Motivation

Overview

1. Introduction
2. Background: PDR on Hardware
3. PDR on Software
4. Implementation in Ultimate
5. Evaluation
6. Related Work
7. Future Work
8. Conclusion

Overview

1. Introduction
2. Background: PDR on Hardware
3. PDR on Software
4. Implementation in Ultimate
5. Evaluation
6. Related Work
7. Future Work
8. Conclusion

1. Introduction

Overview

1. Introduction
2. Background: PDR on Hardware
3. PDR on Software
4. Implementation in Ultimate
5. Evaluation
6. Related Work
7. Future Work
8. Conclusion

2.1 Preliminaries: Boolean Transition System

- A Boolean Transition System *S* = (X, I, T) consists of
  - Set of boolean variables *X*
  - A conjunction representing the initial state *I*
  - A propositional formula *T* over variables in *X* and *X'* = $\{x \in X \mid x' \in X'\}$, called Transition Relation

- States in *S* are cubes containing each variable from *X* with a boolean valuation of it
  - ➔ Finite number of states: $2^{|X|}$

- Transitions @Todo

2.1 Preliminaries: Formulas

- Given a formula φ over X, we get a primed formula φ' by replacing each variable with its corresponding variable in X'

- A literal is a variable or its negation
- A cube is a conjunction of literals
- A clause is a disjunction of literals
  - ➔ Negation of a cube is a clause and vice versa

- A Safety Property P is a formula over X that should be satisfiable by every state reachable from I
  - ➔ $\bar{P}$ being a set of bad states

2.2 Algorithm: Overview

- PDR on hardware checks if states in $\overline{P}$ are reachable from I

- For that it uses cubes of clauses, called Frames
  - Frame F_i represents an over-approximation of reachable states in at most i transitions from I

- PDR maintains sequence of frames $[F_0, F_1, \ldots, F_k]$, called trace

2.2 Algorithm: Pseudo-Code

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

  loop
    while $\exists$ cube c, s.t. $trace.last() \wedge T \wedge c'$ is SAT and $c \Rightarrow \bar{P}$ **do**
      recursively block proof-obligation(c, trace.size() - 1)
      and strengthen the frames of the trace.
      if a proof-obligation(p, 0) is generated **then**
        return false

    $F_{k+1} = new\ frame(P)$
    for all clause c $\in trace.last()$ **do**
      if $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT **then**
        $F_{k+1} = F_{k+1} \wedge c$
    if $trace.last() == F_{k+1}$ **then**
      return true
    $trace.push(F_{k+1})$

2.2 Algorithm: Pseudo-Code

```
procedure PDR-PROVE(I, T, P)
    check for 0-counter-example
    trace.push(new frame(I))

    loop
        while ∃ cube c, s.t. trace.last() ∧ T ∧ c' is SAT and c ⇒ P̄ do
            recursively block proof-obligation(c, trace.size() - 1)
            and strengthen the frames of the trace.
            if a proof-obligation(p, 0) is generated then
                return false

        F_{k+1} = new frame(P)
        for all clause c ∈ trace.last() do
            if trace.last() ∧ T ∧ c̄' is UNSAT then
                F_{k+1} = F_{k+1} ∧ c
        if trace.last() == F_{k+1} then
            return true
        trace.push(F_{k+1})
```

2.2 Algorithm: Checking for 0-Counter-Example

- Is $I \land \bar{P}$ satisfiable?

  → If satisfiable:

  - Algorithm terminates and returns that a bad state is reachable

  → If unsatisfiable:

  - Algorithm initializes the first frame in the trace: $F_0 = I$ and continues

2.2 Algorithm: Pseudo-Code

procedure PDR-PROVE(I, T, P)

  check for 0-counter-example

  $trace.push(new\ frame(I))$

Next Transition Phase

  loop

    while $\exists$ cube c, s.t. $trace.last() \land T \land c'$ is SAT and $c \Rightarrow \bar{P}$ **do**

      recursively block proof-obligation(c, trace.size() - 1)

      and strengthen the frames of the trace.

      if a proof-obligation(p, 0) is generated **then**

        return false

    $F_{k+1} = new\ frame(P)$

    for all clause c $\in trace.last()$ **do**

      if $trace.last() \land T \land \bar{c}'$ is UNSAT **then**

        $F_{k+1} = F_{k+1} \land c$

    if $trace.last() == F_{k+1}$ **then**

      return true

    $trace.push(F_{k+1})$

2.2 Algorithm: Next Transition Phase

- Checking if the next state is a good state:

  - Let $[F_0, F_1, \ldots, F_k]$ be the current trace
  - Is $F_k \land T \land \overline{P'}$ satisfiable?

