Bachelor Thesis

Property Directed Reachability
In Ultimate

Jonas Werner

Examiner: Prof. Dr.

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.

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

foo bar

Zusammenfassung

foo bar aber auf deutsch

Contents

1 Introduction

SAT-based model-checking is a useful technique for both software and hardware verification. Most modern model-checkers are based on interpolation [1]. Recently a novel hardware verification algorithm was devised by Aaron Bradley [2] 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 [3].
The model-checking method behind `IC3` is called *Property Directed Reachability*, *PDR* for short, which is not based on interpolation but on backward-search. It tries to find inductive invariants by constructing an over-approximation of reachable states. During this construction, possible counter-examples are disproved 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 [4] is a software analysis framework consisting of multiple plugins that perform steps of a program analysis, like parsing source code, trans- forming programs from one representation to another, or analyse programs. ULTIMATE already has analysis-plugins using different model-checking techniques like trace abstraction [9] or lazy interpolation [10]. The goal of this Bachelor's Thesis is to implement a new analysis-plugin that uses PDR on software in ULTIMATE and to compare it with the other existing techniques.

2 Related Work

Because hardware verification is limited to propositional logic, we need techniques to lift PDR from bit level to first-order logic formulas used in software, for that there are two other important approaches:

The first ever approach of using PDR on first-order formulas came in 2012 by Cimatti and Griggio in [6] who proposed exploiting the partitioning of a program's state space, by unwinding the program's control flow graph into an Abstract Reachability Tree where each node is coupled with a location and a first-order formula, resulting in a so called explicit-symbolic approach. Possible counterexample traces are disproved by computing under-approximations of predecessors.

Another possible way is proposed by Hoder and Bjørner in [7] 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 disproved.

3 PDR Background

In the following I will describe the basic principle behind PDR as a hardware-checker as used in `IC3`, therefore we use only boolean variables.

3.1 Preliminaries

First some preliminary definitions and notations:

A *literal* is a variable or its negation, e.g., x or $\bar{y}$
A *clause* is a disjunction of literals, e.g., $x \vee \bar{y}$
A *cube* is a conjunction of literals, e.g., $x \wedge \bar{y}$
Therefore, the negation of a cube is a clause. $\overline{(x \wedge \bar{y})} \equiv (\bar{x} \vee y)$

A *boolean transition system* is a tuple $S = (X, I, T)$ where X is a finite set of boolean variables, I is a cube 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 transition graph:

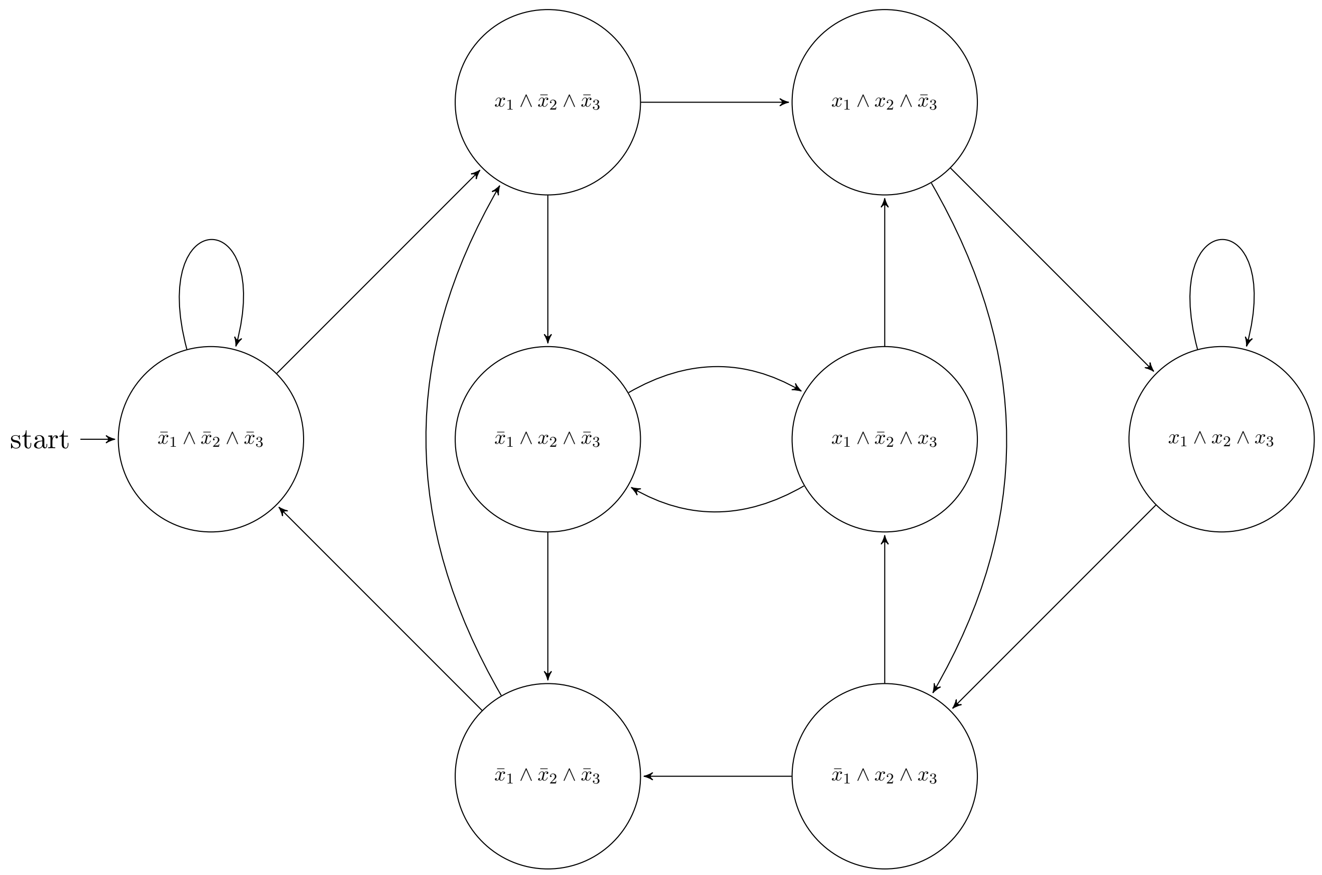

start
$\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3$
$x_1 \wedge \bar{x}_2 \wedge \bar{x}_3$
$x_1 \wedge x_2 \wedge \bar{x}_3$
$\bar{x}_1 \wedge x_2 \wedge \bar{x}_3$
$x_1 \wedge \bar{x}_2 \wedge x_3$
$x_1 \wedge x_2 \wedge x_3$
$\bar{x}_1 \wedge \bar{x}_2 \wedge x_3$
$\bar{x}_1 \wedge x_2 \wedge x_3$

Figure 3.1: Transition Graph of U

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 set of *good states*, conversely $\bar{P}$ represent a 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.
We can use PDR 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.

3.2 Algorithm

A PDR-based 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 $[F_0, ..., F_k]$, called a *trace*, it is organized 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) $F_i \Rightarrow P$

Now to the algorithm itself:
Start 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, initialize the first frame $F_0 = I$, fulfilling (I), and moving on.

Let $[F_0, F_1, ..., F_k]$ be the current trace.
The algorithm repeats the following three phases until termination:

1. Next Transition
Check whether the next state is a good state meaning $F_k \wedge T \Rightarrow P'$ is valid, by testing the satisfiability of $F_k \wedge T \wedge \bar{P}'$

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

2. Blocking-Phase
If there are proof-obligations:
Take proof-obligation (b, i) and try to block the bad state b by checking if frame F_{i-1} can reach b in one transition, i.e., test $F_{i-1} \land T \land b'$ for satisfiability.

