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

A novel hardware verification algorithm was devised by Aaron Bradley [5] called `IC3`, which was surprisingly efficient, winning third place in the hardware model-checking competition (HWMCC) at CAV 2010 [1]. Recently this sparked interest in using the underlying technique, Property Directed Reachability, on software. The goal of this thesis is to extend Property Directed Reachability so that it can be used on software and to implement a Property Directed Reachability-based model-checker in the software analysis framework Ultimate [3].

Zusammenfassung

Der neue Hardware Verifizierungsalgorithmus `IC3`, entwickelt von Aaron Bradley [5], überraschte durch seine Effizienz und gewann den dritten Preis in der Model-checking competition (HWMCC) bei CAV 2010 [1]. Das weckte vor Kurzem das Interesse die unterliegende Technik, Property Directed Reachability, auch auf Software zu nutzen. Das Ziel dieser Arbeit ist Property Directed Reachability so zu erweitern, dass es auf Software anwendbar ist, und einen auf Property Directed Reachability basierten Modell-Checker in das Softwareanalyseframework Ultimate zu implementieren.

Acknowledgements

First and foremost I would like to thank my advisor Daniel, who always guided me in difficult situations. Be it implementing solutions in Ultimate, or helping me with english grammar, and LaTeX aesthetics. I am immensely grateful for your advice and guidance.

Then of course my beloved family who supported me all this time.
My friends in Freiburg, my friends at the Hochrhein, Phil and Vinz who owe me a dinghy, Nik, Dennis, the Helden Gang, my favorite jazz singer Gabri, and whoever I might have forgotten to mention.

Contents

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 analyze 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 discussing 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 on hardware, 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 $S = (X, I, T)$, consisting of

- a finite set of boolean variables X
- a conjunction representing the *initial state* I
- 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 T

Consider the transition system $U = (X, I, T)$ in Figure 2.1 where

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

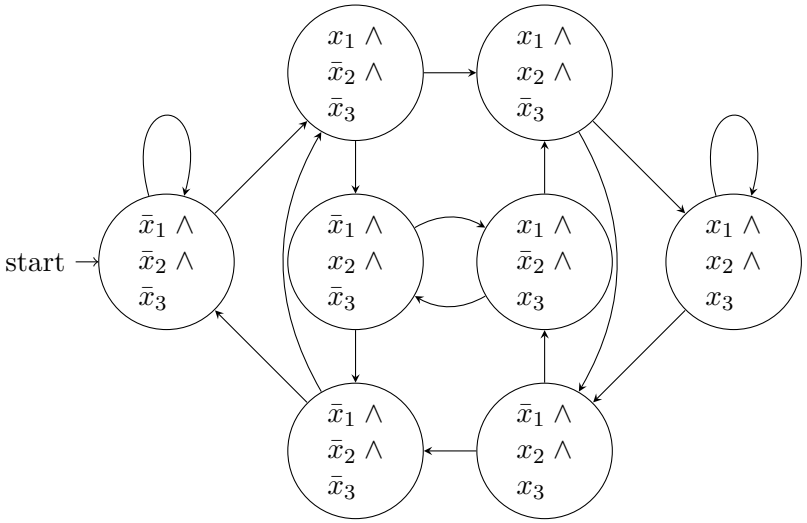


Figure 2.1: Transition Graph of U.

We call a variable or its negation, e.g., x or $\bar{y}$, a *literal*
Furthermore, a conjunction of literals is called a *cube*, and a disjunction a *clause*.
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. $\bar{P}$ describes 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.
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 counterexample.

