



Symbolic Methods for Formal Verification of Industrial Control Software

Dimitri Bohlender, M. Sc. RWTH

i11 Summercolloquium, Schmidt, 16th August 2019

Outline

Introduction

- Formal Methods
- Setting
- Approach

CHC-based Verification

Design and Verification of Restart-robust Software

Software-driven Systems



- ▶ Software drives the systems we rely on – hardware often off-the-shelf
- ▶ While many software bugs are not grave, some may be catastrophic:
 - Misinterpretation & no input validation led to radiation fatalities [Bor06]
 - Blackout after race condition affected 50 million people [Pow04]
- ▶ Writing “correct” software is hard – 50% of resources in testing [Mye12]

Software-driven Systems



- ▶ Software drives the systems we rely on – hardware often off-the-shelf
- ▶ While many software bugs are not grave, some may be catastrophic:
 - Misinterpretation & no input validation led to radiation fatalities [Bor06]
 - Blackout after race condition affected 50 million people [Pow04]
- ▶ Writing “correct” software is hard – 50% of resources in testing [Mye12]

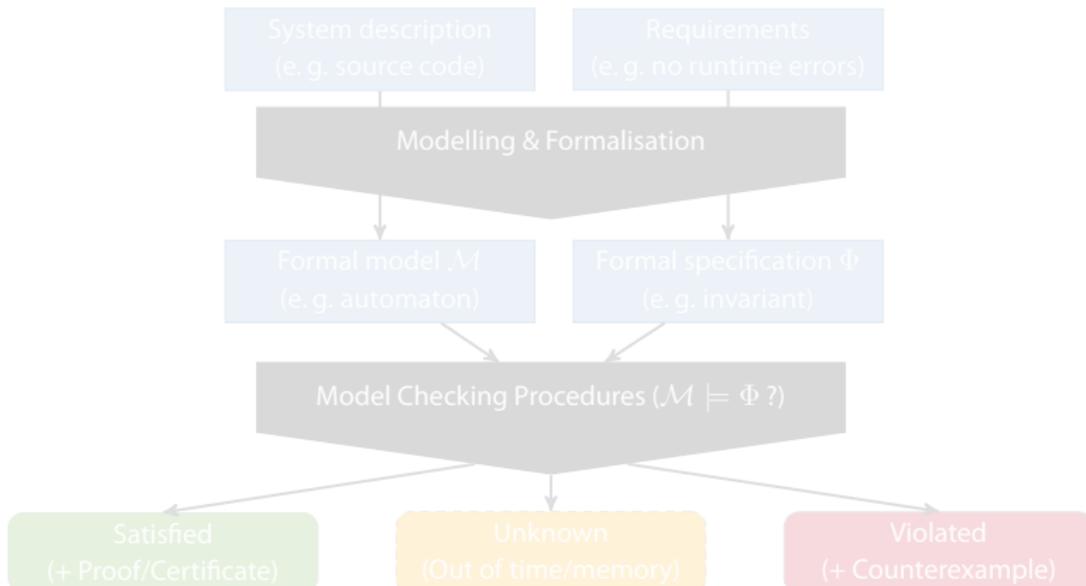
Software-driven Systems



- ▶ Software drives the systems we rely on – hardware often off-the-shelf
- ▶ While many software bugs are not grave, some may be catastrophic:
 - Misinterpretation & no input validation led to radiation fatalities [Bor06]
 - Blackout after race condition affected 50 million people [Pow04]
- ▶ Writing “correct” software is hard – 50% of resources in testing [Mye12]

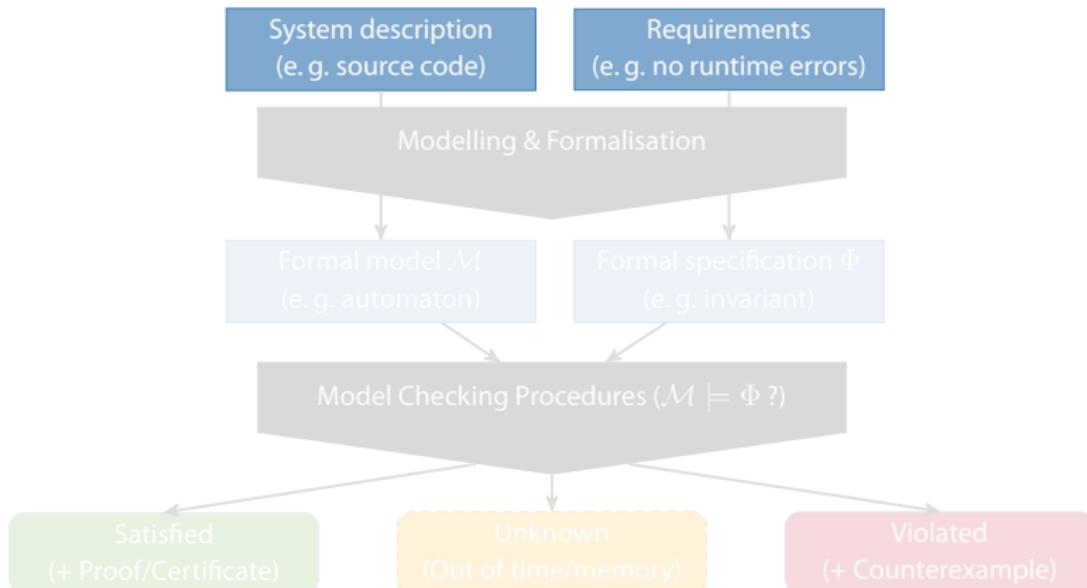
Formal Methods

- ▶ Based on mathematics, they enable **rigorous modelling & reasoning**
- ▶ Model checking (dis-)proves properties of interest



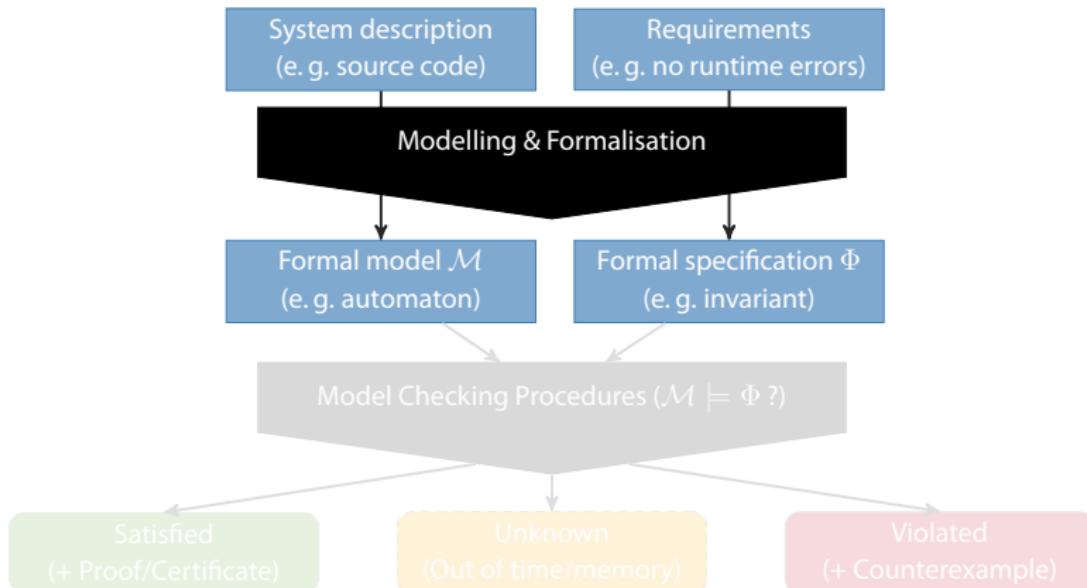
Formal Methods

- ▶ Based on mathematics, they enable **rigorous modelling & reasoning**
- ▶ Model checking (dis-)proves properties of interest



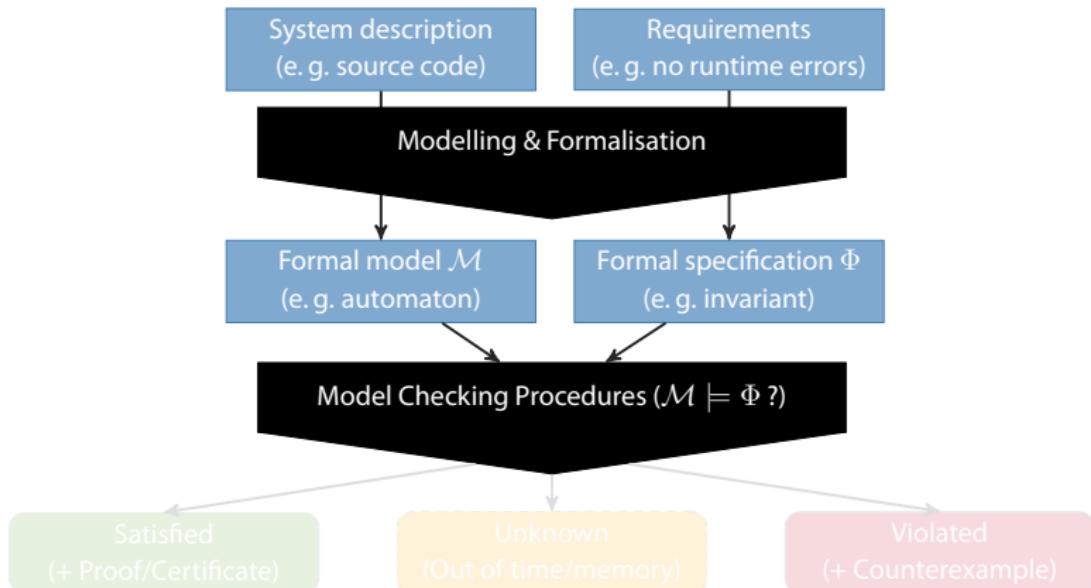
Formal Methods

- ▶ Based on mathematics, they enable **rigorous modelling & reasoning**
- ▶ Model checking (dis-)proves properties of interest