    → If satisfiable:

    - Take satisfying assignment $\vec{x} = \{x_1, x_2, \ldots, x_{|X|}, x'_1, x'_2, \ldots, x'_{|X'|}\}$
    - The algorithm gets new bad state: $b = x_1 \land x_2 \land \ldots \land x_{|X|}$
    - Construct the tuple $t = (b, k)$, called proof-obligation

2.2 Algorithm: Next Transition Phase

- Checking if the next state is a good state:
  - Let $[F_0, F_1, \dots, F_k]$ be the current trace
  - Is $F_k \wedge T \wedge \overline{P'}$ satisfiable?

➔ If unsatisfiable:

- Continue with the next phase

2.2 Algorithm: Pseudo-Code

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

  loop
    while $\exists$ cube c, s.t. $trace.last() \wedge T \wedge c'$ is SAT and $c \Rightarrow \bar{P}$ **do**
      recursively block proof-obligation(c, trace.size() - 1)
      and strengthen the frames of the trace.
      if a proof-obligation(p, 0) is generated **then**
        return false

Blocking-Phase

    $F_{k+1} = new\ frame(P)$
    for all clause c $\in$ $trace.last()$ **do**
      if $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT **then**
        $F_{k+1} = F_{k+1} \wedge c$
    if $trace.last() == F_{k+1}$ **then**
      return true
    $trace.push(F_{k+1})$

```
loop
    while ∃ cube c, s.t. trace.last() ∧ T ∧ c' is SAT and c ⇒ P̄ do
        recursively block proof-obligation(c, trace.size() - 1)
        and strengthen the frames of the trace.
        if a proof-obligation(p, 0) is generated then
            return false
```

2.2 Algorithm: Blocking-Phase

Note: useful to have
Piece of pseudo-code?

- Proving that new bad states are not reachable
  - If there are proof-obligations:
    - Algorithm takes proof-obligation (b, i)
    - Tries to block bad state b by checking $F_{i-1} \land T \land b'$ for satisfiability

    ➔ If satisfiable:
      - Frame F_{i-1} is not strong enough to block b
      - Take satisfying assignment $\vec{x} = \left\{x_1, x_2, \ldots, x_{|X|}, x'_1, x'_2, \ldots, x'_{|X'|}\right\}$
      - The algorithm gets another new bad state: $c = x_1 \land x_2 \land \ldots \land x_{|X|}$
      - Construct new proof-obligation $u = (c, i - 1)$

2.2 **Algorithm:** Blocking-Phase

- Proving that new bad states are not reachable

  - If there are proof-obligations:
    - Algorithm takes proof-obligation (b, i)
    - Tries to block bad state b by checking $F_{i-1} \land T \land b'$ for satisfiability

      ➔ If unsatisfiable:
      - Algorithm strengthens F_i with $\bar{b}$

        ➔ $F_i = F_i \land \bar{b}$
      - Blocking bad state b at F_i

2.2 Algorithm: Blocking-Phase

- This continues recursively until:
  - There are no proof-obligations left
    - ➔ Algorithm continues with the next phase
  - A proof-obligation $(d, 0)$ is created
    - ➔ Proving that a bad state can be reached, terminating the algorithm

2.2 Algorithm: Pseudo-Code

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

    loop
        while $\exists$ cube c, s.t. $trace.last() \wedge T \wedge c'$ is SAT and $c \Rightarrow \bar{P}$ **do**
            recursively block proof-obligation(c, trace.size() - 1)
            and strengthen the frames of the trace.
            if a proof-obligation(p, 0) is generated **then**
                return false

        $F_{k+1} = new\ frame(P)$
        for all clause c $\in trace.last()$ **do**
            if $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT **then**
                $F_{k+1} = F_{k+1} \wedge c$
        if $trace.last() == F_{k+1}$ **then**
            return true
        $trace.push(F_{k+1})$

