Induction-based Verification of Synchronous and Hybrid Programs

Xian Li

Embedded Systems Chair
Department of Computer Science
University of Kaiserslautern

November 10th 2017

Contributions

Induction-based VCG methods for Synchronous and Hybrid Programs

- users choose a VCG method and provide inductive assertions
- VCs are generated automatically for induction bases and steps
- external SMT solvers verify the VCs

Control-flow Guided PDR for Synchronous Programs

- modify transition relation to generate less CTIs
- identify CTIs with simpler unreachability tests in $\mathcal{K}^{\text{cf}}$
- generalize CTIs by omitting dataflow literals

Table of Contents

Outline

1. Motivation
   - Synchronous and Hybrid Programs
   - Proving Safety Properties
2. VCG using Inductive Assertions
   - VCG using Control-flow Assertions
   - VCG using SCC Assertions
   - VCG using Loop Assertions
3. Control-flow Guided PDR Optimizations
   - Transition Relation Modification
   - CTI Identification and Generalization
4. Summary

Embedded Reactive Systems in Physical Environment

discrete reactions by embedded reactive system

- program variables are assigned by the system
- program variables = output of the system
- no physical time is consumed

⇒ **synchronous reactive system**

continuous phase by physical environment

- environment variables defined by differential equations
- environment variables = input of the system
- physical time is consumed

⇒ **hybrid system**

Averest and Quartz

Averest (www.averest.org)

- Tool-set for the development of reactive systems

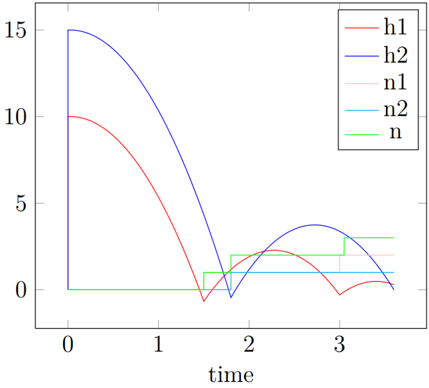
Quartz [Schneider, 2009] [Bauer, 2012]

- Synchronous language for modeling, simulation, and verification of hybrid systems

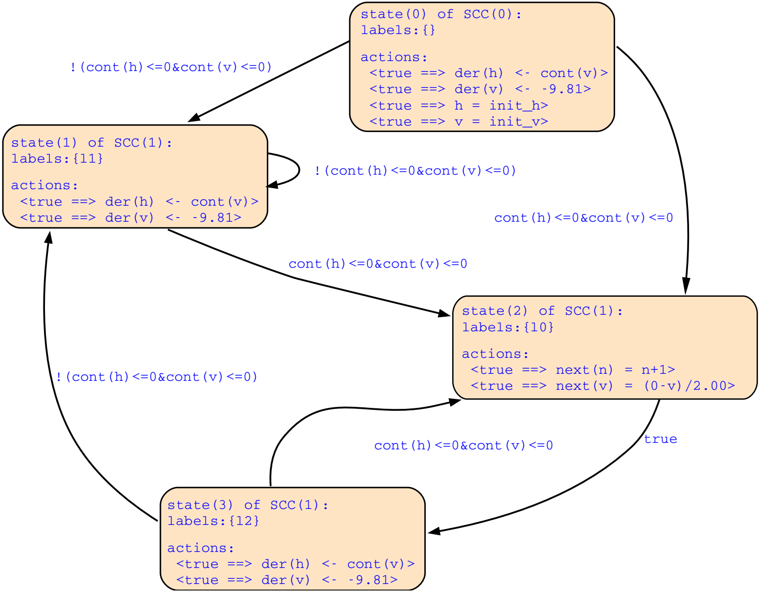
Bouncing Ball

```
module Ball(real ? init_h,? init_v, int n)
{
  hybrid real h, v;
  h = init_h; v = init_v;
  loop{
   l0,l1:flow{
     drv(h) <- cont(v);
     drv(v) <- -9.81;
   }until(cont(h)<=0 and cont(v)<=0);
  next(v) = -v/2.0;
  next(n) = n + 1 ;
  l2,l3:flow {} until(true);
  }
}

module TwoBalls(){
  int n, n1, n2;
  Ball(15.0,0,0,n1);
  ||
  Ball(10.0,1.0,n2)
  ||
  loop{ n = n1+n2; pause; }
}
```



Bouncing Ball



Outline

1. Motivation
   - Synchronous and Hybrid Programs
   - Proving Safety Properties
2. VCG using Inductive Assertions
   - VCG using Control-flow Assertions
   - VCG using SCC Assertions
   - VCG using Loop Assertions
3. Control-flow Guided PDR Optimizations
   - Transition Relation Modification
   - CTI Identification and Generalization
4. Summary