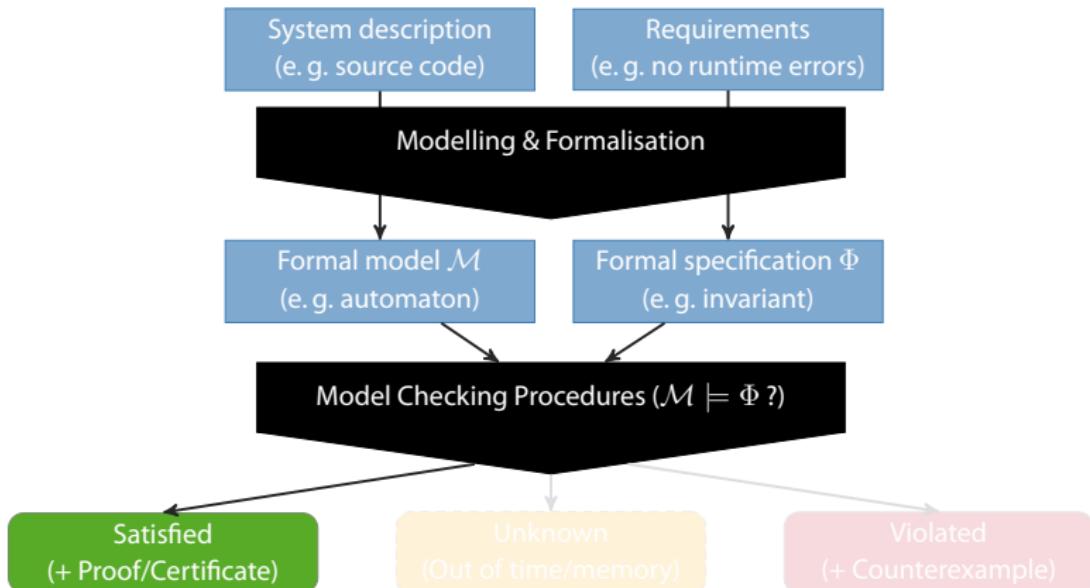
Formal Methods

- Based on mathematics, they enable **rigorous modelling & reasoning**
- Model checking (**dis-**)proves properties of interest



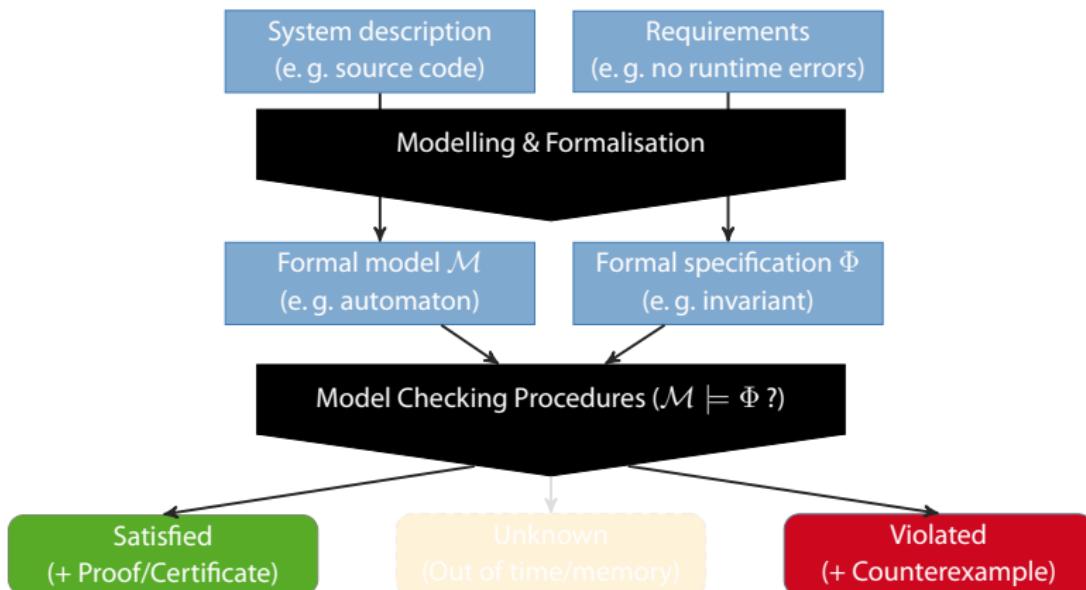
Formal Methods

- Based on mathematics, they enable **rigorous modelling & reasoning**
- Model checking (**dis-**)proves properties of interest



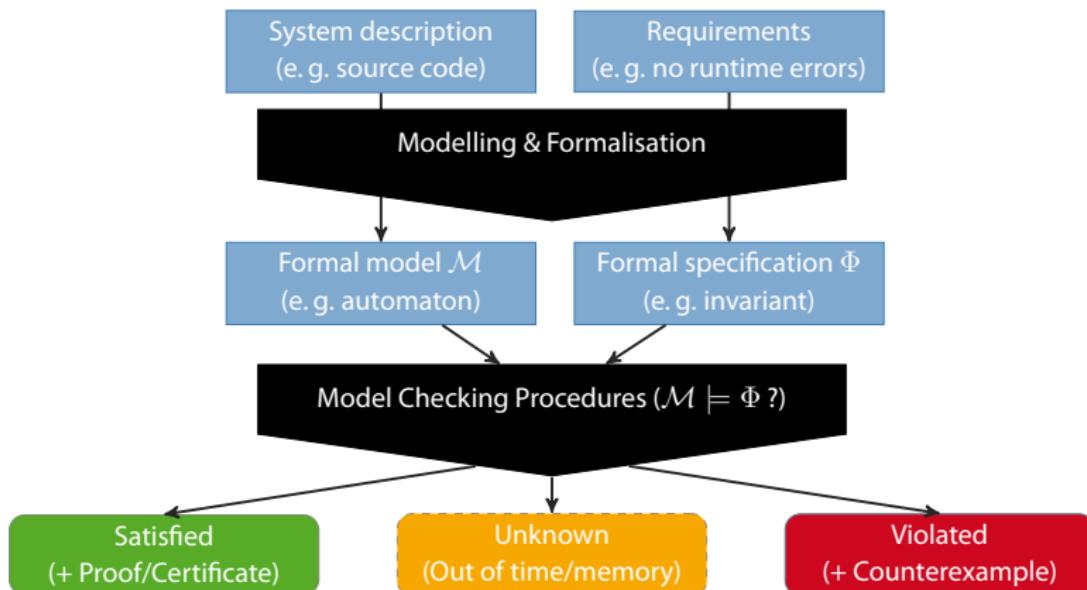
Formal Methods

- Based on mathematics, they enable **rigorous modelling & reasoning**
- Model checking (**dis-**)proves properties of interest

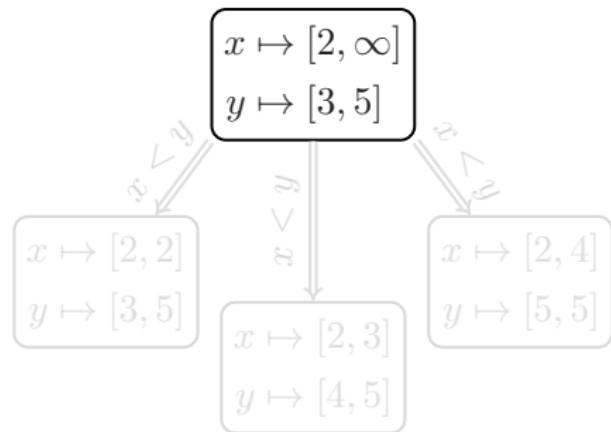


Formal Methods

- Based on mathematics, they enable **rigorous modelling & reasoning**
- Model checking (**dis-)proves** properties of interest



Explicit vs. Symbolic Methods



$$\text{src}(x, y) := 2 \leq x$$

$$\wedge 3 \leq y \leq 5$$

$$T(x, y, x', y') := x < y$$

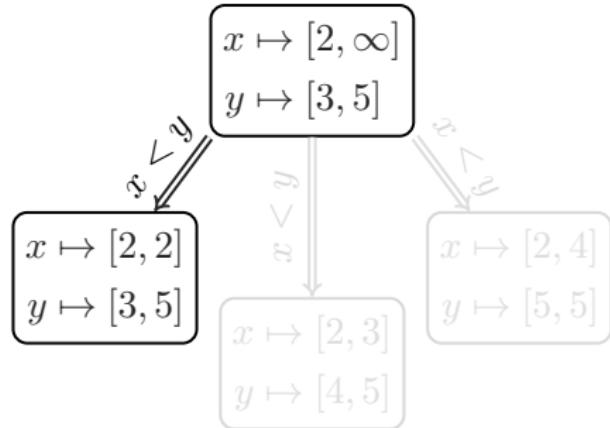
$$\wedge x' = x \wedge y' = y$$

$$\text{bad}(x', y') := x = 4$$

- ▶ Explicit construction & search
- ▶ Processes one state at a time
- ▶ Precise representation needs space

- ▶ Implicit reasoning via SAT
- ▶ Operates on sets of states
- ▶ Exploits structure & symmetry