Propagation-Phase

2.2 Algorithm: Propagation-Phase

- Propagating learned information:
  - After no proof-obligations are left, the algorithm initializes new frame $F_{k+1} = \mathrm{P}$
  - Algorithm passes on learned information, e.g., which states are blocked:
    - For each clause c in F_k check: $F_k \wedge T \wedge \bar{c}'$ for satisfiability

      ➔ If satisfiable:
      - Do nothing, continue with next clause

2.2 Algorithm: Propagation-Phase

- Propagating learned information:
  - After no proof-obligations are left, the algorithm initializes new frame $F_{k+1} = \mathrm{P}$
  - Algorithm passes on learned information, e.g., which states are blocked:
    - For each clause c in F_k check: $F_k \land T \land \bar{c}'$ for satisfiability

      ➔ If unsatisfiable:
      - Algorithm strengthens F_{k+1} with c

        ➔ $F_{k+1} = F_{k+1} \land c$

2.2 Algorithm: Propagation-Phase

- Check for termination:
  - ➢ After all clauses have been tested, algorithm checks if $F_k \equiv F_{k+1}$
    - If so, the algorithm has found a fixpoint and terminates
      ➔ No states of $\overline{P}$ are reachable

2.2 Algorithm: Propagation-Phase

- Check for termination:
  - After all clauses have been tested, algorithm checks if $F_k \equiv F_{k+1}$
    - If not, the algorithm continues with a new Next Transition Phase

2.2 **Algorithm:** Propagation-Phase

- Check for termination:

  - After all clauses have been tested, algorithm checks if $F_k \equiv F_{k+1}$
    - If so, the algorithm has found a fixpoint and terminates
      - ➔ No states of $\bar{P}$ are reachable

    - If not, the algorithm continues with a new Next Transition Phase

- Algorithm repeats the three phases until a fixpoint is found, or a proof-obligation $(d, 0)$ is created

2.2 Algorithm: Pseudo-Code TEMPLATE

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

    loop
        while $\exists$ cube c, s.t. $trace.last() \wedge T \wedge c'$ is SAT and $c \Rightarrow \bar{P}$ **do**
            recursively block proof-obligation(c, trace.size() - 1)
            and strengthen the frames of the trace.
            if a proof-obligation(p, 0) is generated **then**
                return false

        $F_{k+1} = new\ frame(P)$
        for all clause c $\in trace.last()$ **do**
            if $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT **then**
                $F_{k+1} = F_{k+1} \wedge c$
        if $trace.last() == F_{k+1}$ **then**
            return true
        $trace.push(F_{k+1})$

2.3 Example

2.4 Possible Improvements

- Blocking one state at a time is ineffective:
  - Generalize blocked states

    → Eliminate insignificant cubes from states, that are not used by UNSAT-cores

- Ternary Simulation to reduce proof-obligations:
  - Extend binary variables with a new value: unknown
  - Check state variables of proof-obligations for importance

    → Eliminate unimportant state variables

Overview

1. Introduction
2. Background: PDR on Hardware
3. PDR on Software
4. Implementation in Ultimate
5. Evaluation
6. Related Work
7. Future Work
8. Conclusion

3.1 Preliminaries

- For using PDR on software, lift the algorithm from propositional logic to first-order logic

➔ We base our approach on the technique described by Lange et al.

1. Tim Lange, Martin R. Neuh¨außer, and Thomas Noll. IC3 software model checking on control flow automata. In *FMCAD*, pages 97–104. IEEE, 2015.

3.1 Preliminaries: Control Flow Graph

- A control flow graph (CFG) $A = (X, L, G, \ell_0, \ell_E)$ is a graph consisting of
  - A finite set of first-order 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 $\ell_E \in L$

3.1 Preliminaries: Control Flow Graph

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

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

3.1 Preliminaries: Control Flow Graph

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

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

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

3.2 Lifted Algorithm: Pseudo-Code

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

  loop
    while $\exists$ cube c, s.t. $trace.last() \wedge T \wedge c'$ is SAT and $c \Rightarrow \bar{P}$ **do**
      recursively block proof-obligation(c, trace.size() - 1)
      and strengthen the frames of the trace.
      if a proof-obligation(p, 0) is generated **then**
        return false

