

BACHELOR THESIS

PROPERTY DIRECTED REACHABILITY
IN ULTIMATE

JONAS WERNER

EXAMINER: PROF. DR. ANDREAS PODELSKI

ADVISER: DR. DANIEL DIETSCH

ALBERT-LUDWIGS-UNIVERSITY FREIBURG
FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE
CHAIR OF SOFTWARE ENGINEERING

AUGUST 3RD, 2018

Writing period

03.05.2018 – 03.08.2018

Examiner

Prof. Dr. Andreas Podelski

Adviser

Dr. Daniel Dietsch

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date

Jonas Werner

Abstract

Zusammenfassung

foo bar aber auf deutsch

Contents

1	Introduction	1
2	Background: PDR for Hardware	3
2.1	Preliminaries	3
2.2	Algorithm	5
2.3	Examples	8
2.3.1	Case 1: Property not Satisfied	8
2.3.2	Case 2: Property Satisfied	12
2.4	Possible Improvements	14
2.4.1	Generalization of States	14
2.4.2	Ternary Simulation	14
3	PDR for Software	17
3.1	Preliminaries	17
3.2	Lifted Algorithm	18
3.3	Example	21
3.3.1	Case 1: Error State is Reachable	21
3.3.2	Case 2: Error State is Unreachable	25
3.4	Possible Improvements	28
3.4.1	Syntactical Analysis	28
3.4.2	Weakest Precondition	29
3.4.3	Disjunctive Normal Form	29
3.4.4	Interpolation	29
4	PDR in Ultimate	31
4.1	Introduction ULTIMATE	31
4.2	Implementation	31
4.3	Improvements	32
4.3.1	Caching of Proof-Obligation Queue	32
4.3.2	Ignoring Already Blocked Proof-Obligations	33
4.3.3	Using Preconditions	34

5	Evaluation	37
5.1	Comparison	37
5.2	Discussion	38
6	Related Work	41
7	Future Work	43
7.1	Further Improvements	43
7.1.1	Interpolation	43
7.1.2	Procedures	43
8	Conclusion	45

1 Introduction

SAT-based model-checking is a useful technique for both software and hardware verification. Recently a novel hardware verification algorithm was devised by Aaron Bradley [5] called **IC3**. Because it was so new, it came as a surprise that it won third place in the hardware model-checking competition (HWMCC) at CAV 2010 [1].

The model-checking method behind **IC3** is called *Property Directed Reachability*, *PDR* for short, which is based on backward-search. It tries to find inductive invariants by constructing an over-approximation of reachable states. During this construction, possible counter-examples are disproven using Boolean SAT queries. Because this approach turned out to be very efficient for hardware verification, it could be interesting for software verification as well.

ULTIMATE [3] is a software analysis framework consisting of multiple plugins that perform steps of a program analysis, like parsing source code, transforming programs from one representation to another, or analyse programs. **ULTIMATE** already has analysis-plugins using different model-checking techniques like trace abstraction [7] or lazy interpolation [9]. The goal of this Bachelor's Thesis is to implement a new analysis-plugin that is based on PDR but applicable for software in **ULTIMATE** and to compare it with the other existing techniques.

This thesis is structured as follows: in Chapter 2 we introduce how PDR is used on hardware, we present an algorithm, and discuss possible methods to make it more efficient. In Chapter 3 we explain our approach of lifting PDR from hardware to software, presenting a lifted algorithm, and again discuss possible improvements. Chapter 4 focuses on our implementation of PDR in **Ultimate**, followed by an evaluation in Chapter 5 where we compare PDR to an already existing analysis-plugin. And last but not least we give an outlook on possible future work in Chapter 6.

2 Background: PDR for Hardware

In this section we describe how PDR works as a hardware-checker, which requires propositional logic, so that every variable is a boolean.

2.1 Preliminaries

First some preliminary definitions and notations:

A *boolean transition system* is a tuple $S = (X, I, T)$, where X is a finite set of boolean variables, I is a disjunction representing the *initial state*, and T is a propositional formula over variables in X and $X' = \{x \in X \mid x' \in X'\}$, called transition relation, that describes updates to the variables.

For example, consider the transition system $U = (X, I, T)$ where

- $X = \{x_1, x_2, x_3\}$
- $I = \bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3$
- $T = (x_1 \vee \neg x'_2) \wedge (\bar{x}_1 \vee x'_2) \wedge (x_2 \vee \bar{x}'_3) \wedge (\bar{x}_2 \vee x'_3)$

With corresponding transition graph:

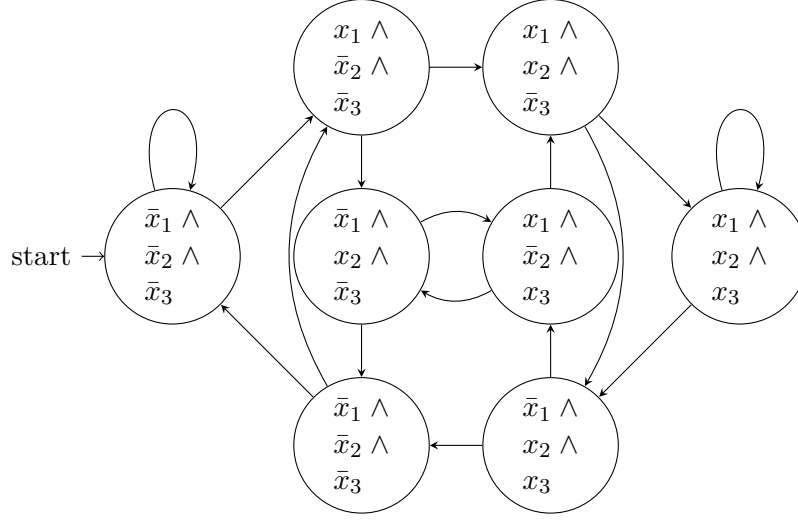


Figure 2.1: Transition Graph of U .

We call a variable or its negation, e.g., x or \bar{y} , a *literal*

Furthermore, a conjunction of literals is called a *cube*, e.g., $x \wedge \bar{y}$, and a disjunction of literals a *clause*, e.g., $x \vee \bar{y}$.

Note that the negation of a cube is a clause and vice versa.

Given a propositional formula ϕ over X we get a *primed formula* ϕ' by replacing each variable with its corresponding variable in X' .

A *state* in S is a cube containing each variable from X with a boolean valuation of it. For each possible valuation there is a corresponding state, resulting in $2^{|X|}$ states in S . Like we see in the graph of U we have $2^{|X|} = 2^3 = 8$ states.

A *transition* from one state s to another state q exists if the conjunction of s , the transition relation, and q' is satisfiable.

For example in U the transition between the initial state $I = \bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3$ and state $r = x_1 \wedge \bar{x}_2 \wedge \bar{x}_3$ exists because

$$\underbrace{\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_I \wedge \underbrace{(x_1 \vee \bar{x}_2') \wedge (\bar{x}_1 \vee x_2') \wedge (x_2 \vee \bar{x}_3') \wedge (\bar{x}_2 \vee x_3')}_T \wedge \underbrace{x_1' \wedge \bar{x}_2' \wedge \bar{x}_3'}_{r'}$$

is satisfiable.

Given a propositional formula P over X , called *property*, we want to verify that every state in S that is reachable from I satisfies P such that, P describes a finite set of *good states*, conversely \bar{P} represent a finite set of *bad states*.

Regarding U , let $P = \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$ be given, making $\bar{P} = x_1 \wedge x_2 \wedge x_3$ a bad state.

In the next section we explain how PDR can be used to show that either \bar{P} is unreachable from I or that there exists a sequence of transitions leading to \bar{P} as counter-example.

2.2 Algorithm

A PDR algorithm tries to prove that a transition system $S = (X, I, T)$ satisfies a given property P by trying to find a formula F over X with the following qualities:

- (1) $I \Rightarrow F$
- (2) $F \wedge T \Rightarrow F'$
- (3) $F \Rightarrow P$

F is called an *inductive invariant*.

To calculate an inductive invariant, PDR uses *frames* which are cubes of clauses representing an over-approximation of reachable states in at most i transitions from I .

PDR maintains a sequence of frames $[F_0, \dots, F_k]$, called a *trace*, which is ordered so that it fulfills the following characteristics:

- (I) $F_0 = I$
- (II) $F_{i+1} \subseteq F_i$, therefore $F_i \Rightarrow F_{i+1}$
- (III) $F_i \wedge T \Rightarrow F'_{i+1}$
- (IV) $F_i \Rightarrow P$

```

1: procedure PDR-PROVE( $I, T, P$ )
2:   check for 0-counter-example
3:    $trace.push(new\ frame(I))$ 

4:   loop
5:     while  $\exists$  cube  $c$ , s.t.  $trace.last() \wedge T \wedge c'$  is SAT and  $c \Rightarrow \bar{P}$  do
6:       recursively block proof-obligation( $c$ ,  $trace.size() - 1$ )
7:       and strengthen the frames of the trace.
8:       if a proof-obligation( $p$ , 0) is generated then
9:         return false ▷ counter-example found

10:     $F_{k+1} = new\ frame(P)$ 
11:    for all clause  $c \in trace.last()$  do
12:      if  $trace.last() \wedge T \wedge c'$  is UNSAT then
13:         $F_{k+1} = F_{k+1} \wedge c$ 
14:    if  $trace.last() == F_{k+1}$  then
15:      return true
16:     $trace.push(F_{k+1})$ 

```

Figure 2.2: Bit-Level PDR Algorithm.