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 $[F_0, ..., F_k]$, 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 ∃ cube c, s.t. trace.last() ∧ T ∧ c′ is SAT and c ⇒ P̄ 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 ∈ trace.last() do
12:             if trace.last() ∧ T ∧ c̄′ is UNSAT then
13:                 F_{k+1} = F_{k+1} ∧ 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 in 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 $[F_0, F_1, ..., F_k]$ 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 $\vec{x} = (x_1, x_2, ..., x_{|X|}, x'_1, x'_2, ..., x'_{|X'|})$ the algorithm gets a new bad state $a = x_1 \wedge x_2 \wedge ... \wedge x_{|X|}$ and creates tuple (a, k), this tuple is called a *proof-obligation*.
- If the formula is *unsatisfiable*, the algorithm 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
  $\vec{x} = (x_1, x_2, ..., x_{|X|}, x'_1, x'_2, ..., x'_{|X'|})$ the algorithm gets a new bad state
  $c = x_1 \wedge x_2 \wedge ... \wedge x_{|X|}$ 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.

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:

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

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 $F_1 \not\equiv F_0$ 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 new frame $F_2 = P$
We need to check each clause c in F_1 if $F_1 \wedge T \wedge \bar{c}'$ 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:

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

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

  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

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

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

  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_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 $F_2 \not\equiv F_1$ 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 $(\underbrace{x_1, \bar{x}_2, \bar{x}_3}_{\mathrm{q}}, x'_1, x'_2, \bar{x}'_3)$, 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 $F_2 = F_2 \wedge \bar{s}$ making absolutely sure that s is not reachable, every future proof-obligation containing s would have been blocked by F_2.

2.3.2 Case 2: Property Satisfied

To show a transition system with an inductive invariant, consider $B = (X, I, T)$ shown in Figure 2.3, where

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

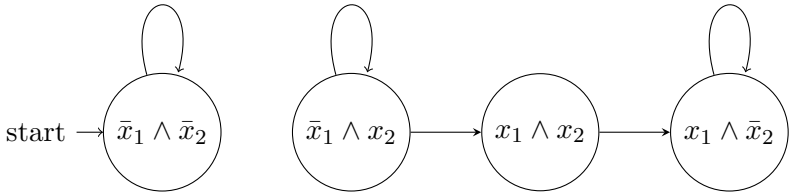


Figure 2.3: Transition Graph of B.

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. 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

After we have seen how PDR works on systems based on propositional logic, we show in this chapter how to extend the approach to work on systems based on first-order logic. We need to *lift* the approach from bit-level 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

The lifted algorithm no longer works on boolean transition systems but on control flow graphs.

A control flow graph (CFG) $\mathcal{A} = (X, L, G, \ell_0, \ell_E)$ is a graph, 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 $\ell_0 \in L$
- An error location $\ell_E \in L$

The lifted algorithm tries to prove whether ℓ_E is reachable by finding a feasible path from ℓ_0 to ℓ_E.

For example consider the CFG $\mathcal{A} = (X, L, G, \ell_0, \ell_E)$ in Figure 3.1, where

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

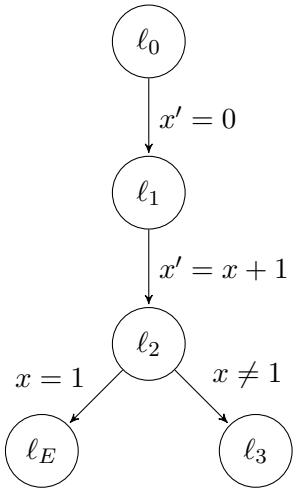


Figure 3.1: Graph of $\mathcal{A}$.

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:

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

3.2 Lifted Algorithm

To lift our on propositional logic based PDR algorithm to first-order logic we have to modify four aspects of the algorithm:

- Instead of a global set of Frames $[F_0, ..., F_k]$ we 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 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 $\ell \in L\backslash\{l_E\}$, i is called a global fixpoint position. There is no more propagating formulae forward.

```
1:  procedure LIFTED-PDR-PROVE(L, G)
2:      check for 0-counter-example
3:      ℓ_0.trace.push(new frame(true))
4:      for all ℓ ∈ L\{ℓ_0, ℓ_E} do
5:          ℓ.trace.push(new frame(false))
6:      level := 0

7:      loop
8:          for all ℓ ∈ L\{ℓ_E} do
9:              ℓ.trace.push(new frame(true))
10:         level := level + 1
11:         get initial proof-obligations

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

16:         for i = 0; i ≤ level; i := i + 1 do
17:             for ℓ ∈ L\{l_E} do
18:                 if ℓ.trace[i] ≠ ℓ.trace[i − 1] then
19:                     break
20:             return true
```

Figure 3.2: Lifted PDR Algorithm.