    $F_{k+1} = new\ frame(P)$
    for all clause c $\in trace.last()$ **do**
      if $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT **then**
        $F_{k+1} = F_{k+1} \wedge c$
    if $trace.last() == F_{k+1}$ **then**
      return true
    $trace.push(F_{k+1})$

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

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

    while $\exists$ proof-obligation (t, ℓ, i), **do**
      Recursively block proof-obligation
      if a proof-obligation$(p, \ell, 0)$ is generated **then**
        return false

    for $i = 0;\ i \leq level;\ i := i + 1$ **do**
      for $\ell \in L \setminus \{l_E\}$ **do**
        if $\ell.trace[i] \neq \ell.trace[i-1]$ **then**
          break
      return true

3.2 Lifted Algorithm: Pseudo-Code

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

  loop
    while $\exists$ cube c, s.t. $trace.last() \wedge T \wedge c'$ is SAT and $c \Rightarrow \bar{P}$ **do**
      recursively block proof-obligation(c, trace.size() - 1)
      and strengthen the frames of the trace.
      if a proof-obligation(p, 0) is generated **then**
        return false

    $F_{k+1} = new\ frame(P)$
    for all clause c $\in trace.last()$ **do**
      if $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT **then**
        $F_{k+1} = F_{k+1} \wedge c$
    if $trace.last() == F_{k+1}$ **then**
      return true
    $trace.push(F_{k+1})$

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

  loop
    for all $\ell \in L \backslash \{\ell_E\}$ **do**
      $\ell.trace.push(new\ frame(true))$
    $level := level + 1$
    get initial proof-obligations

    while $\exists$ proof-obligation (t, ℓ, i), **do**
      Recursively block proof-obligation
      if a proof-obligation$(p, \ell, 0)$ is generated **then**
        return false

    for $i = 0;\ i \leq level;\ i := i + 1$ **do**
      for $\ell \in L \backslash \{l_E\}$ **do**
        if $\ell.trace[i] \neq \ell.trace[i-1]$ **then**
          break
      return true

3.2 Lifted Algorithm: Checking for 0-Counter-Example

- Is $\ell_0 = \ell_E$?

➔ Yes:

- Algorithm terminates, returning that ℓ_E is reachable

➔ No:

- Algorithm continues

3.2 Lifted Algorithm: Pseudo-Code

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

  loop
    while $\exists$ cube c, s.t. $trace.last() \wedge T \wedge c'$ is SAT and $c \Rightarrow \bar{P}$ **do**
      recursively block proof-obligation(c, trace.size() - 1)
      and strengthen the frames of the trace.
      if a proof-obligation(p, 0) is generated **then**
        return false

    $F_{k+1} = new\ frame(P)$
    for all clause c $\in trace.last()$ **do**
      if $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT **then**
        $F_{k+1} = F_{k+1} \wedge c$
    if $trace.last() == F_{k+1}$ **then**
      return true
    $trace.push(F_{k+1})$

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

  loop
    for all $\ell \in L \backslash \{\ell_E\}$ **do**
      $\ell.trace.push(new\ frame(true))$
    $level := level + 1$
    get initial proof-obligations

    while $\exists$ proof-obligation (t, ℓ, i), **do**
      Recursively block proof-obligation
      if a proof-obligation$(p, \ell, 0)$ is generated **then**
        return false

    for $i = 0;\ i \leq level;\ i := i + 1$ **do**
      for $\ell \in L \backslash \{l_E\}$ **do**
        if $\ell.trace[i] \neq \ell.trace[i-1]$ **then**
          break
      return true

3.2 Lifted Algorithm: Pseudo-Code

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

  loop
    while $\exists$ cube c, s.t. $trace.last() \wedge T \wedge c'$ is SAT and $c \Rightarrow \bar{P}$ **do**
      recursively block proof-obligation(c, trace.size() - 1)
      and strengthen the frames of the trace.
      if a proof-obligation(p, 0) is generated **then**
        return false

    $F_{k+1} = new\ frame(P)$
    for all clause c $\in trace.last()$ **do**
      if $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT **then**
        $F_{k+1} = F_{k+1} \wedge c$
    if $trace.last() == F_{k+1}$ **then**
      return true
    $trace.push(F_{k+1})$