- If the formula is *satisfiable*, it means that F_i is not strong enough to block b. For each satisfying assignment $\vec{x} = (x_1, x_2, ..., x_{|X|}, x'_1, x'_2, ..., x'_{|X'|})$ get a new bad state $c = x_1 \land x_2 \land ... \land x_{|X|}$ creating the new proof-obligation $(c, i-1)$.
- If the formula is *unsatisfiable*, strengthen frame F_i with $\bar{b}$ meaning $F_i = F_i \land \bar{b}$, 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
Add a new frame $F_{k+1} = P$ and propagate clauses from F_k forward, meaning for all clauses c in F_k check $F_k \land T \land \bar{c}'$ for satisfiability. If that formula is unsatisfiable, strengthen F_{k+1} with c: $F_{k+1} = F_{k+1} \land c$, else do nothing and continue 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 fixpoint and terminates returning that P always holds with F_k being the inductive invariant.

To illustrate the procedure further consider the pseudo-code:

Algorithm 1 PDR-prove

```
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 $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT then
13:                $F_{k+1} = F_{k+1} \wedge c$
14:        if $trace.last() == F_{k+1}$ then
15:            return true
16:        $trace.push(F_{k+1})$
```

3.3 Examples

3.3.1 With Failing Property

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

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

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:

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

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 it is not because $\bar{x}_1 \wedge (x_1 \vee \bar{x}_2') \wedge x_2'$ is unsatisfiable. We do not generate a proof-obligation so we can skip the blocking-phase and continue on with the propagation-phase.

3. Step: First Propagation-Phase
Initialize $F_1 = P$
Check each clause c in F_0 for $F_0 \wedge T \wedge \bar{c}'$ to strengthen F_1.

- $c = \bar{x}_1$

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

  Satisfiable with $(\bar{x}_1, \bar{x}_2, \bar{x}_3, x_1', \bar{x}_2', \bar{x}_3')$
  $\rightarrow$ Do not add $\bar{x}_1$ to F_1.

- $c = \bar{x}_2$

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

  Unsatisfiable because $\bar{x}_1 \wedge (x_1 \vee \bar{x}_2') \wedge x_2'$ is not satisfiable
  $\rightarrow$ Add $\bar{x}_2$ to F_1
  $\rightarrow F_1 = P \wedge \bar{x}_2$.

- $c = \bar{x}_3$

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

  Unsatisfiable because $\bar{x}_2 \wedge (x_2 \vee \bar{x}'_3) \wedge x'_3$ is not satisfiable
  $\rightarrow$ Add $\bar{x}_3$ to F_1
  $\rightarrow F_1 = P \wedge \bar{x}_2 \wedge \bar{x}_3$

With that the first propagation-phase is done resulting in

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

and because $F_1 \not\equiv F_0$ 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 $F_2 = P$
Check each clause c in F_1 for $F_1 \wedge T \wedge \bar{c}'$ to strengthen F_2. We skip P, as it is already part of F_2.

This works exactly as in the 3. step:

- $c = \bar{x}_2$

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

  Satisfiable with $(x_1, \bar{x}_2, \bar{x}_3, x'_1, x'_2, \bar{x}'_3)$
  $\rightarrow$ Do not add $\bar{x}_2$ to F_2

- $c = \bar{x}_3$

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

  Unsatisfiable because $\bar{x}_2 \wedge (x_2 \vee \bar{x}'_3) \wedge x'_3$ is not satisfiable.
  $\rightarrow$ Add $\bar{x}_3$ to F_2
  $\rightarrow F_2 = P \wedge \bar{x}_3$

That concludes the second propagation-phase resulting in

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

and because $F_2 \not\equiv F_1$ we continue.

6. Step: Third Transition Step
Check $F_2 \wedge T \Rightarrow \bar{P}'$ 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 $(\underbrace{x_1, \bar{x}_2, \bar{x}_3}_{\text{q}}, x'_1, x'_2, \bar{x}'_3)$, we get the bad state $q = x_1 \wedge \bar{x}_2 \wedge \bar{x}_3$ and generate a new proof-obligation $(q, 1)$.