Explicit vs. Symbolic Methods



$$\text{src}(x, y) := 2 \leq x$$

$$\wedge 3 \leq y \leq 5$$

$$T(x, y, x', y') := x < y$$

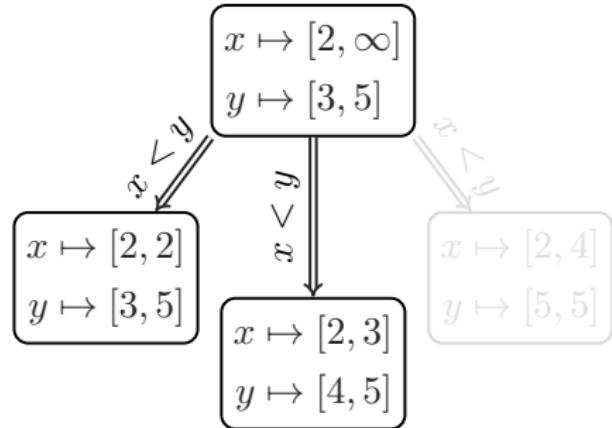
$$\wedge x' = x \wedge y' = y$$

$$\text{bad}(x', y') := x = 4$$

- ▶ Explicit construction & search
- ▶ Processes one state at a time
- ▶ Precise representation needs space

- ▶ Implicit reasoning via SAT
- ▶ Operates on sets of states
- ▶ Exploits structure & symmetry

Explicit vs. Symbolic Methods



$$\text{src}(x, y) := 2 \leq x$$

$$\wedge 3 \leq y \leq 5$$

$$T(x, y, x', y') := x < y$$

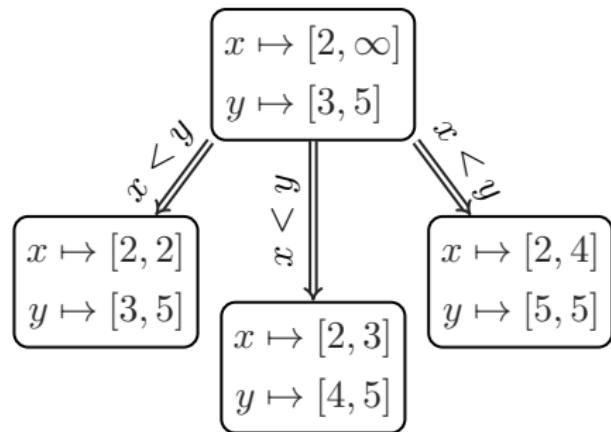
$$\wedge x' = x \wedge y' = y$$

$$\text{bad}(x', y') := x' = 4$$

- ▶ Explicit construction & search
- ▶ Processes one state at a time
- ▶ Precise representation needs space

- ▶ Implicit reasoning via SAT
- ▶ Operates on sets of states
- ▶ Exploits structure & symmetry

Explicit vs. Symbolic Methods



$$\text{src}(x, y) := 2 \leq x$$

$$\wedge 3 \leq y \leq 5$$

$$T(x, y, x', y') := x < y$$

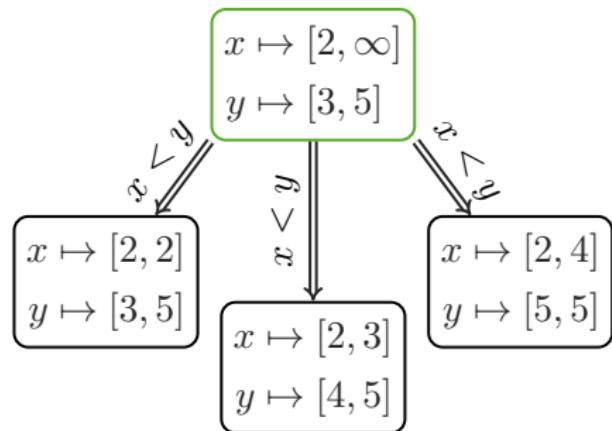
$$\wedge x' = x \wedge y' = y$$

$$\text{bad}(x', y') := x' = 4$$

- ▶ Explicit construction & search
- ▶ Processes one state at a time
- ▶ Precise representation needs space

- ▶ Implicit reasoning via SAT
- ▶ Operates on sets of states
- ▶ Exploits structure & symmetry

Explicit vs. Symbolic Methods



$$\text{src}(x, y) := 2 \leq x$$

$$\wedge 3 \leq y \leq 5$$

$$T(x, y, x', y') := x < y$$

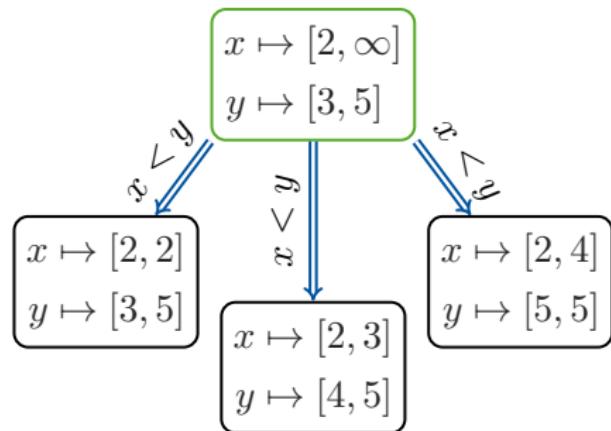
$$\wedge x' = x \wedge y' = y$$

$$\text{bad}(x', y') := x' = 4$$

- ▶ Explicit construction & search
- ▶ Processes one state at a time
- ▶ Precise representation needs space

- ▶ Implicit reasoning via SAT
- ▶ Operates on sets of states
- ▶ Exploits structure & symmetry

Explicit vs. Symbolic Methods



$$\text{src}(x, y) := 2 \leq x$$

$$\wedge 3 \leq y \leq 5$$

$$T(x, y, x', y') := x < y$$

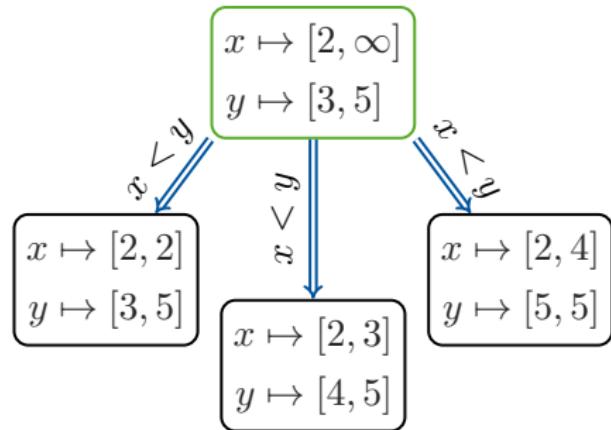
$$\wedge x' = x \wedge y' = y$$

$$\text{bad}(x', y') := x = 4$$

- ▶ Explicit construction & search
- ▶ Processes one state at a time
- ▶ Precise representation needs space

- ▶ Implicit reasoning via SAT
- ▶ Operates on sets of states
- ▶ Exploits structure & symmetry

Explicit vs. Symbolic Methods



$$\text{src}(x, y) := 2 \leq x$$

$$\wedge 3 \leq y \leq 5$$

$$T(x, y, x', y') := x < y$$

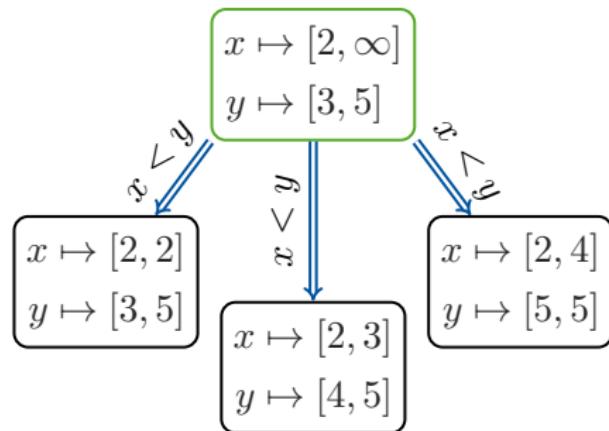
$$\wedge x' = x \wedge y' = y$$

$$\text{bad}(x', y') := x' = 4$$

- ▶ Explicit construction & search
- ▶ Processes one state at a time
- ▶ Precise representation needs space

- ▶ Implicit reasoning via SAT
- ▶ Operates on sets of states
- ▶ Exploits structure & symmetry

Explicit vs. Symbolic Methods

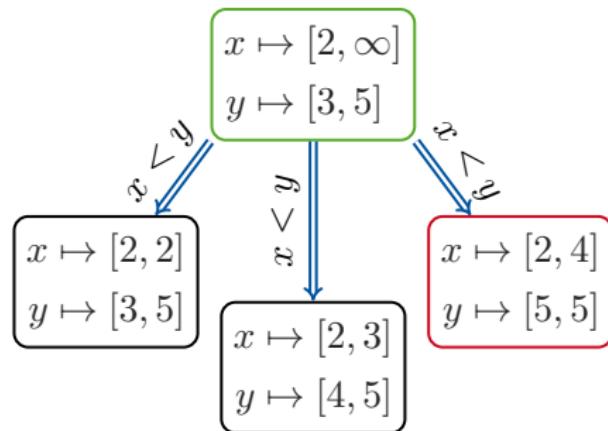


$$\begin{aligned} & \text{src}(x, y) \\ \wedge \\ & T(x, y, x', y') \\ \wedge \\ & \text{bad}(x', y') \end{aligned}$$

- ▶ Explicit construction & search
- ▶ Processes one state at a time
- ▶ Precise representation needs space

- ▶ Implicit reasoning via SAT
- ▶ Operates on sets of states
- ▶ Exploits structure & symmetry

Explicit vs. Symbolic Methods



$$\text{src}(x, y) := 2 \leq x$$

$$\wedge 3 \leq y \leq 5$$

$$T(x, y, x', y') := x < y$$

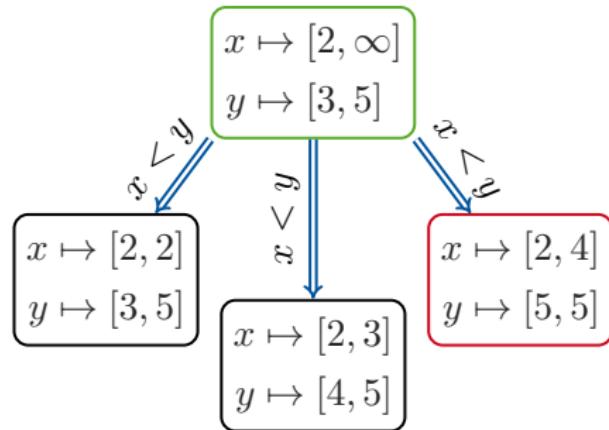
$$\wedge x' = x \wedge y' = y$$

$$\text{bad}(x', y') := x = 4$$

- ▶ **Explicit** construction & search
- ▶ Processes one state at a time
- ▶ Precise representation needs space