Figure 3.2 shows the updated PDR algorithm. Given a CFG $\mathcal{A} = (X, L, G, \ell_0, \ell_E)$ we want to check if ℓ_E is reachable:

Like the bit-level algorithm it starts on line 2 with checking for a 0-counter-example, the algorithm checks whether $\ell_0 = \ell_E$. If so it terminates and returns that ℓ_E is indeed

reachable, if not it initializes 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
In lines 7 to 11, the algorithm initializes for each $\ell \in L \backslash \{\ell_E\}$ a new frame $k+1$ as `true`. For each location $\ell \in L$ where $(\ell, t, \ell_E) \in G$ it generates an initial proof-obligation (t, ℓ, k).

2. Blocking-Phase
With the lifted algorithm the blocking-phase is no longer nested in the preceding phase, as we see in line 12 to 15 int he code. Here the algorithm takes proof-obligation (t, ℓ, i) with the lowest i and checks 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, the algorithm generates a new proof-obligation $(p, \ell_{pre}, i-1)$ where p is the weakest precondition of t with regard to $T_{\ell_{pre} \to \ell}$.
- If the formula is *unsatisfiable*, the algorithm strengthens 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
In lines 16 to 20 the algorithm checks 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.

3.3 Example

In this section we show an application of the lifted algorithm on two cases, one where the error state is reachable, and one where it is not.

3.3.1 Case 1: Error State is Reachable

To show an application of the lifted algorithm consider again the example in Figure 3.1, 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 $\ell \in L \backslash \{\ell_0, \ell_E\}$ a new frame $F_{0,\ell} = false$, for ℓ_0 adding $F_{0,\ell_0} = true$:

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

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

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

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:

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

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

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

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.

	level	
location	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 $\ell \in L \backslash \{l_E\}$.
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`:

	level		
location	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, 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 checking

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

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:

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

3.3.2 Case 2: Error State is Unreachable

Figure 3.3 shows an CFG $\mathcal{B} = (X, L, G, \ell_0, \ell_E)$ with reachable error state, where

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

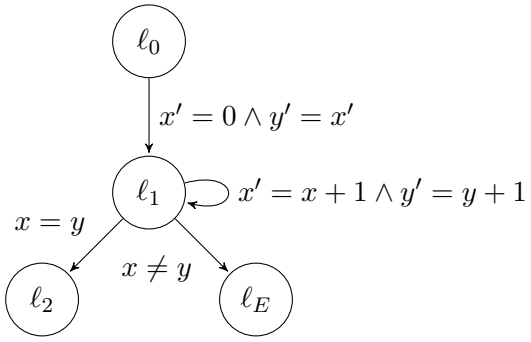


Figure 3.3: Graph of $\mathcal{B}$.

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

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

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

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

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

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

  Which is unsatisfiable, we 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

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

  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 $\ell \in L \backslash \{l_E\}$.
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`:

	level		
location	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 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

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

  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}:

	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	$false$	$true$	$true$

- predecessor: ℓ_1

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

  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

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

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.

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, because as we have seen in section 2.4.2 ternary simulation extends the binary propositional logic by a third value and checks whether that value propagates through formulae. First-order logic is not binary, extending its value space would not work the way it does on propositional logic, making it impossible to use to reduce lifted proof-obligations.

In the following we present techniques to improve our base algorithm.

3.4.1 Weakest Precondition

The definition of the lifted algorithm above already contains an improvement, using weakest preconditions to overapproximate predecessor states. Without weakest preconditions we would have to generate a new explicit proof-obligation for each possible predecessor state. Now we have one proof-obligation containing every possible predecessor.