Figure 2.2 shows the pseudo-code of the PDR algorithm. It starts on line 2 with checking for a *0-counter-example*, that means checking if $I \Rightarrow P$, by testing whether the formula $I \wedge \bar{P}$ is satisfiable. If it is, then I is a 0-counter-example, the algorithm terminates. If the formula is unsatisfiable the algorithm continues on to line 3, where it initializes the first frame $F_0 = I$, fulfilling (I), and moving on.

Let $[F_0, F_1, \dots, F_k]$ be the current trace.

The algorithm repeats the following three phases until termination:

1. Next Transition

In the code-block from lines 4 to 9 the algorithm checks whether the next state is a good state meaning if $F_k \wedge T \Rightarrow P'$ is valid, by testing the satisfiability of $F_k \wedge T \wedge \bar{P}'$

- If the formula is *satisfiable*, for each satisfying assignment $\vec{x} = (x_1, x_2, \dots, x_{|X|}, x'_1, x'_2, \dots, x'_{|X'|})$ the algorithm gets a new bad state $a = x_1 \wedge x_2 \wedge \dots \wedge x_{|X|}$ and creates tuple (a, k) , this tuple is called a *proof-obligation*.
- If the formula is *unsatisfiable*, the procedure continues with the next phase.

2. Blocking-Phase

The blocking-phase is nested in the next transition phase, it can be found in lines 6 to

9. In it the algorithm checks whether there are proof-obligations, if so, it takes proof-obligation (b, i) and tries to block the bad state b by checking if frame F_{i-1} can reach b in one transition, i.e., it tests $F_{i-1} \wedge T \wedge b'$ for satisfiability.

- If the formula is *satisfiable*, it means that F_i is not strong enough to block b . For each satisfying assignment $\vec{x} = (x_1, x_2, \dots, x_{|X|}, x'_1, x'_2, \dots, x'_{|X'|})$ the algorithm gets a new bad state $c = x_1 \wedge x_2 \wedge \dots \wedge x_{|X|}$ creating the new proof-obligation $(c, i - 1)$.
- If the formula is *unsatisfiable*, the algorithm strengthens frame F_i with \bar{b} meaning $F_i = F_i \wedge \bar{b}$, blocking b at F_i

This continues recursively until either a proof-obligation $(d, 0)$ is created proving that there exists a counter-example terminating the algorithm, or there is no proof-obligation left.

3. Propagation-Phase

Found in the final code-block of the pseudo-code from line 10 to 16. In it the algorithm adds a new frame $F_{k+1} = P$ and propagates clauses from F_k forward, meaning for all clauses c in F_k it checks $F_k \wedge T \wedge \bar{c}'$ for satisfiability. If that conjunction is unsatisfiable, the algorithm strengthens F_{k+1} with c : $F_{k+1} = F_{k+1} \wedge c$, if it is satisfiable the algorithm does nothing and continues with the next clause. Because of this phase rule (II) is fulfilled.

After propagating all possible clauses, if $F_{k+1} \equiv F_k$ the algorithm found a fix-point and terminates returning that P always holds with F_k being the inductive invariant.

To illustrate the procedure further consider the pseudo-code:

2.3 Examples

In this section we demonstrate the application of the PDR algorithm on both possible cases. Example 2.3.1 shows an example where the property is not satisfied, followed by Example 2.3.2 showing the contrary.

2.3.1 Case 1: Property not Satisfied

To show an application of the algorithm reconsider the example transition system $U = (X, I, T)$ in Figure 3.1, where

- $X = \{x_1, x_2, x_3\}$
- $I = \bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3$
- $T = (x_1 \vee \bar{x}_2') \wedge (\bar{x}_1 \vee x_2') \wedge (x_2 \vee \bar{x}_3') \wedge (\bar{x}_2 \vee x_3')$

and the property:

- $P = \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$ with bad state $\bar{P} = x_1 \wedge x_2 \wedge x_3$

We now want to verify whether P holds or if there is a counter-example.

1. Step: Check for 0-Counter-Example

We need to make sure that $I \Rightarrow P$, we do that by testing if $I \wedge \bar{P}$ is satisfiable:

$$\underbrace{\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_I \wedge \underbrace{x_1 \wedge x_2 \wedge x_3}_{\bar{P}}$$

The formula is unsatisfiable meaning there is no 0-counter-example, we continue by initializing $F_0 = I$

2. Step: First Transition

Check if $F_0 \wedge T \Rightarrow P'$, by testing if $F_0 \wedge T \wedge \bar{P}'$ is satisfiable:

$$\underbrace{\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_{F_0} \wedge \underbrace{(x_1 \vee \bar{x}_2') \wedge (\bar{x}_1 \vee x_2') \wedge (x_2 \vee \bar{x}_3') \wedge (\bar{x}_2 \vee x_3')}_{T} \wedge \underbrace{x_1' \wedge x_2' \wedge x_3'}_{\bar{P}'}$$

Which is not satisfiable because $\bar{x}_1 \wedge (x_1 \vee \bar{x}'_2) \wedge x'_2$ is unsatisfiable. We do not generate a proof-obligation, so that we can skip the blocking-phase and continue on with the propagation-phase.

3. Step: First Propagation-Phase

Initialize the new frame $F_1 = P$

We need to check each clause c in F_0 if $F_0 \wedge T \wedge \bar{c}'$ is unsatisfiable to strengthen F_1 .

- $c = \bar{x}_1$:

$$\underbrace{\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_{F_0} \wedge \underbrace{(x_1 \vee \bar{x}'_2) \wedge (\bar{x}_1 \vee x'_2) \wedge (x_2 \vee \bar{x}'_3) \wedge (\bar{x}_2 \vee x'_3)}_T \wedge \underbrace{x'_1}_{\bar{c}'}$$

The conjunction is satisfiable with assignment $(\bar{x}_1, \bar{x}_2, \bar{x}_3, x'_1, \bar{x}'_2, \bar{x}'_3)$

We do not need to add \bar{x}_1 to F_1 .

- $c = \bar{x}_2$:

$$\underbrace{\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_{F_0} \wedge \underbrace{(x_1 \vee \bar{x}'_2) \wedge (\bar{x}_1 \vee x'_2) \wedge (x_2 \vee \bar{x}'_3) \wedge (\bar{x}_2 \vee x'_3)}_T \wedge \underbrace{x'_2}_{\bar{c}'}$$

The conjunction is unsatisfiable because $\bar{x}_1 \wedge (x_1 \vee \bar{x}'_2) \wedge x'_2$ is not satisfiable

We add \bar{x}_2 to F_1 resulting in $F_1 = P \wedge \bar{x}_2$.

- $c = \bar{x}_3$:

$$\underbrace{\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_{F_0} \wedge \underbrace{(x_1 \vee \bar{x}'_2) \wedge (\bar{x}_1 \vee x'_2) \wedge (x_2 \vee \bar{x}'_3) \wedge (\bar{x}_2 \vee x'_3)}_T \wedge \underbrace{x'_3}_{\bar{c}'}$$

The conjunction is unsatisfiable because $\bar{x}_2 \wedge (x_2 \vee \bar{x}'_3) \wedge x'_3$ is not satisfiable

We add \bar{x}_3 to F_1 resulting in $\rightarrow F_1 = P \wedge \bar{x}_2 \wedge \bar{x}_3$

There are no clauses left, the first propagation-phase is done, resulting in the new frame:

$$F_1 = (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge \bar{x}_2 \wedge \bar{x}_3$$

and because $F_1 \neq F_0$ we continue.

4. Step: Second Transition

Check if $F_1 \wedge T \Rightarrow P'$ by testing $F_1 \wedge T \wedge \bar{P}'$ for satisfiability:

$$\underbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge \bar{x}_2 \wedge \bar{x}_3}_{F_1} \wedge \underbrace{(x_1 \vee \bar{x}'_2) \wedge (\bar{x}_1 \vee x'_2) \wedge (x_2 \vee \bar{x}'_3) \wedge (\bar{x}_2 \vee x'_3)}_T \wedge \underbrace{x'_1 \wedge x'_2 \wedge x'_3}_{\bar{P}'}$$

Which is unsatisfiable because $\bar{x}_2 \wedge (x_2 \vee \bar{x}_3') \wedge x_3'$ is not satisfiable. We do not generate a proof-obligation so we continue with the second propagation-phase.