The Satisfiability Problem

SMT Problem

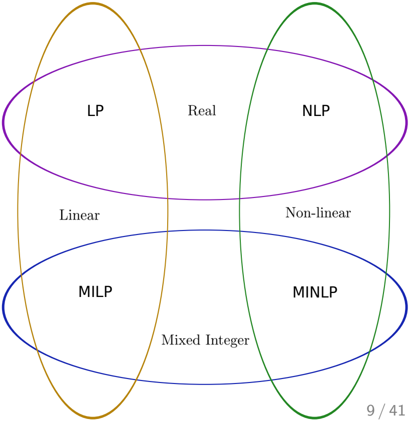
- combinations of propositional logic and non-linear arithmetic theories over integers and reals with $\exists$-quantifiers.

Constraint Problem

- The general form of a MINLP:

$$\begin{array}{lll} \text{minimize} & f(\vec{x},\vec{y}) & \\ \text{subject to} & g(\vec{x},\vec{y}) \leq 0 & \\ & \vec{x}_l \leq \vec{x} \leq \vec{x}_u & x_i \in \mathbb{R} \\ & \vec{y}_l \leq \vec{y} \leq \vec{y}_u & y_i \in \mathbb{Z} \end{array}$$

$f(\vec{x},\vec{y})$, $g(\vec{x},\vec{y})$: nonlinear functions
e.g. $g(x,y) = x^2 + xy^2$



The Satisfiability Problem

SMT Problem

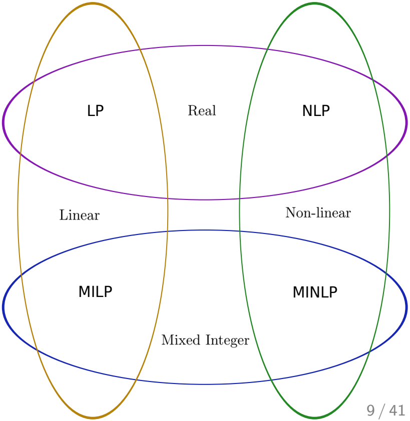
- combinations of propositional logic and non-linear arithmetic theories over integers and reals with $\exists$-quantifiers.

Constraint Problem

- The general form of a MILP:

$$\begin{array}{lll} \text{minimize} & f(\vec{x},\vec{y}) & \\ \text{subject to} & g(\vec{x},\vec{y}) \leq 0 & \\ & \vec{x}_l \leq \vec{x} \leq \vec{x}_u & x_i \in \mathbb{R} \\ & \vec{y}_l \leq \vec{y} \leq \vec{y}_u & y_i \in \mathbb{Z} \end{array}$$

$f(\vec{x},\vec{y})$, $g(\vec{x},\vec{y})$: linear functions
e.g. $g(x,y) = ax + by$



The Satisfiability Problem

SMT Problem

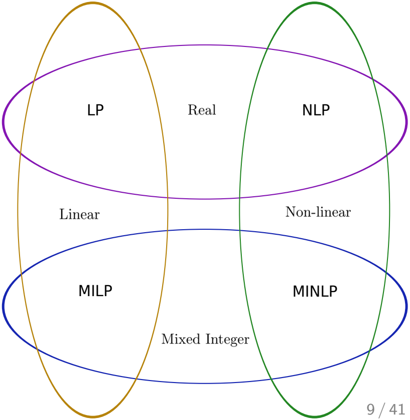
- combinations of propositional logic and non-linear arithmetic theories over integers and reals with ∃-quantifiers.

Constraint Problem

- The general form of a NLP:

$$\begin{aligned} \text{minimize} \quad & f(\vec{x}) \\ \text{subject to} \quad & g(\vec{x}) \leq 0 \\ & \vec{x}_l \leq \vec{x} \leq \vec{x}_u \quad x_i \in \mathbb{R} \end{aligned}$$

$f(\vec{x}, \vec{y})$, $g(\vec{x}, \vec{y})$: nonlinear functions
e.g. $g(x, y) = x^2 + xy^2$



The Satisfiability Problem

SMT Problem

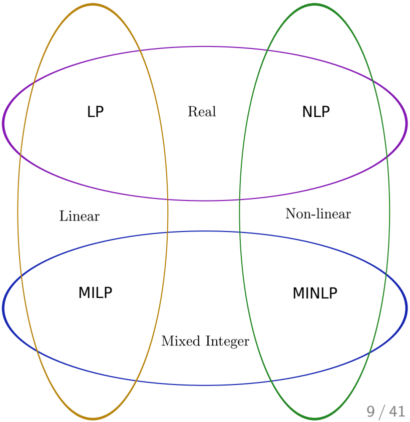
- combinations of propositional logic and non-linear arithmetic theories over integers and reals with ∃-quantifiers.

