Control-flow Guided Property Directed Reachability for Imperative Synchronous Programs

Xian Li and Klaus Schneider

Embedded Systems Chair
Department of Computer Science
University of Kaiserslautern, Germany

MEMOCODE 2016 at IIT Kanpur, November 18 – 20, 2016
Table of Contents

1. Motivation

2. Property Directed Reachability

3. Control-flow Guided PDR for Imperative Synchronous Programs
## Outline

1. Motivation

2. Property Directed Reachability

3. Control-flow Guided PDR for Imperative Synchronous Programs
Formal Verification of Synchronous Hardware Circuits

- PDR: a very efficient verification method based on induction
Formal Verification of Synchronous Programs

- PDR: a very efficient verification method based on induction

**Synchronous Programs**

```plaintext
module M(event bool ?a, ?b, o1, o2) {
    loop {
        l1: pause;
        if(o1 & (a|b)) {
            emit(o2);
            l2: await(a);
        }
    }
}
```

**Synchronous Circuits**

![Synchronous Circuits Diagram]
Imperative Synchronous Programs

Imperative Synchronous Languages: e.g. Quartz

- macro steps: consumption of one logical time unit
- micro steps: no logical time consumption

⇒ synchronous reactive model of computation

Control-flow Information

- not needed for synthesis
- useful for formal verification
Goals

**Target:** Safety Property Verification of Imperative Synchronous Programs

- PDR: relies on good estimation of the reachable states

**Our Heuristic:** Improve it by Reachable Control-flow States Computation

- linear-time static analysis
- symbolic reachability analysis
Outline

1. Motivation

2. Property Directed Reachability

3. Control-flow Guided PDR for Imperative Synchronous Programs
Safety Property Verification

Target: Prove $\Phi$ is valid w.r.t. $\mathcal{K}$

- a state transition system: $\mathcal{K} := (V, I, T)$
- a safety property: $\Phi$
- $\Phi$ holds on all reachable states of $\mathcal{K}$

module CfSeq (){
    p1: pause;
    p2: pause;
}

\[ V := \{ \text{run, p1, p2} \} \]
\[ I := \neg (\text{run} \lor \text{p1} \lor \text{p2}) \]
\[ T := \text{next}(\text{run}) \leftrightarrow \text{true} \]
\[ \land (\text{next}(\text{p1}) \leftrightarrow \neg \text{run}) \]
\[ \land (\text{next}(\text{p2}) \leftrightarrow \text{p1}) \]
\[ \Phi := \neg (\text{p1} \land \text{p2}) \]
Safety Property Verification by Induction

Target: Prove $\Phi$ is valid w.r.t. $\mathcal{K}$
- a state transition system: $\mathcal{K} := (\mathcal{V}, \mathcal{I}, \mathcal{T})$
- a safety property: $\Phi$
- $\Phi$ holds on all reachable states of $\mathcal{K}$

$\Phi$ is inductive w.r.t. $\mathcal{K}$
- induction base: $\Phi$ holds in all initial states
- induction step: $\Phi$-states have no successor violating $\Phi$
Safety Property Verification by Induction

Target: Prove $\Phi$ is valid w.r.t. $\mathcal{K}$
- a state transition system: $\mathcal{K} := (\mathcal{V}, \mathcal{I}, \mathcal{T})$
- a safety property: $\Phi$
- $\Phi$ holds on all reachable states of $\mathcal{K}$

$\Phi$ is inductive w.r.t. $\mathcal{K}$
- induction base: $\Phi$ holds in all initial states
- induction step: $\Phi$-states have no successor violating $\Phi$
Property Directed Reachability

PDR method constructs a sequence of clause sets $\Psi_0, \ldots, \Psi_k$ that overapproximate the states reachable in $0, \ldots, k$ steps.

- **incremental induction:** extend the sequence $\Psi_0, \ldots, \Psi_k$
- **unreachability checking:** identify counterexamples to induction (CTIs)
Property Directed Reachability

PDR method constructs a sequence of clause sets \( \psi_0, \ldots, \psi_k \) that overapproximate the states reachable in \( 0, \ldots, k \) steps.