5. Step: Second Propagation-Phase

Initialize new frame $F_2 = P$

We need to check each clause c in F_1 if $F_1 \wedge T \wedge c'$ is unsatisfiable to strengthen F_2 . We skip P , as it is already part of F_2 .

This works exactly as in the 3. step:

- $c = \bar{x}_2$:

$$\underbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge \bar{x}_2 \wedge \bar{x}_3}_{F_1} \wedge \underbrace{(x_1 \vee \bar{x}_2') \wedge (\bar{x}_1 \vee x_2') \wedge (x_2 \vee \bar{x}_3') \wedge (\bar{x}_2 \vee x_3')}_{T} \wedge \underbrace{x_2'}_{c'}$$

The conjunction is satisfiable with assignment $(x_1, \bar{x}_2, \bar{x}_3, x_1', x_2', \bar{x}_3')$

We do not need to add \bar{x}_2 to F_2

- $c = \bar{x}_3$:

$$\underbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge \bar{x}_2 \wedge \bar{x}_3}_{F_1} \wedge \underbrace{(x_1 \vee \bar{x}_2') \wedge (\bar{x}_1 \vee x_2') \wedge (x_2 \vee \bar{x}_3') \wedge (\bar{x}_2 \vee x_3')}_{T} \wedge \underbrace{x_3'}_{c'}$$

The conjunction is unsatisfiable because $\bar{x}_2 \wedge (x_2 \vee \bar{x}_3') \wedge x_3'$ is not satisfiable.

We add \bar{x}_3 to F_2 resulting in $F_2 = P \wedge \bar{x}_3$

As there are no clauses left, the second propagation-phase concludes, resulting in the new frame:

$$F_2 = (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge \bar{x}_3$$

and because $F_2 \neq F_1$ we continue.

6. Step: Third Transition

Check if $F_2 \wedge T \Rightarrow P'$ by testing $F_2 \wedge T \wedge \bar{P}'$ for satisfiability:

$$\underbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge \bar{x}_3}_{F_2} \wedge \underbrace{(x_1 \vee \bar{x}_2') \wedge (\bar{x}_1 \vee x_2') \wedge (x_2 \vee \bar{x}_3') \wedge (\bar{x}_2 \vee x_3')}_{T} \wedge \underbrace{x_1' \wedge x_2' \wedge x_3'}_{\bar{P}'}$$

This time $F_2 \wedge T \wedge \bar{P}'$ is satisfiable with assignment $(\underbrace{x_1, x_2, \bar{x}_3}_s, x_1', x_2', x_3')$, we get the

new bad state $s = x_1 \wedge x_2 \wedge \bar{x}_3$, and generate a proof-obligation $(s, 2)$, which we now try to block in the blocking-phase.

7. Step: First Blocking-Phase

Try to block proof-obligation $(s, 2)$ by checking if $F_1 \wedge T \wedge s'$ is satisfiable.

$$\underbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge \bar{x}_2 \wedge \bar{x}_3}_{F_1} \wedge \underbrace{(x_1 \vee \bar{x}'_2) \wedge (\bar{x}_1 \vee x'_2) \wedge (x_2 \vee \bar{x}'_3) \wedge (\bar{x}_2 \vee x'_3)}_T \wedge \underbrace{x'_1 \wedge x'_2 \wedge \bar{x}'_3}_{s'}$$

This is again satisfiable with assignment $\underbrace{(x_1, \bar{x}_2, \bar{x}_3)}_q, x'_1, x'_2, \bar{x}'_3$, we get the bad state $q = x_1 \wedge \bar{x}_2 \wedge \bar{x}_3$ and generate a new proof-obligation $(q, 1)$.

We try to block proof-obligation $(q, 1)$ by checking if $F_0 \wedge T \wedge q'$ is satisfiable.

$$\underbrace{\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_{F_0} \wedge \underbrace{(x_1 \vee \bar{x}'_2) \wedge (\bar{x}_1 \vee x'_2) \wedge (x_2 \vee \bar{x}'_3) \wedge (\bar{x}_2 \vee x'_3)}_T \wedge \underbrace{x'_1 \wedge \bar{x}'_2 \wedge \bar{x}'_3}_{q'}$$

This too is satisfiable with assignment $\underbrace{(\bar{x}_1, \bar{x}_2, \bar{x}_3)}_I, x'_1, \bar{x}'_2, \bar{x}'_3$, we get the bad state $I = x_1 \wedge \bar{x}_2 \wedge \bar{x}_3$ and generate a new proof-obligation $(I, 0)$.

With that we have found a counter-example, resulting in the termination of the algorithm, returning the counter-example trace:

$$\underbrace{\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_I \rightarrow \underbrace{x_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_q \rightarrow \underbrace{x_1 \wedge x_2 \wedge \bar{x}_3}_s \rightarrow \underbrace{x_1 \wedge x_2 \wedge x_3}_{\bar{P}}$$

If we now assume proof-obligation $(s, 2)$ would have been blocked, meaning $F_1 \wedge T \wedge s'$ was unsatisfiable, then we would have updated $F_2 = F_1 \wedge \bar{s}$ making absolutely sure that s is not reachable, every future proof-obligation containing s would have been blocked by F_2 .

2.3.2 Case 2: Property Satisfied

To show a transition system with an inductive invariant, consider $B = (X, I, T)$, where

- $X = \{x_1, x_2\}$
- $I = \bar{x}_1 \wedge \bar{x}_2$
- $T = (x_1 \vee \bar{x}_2 \vee x'_2) \wedge (x_1 \vee x_2 \vee \bar{x}'_1) \wedge (\bar{x}_1 \vee x'_1) \wedge (\bar{x}_1 \vee \bar{x}'_2) \wedge (x_2 \vee \bar{x}'_2)$

and corresponding transition graph:

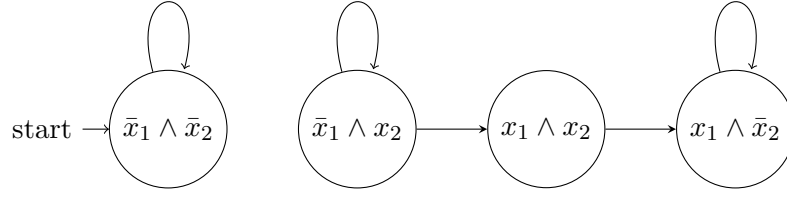


Figure 2.3: Transition Graph of B .

Now given the property $P = \bar{x}_1 \vee x_2$, we want to check whether the bad state $\bar{P} = x_1 \wedge \bar{x}_2$ is reachable:

1. Step: Check for 0-Counter-Example

We need to make sure that $I \Rightarrow P$, we do that by testing if $I \wedge \bar{P}$ is satisfiable:

$$\underbrace{\bar{x}_1 \wedge \bar{x}_2}_I \wedge \underbrace{x_1 \wedge \bar{x}_2}_{\bar{P}}$$

Which is unsatisfiable because $\bar{x}_1 \wedge x_1$, that means there is no 0-counter-example, we continue by initializing $F_0 = I$

2. Step: First Transition

Check if $F_0 \wedge T \Rightarrow P'$ by testing if $F_0 \wedge T \wedge \bar{P}'$ is satisfiable:

$$\underbrace{\bar{x}_1 \wedge \bar{x}_2}_{F_0} \wedge \underbrace{(x_1 \vee \bar{x}_2 \vee x'_2) \wedge (x_1 \vee x_2 \vee \bar{x}'_1) \wedge (\bar{x}_1 \vee x'_1) \wedge (\bar{x}_1 \vee \bar{x}'_2) \wedge (x_2 \vee \bar{x}'_2)}_T \wedge \underbrace{x'_1 \wedge \bar{x}'_2}_{\bar{P}'}$$

Which is unsatisfiable because $\bar{x}_1 \wedge \bar{x}_2 \wedge (x_1 \vee x_2 \vee x'_1) \wedge \bar{x}'_1$ is not satisfiable. We generate no proof-obligation and continue with the propagation-phase.

3. Step: First Propagation-Phase

Initialize the new frame $F_1 = P$

For each clause c in F_0 we check $F_0 \wedge T \wedge \bar{c}'$ for unsatisfiability to strengthen F_1 .

- $c = \bar{x}_1$:

$$\bar{x}_1 \wedge \bar{x}_2 \wedge T \wedge x'_1$$

The conjunction is unsatisfiable because $\bar{x}_1 \wedge \bar{x}_2 \wedge (x_1 \vee x_2 \vee x'_1) \wedge \bar{x}'_1$ is not satisfiable.