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 $(\underbrace{\bar{x}_1, \bar{x}_2, \bar{x}_3}_{\mathrm{I}}, x'_1, \bar{x}'_2, \bar{x}'_3)$, 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:

$$\underbrace{\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_{I} \rightarrow \underbrace{x_1 \wedge \bar{x}_2 \wedge \bar{x}_3}_{q} \rightarrow \underbrace{x_1 \wedge x_2 \wedge \bar{x}_3}_{s} \rightarrow \underbrace{x_1 \wedge x_2 \wedge x_3}_{\bar{P}}$$

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

3.3.2 With Passing Property

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 transition graph:

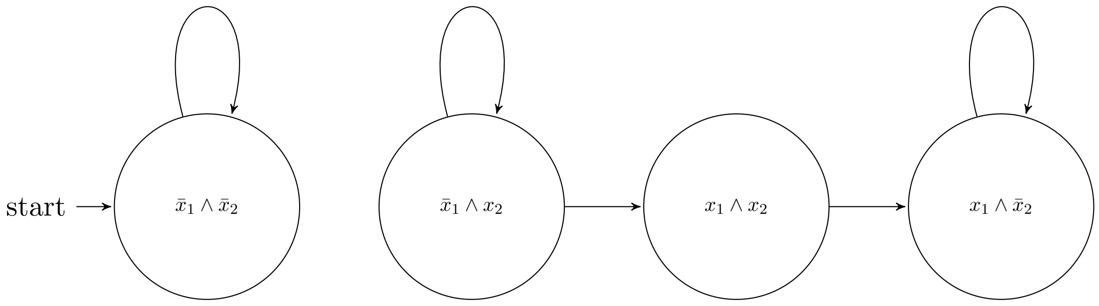


Figure 3.2: Transition Graph of B

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
Check for 0-counter-example to verify $I \Rightarrow P$ by testing $I \wedge \bar{P}$ for satisfiability:

$$\bar{x}_1 \wedge \bar{x}_2 \wedge x_1 \wedge \bar{x}_2$$

The formula is unsatisfiable because $\bar{x}_1 \wedge x_1$ that means there is no 0-counter-example.

2. Step: First Transition
Initialize $F_0 = I$ and check if $F_0 \wedge T \Rightarrow P'$ by testing $F_0 \wedge T \wedge \bar{P}'$ for satisfiability:

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

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

- $c = \bar{x}_1$

$$\bar{x}_1 \wedge \bar{x}_2 \wedge T \wedge x_1'$$

  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.
  $\rightarrow$ Add $\bar{x}_1$ to F_1
  $\rightarrow F_1 = P \wedge \bar{x}_1$

- $c = \bar{x}_2$

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

  Unsatisfiable because $\bar{x}_1 \wedge \bar{x}_2 \wedge (x_2 \vee \bar{x}_2') \wedge x_2'$ is not satisfiable.
  $\rightarrow$ Add $\bar{x}_2$ to F_1
  $\rightarrow F_1 = P \wedge \bar{x}_1 \wedge \bar{x}_2$

That concludes the propagation-phase resulting in

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

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

4. Step: Second Transition
Check if $F_1 \wedge T \Rightarrow P'$ by testing $F_1 \wedge T \wedge \bar{P}'$ for satisfiability:

$$(\bar{x}_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2 \wedge T \wedge x_1' \wedge \bar{x}_2'$$

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 $F_2 = P$
For every clause c in F_1 check $F_1 \wedge T \wedge \bar{c}'$ for satisfiability, again skipping P.

