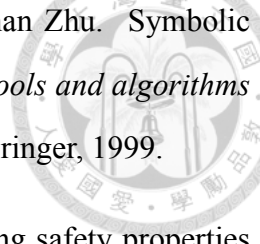
Every thread is associated with its own kind of predicate and is not interrupted by the other ones. The second one is more aggressive to integrate to a single engine. We use local information again to trigger the corresponding predicate. Hence, the activated predicate may be different throughout the process. There is also possible to be no or more than one predicate at a time.


As indicated in Subsection 4.2.7, we can merge all the clauses back after solving the predicate. That is, we are not limited by  $F_\infty$ . A possible scheme is when observing similar clauses in a frame. We can change the termination criteria to be up to that frame. If reaching that frame successfully, we know the common part is unreachable up to that frame. Hence, we get a trace of frame that can eliminate all the similar clauses at that frame, which leads to a smoother representation of the reachability.




## Reference

- [1] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.
- [2] João P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
- [3] Hantao Zhang. Sato: An efficient propositional prover. In *International Conference on Automated Deduction*, pages 272–275. Springer, 1997.
- [4] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.
- [5] E Goldberg and Y Novikov. Berkmin: A fast and robust sat-solver. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 142–149. IEEE, 2002.
- [6] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.
- [7] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Twenty-first International Joint Conference on Artificial Intelligence*, 2009.

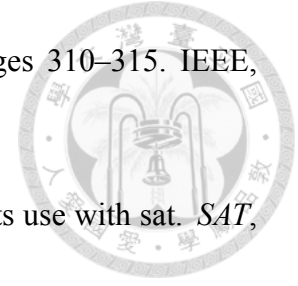
- 
- [8] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.
- [9] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000.
- [10] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.
- [11] Kenneth L McMillan. Interpolation and sat-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 2003.
- [12] William Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *The Journal of Symbolic Logic*, 22(3):250–268, 1957.
- [13] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *2009 Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2009.
- [14] Vijay D’ Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 129–145. Springer, 2010.
- [15] Simone Fulvio Rollini, Ondrej Sery, and Natasha Sharygina. Leveraging interpolant strength in model checking. In *International Conference on Computer Aided Verification*, pages 193–209. Springer, 2012.
- [16] Aaron R Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

- 
- [17] Aaron R Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 173–180. IEEE, 2007.
- [18] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134. IEEE, 2011.
- [19] Hong-Syun Jiang and Chung-Yang (Ric) Huang. Enhancing property directed reachability technique through cube analysis. *Master Thesis, National Taiwan University*, 2015.
- [20] Kuan Fan, Ming-Jen Yang, and Chung-Yang Huang. Automatic abstraction refinement of tr for pdr. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 121–126. IEEE, 2016.
- [21] Ming-Jen Yang and Chung-Yang (Ric) Huang. Improving property directed reachability with temporal decomposition. *Master Thesis, National Taiwan University*, 2016.
- [22] Cheng-Han Yang and Chung-Yang (Ric) Huang. Improving property directed reachability using dynamic timeframe expansion. *Master Thesis, National Taiwan University*, 2017.
- [23] Shih-Yu Chuang and Chung-Yang (Ric) Huang. Property directed reachability with interpolation refinement. *Master Thesis, National Taiwan University*, 2019.
- [24] Alexander Ivrii and Arie Gurfinkel. Pushing to the top. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 65–72. IEEE, 2015.
- [25] Ken L McMillan. Applying sat methods in unbounded symbolic model checking. In *International Conference on Computer Aided Verification*, pages 250–264. Springer, 2002.



- 
- [26] Armin Biere. The aiger and-inverter graph (aig) format version 20071012. *FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr, 69:4040*, 2007.
- [27] Armin Biere, Keijo Heljanko, and Siert Wieringa. Aiger 1.9 and beyond. *Available at fmv.jku.at/hwmcc11/beyond1.pdf*, 2011.
- [28] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert K Brayton. Fraigs: A unifying representation for logic synthesis and verification. Technical report, ERL Technical Report, 2005.
- [29] Robert Brummayer and Armin Biere. Local two-level and-inverter graph minimization without blowup. *Proc. MEMICS*, 6:32–38, 2006.
- [30] Jordi Cortadella. Timing-driven logic bi-decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):675–685, 2003.
- [31] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *International conference on theory and applications of satisfiability testing*, pages 61–75. Springer, 2005.
- [32] Niklas Eén and Niklas Sörensson. *The MiniSat Page*. <http://minisat.se/>.
- [33] Gilles Audemard and Laurent Simon. *Glucose's home page*. <https://www.labri.fr/perso/lsimon/glucose/>.
- [34] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*. Leningrad:Steklov Math. Institute, 1968.
- [35] David A Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.
- [36] Miroslav N Velev. Efficient translation of boolean formulas to cnf in formal verification of microprocessors. In *ASP-DAC 2004: Asia and South Pacific Design*

*Automation Conference 2004 (IEEE Cat. No. 04EX753)*, pages 310–315. IEEE, 2004.



- [37] Daniel Sheridan. The optimality of a fast cnf conversion and its use with sat. *SAT*, 2, 2004.
- [38] Niklas Een, Alan Mishchenko, and Niklas Sörensson. Applying logic synthesis for speeding up sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 272–286. Springer, 2007.
- [39] Alberto Griggio and Marco Roveri. Comparing different variants of the ic3 algorithm for hardware model checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(6):1026–1039, 2015.
- [40] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.
- [41] Cheng-Yin Wu and Chung-Yang (Ric) Huang. *V3: An extensible framework for hardware verification*. <https://github.com/chengyinwe/V3>.
- [42] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. <https://people.eecs.berkeley.edu/~alanmi/abc/>.