- incremental induction: extend the sequence \( \psi_0, \ldots, \psi_k \)
- unreachability checking: identify counterexamples to induction (CTIs)

\[
\begin{array}{c|c|c}
\text{Reachable States} & \phi \text{ holds} & \phi \text{ doesn't hold} \\
\hline
s_0: \{\} & \text{green} & \text{green} \\
\hline
s_1: \{p_2\} & \text{green} & \text{orange} \\
\hline
s_2: \{p_1\} & \text{green} & \text{orange} \\
\hline
s_3: \{p_1, p_2\} & \text{green} & \text{orange} \\
\hline
s_4: \{\text{run}\} & \text{green} & \text{orange} \\
\hline
s_5: \{\text{run}, p_2\} & \text{green} & \text{orange} \\
\hline
s_6: \{\text{run}, p_1\} & \text{green} & \text{orange} \\
\hline
s_7: \{\text{run}, p_1, p_2\} & \text{green} & \text{orange} \\
\end{array}
\]
Property Directed Reachability

PDR method constructs a sequence of clause sets $\Psi_0, \ldots, \Psi_k$ that overapproximate the states reachable in $0, \ldots, k$ steps.

- incremental induction: extend the sequence $\Psi_0, \ldots, \Psi_k$
- unreachability checking: identify counterexamples to induction (CTIs)
Outline

1. Motivation

2. Property Directed Reachability

3. Control-flow Guided PDR for Imperative Synchronous Programs
Main Idea: Modify Transition Relation to generate less CTIs

Original Transition Relation:

```
Original Transition Relation:
holds  doesn't hold
Reachable States
s0: {}
s6: {run,p1}
s1: {p2}
s2: {p1}
s7: {run,p1,p2}
s3: {p1,p2}s4: {run}
s5: {run,p2}
```

$s_2$ has successor $s_7$ violating $\Phi$

```
Enhanced Transition Relation:
holds  doesn't hold
Reachable States
s0: {}
s6: {run,p1}
s1: {p2}
s2: {p1}
s7: {run,p1,p2}
s3: {p1,p2}s4: {run}
s5: {run,p2}
```

$s_2$ has no successor

⇒ remove transitions from unreachable states by **control-flow invariants**
Control-flow Invariants by static Analysis

Control-flow can never be active at both substatements of sequences and conditional statements:

```plaintext
module CfSeq(){
    p1: pause;
    p2: pause;
}
¬(p1 ∧ p2)
```
Control-flow can never be active at both sub-statements of sequences and conditional statements:

```plaintext
module Ite(){
    mem bool i;
    if (i) {
        p1: pause;
    } else {
        q1: pause;
    }
}
¬(p1 ∧ q1)
```
Control-flow can never be active at both substatements of sequences and conditional statements:

\[ \neg(p_1 \land p_2) \land \neg(q_1 \land q_2) \land \neg((p_1 \lor p_2) \land (q_1 \lor q_2)) \]
module CfIte()
    mem bool i;
    if (i) {
        p1: pause;
        p2: pause;
    } else {
        q1: pause;
        q2: pause;
    }
}
Control-flow Invariants by static Analysis

module CfIte(){
    mem bool i;
    if (i) {
        p1: pause;
        p2: pause;
    } else {
        q1: pause;
        q2: pause;
    }
}

Enhanced Transition Relation:

with control-flow invariant by static analysis:
\neg(p1 \land p2) \land \neg(q1 \land q2) \land \neg((p1 \lor p2) \land (q1 \lor q2))
module CfPar(){
    {
        p1: pause;
        p2: pause;
    } ||
    {
        q1: pause;
        q2: pause;
    }
}

Original Transition Relation:

Motivation
Property Directed Reachability
Control-flow Guided PDR for Imperative Synchronous Programs

Control-flow Invariants by symbolic Analysis

Reachable States

14 / 17
Control-flow Invariants by **symbolic** Analysis

```
module CfPar(){
  { p1: pause;  p2: pause; } ||
  { q1: pause;  q2: pause; }
}
```