- $c = \bar{x}_1$

  $$(\bar{x}_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2 \wedge T \wedge x_1'$$

  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
  $\rightarrow$ Add $\bar{x}_1$ to F_2
  $\rightarrow$ $F_2 = P \wedge \bar{x}_1$

- $c = \bar{x}_2$

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

  Unsatisfiable because $\bar{x}_2 \wedge (x_2 \vee \bar{x}_2') \wedge x_2'$ is not satisfiable.
  $\rightarrow$ Add $\bar{x}_2$ to F_2
  $\rightarrow$ $F_2 = P \wedge \bar{x}_1 \wedge \bar{x}_2$

With that 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.

3.4 Possible Improvements

The most time consuming part of the 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:

3.4.1 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 a c just remove unused clauses of s.

3.4.2 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$(x_1 \wedge \bar{x}_2, 1)$

4 PDR as Software Checker

We see that PDR is a useful 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. There are multiple ways to do that, the following approach is based on the technique described in [8].
To use PDR on software we first need some new definitions and other preliminaries.

4.1 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:

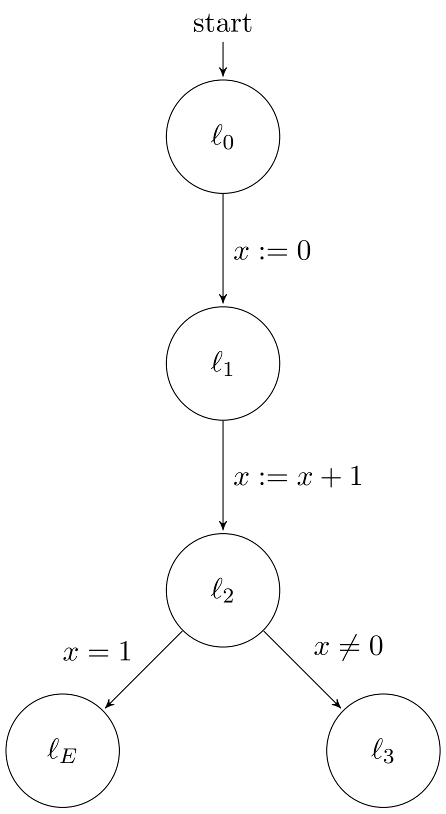


Figure 4.1: Graph of $\mathcal{A}$

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

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

Resulting in the global transition formula:

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

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.

4.2 Lifted Algorithm

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

- Instead of a global set of Frames $[F_0, ..., F_k]$ assign each program location $\ell \in L\backslash\{\ell_E\}$ a local set of frames $[F_{0,\ell}, ..., F_{k,\ell}]$. Each frame is now a cube of first-order formulas. 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 formulas, the lifted algorithm no longer blocks states but transitions, there are no bad states only bad transitions.
- Because of the structure of the CFA, 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 $\ell \in L\backslash\{l_E\}$, i is called a global fixpoint position. There is no more propagating formulas 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 $\ell \in L\backslash\{\ell_0, \ell_E\}$ as `false`, and for ℓ_0 as `true`.

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

1. Next Level
Initialize for each $\ell \in L\backslash\{\ell_E\}$ 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:

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

is satisfiable.

- If it is *satisfiable*, it means that t could not be blocked at ℓ on level i, generate an new proof-obligation $(p, \ell_{pre}, i-1)$ where p is the weakest precondition of t.
- 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 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 $\ell \in L \backslash \{l_E\}$.
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:

Algorithm 2 lifted-PDR-prove

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

7:     loop
8:         for all $\ell \in L \backslash \{\ell_E\}$ 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 $\ell \in L \backslash \{l_E\}$ do
18:                if $\ell.trace[i] \neq \ell.trace[i-1]$ then
19:                    break
20:            return true
```

4.3 Example

4.3.1 Reachable Error State

To show an application of the lifted algorithm reconsider the example from earlier, we have CFA $\mathcal{A} = (X, L, G, \ell_0, \ell_E)$ 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 $\ell \in L\backslash\{\ell_0, \ell_E\}$ a new frame $F_{0,\ell} = false$, for ℓ_0 add $F_{0,\ell_0} = true$:

location \ level	0
ℓ_0	$true$
ℓ_1	$false$
ℓ_2	$false$
ℓ_3	$false$

2. Step: Next Level
Initialize new frames for level 1 as `true`:

location \ level	0	1
ℓ_0	$true$	$true$
ℓ_1	$false$	$true$
ℓ_2	$false$	$true$
ℓ_3	$false$	$true$

To generate the initial proof-obligations, check G and take the transitions where ℓ_E is the target.
There is one transition $(\ell_2, x = 1, \ell_E)$, that means we have to block $x = 1$ at ℓ_2 on level 1
$\rightarrow$ 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 $F_{0,l_{pre}} \wedge T_{\ell_{pre} \rightarrow \ell_2} \wedge x' = 1$ 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
$\rightarrow$ 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	$true$	$true$
ℓ_1	$false$	$true$
ℓ_2	$false \wedge x \neq 1$	$true \wedge x \neq 1$
ℓ_3	$false$	$true$

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 $\ell \in L \backslash \{l_E\}$.
$\rightarrow$ There is no such i, we continue with the next level.

5. Step: Next Level
Initialize new frames for level 2 as `true`:

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

Again generate the initial proof-obligation which is the same as before but on level 2 now:
$\rightarrow$ 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

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

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

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

for satisfiability.
Which is valid, with `true` being the weakest precondition, we generate the new proof-obligation $(true, l_0, 0)$ and because this obligation is on level 0 we terminate, stating that ℓ_E is reachable by the counter-example trace:

$$\ell_0 \to \ell_1 \to \ell_2 \to \ell_E$$

4.3.2 Unreachable Error State

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

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

with graph:

$$\begin{tikzcd}
 & \text{start} \arrow[d] & \\
 & \ell_0 \arrow[d, "x := 0 \land y := x"] & \\
 & \ell_1 \arrow[loop right, "x := x + 1 \land y := y + 1"] \arrow[dl, "x = y"'] \arrow[dr, "x \neq y"] & \\
\ell_2 & & \ell_E
\end{tikzcd}$$

Figure 4.2: 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 $\ell \in L \backslash \{\ell_0, \ell_E\}$ a new frame $F_{0,\ell} = false$, for ℓ_0 add $F_{0,\ell_0} = true$.

level location	0
ℓ_0	$true$
ℓ_1	$false$
ℓ_2	$false$

2. Step: Next Level
Initialize new frames for level 1 as `true`:

level location	0	1
ℓ_0	$true$	$true$
ℓ_1	$false$	$true$
ℓ_2	$false$	$true$

Generate the initial proof-obligations:
There is only one transition leading to ℓ_E, $(\ell_1, x \neq y, \ell_E)$
$\rightarrow$ proof-obligation $(x \neq y, \ell_1, 1)$.

3. Step: First Blocking Phase
To block the proof-obligation $(x \neq y, \ell_1, 1)$ check each predecessor of ℓ_1:

- predecessor: ℓ_0

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

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