- ▶ **Implicit** reasoning via SAT
- ▶ Operates on sets of states
- ▶ Exploits structure & symmetry

Explicit vs. Symbolic Methods



$$\text{src}(x, y) := 2 \leq x$$

$$\wedge 3 \leq y \leq 5$$

$$T(x, y, x', y') := x < y$$

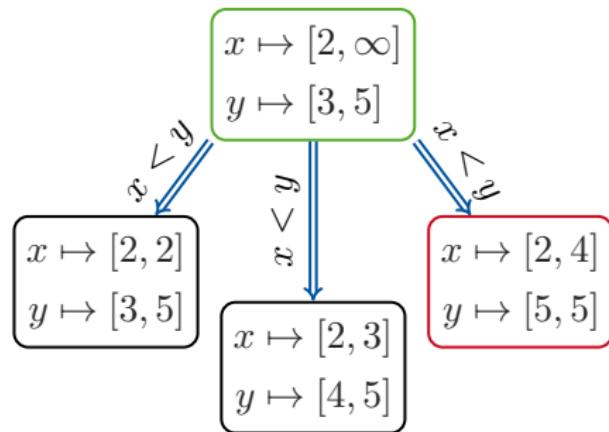
$$\wedge x' = x \wedge y' = y$$

$$\text{bad}(x', y') := x' = 4$$

- ▶ **Explicit** construction & search
- ▶ Processes **one state** at a time
- ▶ Precise representation needs space

- ▶ **Implicit** reasoning via SAT
- ▶ Operates on **sets** of states
- ▶ Exploits structure & symmetry

Explicit vs. Symbolic Methods



$$\text{src}(x, y) := 2 \leq x$$

$$\wedge 3 \leq y \leq 5$$

$$T(x, y, x', y') := x < y$$

$$\wedge x' = x \wedge y' = y$$

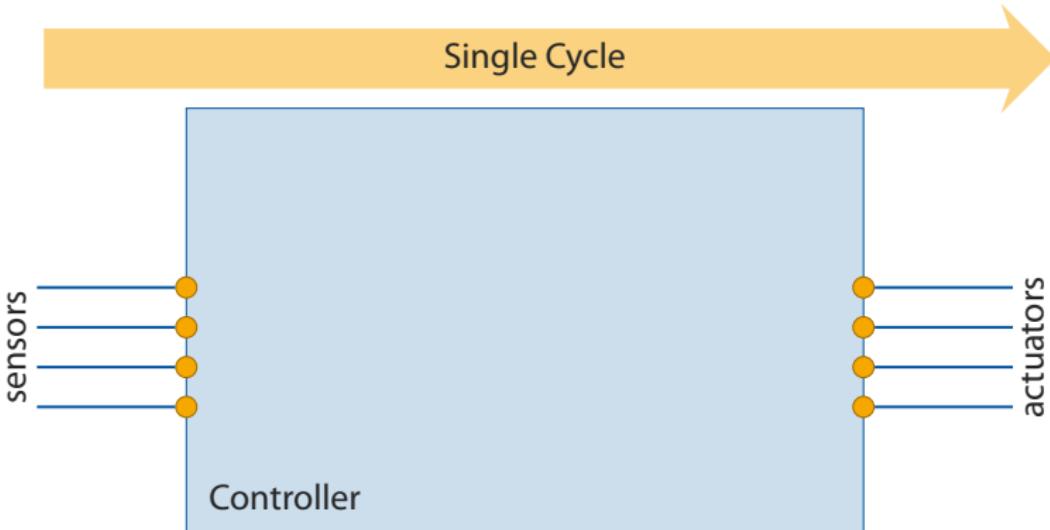
$$\text{bad}(x', y') := x' = 4$$

- ▶ **Explicit** construction & search
- ▶ Processes **one state** at a time
- ▶ Precise representation **needs space**

- ▶ **Implicit** reasoning via SAT
- ▶ Operates on **sets** of states
- ▶ Exploits **structure & symmetry**

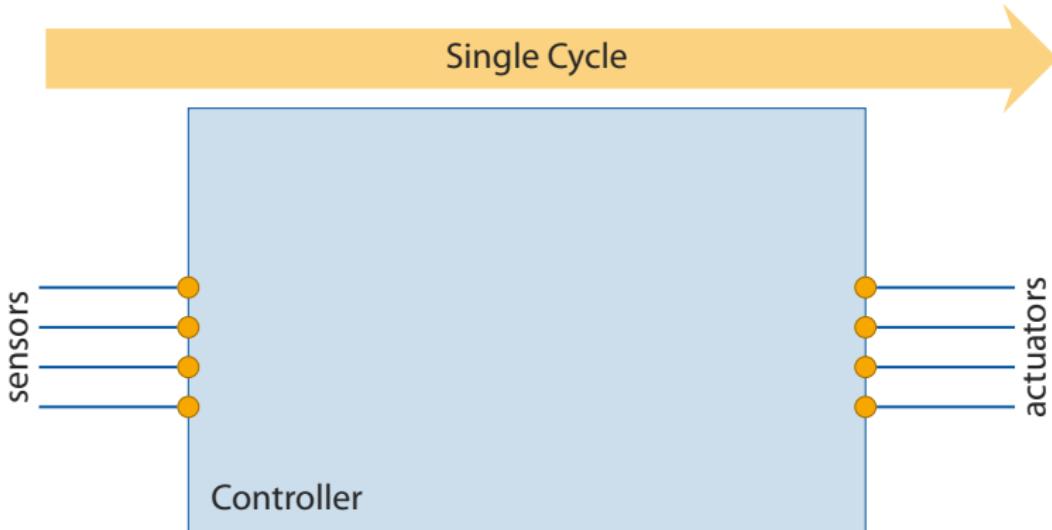
Programmable Logic Controllers (PLCs)

- ▶ Controllers realise **reactive systems**, repeatedly executing some task
- ▶ PLCs are rugged computers especially tailored to **industrial control**, e.g. for actuating assembly lines



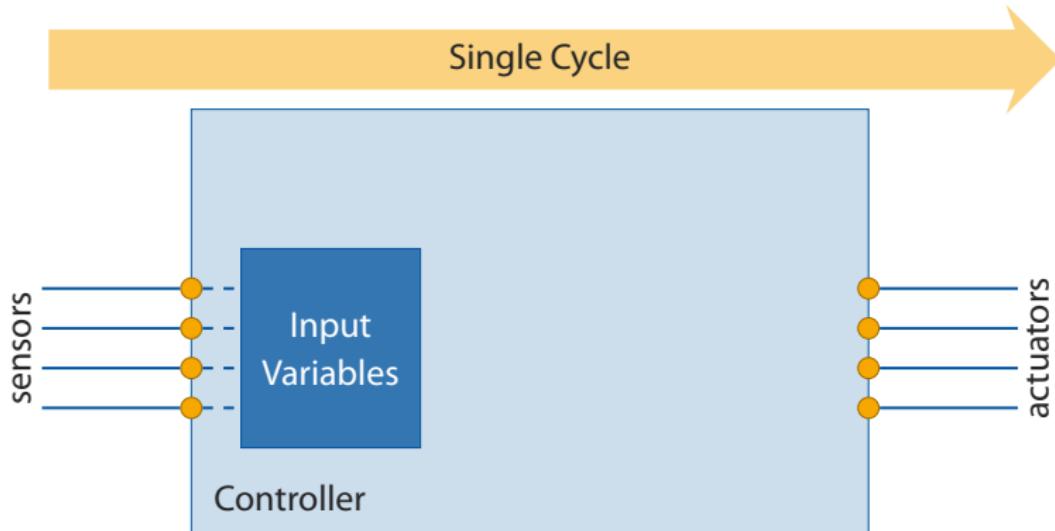
Programmable Logic Controllers (PLCs)

- ▶ Controllers realise **reactive systems**, repeatedly executing some task
- ▶ PLCs are rugged computers especially tailored to **industrial control**, e. g. for actuating assembly lines



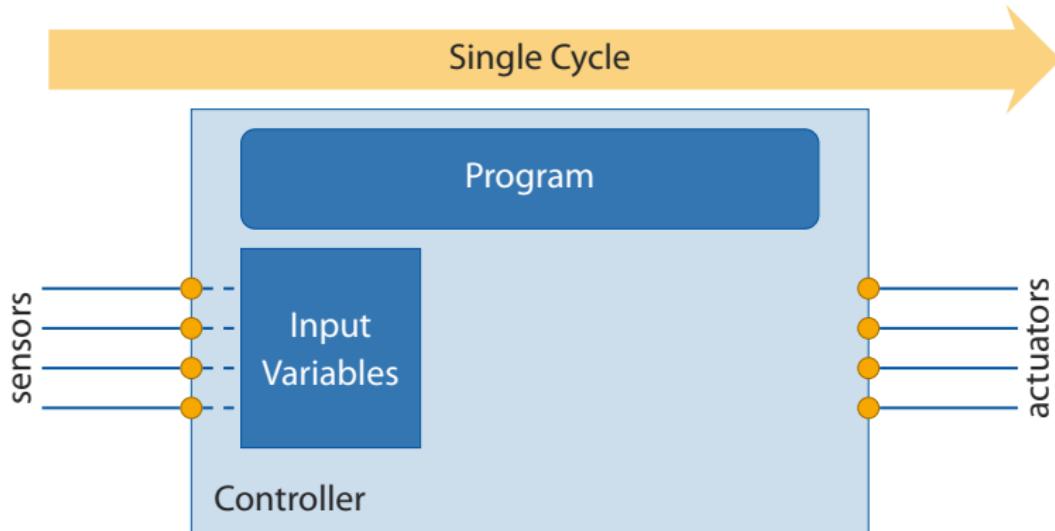
Programmable Logic Controllers (PLCs)