3.4.2 Disjunctive Normal Form

It is possible to transform cubes into clauses by simply 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.

3.4.3 Interpolation

Adding the negated proof-obligation to the frames works well in most cases. Consider program $\mathcal{A}$ in Figure 3.4. The initial proof-obligation on level i is $(x = 1, \ell_1, i)$, because ℓ_1 has two predecessors we have to check:

$$F_{\ell_1,i-1} \wedge x' = x + 1 \wedge x' = 1 \tag{3.1}$$

$$F_{\ell_0,i-1} \wedge x' = 2 \wedge x' = 1 \tag{3.2}$$

for satisfiability.
Formula (3.1) is satisfiable until we get a proof-obligation on level 1, because frame $F_{\ell_1,0}$ is always false.
Formula (3.2) is always unsatisfiable, in each level we add $x \neq 1$ to ℓ_1's frames. That formula however is not strong enough to block the chain of satisfiable obligations of (3.1) in the next level, which is inefficient.

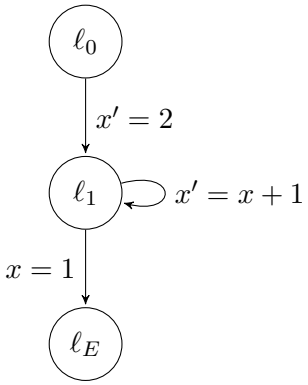


Figure 3.4: Program $\mathcal{A}$.

Instead, consider using interpolants. 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 $A \cap B$

Now, an interpolant for the unsatisfiable formula (3.2) would be $x' = 2$, if we add that to ℓ_1's frames, we would be able to block formula (3.1) much earlier.

4 PDR in Ultimate

In this chapter we introduce the program analysis framework Ultimate [1], 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. An 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]

[1] https://github.com/ultimate-pa

instance of the CEGAR scheme, that works like this:

1. Get an Error Trace:
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, 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.

2. Check Feasability:
To check a trace for feasibility we use our newly integrated Library-PDR. Our implementation does not consider the trace in isolation but in the context of the analyzed program. We project the given program to the transitions found in the trace. This projection is then considered as a new standalone program, called *path program*. We then use PDR on that.
Our Library-PDR works like this:

- 1. We get the analyzed program's CFA, and a possible trace as a list of transitions. From the trace and CFA we generate the corresponding path program.
- 2. Check for 0-Counter-Example: we check if the source of the first transition in the trace is equal to the target of the last transition.
  - If it is we return that the trace is feasible.
  - If not we initialize the frames as a mapping of location to a list of formulas
- 3. Blocking-Phase: We get the initial proof obligation $(f, \ell, 1)$, where f is the formula of the last transition in the trace, and ℓ being its source. We initialize a priority queue of proof-obligations and add the initial proof-obligation.

  We take the first obligation in the queue and check it for satisfiability. This continues until either a proof-obligation on level 0 is created or the queue is empty.
- 4. Propagation-Phase: In it we search for a global fixpoint by iterating through the frames of every location. If we detect a position where the formula equals the following formula, we have possible global fixpoint candidate. We check every other location if its frames are equal on that position too.
  If so we indeed have a global fixpoint and return that the trace is infeasible. If not we continue on.

3. Get Interpolants Sequence:
If Library-PDR returns that the trace is infeasible we need a sequence of interpolants. These are the formulas in the trace on the global fixpoint location, we return the conjunction of those formulas.

4.3 Improvements

As we have seen in Section 4.4 there is 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. In Figure 4.1 we see a representation of both chains of proof-obligation where the chain leading down the most recent one is highlighted, signaling that this chain can be skipped.