level location	0	1
ℓ_0	$true$	$true$
ℓ_1	$false \wedge x = y$	$true \wedge x = y$
ℓ_2	$false$	$true$

- predecessor: ℓ_1

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

Which is unsatisfiable as well
$\rightarrow$ 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 $\ell \in L \backslash \{l_E\}$.
$\rightarrow$ There is no such i, we continue with the next level.

5. Step: Next Level
Initialize new frames for level 2 as `true`:

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

Again generate the initial proof-obligation which is the same as before but on level 2 now:
$\rightarrow$ 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

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

  Which is unsatisfiable
  $\rightarrow$ 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	$true$	$true$	$true$
ℓ_1	$false \wedge x = y$	$true \wedge x = y$	$true \wedge x = y$
ℓ_2	$false$	$true$	$true$

- predecessor: ℓ_1

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

  Which is unsatisfiable as well
  $\rightarrow$ 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

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

$$\underbrace{\qquad\qquad\qquad\qquad}_{global fixpoint}$$

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

4.4 Possible Improvements

As shown above, 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 formulas making it impossible to use it to reduce lifted proof-obligations.

Different generalization techniques are possible:

4.4.1 Syntactical Analysis

Given a cube c remove $a \subseteq c$, if no variable of a is assigned in T and

1. a is already contained in a frame, or
2. there exists an `assume a` in T

4.4.2 Weakest Precondition

The definition of the lifted algorithm above already contains an improvement, using weakest preconditions to find predecessors. Instead of generating multiple proof-obligations for each individual predecessor state, the weakest precondition covers all of them in a single one.

4.4.3 the Disjunctive Normal Form

After transforming the weakest precondition into its disjunctive normal form, each cube can be considered as a separate smaller proof-obligation, saving time on larger formulas.

4.4.4 Using Interpolation

TODO We are using interpolation on frames and pos not on traces.

5 Implementation in Ultimate

5.1 Introduction Ultimate

Ultimate 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

5.2 Implementation

To implement PDR in Ultimate we chose to implement a new library plugin *Library-PDR*, that is used in the generator plugin *traceabstraction*.

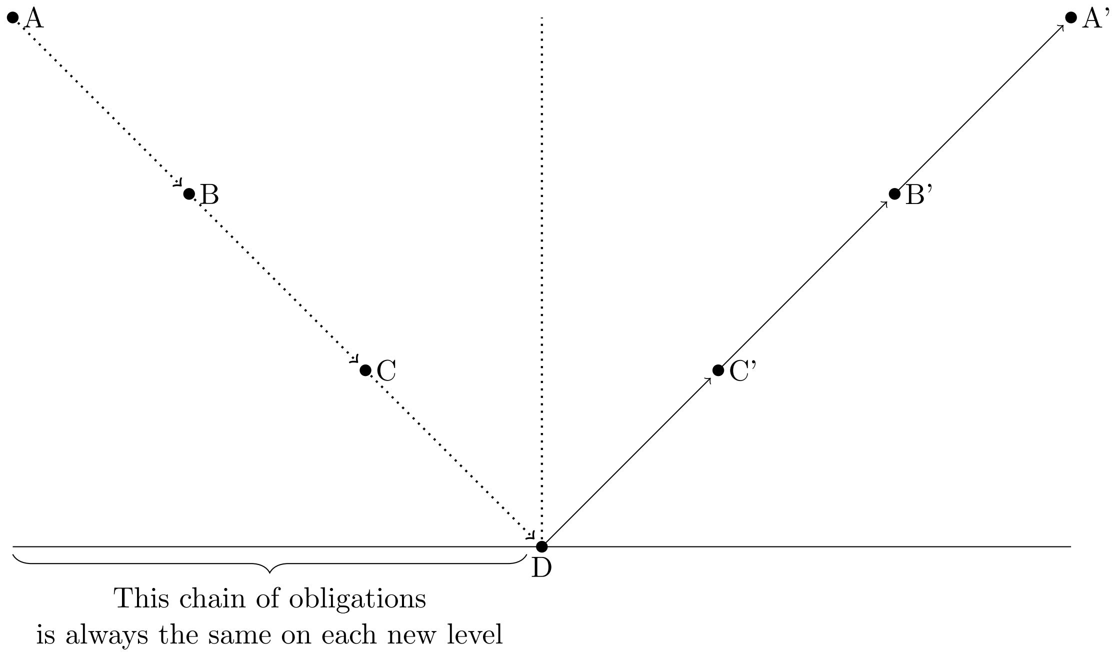
5.3 Improvements

5.3.1 Caching of Proof-Obligation Queue

Because of the backwards-search nature of PDR, we always start with the initial proof-obligation and making our way back to the latest blocked proof-obligation iteratively. Most of the proof-obligations that are generated have already been seen. For example, let i be the current level, we have seen and blocked the initial proof-obligation i times, let there be a satisfying assignment of the initial proof-obligation

meaning we have a follow up proof-obligation on level i-1, which we have also seen i-1 times.
Our idea was to cache that proof-obligation queue and always start a new level not with the initial proof-obligation but with the newest generated one, so that we save the whole chain of the previous obligations as we always end up at the newest one. For that we introduced a global level and changed the levels of each proof-obligation to be subtracted from the global level, with that every proof-obligation is at its corresponding level, for example, the initial proof-obligation has its local level 0, so that 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 global level - 1 = global level - 1, meaning that this proof-obligation comes immediately after the initial one.