Constraint Problem

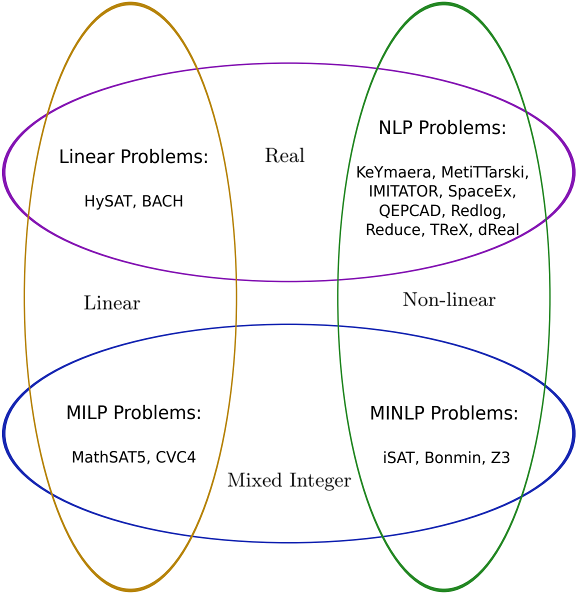
- The general form of a MINLP:

$$\begin{array}{lll} \text{minimize} & f(\vec{x},\vec{y}) & \\ \text{subject to} & g(\vec{x},\vec{y}) \le 0 & \\ & \vec{x}_l \le \vec{x} \le \vec{x}_u & x_i \in \mathbb{R} \\ & \vec{y}_l \le \vec{y} \le \vec{y}_u & y_i \in \mathbb{Z} \end{array}$$

$f(\vec{x},\vec{y})$, $g(\vec{x},\vec{y})$: nonlinear functions
e.g. $g(x,y) = x^2 + xy^2$



Decidability and Tools



Safety Property Verification

model checking

- reachability of states is undecidable
- approximation and abstraction

theorem proving

- interaction with users
- set up proof goals and apply proof rules until a proof is obtained
- Hoare calculus: $\{\varphi\}\ S\ \{\psi\}$ for Software Verification
  - users provide invariants, pre- and postconditions
  - verification condition generation (VCG): automatic
  - proving VCs is done separately, e.g., using SMT solvers

Hoare calculus for Synchronous Programs

Hoare calculus can not be directly applied! [Gesell, 2014]

- impossible to decompose proof goal along the program syntax
  - abstraction of several micro steps to one macro step
  - control-flow can rest at many places at the same time
  - micro steps may correspond to different places in the program
- unless using goto statements or additional label variables

$\Rightarrow$ **VCG using Inductive Assertions**

Property Directed Reachability

PDR

- very efficient verification method for hardware circuit verification

Synchronous Languages: e.g. Quartz

- high-level languages for hardware synthesis

$\Rightarrow$ **Control-flow Guided PDR Optimizations**

Property Directed Reachability

PDR

- very efficient verification method for hardware circuit verification
- relies on good estimation of the reachable states

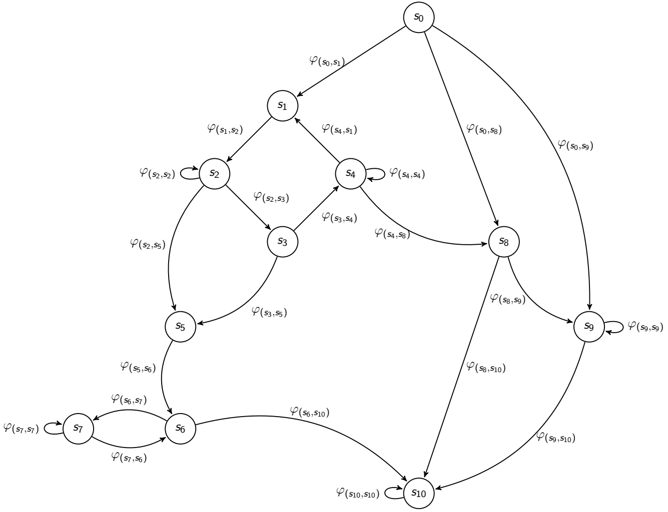
Synchronous Languages: e.g. Quartz

- high-level languages for hardware synthesis
- useful control-flow information for verification