- ▶ Controllers realise **reactive systems**, repeatedly executing some task
- ▶ PLCs are rugged computers especially tailored to **industrial control**, e. g. for actuating assembly lines



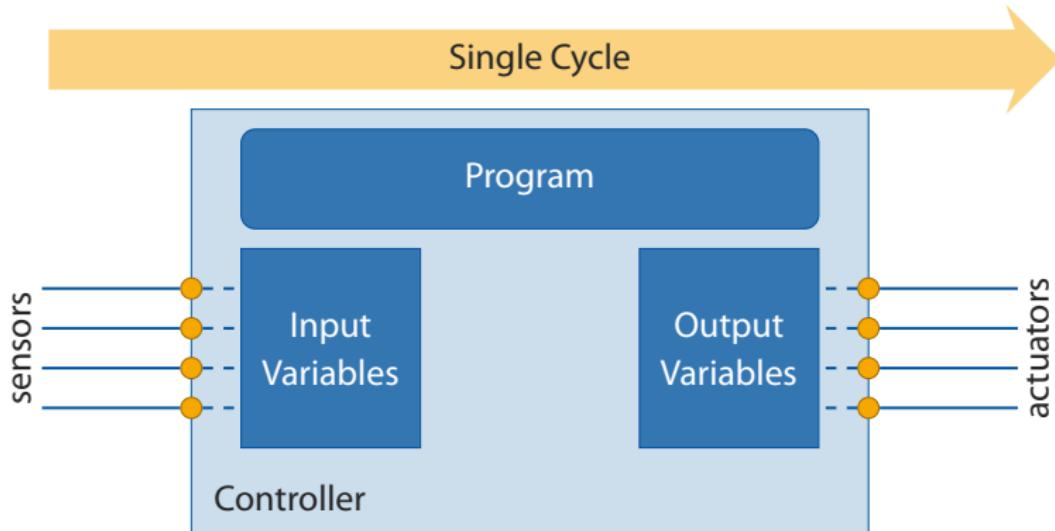
Programmable Logic Controllers (PLCs)

- ▶ Controllers realise **reactive systems**, repeatedly executing some task
- ▶ PLCs are rugged computers especially tailored to **industrial control**, e. g. for actuating assembly lines



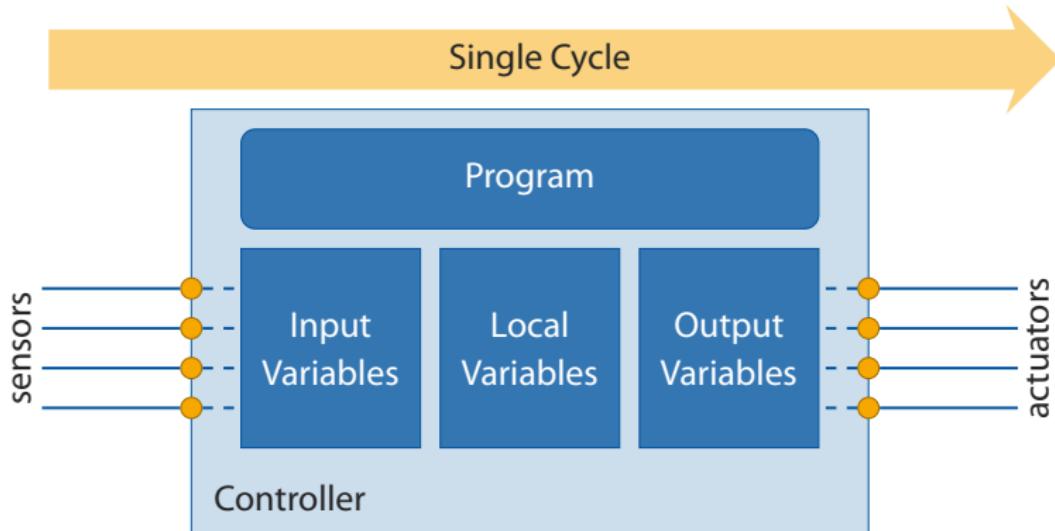
Programmable Logic Controllers (PLCs)

- ▶ Controllers realise **reactive systems**, repeatedly executing some task
- ▶ PLCs are rugged computers especially tailored to **industrial control**, e. g. for actuating assembly lines



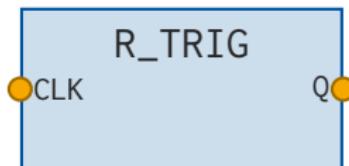
Programmable Logic Controllers (PLCs)

- ▶ Controllers realise **reactive systems**, repeatedly executing some task
- ▶ PLCs are rugged computers especially tailored to **industrial control**, e. g. for actuating assembly lines



PLC Software

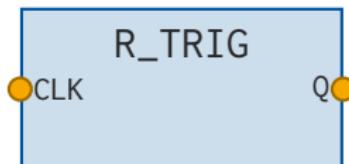
```
1 PROGRAM R_TRIG
2   VAR_INPUT
3     CLK:BOOL;
4   END_VAR
5   VAR
6     M:BOOL;
7   END_VAR
8   VAR_OUTPUT
9     Q:BOOL;
10  END_VAR
11
12  IF CLK AND NOT M THEN
13    Q:=TRUE;
14  ELSE
15    Q:=FALSE;
16  END_IF;
17  M:=CLK;
18 END_PROGRAM
```



- ▶ Programmed via textual and graphical languages from the IEC 61131-3
- ▶ Modularisation via function blocks
- ▶ Has static allocation and no recursion

PLC Software

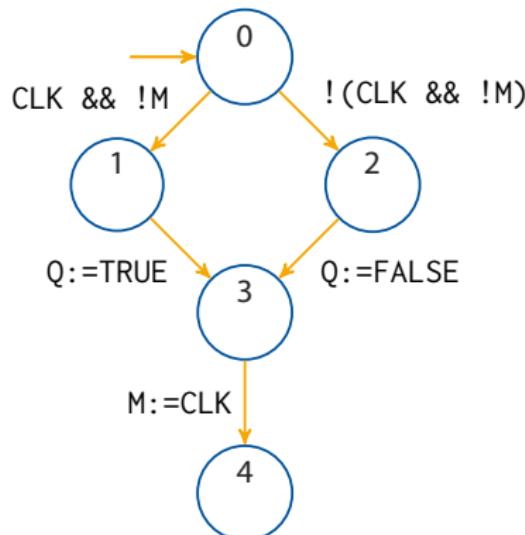
```
1  FUNCTION_BLOCK R_TRIG
2    VAR_INPUT
3      CLK: BOOL;
4    END_VAR
5    VAR
6      M: BOOL;
7    END_VAR
8    VAR_OUTPUT
9      Q: BOOL;
10   END_VAR
11
12  IF CLK AND NOT M THEN
13    Q:=TRUE;
14  ELSE
15    Q:=FALSE;
16  END_IF;
17  M:=CLK;
18 END_FUNCTION_BLOCK
```



- ▶ Programmed via textual and graphical languages from the [IEC 61131-3](#)
- ▶ Modularisation via **function blocks**
- ▶ Has **static allocation** and **no recursion**

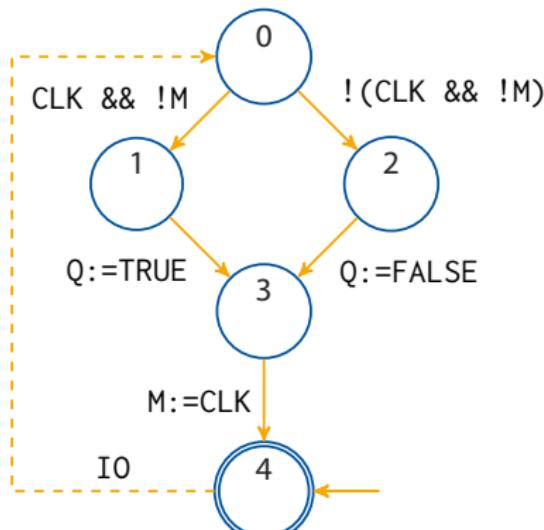
Control Flow Automaton (CFA)

```
1   FUNCTION_BLOCK R_TRIGGER
2     VAR_INPUT
3       CLK: BOOL;
4     END_VAR
5     VAR
6       M: BOOL;
7     END_VAR
8     VAR_OUTPUT
9       Q: BOOL;
10    END_VAR
11
12    IF CLK AND NOT M THEN
13      Q:=TRUE;
14    ELSE
15      Q:=FALSE;
16    END_IF;
17    M:=CLK;
18  END_FUNCTION_BLOCK
```



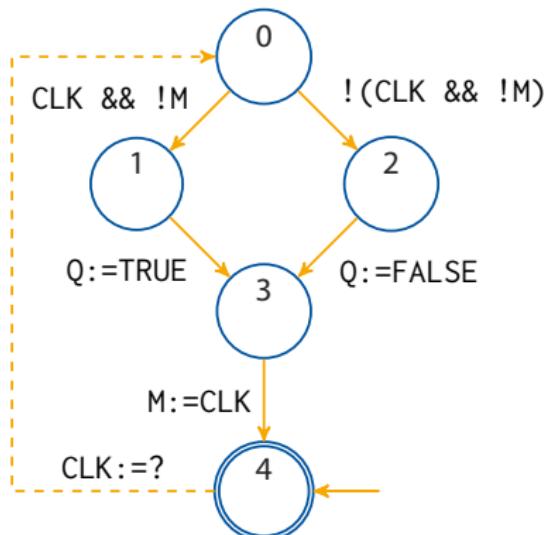
Control Flow Automaton (CFA)

```
1 PROGRAM R_TRIG
2   VAR_INPUT
3     CLK: BOOL;
4   END_VAR
5   VAR
6     M: BOOL;
7   END_VAR
8   VAR_OUTPUT
9     Q: BOOL;
10  END_VAR
11
12  IF CLK AND NOT M THEN
13    Q:=TRUE;
14  ELSE
15    Q:=FALSE;
16  END_IF;
17  M:=CLK;
18 END_PROGRAM
```