**Enhanced Transition Relation:**

with control-flow invariant by **static** analysis:

\[ \neg(p_1 \land p_2) \land \neg(q_1 \land q_2) \]
Symbolic traversal of the state space of the control-flow system:

\[
\neg (p_1 \land p_2) \land \neg (q_1 \land q_2) \land \neg ((p_1 \lor p_2) \land (q_1 \lor q_2))
\]

module CfPar(){
{
  p1: pause;
  p2: pause;
}
||
{
  q1: pause;
  q2: pause;
}
}

Control-flow Invariants by symbolic Analysis
Control-flow Invariants by **symbolic** Analysis

```plaintext
module CfPar(){
  {
    p1: pause;
p2: pause;
  } ||
  {
    q1: pause;
    q2: pause;
  }
}
```

Enhanced Transition Relation:

<table>
<thead>
<tr>
<th>State</th>
<th>Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0: {}</td>
<td></td>
</tr>
<tr>
<td>s1: {q2}</td>
<td></td>
</tr>
<tr>
<td>s2: {q1}</td>
<td></td>
</tr>
<tr>
<td>s3: {q1,q2}</td>
<td></td>
</tr>
<tr>
<td>s4: {p2}</td>
<td></td>
</tr>
<tr>
<td>s5: {p2,q2}</td>
<td></td>
</tr>
<tr>
<td>s6: {p2,q1}</td>
<td></td>
</tr>
<tr>
<td>s7: {p2,q1,q2}</td>
<td></td>
</tr>
<tr>
<td>s8: {p1}</td>
<td></td>
</tr>
<tr>
<td>s9: {p1,q2}</td>
<td></td>
</tr>
<tr>
<td>s10: {p1,q1}</td>
<td></td>
</tr>
<tr>
<td>s11: {p1,q1,q2}</td>
<td></td>
</tr>
<tr>
<td>s12: {p1,p2}</td>
<td></td>
</tr>
<tr>
<td>s13: {p1,p2,q2}</td>
<td></td>
</tr>
<tr>
<td>s14: {p1,p2,q1}</td>
<td></td>
</tr>
<tr>
<td>s15: {p1,p2,q1,q2}</td>
<td></td>
</tr>
<tr>
<td>s16: {run}</td>
<td></td>
</tr>
<tr>
<td>s17: {run,q2}</td>
<td></td>
</tr>
<tr>
<td>s18: {run,q1}</td>
<td></td>
</tr>
<tr>
<td>s19: {run,q1,q2}</td>
<td></td>
</tr>
<tr>
<td>s20: {run,p2}</td>
<td></td>
</tr>
<tr>
<td>s21: {run,p2,q2}</td>
<td></td>
</tr>
<tr>
<td>s22: {run,p2,q1}</td>
<td></td>
</tr>
<tr>
<td>s23: {run,p2,q1,q2}</td>
<td></td>
</tr>
<tr>
<td>s24: {run,p1}</td>
<td></td>
</tr>
<tr>
<td>s25: {run,p1,q2}</td>
<td></td>
</tr>
<tr>
<td>s26: {run,p1,q1}</td>
<td></td>
</tr>
<tr>
<td>s27: {run,p1,q1,q2}</td>
<td></td>
</tr>
<tr>
<td>s28: {run,p1,p2}</td>
<td></td>
</tr>
<tr>
<td>s29: {run,p1,p2,q2}</td>
<td></td>
</tr>
<tr>
<td>s30: {run,p1,p2,q1}</td>
<td></td>
</tr>
<tr>
<td>s31: {run,p1,p2,q1,q2}</td>
<td></td>
</tr>
</tbody>
</table>

Reachable States

with control-flow invariant by **symbolic** analysis:

$$\neg (p1 \land p2) \land \neg (q1 \land q2) \land \neg ((p1 \lor p2) \land (q1 \lor q2))$$
Control-flow Guided PDR for Imperative Synchronous Programs

- two methods for reachable control-flow states computation
  - linear-time static analysis
  - symbolic reachability analysis

⇒ different precision and runtime complexities

- enhanced transition relation makes PDR more efficient
  ⇒ save arbitrarily many incrementation steps of PDR