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

  loop
    for all $\ell \in L \backslash \{\ell_E\}$ **do**
      $\ell.trace.push(new\ frame(true))$
    $level := level + 1$
    get initial proof-obligations

    while $\exists$ proof-obligation (t, ℓ, i), **do**
      Recursively block proof-obligation
      if a proof-obligation$(p, \ell, 0)$ is generated **then**
        return false

    for $i = 0;\ i \leq level;\ i := i + 1$ **do**
      for $\ell \in L \backslash \{l_E\}$ **do**
        if $\ell.trace[i] \neq \ell.trace[i-1]$ **then**
          break
      return true

3.2 Lifted Algorithm: Local Traces

- There is no global trace $[F_0, F_1, \dots, F_k]$

➔ Every location $\ell \in L \setminus \{\ell_E\}$ has its own local trace $[F_{0,\ell}, F_{1,\ell}, \dots, F_{k,\ell}]$

➔ Lifted frames are cubes of first-order formulas

➔ @ToDo, explain changes to proofobligations

3.2 Lifted Algorithm: Pseudo-Code

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

    loop
        while $\exists$ cube c, s.t. $trace.last() \wedge T \wedge c'$ is SAT and $c \Rightarrow \bar{P}$ **do**
            recursively block proof-obligation(c, trace.size() - 1)
            and strengthen the frames of the trace.
            if a proof-obligation(p, 0) is generated **then**
                return false

        $F_{k+1} = new\ frame(P)$
        for all clause c $\in trace.last()$ **do**
            if $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT **then**
                $F_{k+1} = F_{k+1} \wedge c$
        if $trace.last() == F_{k+1}$ **then**
            return true
        $trace.push(F_{k+1})$

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

    loop
        for all $\ell \in L \backslash \{\ell_E\}$ **do**
            $\ell.trace.push(new\ frame(true))$
        $level := level + 1$
        get initial proof-obligations

        while $\exists$ proof-obligation (t, ℓ, i), **do**
            Recursively block proof-obligation
            if a proof-obligation$(p, \ell, 0)$ is generated **then**
                return false

        for $i = 0;\ i \leq level;\ i := i + 1$ **do**
            for $\ell \in L \backslash \{l_E\}$ **do**
                if $\ell.trace[i] \neq \ell.trace[i-1]$ **then**
                    break
            return true

3.2 Lifted Algorithm: Initialization

- Initialize each local frames:

  - $F_{0,\ell} = \begin{cases} true, & \ell = \ell_0 \\ false, & otherwise \end{cases}$

3.2 Lifted Algorithm: Pseudo-Code

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

    loop
        while $\exists$ cube c, s.t. $trace.last() \wedge T \wedge c'$ is SAT and $c \Rightarrow \bar{P}$ **do**
            recursively block proof-obligation(c, trace.size() - 1)
            and strengthen the frames of the trace.
            if a proof-obligation(p, 0) is generated **then**
                return false

        $F_{k+1} = new\ frame(P)$
        for all clause c $\in trace.last()$ **do**
            if $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT **then**
                $F_{k+1} = F_{k+1} \wedge c$
        if $trace.last() == F_{k+1}$ **then**
            return true
        $trace.push(F_{k+1})$

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

Next Level Phase

    loop
        for all $\ell \in L \backslash \{\ell_E\}$ **do**
            $\ell.trace.push(new\ frame(true))$
        $level := level + 1$
        get initial proof-obligations

        while $\exists$ proof-obligation (t, ℓ, i), **do**
            Recursively block proof-obligation
            if a proof-obligation$(p, \ell, 0)$ is generated **then**
                return false

        for $i = 0;\ i \leq level;\ i := i + 1$ **do**
            for $\ell \in L \backslash \{l_E\}$ **do**
                if $\ell.trace[i] \neq \ell.trace[i-1]$ **then**
                    break
            return true

3.2 Lifted Algorithm: Next Level Phase

- Initializing the next level:
  - Let k be the current level:
    - ➔ Every location $\ell \in L \setminus \{\ell_E\}$ has trace $[F_{0,\ell}, \ldots, F_{k,\ell}]$
  - Algorithm initzializes new level $k+1$ for all locations $\ell \in L \setminus \{\ell_E\}$
    - ➔ Adding new frame $F_{k+1,\ell} = true$