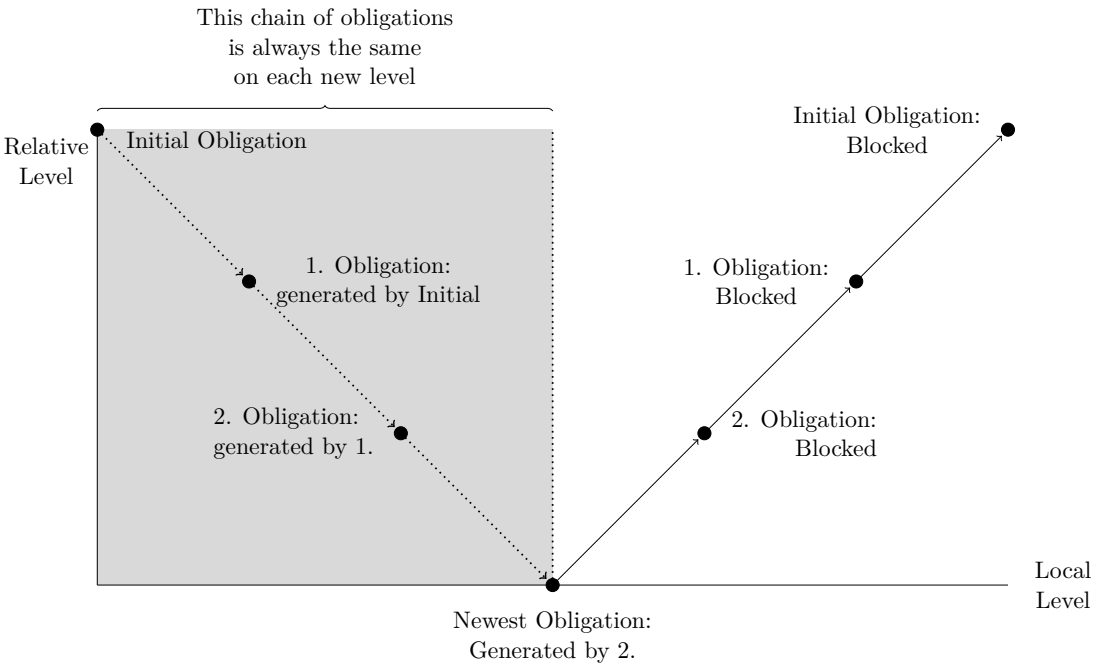


Figure 4.1: Chain of Proof-Obligation Generation.

4.3.2 Ignoring Already Blocked Proof-Obligations

Besides changing the levels of proof-obligations to be relative, we also expanded the proof-obligations 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.

5 Evaluation

In this chapter we evaluate our implementation.

We compare Automizer using Library-PDR as trace checker with Automizer using trace abstraction using SMTInterpol and nested interpolants.
We benchmarked Library-PDR with two different SMT-solvers:

- SMTInterpol [2]
- Z3 [4]

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

We tested it on Ultimate version 0.1.23-e6fd87c with time limit: 300s and memory limit: 8000MB