We add \bar{x}_1 to F_1 resulting in $F_1 = P \wedge \bar{x}_1$

- $c = \bar{x}_2$:

$$\bar{x}_1 \wedge \bar{x}_2 \wedge T \wedge x'_2$$

The conjunction is unsatisfiable because $\bar{x}_1 \wedge \bar{x}_2 \wedge (x_2 \vee \bar{x}'_2) \wedge x'_2$ is not satisfiable.

We add \bar{x}_2 to F_1 resulting in $F_1 = P \wedge \bar{x}_1 \wedge \bar{x}_2$

That concludes the propagation-phase resulting in the new frame

$$F_1 = (\bar{x}_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2$$

and because $F_1 \not\equiv F_0$ we continue.

4. Step: Second Transition

Check if $F_1 \wedge T \Rightarrow P'$ by testing if $F_1 \wedge T \wedge \bar{P}'$ is satisfiable:

$$(\bar{x}_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2 \wedge T \wedge x'_1 \wedge \bar{x}'_2$$

Which is unsatisfiable because $\bar{x}_1 \wedge \bar{x}_2 \wedge (x_1 \vee x_2 \vee \bar{x}'_1) \wedge x'_1$ is not satisfiable. We again do not generate a proof-obligation, so that we continue with the second propagation-phase.

5. Step: Second Propagation-Phase

Initialize the new frame $F_2 = P$

We need to check every clause c in F_1 if $F_1 \wedge T \wedge \bar{c}'$ is unsatisfiable to strengthen F_2 .

We skip P .

- $c = \bar{x}_1$:

$$(\bar{x}_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2 \wedge T \wedge x'_1$$

The conjunction is unsatisfiable because $\bar{x}_1 \wedge \bar{x}_2 \wedge (x_1 \vee x_2 \vee \bar{x}'_1) \wedge x'_1$ is not satisfiable

We add \bar{x}_1 to F_2 resulting in $F_2 = P \wedge \bar{x}_1$

- $c = \bar{x}_2$:

$$(\bar{x}_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2 \wedge T \wedge x'_2$$

The conjunction is unsatisfiable because $\bar{x}_2 \wedge (x_2 \vee \bar{x}'_2) \wedge x'_2$ is not satisfiable.

We add \bar{x}_2 to F_2 resulting in $F_2 = P \wedge \bar{x}_1 \wedge \bar{x}_2$

With no more clauses left the second propagation-phase ends, resulting in

$$F_2 = (\bar{x}_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2 \equiv F_1$$

The algorithm terminates returning that the property always holds and

$$(\bar{x}_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2$$

being an inductive invariant.

2.4 Possible Improvements

In this section we discuss possible improvements to the shown algorithm to make it more efficient, because the most time consuming part of the PDR algorithm is the solving of SAT-queries, the larger the query the more time it takes. To improve this there are several ways to keep SAT-queries small.

2.4.1 Generalization of States

Blocking one state at a time is ineffective.

When blocking a state s do not add \bar{s} but try to find and add a cube $c \subseteq \bar{s}$.

Most modern SAT-solver not only return unsatisfiable but also a reason for it, either by an UNSAT-core or through a final conflict-clause. Both of them deliver information about which clauses were actually used in the proof. To find c just remove unused clauses of s .

2.4.2 Ternary Simulation

To reduce proof-obligations it is possible to eliminate not needed state variables by checking a satisfying assignment using ternary simulation.

Ternary logic extends the binary logic by introducing a new valuation: X , called unknown, and new rules:

$$\begin{aligned} (X \wedge false) &= false, \\ (X \wedge true) &= X, \\ (X \wedge X) &= X, \\ \bar{X} &= X \end{aligned}$$

To remove state variables, set one variable at a time to X and try to transition to a next state using the transition relation, the variable is needed if X propagates into the next state, if it does not remove the variable from the proof-obligation.

Reconsider the prior example's first blocking phase resulting in the proof-obligation($q, 1$) with bad state $q = x_1 \wedge \bar{x}_2 \wedge \bar{x}_3$, we now want to reduce that proof-obligation using ternary simulation:

First of all, the transition formula:

$$\underbrace{x_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_q \wedge \underbrace{(x_1 \vee \bar{x}'_2) \wedge (\bar{x}_1 \vee x'_2) \wedge (x_2 \vee \bar{x}'_3) \wedge (\bar{x}_2 \vee x'_3)}_T$$

Now set $x_1 = X$:

$$\underbrace{X \wedge \bar{x}_2 \wedge \bar{x}_3}_q \wedge \underbrace{(X \vee \bar{x}'_2) \wedge (\bar{x}_1 \vee x'_2) \wedge (x_2 \vee \bar{x}'_3) \wedge (\bar{x}_2 \vee x'_3)}_T$$

$(X \vee \bar{x}'_2)$ is unknown meaning that x_1 is needed.

Now set $x_2 = X$:

$$\underbrace{x_1 \wedge X \wedge \bar{x}_3}_q \wedge \underbrace{(x_1 \vee \bar{x}'_2) \wedge (\bar{x}_1 \vee x'_2) \wedge (X \vee \bar{x}'_3) \wedge (X \vee x'_3)}_T$$

$(X \vee \bar{x}'_3)$ is unknown meaning that x_2 is needed as well.

Now set $x_3 = X$:

$$\underbrace{x_1 \wedge \bar{x}_2 \wedge X}_q \wedge \underbrace{(x_1 \vee \bar{x}'_2) \wedge (\bar{x}_1 \vee x'_2) \wedge (x_2 \vee \bar{x}'_3) \wedge (\bar{x}_2 \vee x'_3)}_T$$

Because there is no clause being unknown, x_3 can be removed from the proof-obligation. We get the reduced proof-obligation($x_1 \wedge \bar{x}_2, 1$)

3 PDR for Software

We see that PDR is a useful and efficient hardware-model checking technique. If we want to use it on software, we need to *lift* the algorithm from bit-level propositional logic to first-order logic. We base our approach on the technique described by Lange et al [11]. There are however other ways to accomplish this, see chapter 5 for an overview of other techniques.

To use PDR on software we first need some new definitions and other preliminaries.

3.1 Preliminaries

A control flow graph (CFG) $\mathcal{A} = (X, L, G, \ell_0, \ell_E)$ is a tuple, consisting of a finite set of variables X , a finite set of locations L , a finite set of transitions $G \subseteq L \times FO \times L$, FO being a quantifier free first-order logic formula over variables in X and $X' = \{x \in X \mid x' \in X'\}$, an initial location $\ell_0 \in L$, and an error location $\ell_E \in L$.

For example consider the CFG $\mathcal{A} = (X, L, G, \ell_0, \ell_E)$, where

- $X = \{x\}$
- $L = \{\ell_0, \ell_1, \ell_2, \ell_3, \ell_E\}$
- $G = \{(\ell_0, x' := 0, \ell_1), (\ell_1, x' := x + 1, \ell_2), (\ell_2, x = 1, \ell_E), (\ell_2, x \neq 1, \ell_3)\}$

with the graph:

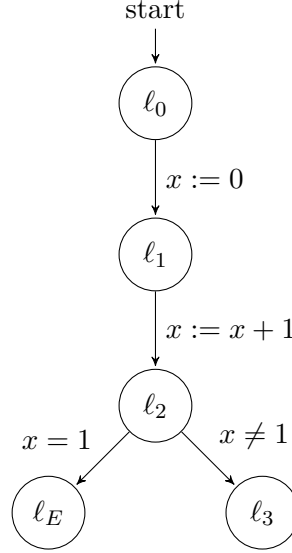


Figure 3.1: Graph of \mathcal{A} .

The transition formula $T_{\ell_1 \rightarrow \ell_2}$ from one location ℓ_1 to another location ℓ_2 is defined as:

$$T_{\ell_1 \rightarrow \ell_2} = \begin{cases} (\ell_1, t, \ell_2), & (\ell_1, t, \ell_2) \in G \\ false, & otherwise \end{cases}$$

Resulting in the global transition formula:

$$T = \bigvee_{(\ell_1, t, \ell_2) \in G} T_{\ell_1 \rightarrow \ell_2}$$

The lifted algorithm no longer works on boolean transition systems but on CFGs. It tries to prove whether ℓ_E is reachable, by finding a feasible path from ℓ_0 to ℓ_E .

3.2 Lifted Algorithm

There are four main differences between bit-level PDR and lifted PDR:

- Instead of a global set of Frames $[F_0, \dots, F_k]$ assign each program location $\ell \in L \setminus \{\ell_E\}$ a local set of frames $[F_{0,\ell}, \dots, F_{k,\ell}]$. Each frame is now a cube of first-order formulae. As there are now multiple traces, proof-obligations get extended

by another parameter, lifted proof-obligations are tuples (t, ℓ, i) where t is a first-order formula, ℓ describes the location where t has to be blocked, and i is a frame number, called level in the lifted algorithm.