3.2 Lifted Algorithm: Next Level Phase

- Initializing the next level:
  - ➢ Let k be the current level:
    - ➔ Every location $\ell \in L \setminus \{\ell_E\}$ has trace $[F_{0,\ell}, \ldots, F_{k,\ell}]$
  - ➢ Additionally, the algorithm computes initial proof-obligations:
    - Because of the structure of CFGs, it is always known which transitions lead to ℓ_E
      - ➔ Check G for transitions of the form (ℓ, t, ℓ_E)
      - ➔ For each transition, get proof-obligation (t, ℓ, k)
      - ➔ @ToDo explain lifted proof-obligations

3.2 Lifted Algorithm: Pseudo-Code

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

loop
while $\exists$ cube c, s.t. $trace.last() \wedge T \wedge c'$ is SAT and $c \Rightarrow \bar{P}$ **do**
recursively block proof-obligation(c, trace.size() - 1)
and strengthen the frames of the trace.
if a proof-obligation(p, 0) is generated **then**
return false

$F_{k+1} = new\ frame(P)$
for all clause c $\in trace.last()$ **do**
if $trace.last() \wedge T \wedge \bar{c}'$ is UNSAT **then**
$F_{k+1} = F_{k+1} \wedge c$
if $trace.last() == F_{k+1}$ **then**
return true
$trace.push(F_{k+1})$

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

loop
for all $\ell \in L \backslash \{\ell_E\}$ **do**
$\ell.trace.push(new\ frame(true))$
$level := level + 1$
get initial proof-obligations

Blocking-Phase

while $\exists$ proof-obligation (t, ℓ, i), **do**
Recursively block proof-obligation
if a proof-obligation$(p, \ell, 0)$ is generated **then**
return false

for $i = 0;\ i \leq level;\ i := i + 1$ **do**
for $\ell \in L \backslash \{l_E\}$ **do**
if $\ell.trace[i] \neq \ell.trace[i-1]$ **then**
break
return true

3.2 Lifted Algorithm: Blocking-Phase

- Blocking-Phase not nested in preceding phase:
  - ➔ No longer optional:
    - In each iteration we have at least the initial proof-obligations

3.2 Lifted Algorithm: Blocking-Phase

- Checking if bad transitions are reachable:
  - Algorithm takes proof-obligation (t, ℓ, i) with the lowest i
    - For each predecessor location ℓ_{pre} of ℓ check if $F_{i-1,\ell_{pre}} \wedge T_{\ell_{pre} \to \ell} \wedge t'$ is satisfiable

      ➔ If satisfiable:
      - t could not be blocked at ℓ on level i
      - Get new proof-obligation $(p, \ell_{pre}, i-1)$

        ➔ p being the weakest precondition of t and $T_{\ell_{pre} \to \ell}$

3.2 Lifted Algorithm: Blocking-Phase

- Checking if bad transitions are reachable:
  - ➢ Algorithm takes proof-obligation (t, ℓ, i) with the lowest i
    - For each predecessor location ℓ_{pre} of ℓ check if $F_{i-1,\ell_{pre}} \wedge T_{\ell_{pre} \to \ell} \wedge t'$ is satisfiable

      ➔ If unsatisfiable:
      - t is blocked at ℓ on level i
      - Strengthen each frame $F_{j,\ell}, j \leq i$ with $\bar{t}$

        ➔ $F_{j,\ell} = F_{j,\ell} \wedge \bar{t}$

3.2 Lifted Algorithm: Blocking-Phase

- This continues recursively until:
  - ➔ There are no proof-obligations left:
    - Algorithm continues with the next phase
  - ➔ A proof-obligation $(d, \ell, 0)$ is created
    - Proving that there exists a feasible path to ℓ_E

3.2 Lifted Algorithm: Pseudo-Code

```
procedure PDR-PROVE(I, T, P)
    check for 0-counter-example
    trace.push(new frame(I))

    loop
        while ∃ cube c, s.t. trace.last() ∧ T ∧ c′ is SAT and c ⇒ P̄ do
            recursively block proof-obligation(c, trace.size() - 1)
            and strengthen the frames of the trace.
            if a proof-obligation(p, 0) is generated then
                return false

        F_{k+1} = new frame(P)
        for all clause c ∈ trace.last() do
            if trace.last() ∧ T ∧ c̄′ is UNSAT then
                F_{k+1} = F_{k+1} ∧ c
        if trace.last() == F_{k+1} then
            return true
        trace.push(F_{k+1})
```

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

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

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

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

Propagation-Phase

3.2 Lifted Algorithm: Propagation-Phase

- No more propagation of learned information

- Only checking for termination

  ➔ Trying to find a global fixpoint:

  - Is there an i where $F_{i-1,\ell} = F_{i,\ell}$ for every $\ell \in L \setminus \{\ell_E\}$?