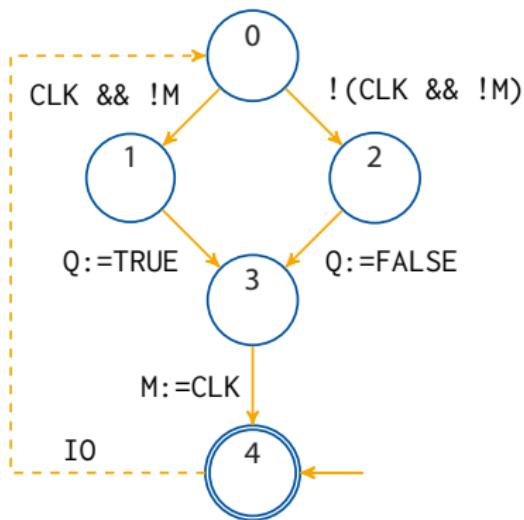
Control Flow Automaton (CFA)

```
1 PROGRAM R_TRIG
2   VAR_INPUT
3     CLK:BOOL;
4   END_VAR
5   VAR
6     M:BOOL;
7   END_VAR
8   VAR_OUTPUT
9     Q:BOOL;
10  END_VAR
11
12  IF CLK AND NOT M THEN
13    Q:=TRUE;
14  ELSE
15    Q:=FALSE;
16  END_IF;
17  M:=CLK;
18 END_PROGRAM
```



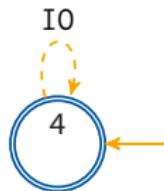
Specifications

- ▶ Intermediate states are not observable
- ⇒ Automation engineers and specifications always refer to the observable state
- ▶ Common specifications can be adapted to such cycle-step semantics, e. g.
 - ($M = CLK$) \rightsquigarrow □($pc = 4 \rightarrow M = CLK$)
 - and checked with off-the-shelf backends
- ▶ Unique specs need dedicated procedures



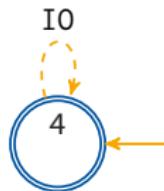
Specifications

- ▶ Intermediate states are not observable
- ⇒ Automation engineers and **specifications** always **refer to the observable state**
- ▶ Common specifications can be adapted to such cycle-step semantics, e. g.
 $\square(M = \text{CLK}) \rightsquigarrow \square(\text{pc} = 4 \rightarrow M = \text{CLK})$
and checked with off-the-shelf backends
- ▶ Unique specs need dedicated procedures



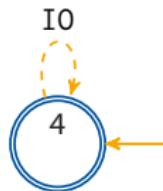
Specifications

- ▶ Intermediate states are not observable
- ⇒ Automation engineers and **specifications** always **refer to the observable state**
- ▶ Common specifications can be adapted to such **cycle-step semantics**, e. g.
 $\square(M = \text{CLK}) \rightsquigarrow \square(\text{pc} = 4 \rightarrow M = \text{CLK})$
and checked with **off-the-shelf backends**
- ▶ Unique specs need dedicated procedures



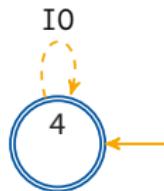
Specifications

- ▶ Intermediate states are not observable
- ⇒ Automation engineers and **specifications** always **refer to the observable state**
- ▶ Common specifications can be adapted to such **cycle-step semantics**, e. g.
 $\square(M = \text{CLK}) \rightsquigarrow \square(\text{pc} = 4 \rightarrow M = \text{CLK})$
and checked with **off-the-shelf backends**
- ▶ Unique specs need dedicated procedures



Specifications

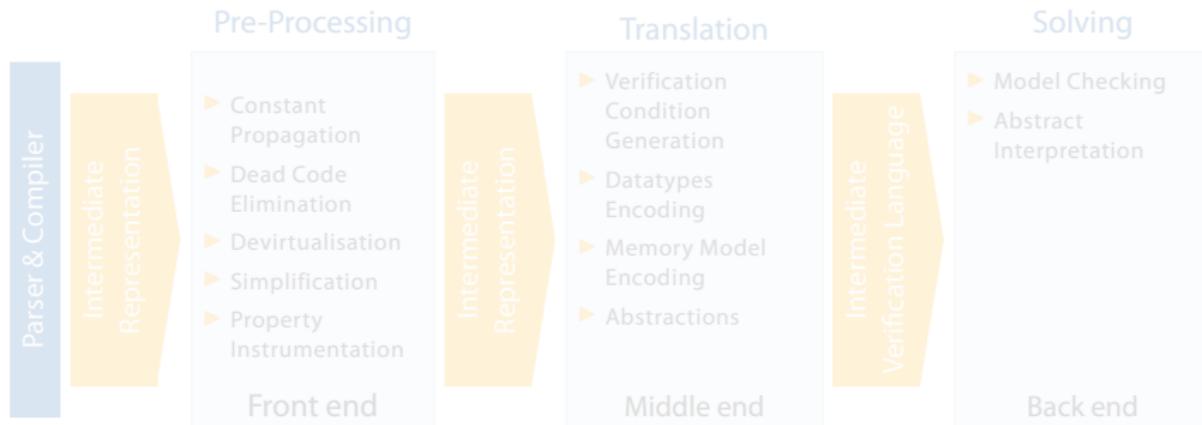
- ▶ Intermediate states are not observable
- ⇒ Automation engineers and **specifications** always **refer to the observable state**
- ▶ Common specifications can be adapted to such **cycle-step semantics**, e. g.
 $\square(M = \text{CLK}) \rightsquigarrow \square(\text{pc} = 4 \rightarrow M = \text{CLK})$
and checked with **off-the-shelf backends**
- ▶ Unique specs need dedicated procedures



Software Verification Pipeline

Recent years:

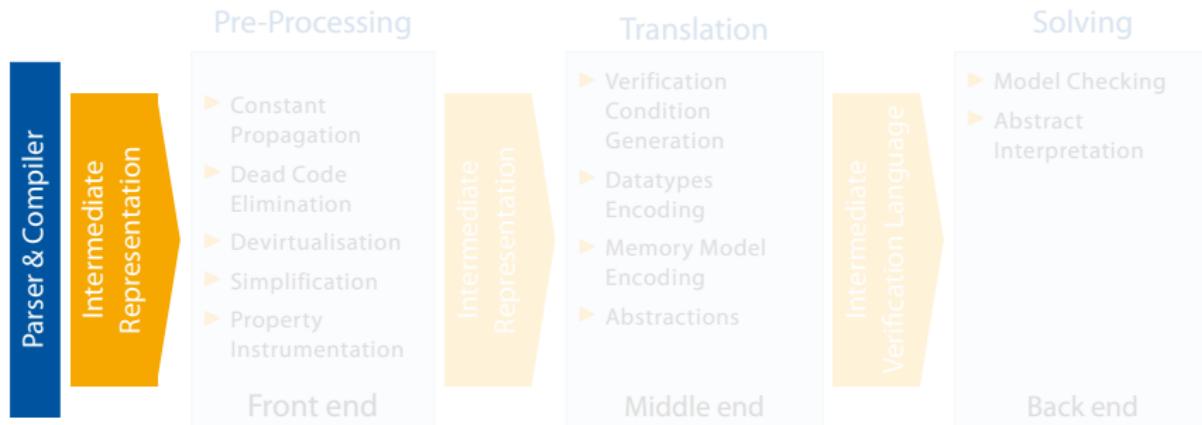
- ▶ Decoupling of language details from verifier implementations
- ▶ Modular structure & off-the-shelf components [Gur+15; RE14]



Software Verification Pipeline

Recent years:

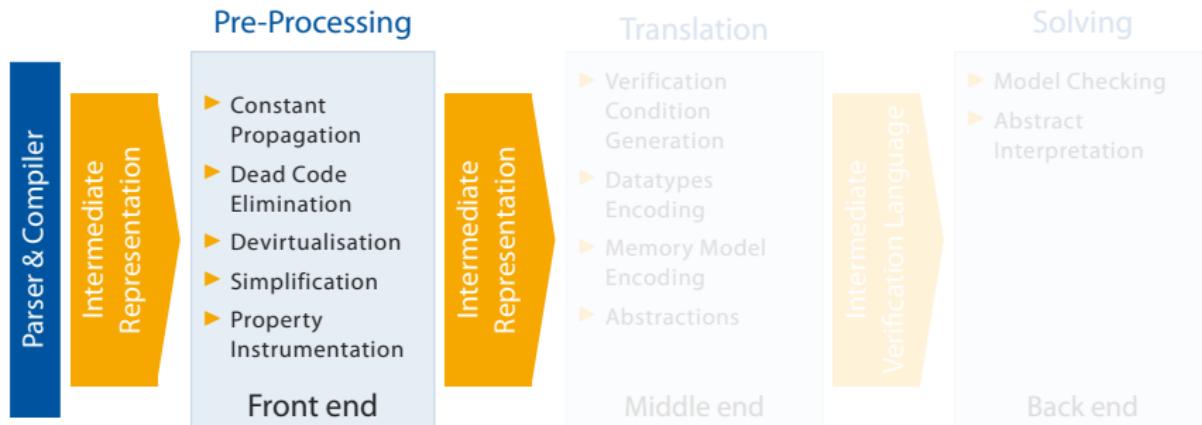
- ▶ Decoupling of language details from verifier implementations
- ▶ Modular structure & off-the-shelf components [Gur+15; RE14]