- Because the states in a CFG are no formulae, the lifted algorithm no longer blocks states but transitions, there are no bad states only bad transitions.
- Because of the structure of the CFG, it is already known which states lead to the error location, as it is easy to extract the transitions in G that have ℓ_E as target, making the next transition phase, that was used to find proof-obligations before, obsolete.
If there exists a transition to ℓ_E there will be an initial proof-obligation in each iteration of the algorithm, making the blocking-phase no longer optional.
- The propagation-phase is slimmed, it only checks for termination. In the phase the algorithm checks the frames to find a level i where all locations have a fixpoint, meaning $F_{i,\ell} = F_{i-1,\ell}$ for every location $\ell \in L \setminus \{\ell_E\}$, i is called a global fixpoint position. There is no more propagating formulae forward.

In more detail:

Given a CFG $\mathcal{A} = (X, L, G, \ell_0, \ell_E)$ we want to check if ℓ_E is reachable:

Again start with checking for a 0-counter-example, this is easily done by looking at ℓ_0 if $\ell_0 = \ell_E$ terminate and return that ℓ_E is indeed reachable, if $\ell_0 \neq \ell_E$ initialize level 0 frames for all locations $\ell \in L \setminus \{\ell_0, \ell_E\}$ as **false**, and for ℓ_0 as **true**.

Let k be the current level, so that each location $\ell \in L \setminus \{\ell_E\}$ has frames $[F_{0,\ell}, \dots, F_{k,\ell}]$.
The algorithm repeats the following phases:

1. Next Level

Initialize for each $\ell \in L \setminus \{\ell_E\}$ a new frame $k + 1$ as **true**.

For each location $\ell \in L$ where $(\ell, t, \ell_E) \in G$ generate an initial proof-obligation (t, ℓ, k) .

2. Blocking-Phase

If there are proof-obligations:

Take proof-obligation (t, ℓ, i) with the lowest i and check for each predecessor location ℓ_{pre} if the formula:

$$F_{i-1, \ell_{pre}} \wedge T_{\ell_{pre} \rightarrow \ell} \wedge t'$$

is satisfiable.

- If it is *satisfiable*, it means that t could not be blocked at ℓ on level i , generate a new proof-obligation $(p, \ell_{pre}, i - 1)$ where p is the weakest precondition of t with regard to $T_{\ell_{pre} \rightarrow \ell}$.
- If the formula is *unsatisfiable*, strengthen each frame $F_{j, \ell}$, $j \leq i$ with \bar{t} , meaning $F_{j, \ell} = F_{j, \ell} \wedge \bar{t}$, blocking t at ℓ on level i .

This continues recursively until either a proof-obligation $(d, \ell, 0)$ is generated, proving that there exists a feasible path to ℓ_E terminating the algorithm, or there is no unblocked proof-obligation left.

3. Propagation-Phase

Check the frames if there exists a global fixpoint position i where

$$F_{i-1, \ell} = F_{i, \ell}$$

for every location $\ell \in L \setminus \{\ell_E\}$.

If there is such an i the algorithm terminates returning that ℓ_E is not reachable.

To illustrate the lifted algorithm further consider the updated pseudo-code:

```

1: procedure LIFTED-PDR-PROVE( $L, G$ )
2:   check for 0-counter-example
3:    $\ell_0.trace.push(new\ frame(true))$ 
4:   for all  $\ell \in L \setminus \{\ell_0, \ell_E\}$  do
5:      $\ell.trace.push(new\ frame(false))$ 
6:    $level := 0$ 

7:   loop
8:     for all  $\ell \in L \setminus \{\ell_E\}$  do
9:        $\ell.trace.push(new\ frame(true))$ 
10:     $level := level + 1$ 
11:    get initial proof-obligations

12:    while  $\exists$  proof-obligation  $(t, \ell, i)$ , do
13:      Recursively block proof-obligation
14:      if a proof-obligation( $p, \ell, 0$ ) is generated then
15:        return false

16:    for  $i = 0; i \leq level; i := i + 1$  do
17:      for  $\ell \in L \setminus \{\ell_E\}$  do
18:        if  $\ell.trace[i] \neq \ell.trace[i - 1]$  then
19:          break
20:    return true

```

Figure 3.2: Lifted PDR Algorithm.

3.3 Example

ToDo Intro text

3.3.1 Case 1: Error State is Reachable

To show an application of the lifted algorithm reconsider the example in Figure 4.1, we have CFG $\mathcal{A} = (X, L, G, \ell_0, \ell_E)$, where

- $X = \{x\}$
- $L = \{\ell_0, \ell_1, \ell_2, \ell_3, \ell_E\}$
- $G = \{(\ell_0, x := 0, \ell_1), (\ell_1, x := x + 1, \ell_2), (\ell_2, x = 1, \ell_E), (\ell_2, x \neq 1, \ell_3)\}$

We now want to verify whether ℓ_E is reachable using the lifted algorithm:

1. Step: Check for 0-Counter-Example

Is $\ell_0 = \ell_E$?

No, we continue with initializing level 0 by adding to each $\ell \in L \setminus \{\ell_0, \ell_E\}$ a new frame $F_{0,\ell} = false$, for ℓ_0 adding $F_{0,\ell_0} = true$:

	level
location	0
ℓ_0	<i>true</i>
ℓ_1	<i>false</i>
ℓ_2	<i>false</i>
ℓ_3	<i>false</i>

2. Step: Next Level

We initialize new frames for level 1 as **true**:

	level	
location	0	1
ℓ_0	<i>true</i>	<i>true</i>
ℓ_1	<i>false</i>	<i>true</i>
ℓ_2	<i>false</i>	<i>true</i>
ℓ_3	<i>false</i>	<i>true</i>

To generate the initial proof-obligations, we check G and take the transitions where ℓ_E is the target.

We see, there is one transition $(\ell_2, x = 1, \ell_E)$, that means we have to block $x = 1$ at ℓ_2 on level 1, resulting in the initial proof-obligation $(x = 1, \ell_2, 1)$.

3. Step: First Blocking Phase

We need to block the initial proof-obligation $(x = 1, \ell_2, 1)$. Let ℓ_{pre} be a predecessor of

ℓ_2 , we need to check the formula:

$$F_{0,\ell_{pre}} \wedge T_{\ell_{pre} \rightarrow \ell_2} \wedge x' = 1$$

for satisfiability. As there is only one predecessor ℓ_1 we test:

$$\underbrace{false}_{F_{0,\ell_1}} \wedge \underbrace{x' := x + 1}_{T_{\ell_1 \rightarrow \ell_2}} \wedge x' = 1$$

Which is unsatisfiable

We add $\overline{(x = 1)} \equiv x \neq 1$ to F_{0,ℓ_2} and F_{1,ℓ_2} , blocking $x = 1$ at ℓ_2 on level 1.

location	level	
	0	1
ℓ_0	<i>true</i>	<i>true</i>
ℓ_1	<i>false</i>	<i>true</i>
ℓ_2	<i>false</i> $\wedge x \neq 1$	<i>true</i> $\wedge x \neq 1$
ℓ_3	<i>false</i>	<i>true</i>

Because there are no proof-obligations left we continue with the propagation-phase.

4. Step: First Propagation-Phase

Check if there exists a global fixpoint position i where

$$F_{i-1,\ell} = F_{i,\ell}$$

for every location $\ell \in L \setminus \{l_E\}$.

We see there is no such i , we continue with the next level.

5. Step: Next Level

We initialize new frames for level 2 as **true**:

location	level		
	0	1	2
ℓ_0	<i>true</i>	<i>true</i>	<i>true</i>
ℓ_1	<i>false</i>	<i>true</i>	<i>true</i>
ℓ_2	<i>false</i> $\wedge x \neq 1$	<i>true</i> $\wedge x \neq 1$	<i>true</i>
ℓ_3	<i>false</i>	<i>true</i>	<i>true</i>

Again generate the initial proof-obligation which is the same as before but on level 2 now, we have initial proof-obligation $(x = 1, \ell_2, 2)$

6. Step: Second-Blocking Phase

We need to block the proof-obligation $(x = 1, \ell_2, 2)$ by testing

$$\underbrace{true}_{F_{1,\ell_1}} \wedge \underbrace{x' := x + 1 \wedge x' = 1}_{T_{\ell_1 \rightarrow \ell_2}} = 1$$

for satisfiability. Which is satisfiable with $p = (x = 0)$. Because p being also the weakest precondition, we generate a new proof-obligation $(p, \ell_1, 1)$, meaning we need to block p at location ℓ_1 on level 1.

Take the new proof-obligation $(x = 0, \ell_1, 1)$ and check

$$\underbrace{true}_{F_{0,\ell_0}} \wedge \underbrace{x' := 0}_{T_{\ell_0 \rightarrow \ell_1}} \wedge \underbrace{x' = 0}_{p'}$$

for satisfiability.