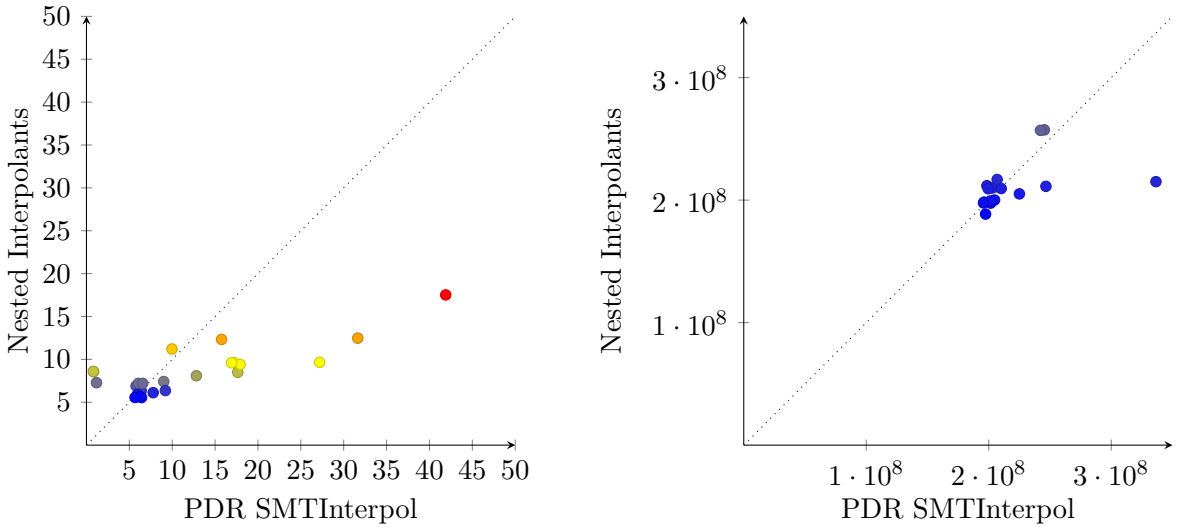
5.1 Comparison

We used a testing set containing 250 Boogie[1] programs, divided into the following categories:

- 31 examples of real-life code: `real-life`
- 40 examples of path programs without disjunctions: `20170319-ConjunctivePathPrograms`
- 134 examples of path programs that Automizer could not solve in three iterations: `20170304-DifficultPathPrograms`
- 37 examples of programs with difficult loop invariants: `tooDifficultLoopInvariants`

[1] https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/

- 8 examples of programs containing non-linear arithmetic: `nonlinear`



(a) CPU Time Comparison.

(b) Memory Usage Comparison.

Figure 5.1: Statistics Collected from Successful Benchmarks.

	Nested Interpolants	PDR SMTInterpol	PDR Z3
Tests Solved	179/250	49/250	62/250
Solve Time	3543s	575s	1332s
Timeouts	65	90	133
Exceptions	6	111	55
`real-life`			
Tests Solved	20/31	3/31	9/31
Solve Time	598s	8s	76s
Timeouts	11	10	14
Exceptions	0	18	8
`20170319-ConjunctivePathPrograms`			
Tests Solved	29/40	6/40	16/40
Solve Time	531s	35s	191s
Timeouts	11	15	20
Exceptions	0	19	4
`20170304-DifficultPathPrograms`			
Tests Solved	105/134	24/134	24/134
Solve Time	1435s	449s	975s
Timeouts	24	44	74
Exceptions	5	66	36
`tooDifficultLoopInvariants`			
Tests Solved	17/37	8/37	8/37
Solve Time	944s	42s	57s
Timeouts	19	21	22
Exceptions	1	8	7
`nonlinear`			
Tests Solved	8/8	8/8	5/8
Solve Time	35s	41s	33s
Timeouts	0	0	3
Exceptions	0	0	0

Table 5.1: Evaluation Results Divided by Category.

	Nested Interpolants	PDR
Exclusively solved	116	13

Table 5.2: Number of Programs That Were Solved Only by One Method. We United the Results of Both PDR Variants.

5.2 Discussion

In Figure 5.1 we can see that the most benchmark results were equally fast for both techniques. There are some programs where PDR was significantly slower than the Nested Interpolants. This stems from loops, because Library-PDR does not yet use interpolants, we have trouble solving programs that contain loops leading so slower performances and timeouts.
Both techniques were more or less on par regarding memory consumption.

Table 5.1 shows in the first section the overall statistics of each method. Then we see the performance of each category.
We see that PDR with SMTInterpol solved roughly $\frac{1}{5}$ of the benchmarks, and PDR with Z3 roughly $\frac{1}{4}$. The most exceptions of PDR with SMTInterpol were results of there being existential quantors in formulae with which SMTInterpol cannot deal. Most exceptions in PDR with Z3 were unsupported operation exceptions of Z3.

Table 5.2 shows the instances that were solved exclusively by one method. We see that PDR, here both SMT-solver benchmarks were united, were able to solve 13 instances. In each of these instances trace abstraction with nested interpolants timed out. 5 of those instances were solved as a result of using Z3 as both other methods using SMTInterpol were returning unknown.
The other 8 were:

- `20170319-ConjunctivePathPrograms/count_by_1_variant.i_2.bplTransformedIcfg_BEv2_3.bpl`

  Nested Interpolants timed out after 211 Automizer iterations, while PDR proved it in 2.