Software Verification Pipeline

Recent years:

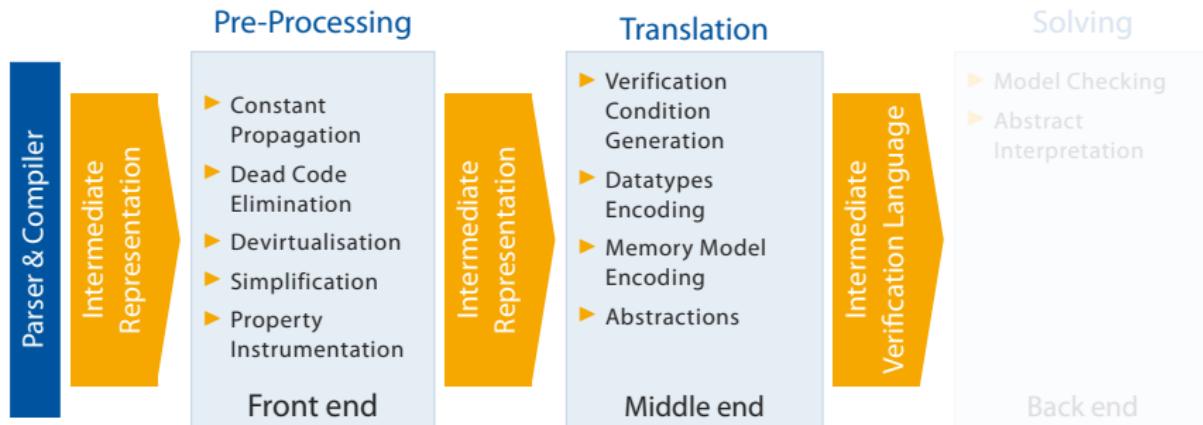
- ▶ Decoupling of language details from verifier implementations
- ▶ Modular structure & off-the-shelf components [Gur+15; RE14]



Software Verification Pipeline

Recent years:

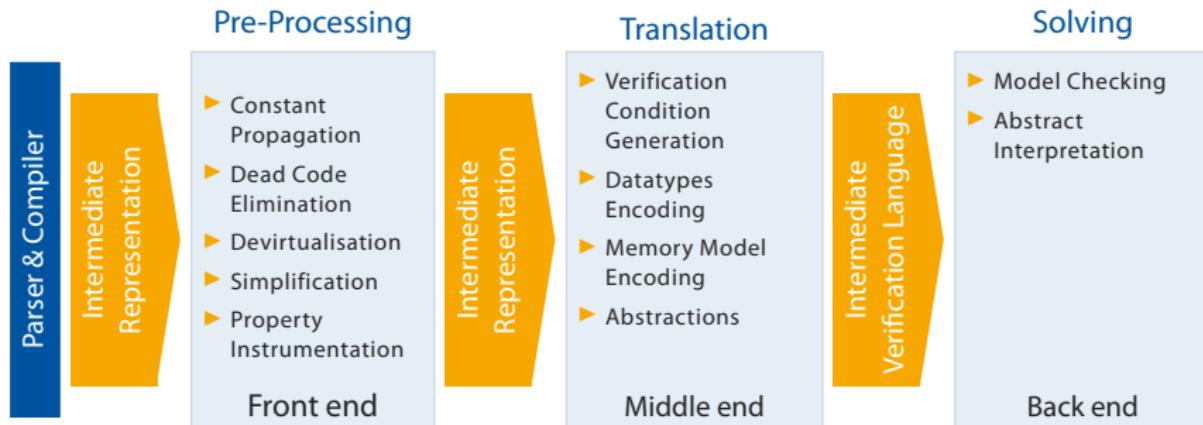
- ▶ Decoupling of language details from verifier implementations
- ▶ Modular structure & off-the-shelf components [Gur+15; RE14]



Software Verification Pipeline

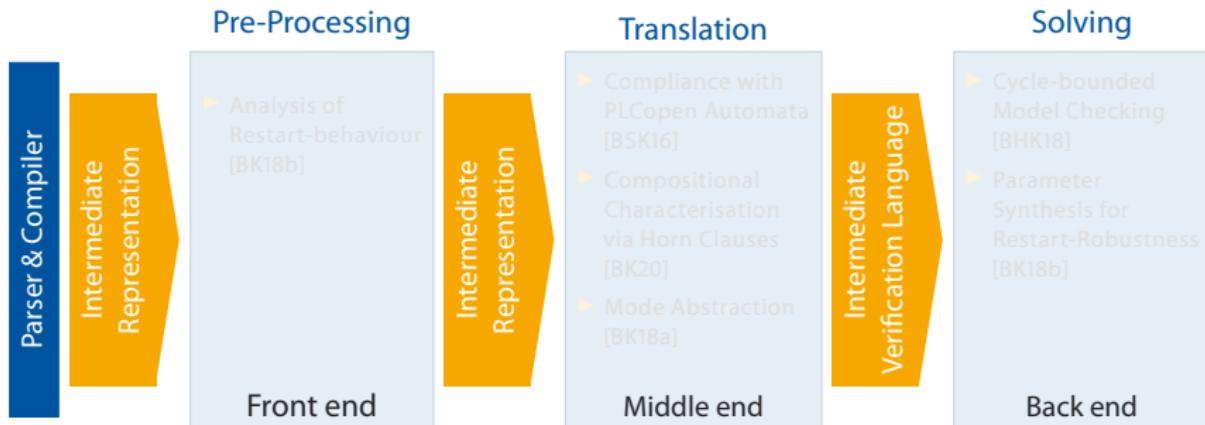
Recent years:

- ▶ Decoupling of language details from verifier implementations
- ▶ Modular structure & off-the-shelf components [Gur+15; RE14]



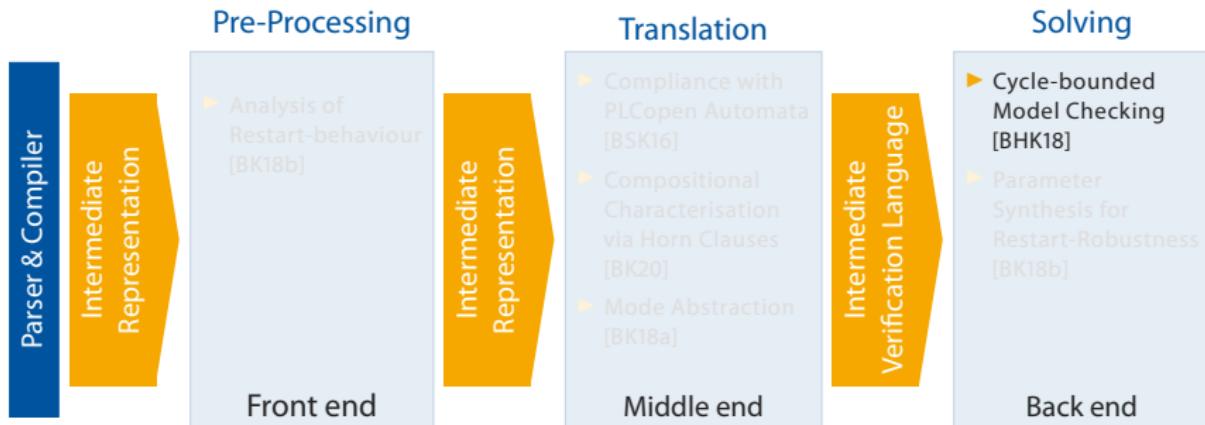
Contributions

- ▶ Symbolic verification procedures for domain-specific issues
- ▶ Implemented in **ARCADE.PLC**, but formulated for CFAs and transferable
- ▶ Not included: Test generation [Boh+16], Explainability [Kö+19]



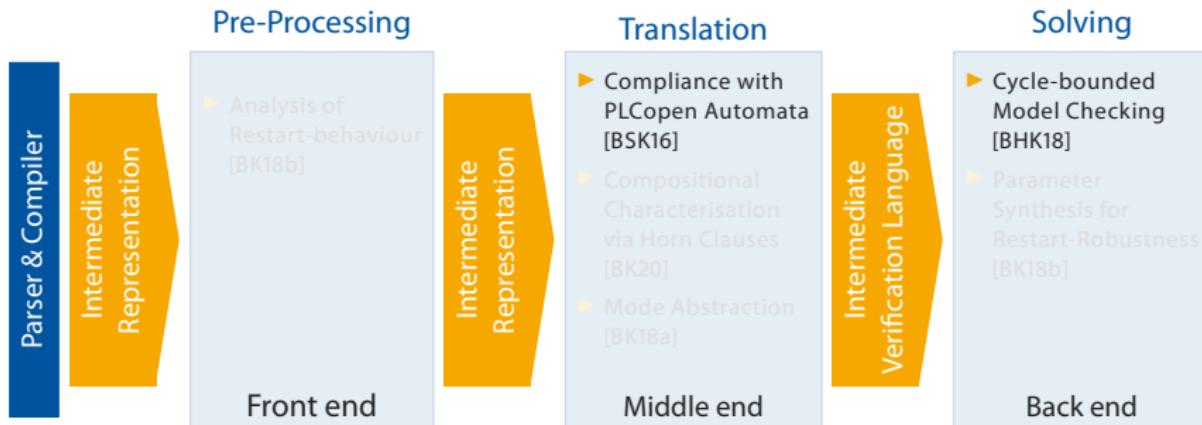
Contributions

- ▶ Symbolic verification procedures for domain-specific issues
- ▶ Implemented in **ARCADE.PLC**, but formulated for CFAs and transferable
- ▶ Not included: Test generation [Boh+16], Explainability [Kö+19]