$\Rightarrow$ **Control-flow Guided PDR Optimizations**

Outline

EFSM Decomposition by Control-flow States



VCG using Control-flow Assertions

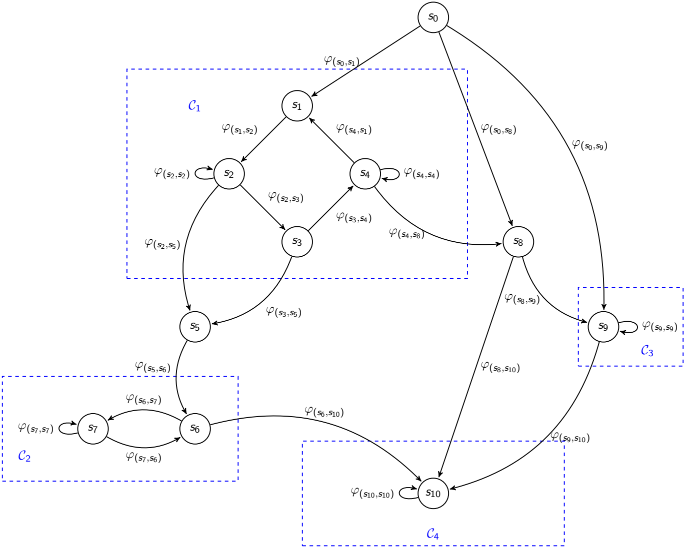
$$\frac{\mathcal{I}_s \rightarrow \Phi}{\Psi_{\text{reach}} \rightarrow \Phi}$$

Transition-based method:

- users provide $\mathcal{I}_s$
- induction base:
  - $\mathcal{I}_{s_{\text{root}}}$ holds on the initial control-flow state
- induction step:
  - $\mathcal{I}_s$ holds on each non-initial control-flow state
    enumerate transitions from one node to the other

Outline

EFSM Decomposition by SCCs



VCG using SCC Assertions

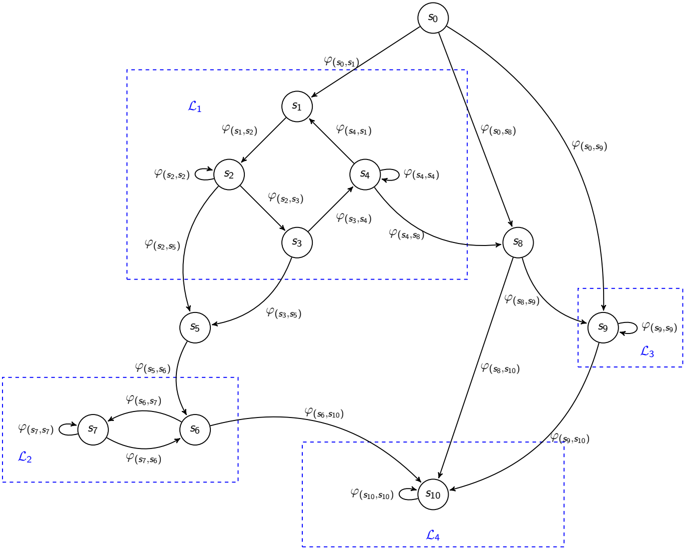
$$\frac{\mathcal{I}_{C_0}(s_{\text{root}}) \qquad s \in C_i \vdash \mathcal{I}_{C_i}(s) \qquad \mathcal{I}_{C_i} \to \Phi}{\Psi_{\text{reach}} \to \Phi}$$

SCC-Path and **SCC-Trans** methods:

- users provide $\mathcal{I}_{C_i}$
- induction base:
  - $\mathcal{I}_{C_i}$ holds on each entering state(s) of C_i
    enumerate paths/transitions from one SCC to the other
- induction step:
  - $\mathcal{I}_{C_i}$ is preserved for the transitions inside C_i
    enumerate transitions from one node to the other inside the same SCC

Outline

EFSM Decomposition by Loop Statemetns




```
...
loop{
  p0: pause;
  ...
  pn: pause;
};
loop{
  q0: pause;
  ...
  qn: pause;
};
...
```

VCG using Loop Assertions

$$\frac{\mathcal{I}_{L_0}(s_{\text{root}}) \qquad s \in L_i \vdash \mathcal{I}_{L_i}(s) \qquad \mathcal{I}_{L_i} \to \Phi}{\Psi_{\text{reach}} \to \Phi}$$

Loop-Path and **Loop-Trans** methods:

- users provide $\mathcal{I}_{L_i}$
- induction base:
  - $\mathcal{I}_{L_i}$ holds on each entering states of L_i
    enumerate paths/transitions from one loop statement to the other
- induction step:
  - $\mathcal{I}_{L_i}$ is preserved for the transitions inside L_i
    enumerate transitions from one node to the other related to the same loop statement

Summary: VCG using Inductive Assertions

Induction-based VCG methods for Synchronous and Hybrid Programs

- users choose a VCG method and provide inductive assertions
- VCs are generated automatically for induction bases and steps
- external SMT solvers verify the VCs

Outline

1. Motivation
   - Synchronous and Hybrid Programs
   - Proving Safety Properties
2. VCG using Inductive Assertions
   - VCG using Control-flow Assertions
   - VCG using SCC Assertions
   - VCG using Loop Assertions
3. Control-flow Guided PDR Optimizations
   - Transition Relation Modification
   - CTI Identification and Generalization
4. Summary

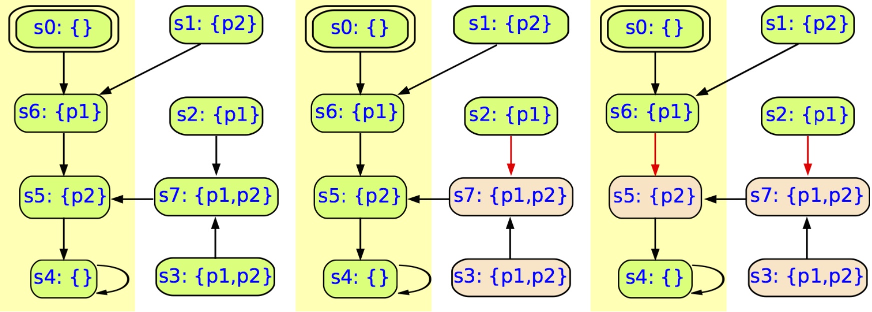
Safety Property Verification by Induction

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
- all reachable states of $\mathcal{K}$ are Φ-states

Φ is inductive w.r.t. $\mathcal{K}$

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



Property Directed Reachability

PDR

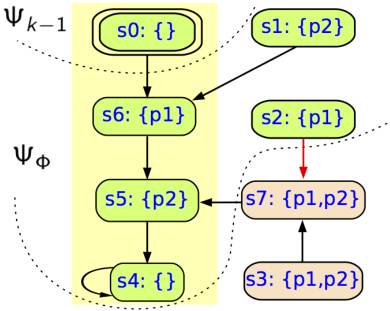
- counterexamples to induction (CTIs) identification and generalization
- relies on good estimation of the reachable states

Control-flow of Synchronous Program

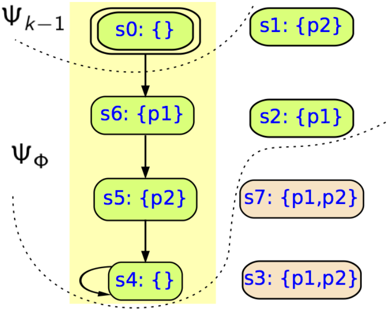
- not needed for synthesis
- useful for formal verification

Heuristic: Modify Transition Relation to generate less CTIs

Original Transition Relation:



Enhanced Transition Relation:



⇒ remove transitions from unreachable states by **control-flow invariants**

- linear-time static analysis
- symbolic reachability analysis restricted to control-flow

Control-flow Invariants by static Analysis

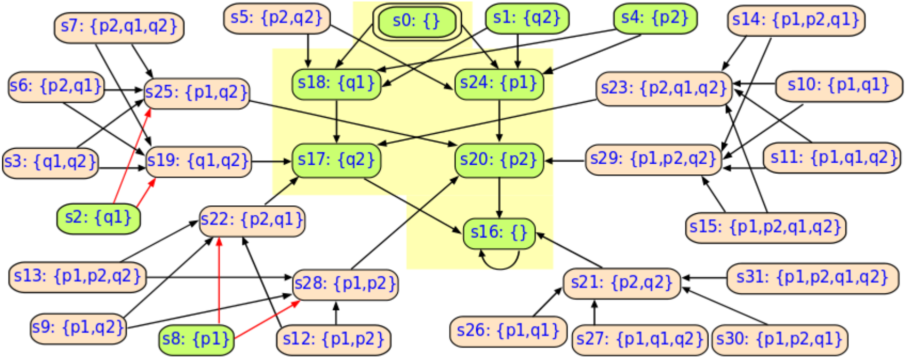
Control-flow can never be active at both substatements of sequences or conditional statements:

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

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

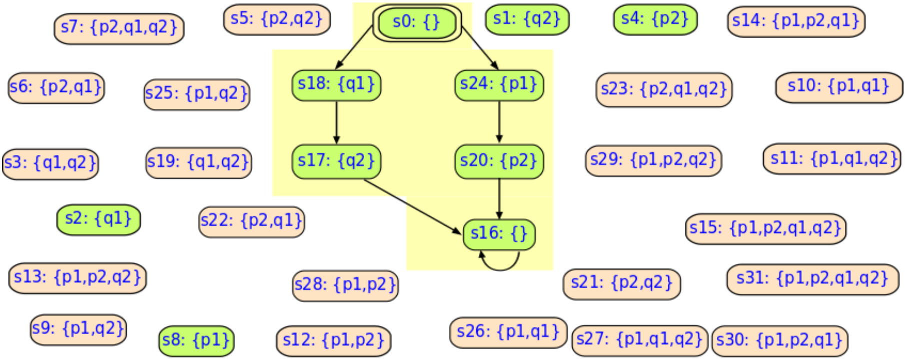
Control-flow Invariants by static Analysis

Original Transition Relation



Control-flow Invariants by static Analysis

Enhanced Transition Relation



with control-flow invariant by static analysis:

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

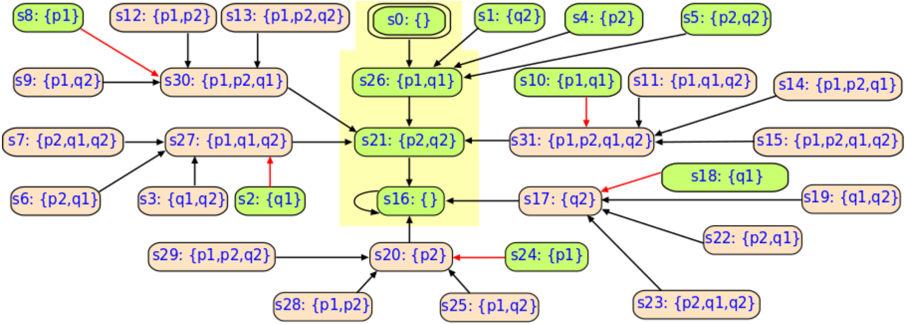
Control-flow Invariants by symbolic Analysis

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

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

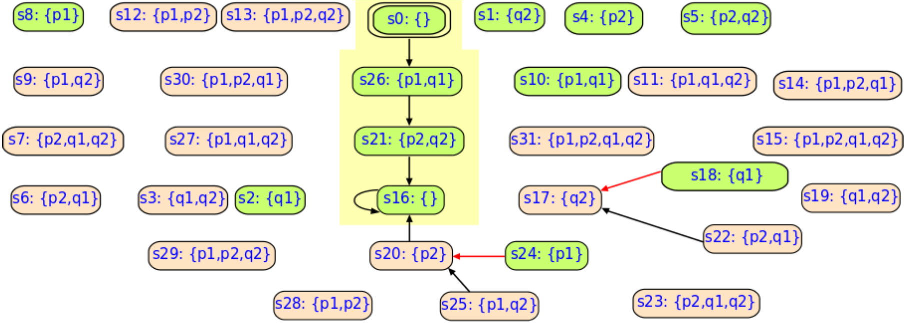
Control-flow Invariants by symbolic Analysis

Original Transition Relation



Control-flow Invariants by symbolic Analysis

Enhanced Transition Relation

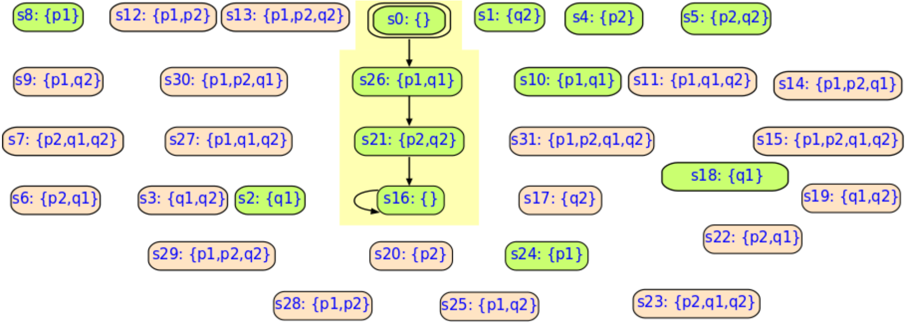


with control-flow invariant by static analysis:

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

Control-flow Invariants by symbolic Analysis

Enhanced Transition Relation



with control-flow invariant by symbolic analysis:

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

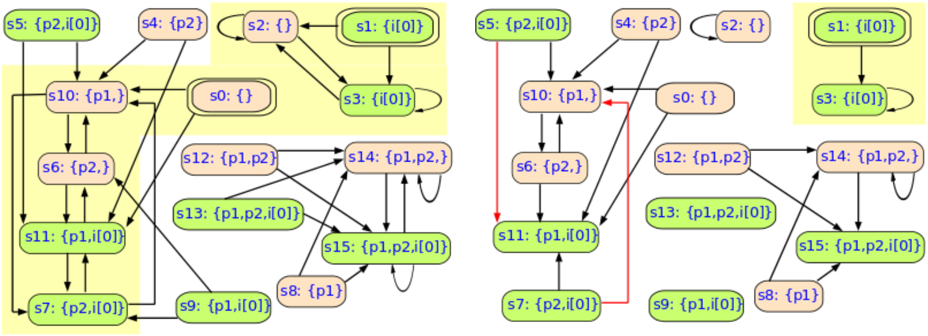
Outline

1. Motivation
   - Synchronous and Hybrid Programs
   - Proving Safety Properties
2. VCG using Inductive Assertions
   - VCG using Control-flow Assertions
   - VCG using SCC Assertions
   - VCG using Loop Assertions
3. Control-flow Guided PDR Optimizations
   - Transition Relation Modification
   - CTI Identification and Generalization
4. Summary

CTI Identification and Generalization

$\mathcal{K} := \mathcal{K}^{\text{cf}} \times \mathcal{K}^{\text{df}}$

- symbolic representation of $\mathcal{K}^{\text{cf}}$ is simpler than $\mathcal{K}$
- $\mathcal{K}^{\text{cf}}$ conserves the approximation of unreachability in $\mathcal{K}$



CTI Identification and Generalization

$\mathcal{K} := \mathcal{K}^{\text{cf}} \times \mathcal{K}^{\text{df}}$

- symbolic representation of $\mathcal{K}^{\text{cf}}$ is simpler than $\mathcal{K}$
- $\mathcal{K}^{\text{cf}}$ conserves the approximation of unreachability in $\mathcal{K}$

unreachability of CTIs in $\mathcal{K}$ can be proved by unreachability in $\mathcal{K}^{\text{cf}}$

- reachability of CTIs in $\mathcal{K}$
  simpler unreachability tests in $\mathcal{K}^{\text{cf}}$

unreachability in $\mathcal{K}^{\text{cf}}$ is independent on the dataflows

- generalize CTIs to narrow the reachable state approximations
  if $\mathcal{C}$ is unreachable in $\mathcal{K}^{\text{cf}}$, then generalize $\neg\mathcal{C}'$ instead of $\neg\mathcal{C}$:
  $\mathcal{C}' := \mathcal{C}_{|\mathcal{V}^{\text{cf}}}$ obtained from omitting the dataflow literals in $\mathcal{C}$

Summary: Control-flow Guided PDR Optimizations

Control-flow Guided PDR for Synchronous Programs

- modify transition relation to generate less CTIs by reachable control-flow states computation
  - linear-time static analysis
  - symbolic reachability analysis restricted to control-flow

  $\Rightarrow$ different precision and runtime complexities
- identify CTIs with simpler unreachability tests in $\mathcal{K}^{\text{cf}}$
- generalize CTIs by omitting dataflow literals

Outline

1. Motivation
   - Synchronous and Hybrid Programs
   - Proving Safety Properties
2. VCG using Inductive Assertions
   - VCG using Control-flow Assertions
   - VCG using SCC Assertions
   - VCG using Loop Assertions
3. Control-flow Guided PDR Optimizations
   - Transition Relation Modification
   - CTI Identification and Generalization
4. Summary

Contributions

Induction-based VCG methods for Synchronous and Hybrid Programs

- users choose a VCG method and provide inductive assertions
- VCs are generated automatically for induction bases and steps
- external SMT solvers verify the VCs

Control-flow Guided PDR for Synchronous Programs

- modify transition relation to generate less CTIs
- identify CTIs with simpler unreachability tests in $\mathcal{K}^{\mathrm{cf}}$
- generalize CTIs by omitting dataflow literals

