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

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

- 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

- 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 ist 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

- PDR on hardware checks if states in $\bar{P}$ 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 $\exists$ 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:** 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. $trace.last() \wedge T \wedge c'$ 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:** 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: 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 satisfiable:

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

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

Blocking-Phase

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

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 $F_{i-1} \wedge T \wedge b'$ for satisfiability

➔ If satisfiable:

- Frame F_{i-1} is not strong enough to block b
- Take satisfying assignment $\vec{x} = \left\{x_1, x_2, \dots, x_{|X|}, x_1', x_2', \dots, x_{|X'|}'\right\}$
- The algorithm gets another new bad state: $c = x_1 \wedge x_2 \wedge \dots \wedge x_{|X|}$
- 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 $F_{i-1} \wedge T \wedge b'$ for satisfiability

➔ If unsatisfiable:

- Algorithm strenghthens F_i with $\bar{b}$

  ➔ $F_i = F_i \wedge \bar{b}$
- 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. $trace.last() \land T \land c'$ 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
        $trace.push(F_{k+1})$

Propagation-Phase

2.2 **Algorithm:** Propagation-Phase

- Propagating learned information

➢ After no proof-obligations are left, the algorithm initzializes new frame $F_{k+1} = \mathrm{P}$

➢ Algorithm passes on learned informations, 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 initzializes new frame $F_{k+1} = \mathrm{P}$

➢ Algorithm passes on learned informations, e.g, which states are blocked:

- For each clause c in F_k check: $F_k \wedge T \wedge \bar{c}'$ for satisfiability

  ➔ If unsatisfiable:

  - Algorithm strengthens F_{k+1} with c

    ➔ $F_{k+1} = F_{k+1} \wedge c$

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 $\bar{P}$ are reachable

- 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 ∃ 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})
```

2.3 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

3.2 Lifted Algorithm

- Algorithm Pseudocode here

3.3 Example

3.4 Possible Improvements

Overview

4.1 Implementation

4.2 Implemented Improvements

Overview

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

Overview

7.1 Further 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

8. Conclusion