Which is valid, with **true** being the weakest precondition, we generate the new proof-obligation $(true, \ell_0, 0)$ and because this obligation is on level 0 we terminate, stating that ℓ_E is reachable by the counter-example trace:

$$\ell_0 \rightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_E$$

3.3.2 Case 2: Error State is Unreachable

To show a CFG with an unreachable error state consider $\mathcal{B} = (X, L, G, \ell_0, \ell_E)$, where

- $X = \{x, y\}$
- $L = \{\ell_0, \ell_1, \ell_2, \ell_E\}$
- $G = \{(\ell_0, x' := 0 \wedge y' := x', \ell_1), (\ell_1, x' := x + 1 \wedge y' := y + 1, \ell_1), (\ell_1, x = y, \ell_2), (\ell_1, x \neq y, \ell_E)\}$

with graph:

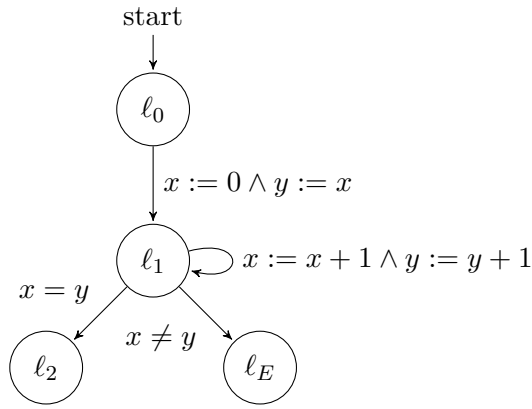


Figure 3.3: Graph of \mathcal{B} .

We now want to check whether ℓ_E is reachable, using the lifted algorithm:

1. Step: Check for 0-Counter-Example

Is $\ell_0 = \ell_E$?

No, we continue with initializing level 0 by adding to each $\ell \in L \setminus \{\ell_0, \ell_E\}$ a new frame $F_{0,\ell} = false$, for ℓ_0 adding $F_{0,\ell_0} = true$.

	level
location	0
ℓ_0	<i>true</i>
ℓ_1	<i>false</i>
ℓ_2	<i>false</i>

2. Step: Next Level

We initialize new frames for level 1 as **true**:

location	level	
	0	1
ℓ_0	<i>true</i>	<i>true</i>
ℓ_1	<i>false</i>	<i>true</i>
ℓ_2	<i>false</i>	<i>true</i>

We see there is only one transition leading to ℓ_E , $(\ell_1, x \neq y, \ell_E)$. We get the initial proof-obligation $(x \neq y, \ell_1, 1)$.

3. Step: First Blocking Phase

To block the initial proof-obligation $(x \neq y, \ell_1, 1)$ we check each predecessor of ℓ_1 :

- predecessor: ℓ_0

$$\underbrace{true}_{F_{0,\ell_0}} \wedge \underbrace{x' := 0 \wedge y' := x' \wedge x' \neq y'}_{T_{\ell_0 \rightarrow \ell_1}}$$

Which is unsatisfiable, we add $\overline{(x \neq y)} \equiv x = y$ to F_{0,ℓ_1} and F_{1,ℓ_1} :

location	level	
	0	1
ℓ_0	<i>true</i>	<i>true</i>
ℓ_1	<i>false</i> \wedge $x = y$	<i>true</i> \wedge $x = y$
ℓ_2	<i>false</i>	<i>true</i>

- predecessor: ℓ_1

$$\underbrace{false \wedge x = y}_{F_{0,\ell_1}} \wedge \underbrace{x' := x + 1 \wedge y' := y + 1' \wedge x' \neq y'}_{T_{\ell_1 \rightarrow \ell_1}}$$

Which is unsatisfiable as well, but because $x = y$ has already been added to F_{0,ℓ_1} and F_{1,ℓ_1} we move on.

As there are no proof-obligations left, we continue with the first propagation-phase.

4. Step: First Propagation-Phase

Check if there exists a global fixpoint position i where

$$F_{i-1,\ell} = F_{i,\ell}$$

for every location $\ell \in L \setminus \{l_E\}$.

We see there is no such i , we continue with the next level.

5. Step: Next Level

We initialize new frames for level 2 as **true**:

location	level		
	0	1	2
ℓ_0	<i>true</i>	<i>true</i>	<i>true</i>
ℓ_1	$false \wedge x = y$	$true \wedge x = y$	<i>true</i>
ℓ_2	<i>false</i>	<i>true</i>	<i>true</i>

Again we generate the initial proof-obligation which is the same as before but on level 2 now, we have the initial proof-obligation $(x \neq y, \ell_1, 2)$.

6. Step: Second Blocking Phase

To block proof-obligation $(x \neq y, \ell_1, 2)$ we check the predecessors of ℓ_1 :

- predecessor: ℓ_0

$$\underbrace{true \wedge x' := 0}_{F_{1,\ell_0}} \wedge \underbrace{y' := x' \wedge x' \neq y'}_{T_{\ell_0 \rightarrow \ell_1}}$$

Which is unsatisfiable, we add $\overline{(x \neq y)} \equiv x = y$ to F_{0,ℓ_1} , F_{1,ℓ_1} and F_{2,ℓ_1} :

location	level		
	0	1	2
ℓ_0	<i>true</i>	<i>true</i>	<i>true</i>
ℓ_1	$false \wedge x = y$	$true \wedge x = y$	$true \wedge x = y$
ℓ_2	<i>false</i>	<i>true</i>	<i>true</i>

- predecessor: ℓ_1

$$\underbrace{true \wedge x = y \wedge x' := x + 1}_{F_{1,\ell_1}} \wedge \underbrace{y' := y + 1 \wedge x' \neq y'}_{T_{\ell_1 \rightarrow \ell_1}}$$

Which is unsatisfiable as well, but because $x = y$ has already been added to F_{0,ℓ_1} , F_{1,ℓ_1} , and F_{2,ℓ_1} we move on.

As there are no proof-obligations left, we continue with the second propagation-phase

7. Step: Second Propagation-Phase

location	level		
	0	1	2
ℓ_0	<i>true</i>	<i>true</i>	<i>true</i>
ℓ_1	$false \wedge x = y$	$true \wedge x = y$	$true \wedge x = y$
ℓ_2	<i>false</i>	<i>true</i>	<i>true</i>

$\underbrace{\hspace{15em}}_{\text{global fixpoint}}$

We see that level 1 equals level 2 on all locations, with that we found global fixpoint position $i = 2$, the formulas at that position are the inductive invariants proving that ℓ_E is not reachable.

3.4 Possible Improvements

As shown before, lifting PDR from bit-level to control flow graphs is possible. The problem of large, time consuming queries to the solver remain however. Is it possible to lift the improvements of the bit-level algorithm too?

Ternary Simulation cannot be used on first-order formulae making it impossible to use it to reduce lifted proof-obligations.

There are possible generalization techniques:

3.4.1 Syntactical Analysis

It is possible to remove a cube c from a proof-obligation if no variable in c is assigned in T , and either c is already contained in a frame or there is an **assume** c statement in T . Making the obligation more general.

3.4.2 Weakest Precondition

The definition of the lifted algorithm above already contains an improvement, using weakest preconditions to find predecessors. Instead of generating a single proof-obligation for every predecessor state and wasting time blocking each of them, calculate the weakest precondition which is a safe overapproximation of all predecessor states.

3.4.3 Disjunctive Normal Form

It is possible to transform cubes into clauses by negating them. When there is a large proof-obligation we can use this to generate the disjunctive normal form and split the large obligation into smaller ones by taking each cube in the disjunctive normal form as a new proof-obligation, saving time because smaller queries are solved faster.

3.4.4 Interpolation

Let (A, B) be a pair of formulae such that $A \wedge B$ is unsatisfiable. An interpolant I for (A, B) is a formula fulfilling the following characteristics:

- $A \Rightarrow I$
- $I \wedge B$ is unsatisfiable
- I consists only of variables found in $A \cap B$

still **ToDo**

4 PDR in Ultimate

In this chapter we introduce the program analysis framework `ULTIMATE`, and its tool `ULTIMATE AUTOMIZER`, we will show how we integrated PDR, and detail which improvements were considered.

4.1 Introduction Ultimate

`ULTIMATE` is a program analysis framework, based on *plugins* that can be executed one after another to form *toolchains* which can perform various tasks. A big advantage of this modularity is that it is relatively easy to implement new toolchains as a lot of plugins can be reused creating much less overhead. There are five types of plugins:

- Source plugins define a file-to-model transformation
- Analyzer plugins take a model as input and modifies it
- Generator plugins have a similar functionality as Analyzer plugins but they can additionally produce new models
- Output plugins do not produce or modify anything, they write models into files
- The last plugin cannot be used in toolchains per say, they act more like a library providing additional functionality to other plugins

4.2 Implementation

To implement PDR in `ULTIMATE` we chose to implement a new library plugin *Library-PDR*, that will be executed as part of `ULTIMATE AUTOMIZER` [8].