Appendix: VCG Methods Comparison

Assertions Numbers

- Loop-based $\geq$ SCC-based $\geq$ Transition-based

Assertions Information

- SCC-based $\approx$ Transition-based $\geq$ Loop-based

Appendix: Backend Tools and Input VC-formats Comparison

Execution Time

- EFSM-Inv Time
- VCG Time
- SMT Time

VC Formats

- $\sum$-Format
- $\bigwedge$-Format
- $\bigvee$-Format

SMT Solvers

- iSAT
- Z3
- Z3 API
- Z3 API async

Appendix: Backend Tools and Input VC-formats Comparison

Execution Time

- EFSM-Inv Time
- VCG Time
- SMT Time

VC Formats

- $\sum$-Format
- $\bigwedge$-Format
- $\bigvee$-Format

SMT Solvers

- iSAT
- Z3
- Z3 API
- Z3 API async

Appendix: Backend Tools and Input VC-formats Comparison

Execution Time

- EFSM-Inv Time
- VCG Time
- SMT Time

VC Formats

- $\sum$-Format
- $\bigwedge$-Format
- $\bigvee$-Format

SMT Solvers

- iSAT
- Z3
- Z3 API
- Z3 API async

Appendix: Backend Tools and Input VC-formats Comparison

Execution Time

- EFSM-Inv Time
- VCG Time
- SMT Time

VC Formats

- $\sum$-Format
- $\bigwedge$-Format
- $\bigvee$-Format

SMT Solvers

- iSAT
- Z3
- Z3 API
- Z3 API async

Appendix: CTI Generalization Example

```
macro N=?;
module ITELoop() {
  [N]bool i;
  i[0] = true;
  if (!i[0]) {
    loop{
      p1: pause;
      i[0] = false;
      p2: pause;
    }
  }
}
```

The set of boolean variables of module ITELoop

$$\mathcal{V}_{\texttt{N}} \;:=\; \underbrace{\{\texttt{i[0]},\ldots,\texttt{i[N-1]}\}}_{\mathcal{V}^{\mathsf{df}}} \cup \underbrace{\{\texttt{p1},\texttt{p2},\texttt{run}\}}_{\mathcal{V}^{\mathsf{cf}}}$$

$\Rightarrow$ reduce at most $2^{\texttt{N}+3}$ to 2^3 times relative inductiveness reasoning

Synchronous Model of Computation

- Macro steps : consumption of 1 logical time unit
- Micro steps : no logical time consumption

Statements of Quartz (incomplete) [Schneider, 2009] [Bauer, 2012]

$x = \tau$ and $\mathbf{next}(x) = \tau$	(assignments)
$\mathbf{assume}(\varphi)$, $\mathbf{assert}(\varphi)$	(assumptions and assertions)
$\ell : \mathbf{pause}$	(start/end of macro step)
$S_1; S_2$	(sequences)
$S_1 \parallel S_2$	(synchronous concurrency)
$\mathbf{if}(\sigma) S_1 \mathbf{else} S_2$	(conditional)
$\mathbf{do} S \mathbf{while}(\sigma)$	(loops)
$\{\alpha\ S\}$	(local variable)
$M([params])$	(module call)
$\mathbf{flow}\{S_1; ...; S_N;\}\mathbf{until}(\sigma)$	(flow statements)
x<-τ	(continuous assignments)
$\mathbf{drv}(x)$<-τ	(derivative assignments)

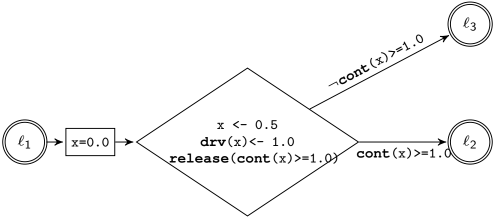
Synchronous Model of Computation

- Macro steps : consumption of 1 logical time unit
- Micro steps : no logical time consumption

Code Fragment

```
l0:pause;
   x = 1.0;
   next(y) = x;
l1:pause;
   x = 0.0;
l2,l3:flow{
     x <- 0.5;
     drv(x)<- 1.0;
   }until(cont(x)>=1.0);
```

EFSM



Synchronous Model of Computation

- Macro steps : consumption of 1 logical time unit
- Micro steps : no logical time consumption

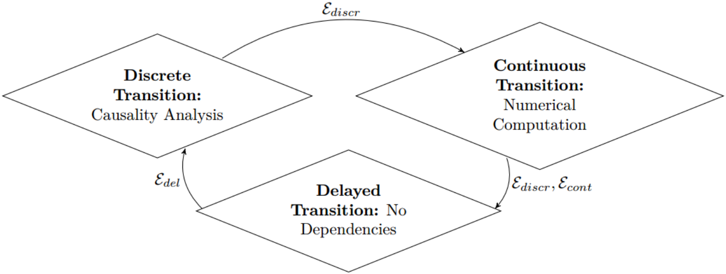


Figure 4.6: Execution of a Macro Step of Hybrid Quartz