Something that cannot be saved is the blocking of the previous proof-obligations, 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.

ToDo Graphic


A
B
C
D
A'
B'
C'
This chain of obligations
is always the same on each new level

5.3.2 Ignoring Already Blocked Proof-Obligations

Besides changing the way the levels work in proof-obligations, we expanded them with a list of already seen and blocked queries to the SMT-solver. With that memoization in place, before we consider a new call to the SMT-solver we first check the given query whether we have already seen and deemed it unsatisfiable, if so, we just copy the the frame from one level before and continue with the next obligation. If we prove an unknown query as unsatisfiable we add them to the list of seen queries.

With this technique we save time on programs with a lot of proof-obligations for example in loops, where we see the same queries very often or when we have blocked the newest obligation and have to block the obligation chain back to the initial one.

5.3.3 Using Preconditions

As discussed in the improvements section of the lifted PDR algorithm using weakest preconditions is a really useful addition to the way proof-obligations are generated. In our implementation we instead use just the precondition, because "assume" edges have a reeeaaaallly funky weakest precondition that lead to diverse errors. **ToDo: A bit nicer**

6 Evaluation

7 Future Work

7.1 Implementation of Further Improvements

7.1.1 Generalization

7.1.2 Procedures

Bibliography

[1] Vizel Y., Gurfinkel A. (2014) Interpolating Property Directed Reachability. In: Biere A., Bloem R. (eds) Computer Aided Verification. CAV 2014. Lecture Notes in Computer Science, vol 8559. Springer, Cham

[2] Bradley A.R. (2011) SAT-Based Model Checking without Unrolling. In: Jhala R., Schmidt D. (eds) Verification, Model Checking, and Abstract Interpretation. VMCAI 2011. Lecture Notes in Computer Science, vol 6538. Springer, Berlin, Heidelberg

[3] `https://fmv.jku.at/hwmcc10/results.html`

[4] ULTIMATE: `https://ultimate.informatik.uni-freiburg.de`

[5] N. Een, A. Mishchenko and R. Brayton, "Efficient implementation of property directed reachability," 2011 Formal Methods in Computer-Aided Design (FMCAD), Austin, TX, 2011, pp. 125-134. Reachability"

[6] Cimatti A., Griggio A. (2012) Software Model Checking via IC3. In: Madhusudan P., Seshia S.A. (eds) Computer Aided Verification. CAV 2012. Lecture Notes in Computer Science, vol 7358. Springer, Berlin, Heidelberg

[7] Hoder K., Bjørner N. (2012) Generalized Property Directed Reachability. In: Cimatti A., Sebastiani R. (eds) Theory and Applications of Satisfiability Testing – SAT 2012. SAT 2012. Lecture Notes in Computer Science, vol 7317. Springer, Berlin, Heidelberg

[8] T. Lange, M. R. Neuhauber and T. Noll, "IC3 software model checking on control flow automata," 2015 Formal Methods in Computer-Aided Design (FMCAD), Austin, TX, 2015, pp. 97-104.

[9] Heizmann M., Hoenicke J., Podelski A. (2013) Software Model Checking for People Who Love Automata. In: Sharygina N., Veith H. (eds) Computer Aided Verification. CAV 2013. Lecture Notes in Computer Science, vol 8044. Springer, Berlin, Heidelberg

[10] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '02). ACM, New York, NY, USA, 58-70