`ULTIMATE AUTOMIZER`, `AUTOMIZER` in the following, is a software verifier that is capable of checking safety and liveness properties. It uses an automata-based [7] instance of the CEGAR scheme, that works like this:

In each new iteration choose a sequence of statements, called trace, that start at the

initial location and lead to an error location. Check whether the trace is feasible or infeasible.

- If the trace is *feasible* then the program is proven unsafe, meaning there exists a possible program execution leading to an error location.
- If the trace is *infeasible*, calculate a proof for the infeasibility by computing a sequence of predicates along the trace, called sequence of interpolants. Then, in a following refinement step, try to eliminate every other trace that can be proven infeasible using that sequence of interpolants, these traces are then being subtracted from the set of potentially error traces.

Now, to check a trace for feasibility we use our newly integrated PDR-Library. The PDR-Library is a path program focused technique, meaning that we do not consider the trace alone but also the context of the analyzed program.

The sequence of interpolants are the entries in the frames on the global fixpoint.

4.3 Improvements

As we have seen in section 4.4 there are a plethora of possible ways to make PDR more efficient. In this section we present implemented improvements to our approach.

4.3.1 Caching of Proof-Obligation Queue

We always start each new level with the initial proof-obligation, and generate a chain of proof-obligations, until the most recent one is blocked, we then recursively block all predecessor obligations, adding new information to the frames. This chain of obligations does not change, once a new obligation is generated it needs to be blocked in each successive level.

Now as a result of the backwards search nature of PDR, each new level has to create that chain of proof-obligations from the beginning again. That results in a lot of overhead because we have to calculate the same chain of obligations times and times again.

Our idea was to cache the proof-obligation queue and always start a new level with the latest generated one, so we save the process of generating already known obligations.

For that we introduced a *global level* variable serving as the level counter, and changed the levels of each proof-obligation to be *subtracted* from the global level, resulting in a relative level = $global\ level - local\ level$, with that every proof-obligation is at its corresponding level of generation.

For example, the initial proof-obligation has its local level 0, so that *relative level* = *global level* - 0 = *global level*, because the initial obligation always is on the global level, take an obligation that we generate from the initial one, it has local level 1 so that *relative level* = *global level* - 1 means that this proof-obligation comes immediately after the initial one.

Something that cannot be saved is blocking of the previous proof-obligation chain, we can only ignore the chain of proof-obligations leading to the newest one, the chain of blocking the ones before up to the initial obligation is necessary, because otherwise the frames are not updated properly. We however found a way to shorten this as well, as described in the next point.

To illustrate this technique further, consider the following graphic:

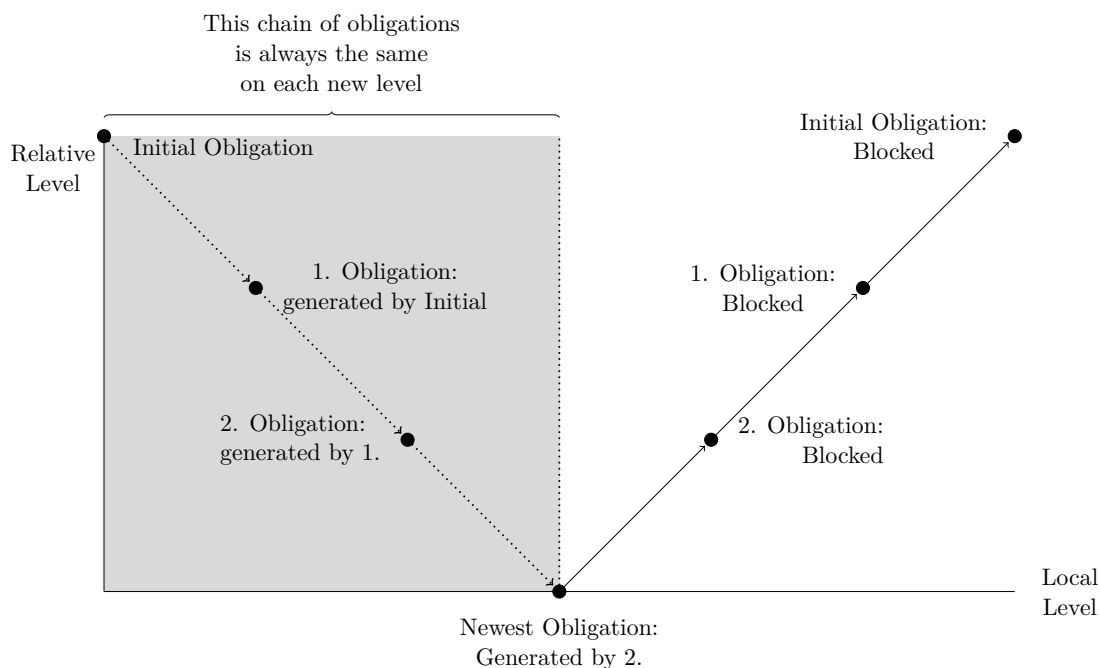


Figure 4.1: Chain of Proof-Obligation Generation.

4.3.2 Ignoring Already Blocked Proof-Obligations

Besides changing proof-obligations's levels, we also expanded them with a list of already seen and blocked SMT-solver queries.

Before we consider a new call to the SMT-solver we first check the given query whether it has already been proven unsatisfiable, if so, we consider the obligation blocked and add the negated formula to the frames, if we prove an unknown query as unsatisfiable

we add them to the list of blocked queries.

With this memoization we save unnecessary calls to the SMT-solver and with that, time on programs with a lot of identical proof-obligations, in loops for example.

4.3.3 Using Preconditions

As discussed in section 3.4, using weakest preconditions is a really useful addition to the way proof-obligations are generated.

We implemented the usage of weakest preconditions, but were met with problems regarding **assume**-statements:

Let φ and ψ be first-order formulae and wp be the weakest precondition function, we have:

$$wp(\varphi, \mathbf{assume} \ \psi) = (\psi \Rightarrow \varphi) \quad (4.1)$$

To illustrate this issue, consider the following CFG:

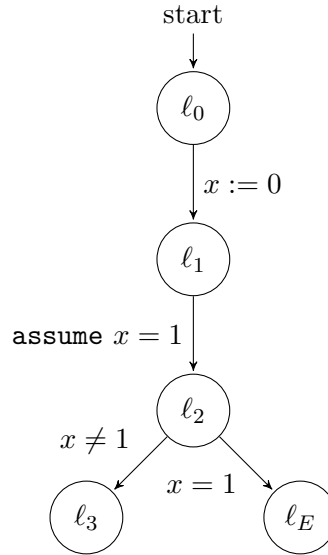


Figure 4.2: Graph of \mathcal{B} .

We now use PDR to prove if ℓ_E is reachable.

We omit the first iteration of the algorithm and start on level 2 with the initial proof-obligation $(x = 0, \ell_2, 2)$, we check the following formula for satisfiability:

$$true \wedge \mathbf{assume} \ x = 1 \wedge x' = 1$$

Which is satisfiable, we calculate

$$wp(x' = 1, \mathbf{assume} \ x = 1) = ((x = 1) \Rightarrow (x' = 1)) \quad (4.2)$$

Because **assume** does not change x , it becomes apparent that $x' = x$, resulting in

$$((x = 1) \Rightarrow (x = 1)) = \mathit{true}$$

We get the new proof-obligation $(\mathit{true}, \ell_1, 1)$.

To block $(\mathit{true}, \ell_1, 1)$, we check the following formula for satisfiability:

$$\mathit{true} \wedge x' := 0 \wedge \mathit{true}$$

which is satisfiable, we would create a proof-obligation on level 0 and with that would terminate, resulting in the wrong outcome that ℓ_E is reachable.

To fix this problem we do not use the weakest precondition but just the precondition, which is characterized like this:

Let φ and ψ be first-order formulae and pre be the precondition function, we have:

$$pre(\varphi, \mathbf{assume} \ \psi) = (\varphi \wedge \psi)$$

Now reconsider formula (4.2), to get the new proof-obligation we use the pre function:

$$pre(x' = 1, \mathbf{assume} \ x = 1) = ((x = 1) \wedge (x' = 1))$$

Because $x = x'$ this can be simplified to $x = 1$. We get the new proof-obligation $(x = 1, \ell_1, 1)$. To block $(x = 1, \ell_1, 1)$, we check the following formula for satisfiability:

$$\mathit{true} \wedge x' := 0 \wedge x' = 1$$

which is unsatisfiable, and with that we have solved the issue and can continue working on the program.

5 Evaluation

In this chapter we evaluate our implementation and compare it to `ULTIMATE`'s Nested Interpolants.

We benchmarked PDR with two different SMT-solvers:

- SMTInterpol [2]
- Z3 [4]

The difference between those is that SMTInterpol interpolates better and is integrated into `ULTIMATE`, but on the other hand cannot deal with quantors and non-linear queries. We however focus mainly on SMTInterpol.

