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

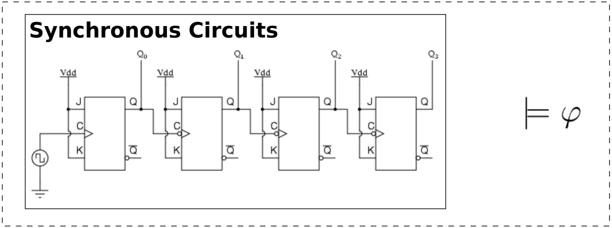
Table of Contents

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

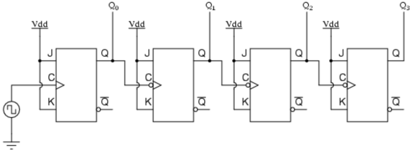
Synchronous Programs

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

$\models \varphi'$

synthesize

Synchronous Circuits



$\models \varphi$

Imperative Synchronous Programs

Imperative Synchronous Languages: e.g. Quartz

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

$\Rightarrow$ 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

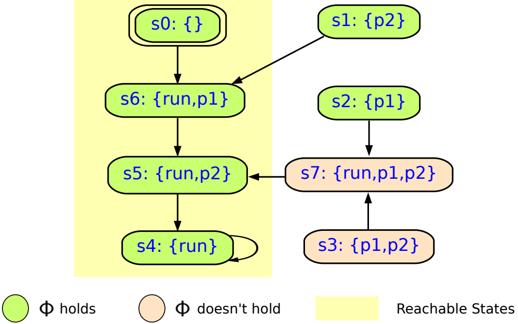
Outline

1. Motivation
2. Property Directed Reachability
3. Control-flow Guided PDR for Imperative Synchronous Programs

Safety Property Verification

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

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




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

$$
\begin{aligned}
\mathcal{V} &:= \{\mathrm{run}, \mathrm{p1}, \mathrm{p2}\} \\
\mathcal{I} &:= \neg(\mathrm{run} \vee \mathrm{p1} \vee \mathrm{p2}) \\
\mathcal{T} &:= \quad \mathbf{next}(\mathrm{run}) \leftrightarrow \mathbf{true} \\
&\quad \wedge (\mathbf{next}(\mathrm{p1}) \leftrightarrow \neg\mathrm{run}) \\
&\quad \wedge (\mathbf{next}(\mathrm{p2}) \leftrightarrow \mathrm{p1})
\end{aligned}
$$

$$\Phi := \neg(\mathrm{p1} \wedge \mathrm{p2})$$

Safety Property Verification by Induction

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

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

Φ is inductive w.r.t. $\mathcal{K}$

- induction base: Φ holds in all initial states
- induction step: Φ-states have no successor violating Φ

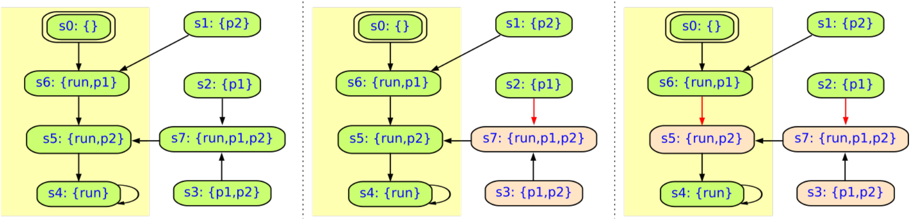
Safety Property Verification by Induction

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

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

Φ is inductive w.r.t. $\mathcal{K}$

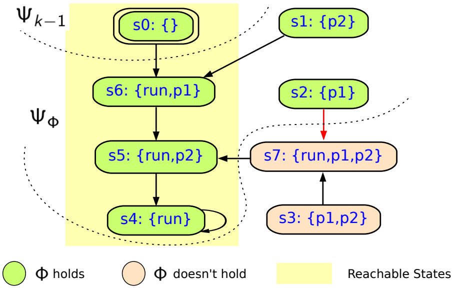
- induction base: Φ holds in all initial states
- induction step: Φ-states have no successor violating Φ



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.