3.2 Lifted Algorithm: Propagation-Phase

- No more propagation of learned information

- Only checking for termination
  - ➔ Trying to find a global fixpoint:
    - Is there an i where $F_{i-1,\ell} = F_{i,\ell}$ for every $\ell \in L \setminus \{\ell_E\}$?

    ➔ Yes:
    - Algorithm terminates returning that there is no feasible path to ℓ_E

3.2 Lifted Algorithm: Propagation-Phase

- No more propagation of learned information

- Only checking for termination
  - ➔ Trying to find a global fixpoint:
    - Is there an i where $F_{i-1,\ell} = F_{i,\ell}$ for every $\ell \in L \setminus \{\ell_E\}$?
  - ➔ No:
    - Algorithm continues with the next level

3.3 Example

3.4 Possible Improvements: Generalization of Proof-Obligations

- Using the weakest precondition:
  - ➔ Over approximation of predecessor states
  - ➔ Algorithm does not need to generate an explicit proof-obligation for each predecessor state

- Using the disjunctive normal form (DNF):
  - ➔ Negation of a cube is a clause:
    - Split large proof-obligations into smaller ones by taking each cube of the DNF as a separate proof-obligation

3.4 Possible Improvements: Generalization of Proof-Obligations

- Using Interpolation:
  - Instead of strengthening frames with the negated proof-obligation, compute an interpolant
  - @ToDo MOAR

Overview

1. Introduction
2. Background: PDR on Hardware
3. PDR on Software
4. Implementation in Ultimate
5. Evaluation
6. Related Work
7. Future Work
8. Conclusion

4.1 Implementation: Introduction Ultimate

4.2 Implementation: CEGAR-Scheme with PDR

4.3 Implemented Improvements

Overview

1. Introduction
2. Background: PDR on Hardware
3. PDR on Software
4. Implementation in Ultimate
5. Evaluation
6. Related Work
7. Future Work
8. Conclusion

5.1 Data Comparison

5.2 Discussion

Overview

1. Introduction
2. Background: PDR on Hardware
3. PDR on Software
4. Implementation in Ultimate
5. Evaluation
6. Related Work
7. Future Work
8. Conclusion

6. Related Work

- There are several other techniques of using PDR on software:

➢ Bit-Blasting:

- Encode variables as bitvectors
- Use new variable pc to keep track of program location
- Use unmodified hardware PDR algorithm on that

➔ Drawback: tedious handling of pc variable

6. Related Work

- There are several other techniques of using PDR on software:

- Using Abstract Reachability Trees (ART):
  - Exploiting the partitioning of program's state space by unwinding the CFG into an ART
  - @ToDo: introducing ARTs and how algo works

Overview

1. Introduction
2. Background: PDR on Hardware
3. PDR on Software
4. Implementation in Ultimate
5. Evaluation
6. Related Work
7. Future Work
8. Conclusion

7.1 Implementing Further Improvements

- There are possible ways to make our PDR algorithm more efficient:

➢ Interpolation:

- Ultimate already supports ways of computing interpolants

➔ Instead of strengthening frames with negated proof-obligation, add interpolant

➔ Helps with loops

7.1 Implementing Further Improvements

- There are possible ways to make our PDR algorithm more efficient:

➢ Dealing with procedures:

- Ultimate verifies C programs that contain procedure calls

➔ Our algorithm cannot deal with them:

- Problems arise due to PDR‘s linear backwards-search nature

➔ Possible solutions:

- Modify PDR to deal with procedures non-linearly
- Calculate procedure summary and attach that to the CFG, removing the procedure altogether

Overview

1. Introduction
2. Background: PDR on Hardware
3. PDR on Software
4. Implementation in Ultimate
5. Evaluation
6. Related Work
7. Future Work
8. Conclusion

8. Conclusion