5.1 Comparison

We used a testing set containing 250 programs.

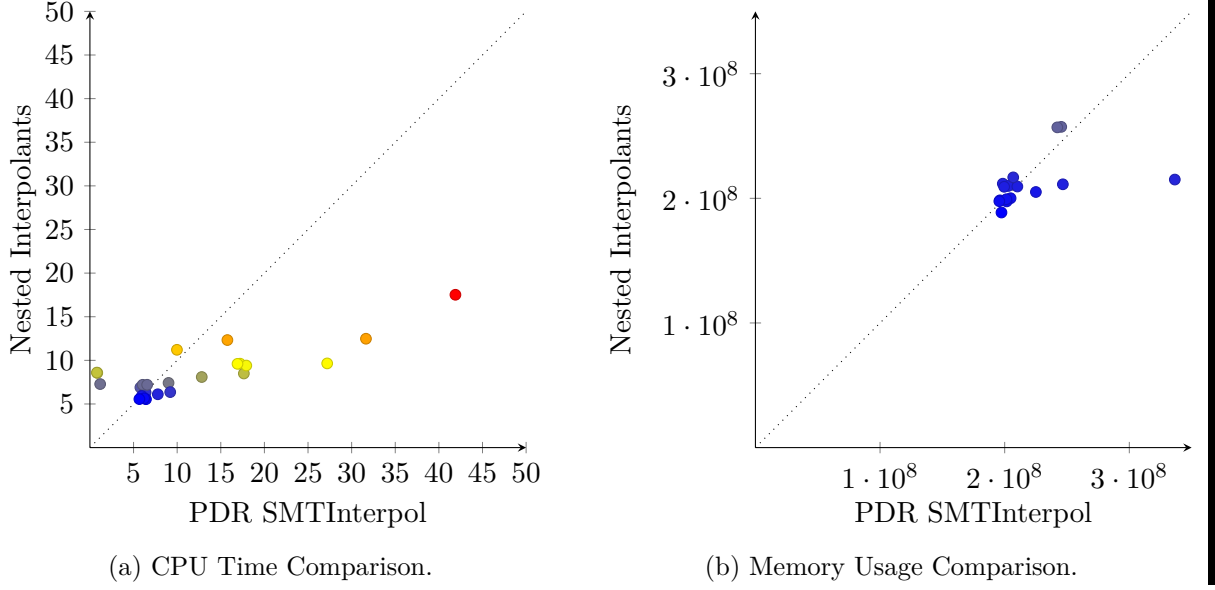


Figure 5.1: Statistics collected from successful benchmarks.

Technique	tests solved	solve time
Nested Interpolants	165/250	3508s
PDR SMTInterpol	35/250	496s
PDR Z3	62/250	1322s

Figure 5.2: Evaluation Results.

5.2 Discussion

In Figure 5.1 we can see that the most benchmark results were equally fast for both techniques, there are however some programs where PDR was significantly slower than the Nested Interpolants. This stems from loops, because PDR does not yet use interpolants, we have trouble solving programs that contain loops, as we have to disprove every single loop iteration, making verification very time consuming. Both techniques were more or less on par regarding memory consumption.

In Figure 5.2 we see that PDR generally only solved $\frac{1}{7}$ of the benchmarks using SMT-

Interpol and $\frac{1}{4}$ using Z3. This is because our benchmarkset contained programs with procedures, a procedure is a call to another function that is not the main program. Our PDR implementation can not yet deal with such programs, and because of that cannot deal with a lot of the benchmarks.

The average time spent per test is promising:

$$\text{Nested Interpolants: } 3508s/165\text{tests} = 21s/test$$

$$\text{PDR SMTInterpol: } 496s/35\text{tests} = 14s/test$$

6 Related Work

Closely related to our work is the approach by Lange et al. [11] who propose a true lifting of IC3 from hardware to software by fully exploiting the control flow of a given program, meaning the control flow graph is not altered in any way. They extend IC3 in the following ways. Instead of using a SAT-solver they use a SMT-solver. Additional information is no longer being used globally over all states, but every program location has its own local information, and if there exists an error location each iteration of the algorithm will have to prove that it is unreachable.

The first approach of using PDR on software is rather straight-forward, Welp et al. propose a way to encode variables as bitvectors, introduce a new variable pc representing the program location, and then using the unmodified IC3-algorithm on that encoding [12]. A big drawback of that approach is the tedious handling of the pc variable rendering it not very competitive.

In 2012 the first ever approach of using PDR on first-order formulae was devised, Cimatti et al. [6] propose exploiting the partitioning of a program's state space by unwinding the program's control flow graph into an Abstract Reachability Tree (ART). An ART \mathcal{A} for a given program is a tree over (V, E) , such that V is a set of triples (pc, φ) , where pc is a program location, and φ is a first-order formula, and E is a set of transitions (pc_i, ψ, pc_j) .

The root node is defined as $(pc_{init}, true)$.

For every non-leaf node (pc_i, φ) holds $\varphi \wedge T_{pc_i \rightarrow pc_j} \models \psi'$, where (pc_j, ψ) is a child node, and $T_{pc_i \rightarrow pc_j}$ is the transition formula.

The algorithm first computes a possible path to an error location pc_E (pc_{init}, \dots, pc_E) then constructs a trace $F = [true, \dots, \varphi_i, \dots, \varphi_{n-1}]$, by taking the coupled formula of each location.

ToDo proofobligaitons.

Another possible way is proposed by Hoder and Bjørner [10] operating on an abstract transition system derived from the program. A non-linear variant of PDR is used, so that counterexamples unfold into trees that can be recursively generalized until either proven or disproven.

7 Future Work

7.1 Further Improvements

We have implemented some improvement methods, but there are still more possible ways to make our PDR algorithm more efficient.

7.1.1 Interpolation

ULTIMATE already supports ways of computing an interpolant for two given formulae. The idea is, that everytime a query to the SMT-solver is unsatisfiable we, instead of adding the negated proof-obligation to the frames, add a calculated interpolant for the query.

Let ℓ and ℓ_{pre} be two locations where ℓ_{pre} is a predecessor of ℓ , $F_{i-1, \ell_{pre}}$ be a frame, $T_{\ell_{pre} \rightarrow \ell}$ be the transition from ℓ_{pre} to ℓ , and t' be a primed formula.

To get an interpolant I for that query we first need to define A and B , we do that by dividing the query the following way:

$$\underbrace{F_{i-1, \ell_{pre}} \wedge T_{\ell_{pre} \rightarrow \ell}}_A \wedge \underbrace{t'}_B$$

Because we have proven the query unsatisfiable generating an interpolant is possible. With this technique our frames contain only interpolants

7.1.2 Procedures

ULTIMATE aims to verify C programs, most of which contain procedure calls that cannot be handled yet.

Procedures are presented like this:

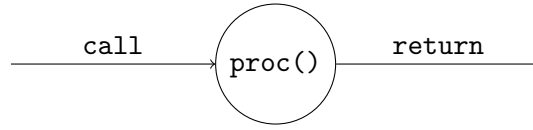


Figure 7.1: A Procedure.

There is a `call`-Transition leading to the procedure, then the procedure `proc()` itself, and finally a `return`-transition leading back to the main program.

Because of PDR's backwards-search nature, we first find the `return` of a procedure, without knowing the effects of the corresponding procedure on variables, and the prior state of the program before the execution, this can lead to the creation of wrong proof-obligations, leading to unsound results.

A solution to that problem is, instead of using a linear approach, we modify our algorithm to be nonlinear and deal with procedures accordingly [10] or to calculate a procedure summary and attach that to the CFG, removing the procedure altogether.

8 Conclusion

Bibliography

- [1] Hwmcc10 results. <https://fmv.jku.at/hwmcc10/results.html>. Accessed: 2018-07-20.
- [2] Smtinterpol. <https://ultimate.informatik.uni-freiburg.de/smtinterpol/>. Accessed: 2018-07-28.
- [3] ULTIMATE. <https://ultimate.informatik.uni-freiburg.de>. Accessed: 2018-07-20.
- [4] Z3. <https://github.com/Z3Prover/z3>. Accessed: 2018-07-28.
- [5] Aaron R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.
- [6] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012.
- [7] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2013.
- [8] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *Proceedings of the 25th International Conference on Computer Aided Verification, CAV’13*, pages 36–52. Springer, 2013.
- [9] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.
- [10] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.
- [11] Tim Lange, Martin R. Neuhäuser, and Thomas Noll. IC3 software model checking on control flow automata. In *FMCAD*, pages 97–104. IEEE, 2015.
- [12] Tobias Welp and Andreas Kuehlmann. QF BV model checking with property directed reachability. In *DATE*, pages 791–796. EDA Consortium San Jose, CA, USA / ACM DL, 2013.