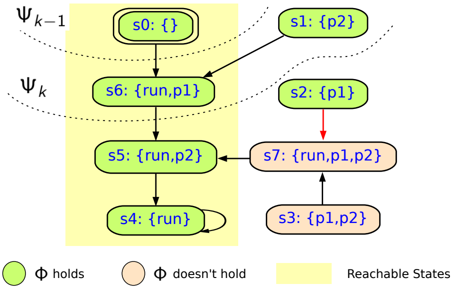
- incremental induction: extend the sequence $\Psi_0, \ldots, \Psi_k$
- unreachability checking:
  indentify counterexamples to induction (CTIs)



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.

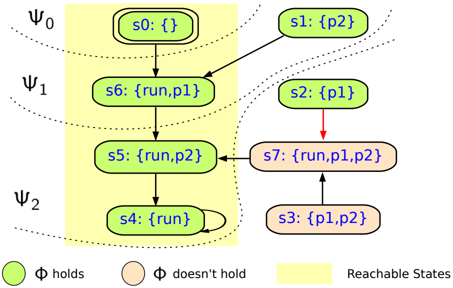
- incremental induction: extend the sequence $\Psi_0, \ldots, \Psi_k$
- unreachability checking: indentify counterexamples to induction (CTIs)



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:
  indentify counterexamples to induction (CTIs)

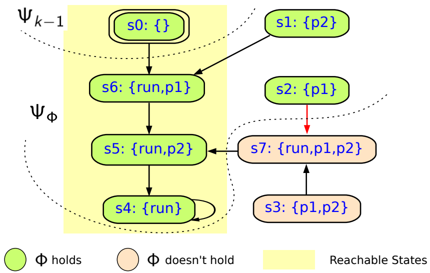


Outline

1. Motivation
2. Property Directed Reachability
3. Control-flow Guided PDR for Imperative Synchronous Programs

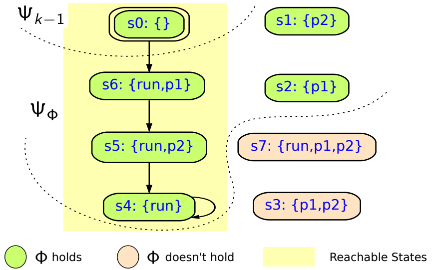
Main Idea: Modify Transition Relation to generate less CTIs

Original Transition Relation:



s_2 has successor s_7 violating Φ

Enhanced Transition Relation:



s_2 has no successor

$\Rightarrow$ 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:

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

$$\neg(\mathrm{p1} \wedge \mathrm{p2})$$

Control-flow Invariants by static Analysis

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

```
module Ite(){
  mem bool i;
  if (i) {
    p1: pause;
  } else {
    q1: pause;
  }
}
```

$$\neg(p1 \wedge q1)$$

Control-flow Invariants by static Analysis

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

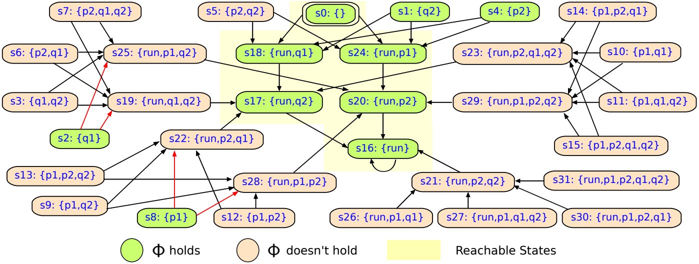
```
module CfIte(){
  mem bool i;
  if (i) {
    p1: pause;
    p2: pause;
  } else {
    q1: pause;
    q2: pause;
  }
}
```

$$\neg(p1 \wedge p2) \wedge \neg(q1 \wedge q2) \wedge \neg((p1 \vee p2) \wedge (q1 \vee q2))$$