- `20170319-ConjunctivePathPrograms/jain_5.i_2.bplTransformedIcfg_BEv2_4.bpl`

  Nested Interpolants timed out after 130 Automizer iterations, while PDR proved it in 2.

- `20170304-DifficultPathPrograms/count_by_1.i_3.bpl`

  Nested Interpolants timed out after 231 Automizer iterations, while PDR proved it in 1.

- `20170304-DifficultPathPrograms/nested.c_4.bpl`

  Nested Interpolants timed out after 223 AUTOMIZER iterations, while PDR proved it in 2.

- `20170304-DifficultPathPrograms/phases1.i_3.bpl`

  Nested Interpolants timed out after 176 AUTOMIZER iterations, while PDR proved it in 1.

- `tooDifficultLoopInvariant/DivisibilityInterpolantRequired01-BackwardSuccess.bpl`

  Nested Interpolants timed out after 201 AUTOMIZER iterations, while PDR proved it in 2.

- `tooDifficultLoopInvariant/LargeConstant-ForwardSuccess.bpl`

  Nested Interpolants timed out after 224 AUTOMIZER iterations, while PDR proved it in 2.

- `tooDifficultLoopInvariant/codeblockAssertOrder01.bpl`

  Nested Interpolants timed out after 250 AUTOMIZER iterations, while PDR proved it in 2.

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 `PDR` 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. [12] 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. 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 tuples (pc, φ), where pc is a program location, φ is a first-order formula. $E \subseteq pc \times \psi \times pc$ being a set of transitions.

The root node is defined as $(pc_{init}, true)$.
For every non-leaf node (pc_i, φ) holds $\varphi \wedge T_{pc_i \to pc_j} \models \psi'$, for child node (pc_j, ψ). $T_{pc_i \to pc_j} \in E$.
The proposed algorithm first computes a possible path to an error location pc_E $(pc_{init}, ..., pc_E)$ then constructs a trace $F = [true, ..., \varphi_i, ..., \varphi_{n-1}]$, by taking the coupled formula of each location.
A proof-obligation in this approach is a tuple $(\varphi, T_{pc_i \to pc_j})$
To block a proof-obligation $(\varphi, T_{pc_i \to pc_j})$ the algorithm checks

$$F[i] \wedge T_{pc_i \to pc_j} \models \neg\varphi'$$

If that formula is satisfiable, the proof-obligation is blocked and the algorithm strengthens its trace $F[i]$ with $\neg c$. If it is unsatisfiable a new proof-obligation is generated.

We see this approach is similar to ours, except that it maintains a global trace like the original PDR and that it works on an ART, omitting levels.

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, $T_{\ell_{pre}\to\ell}$ 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 formulae A and B, we do that by dividing the query the following way:

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

This helps to save time on loops as we have seen in section 3.4.4.

7.1.2 Procedures

Ultimate aims to verify C programs, most of which contain procedure calls that our basic PDR algorithm cannot handle.
An example procedure is shown in Figure 7.1

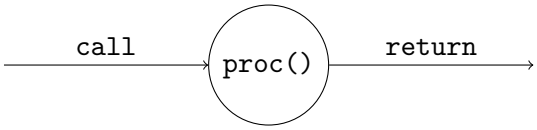


Figure 7.1: A Procedure.

Basically, there is a `call`-Transition leading to the procedure, then follows 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. Another problem is that returns correspond only to certain calls, if we do not know which return belongs to which call, we cannot handle them.

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

Additionally we have shown how to further improve PDR to make it more efficient. In comparison with another checking technique, Nested Interpolants, we have seen that PDR can be faster and can solve programs that the other technique struggled with. However there is still some work to be done like implementing proper use of interpolants and a way to deal with procedures.

Bibliography

[1] Hwmcc10 results. `http://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. Neuhäußer, 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.