Control-flow Invariants by static Analysis

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

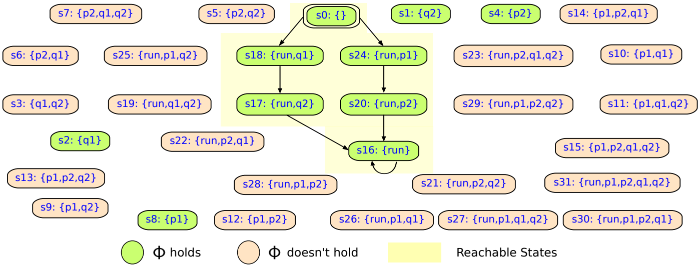
Original Transition Relation:



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:



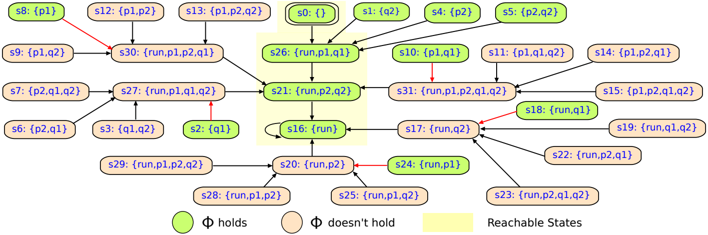
with control-flow invariant by static analysis:

$\neg(p1 \wedge p2) \wedge \neg(q1 \wedge q2) \wedge \neg((p1 \vee p2) \wedge (q1 \vee q2))$

Control-flow Invariants by symbolic Analysis

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

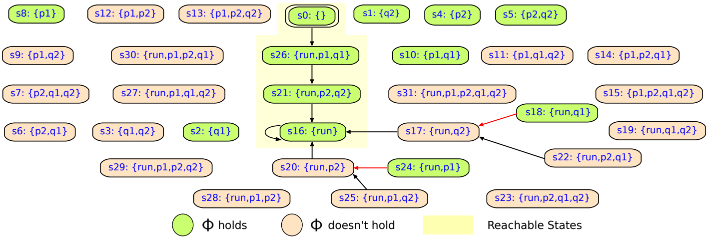
Original Transition Relation:



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:

$\neg(p1 \wedge p2) \wedge \neg(q1 \wedge q2)$

Control-flow Invariants by symbolic Analysis

Symbolic traversal of the state space of the control-flow system:

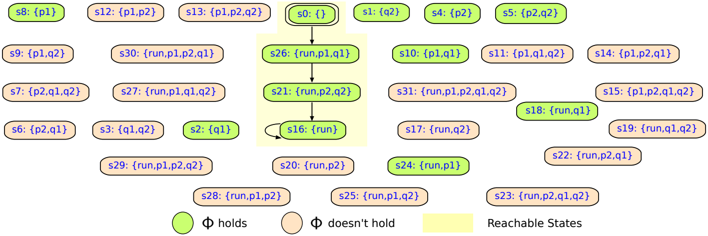
```
module CfPar(){
  {
    p1: pause;
    p2: pause;
  } ||
  {
    q1: pause;
    q2: pause;
  }
}
```

$$\neg(p1 \wedge p2) \wedge \neg(q1 \wedge q2) \wedge \neg((p1 \vee p2) \wedge (q1 \vee q2))$$

Control-flow Invariants by symbolic Analysis

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

Enhanced Transition Relation:



with control-flow invariant by symbolic analysis:

$$\neg(p1 \wedge p2) \wedge \neg(q1 \wedge q2) \wedge \neg((p1 \vee p2) \wedge (q1 \vee q2))$$

Summary

Control-flow Guided PDR for Imperative Synchronous Programs

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

  $\Rightarrow$ different precision and runtime complexities
- enhanced transition relation makes PDR more efficient

  $\Rightarrow$ save arbitrarily many incrementation steps of PDR