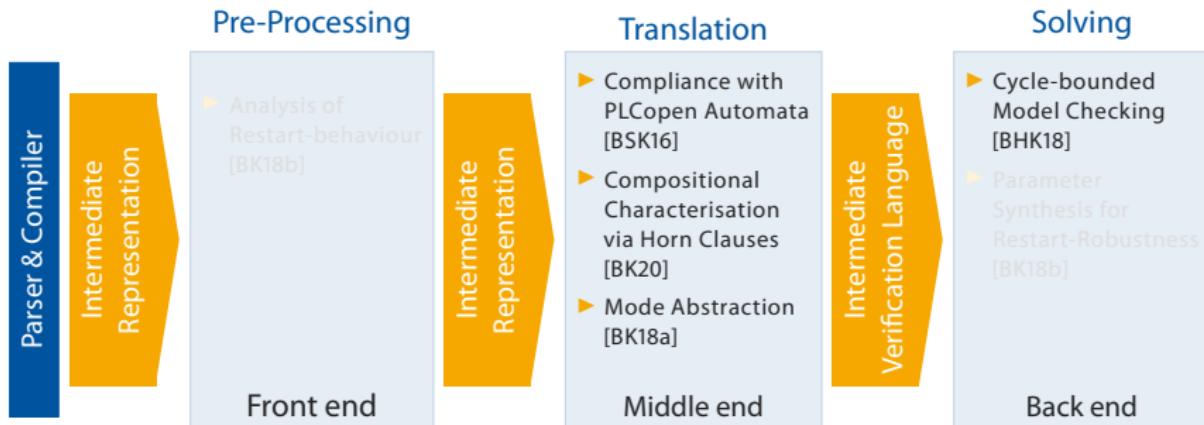
Contributions

- ▶ Symbolic verification procedures for domain-specific issues
- ▶ Implemented in **ARCADE.PLC**, but formulated for CFAs and transferable
- ▶ Not included: Test generation [Boh+16], Explainability [Kö+19]



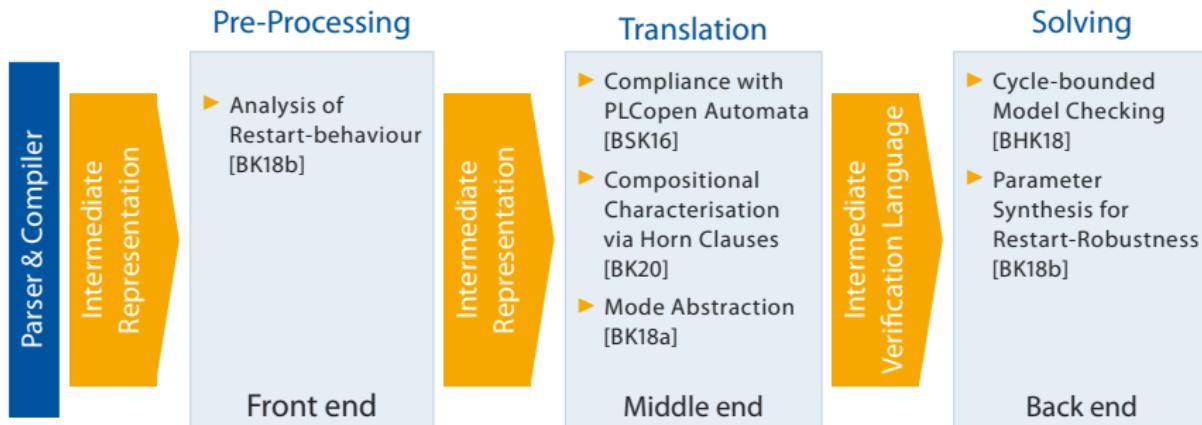
Contributions

- ▶ Symbolic verification procedures for domain-specific issues
- ▶ Implemented in **ARCADE.PLC**, but formulated for CFAs and transferable
- ▶ Not included: Test generation [Boh+16], Explainability [Kö+19]



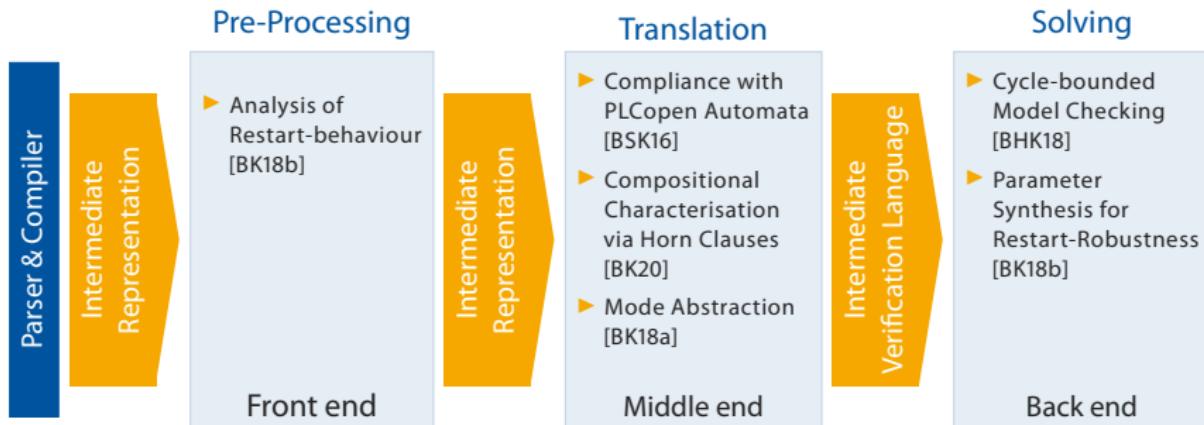
Contributions

- ▶ Symbolic verification procedures for domain-specific issues
- ▶ Implemented in **ARCADE.PLC**, but formulated for CFAs and transferable
- ▶ Not included: Test generation [Boh+16], Explainability [Kö+19]



Contributions

- ▶ Symbolic verification procedures for domain-specific issues
- ▶ Implemented in [ARCADE.PLC](#), but formulated for CFAs and transferable
- ▶ Not included: [Test generation](#) [Boh+16], [Explainability](#) [Kö+19]



PLCopen

- ▶ An independent organisation “for efficiency in automation”
- ▶ Focus on **technical specifications** around IEC 61131-3, e. g.

Performed most experiments on implementation of PLCopen Safety library:

- ▶ Elementary modules implementing particular safety concepts
- ▶ User examples composed of those

PLCopen

- ▶ An independent organisation “for efficiency in automation”
- ▶ Focus on **technical specifications** around IEC 61131-3, e. g.



Exchange format



Standard modules

Performed most experiments on implementation of PLCopen Safety library:

- ▶ Elementary modules implementing particular safety concepts
- ▶ User examples composed of those

PLCopen

- ▶ An independent organisation “for efficiency in automation”
- ▶ Focus on **technical specifications** around IEC 61131-3, e. g.



Exchange format



Standard modules

Performed most experiments on implementation of **PLCopen Safety** library:

- ▶ **Elementary modules** implementing particular safety concepts
- ▶ User examples **composed** of those

Related Work

- ▶ Discrete event systems community targets industrial control, mostly
 - reasoning on **model-level** – problems akin to hardware verification
 - using **binary decision diagrams** (BDDs) based backends [Ova+16]
- ▶ Darvas focuses on translation [DVA15] & BDD-based verification [Dar17]
- ▶ Biallas started ARCADE.PLC with explicit abstract interpretation [Bia16]
- ▶ Lange worked on property directed reachability (PDR) to CFAs [Lan18]
- ▶ Weigl develops methods to assist software evolution [Bec+15; Bec+17]
- ⇒ Although common for “ordinary” software, besides Weigl no one targets SAT-based verification of PLC software or domain-specifics

Related Work

- ▶ Discrete event systems community targets industrial control, mostly
 - reasoning on **model-level** – problems akin to hardware verification
 - using **binary decision diagrams** (BDDs) based backends [Ova+16]
- ▶ Darvas focuses on **translation** [DVA15] & BDD-based verification [Dar17]
- ▶ Biallas started **ARCADE.PLC** with **explicit abstract interpretation** [Bia16]
- ▶ Lange worked on **property directed reachability** (PDR) to CFAs [Lan18]
- ▶ Weigl develops methods to assist **software evolution** [Bec+15; Bec+17]
- ⇒ Although common for “ordinary” software, besides Weigl no one targets **SAT-based verification** of PLC software or domain-specifics

Related Work

- ▶ Discrete event systems community targets industrial control, mostly
 - reasoning on **model-level** – problems akin to hardware verification
 - using **binary decision diagrams** (BDDs) based backends [Ova+16]
- ▶ Darvas focuses on **translation** [DVA15] & BDD-based verification [Dar17]
- ▶ Biallas started **ARCADE.PLC** with **explicit abstract interpretation** [Bia16]
- ▶ Lange worked on **property directed reachability** (PDR) to CFAs [Lan18]
- ▶ Weigl develops methods to assist **software evolution** [Bec+15; Bec+17]
- ⇒ Although common for “ordinary” software, besides Weigl no one targets **SAT-based verification of PLC software or domain-specifics**

Why SAT Modulo Theories (SMT)?

- ▶ Explicit, relational information leads to state space explosion
- ▶ Both decision diagrams (DDs) and SMT enable symbolic reasoning
- ▶ Practice shows that DDs do not scale to software verification problems
- ▶ SMT theories enable modelling and reasoning about software, e. g.
 - datatypes – domains like \mathbb{R} are infeasible for DDs
 - memory via theory of arrays
- ⇒ Software verification machinery is distinctly SAT/SMT-based

Why SAT Modulo Theories (SMT)?

- ▶ Explicit, relational information leads to state space explosion
- ▶ Both **decision diagrams** (DDs) and **SMT** enable symbolic reasoning
- ▶ Practice shows that DDs do not scale to software verification problems
- ▶ SMT theories enable modelling and reasoning about software, e. g.
 - datatypes – domains like \mathbb{R} are infeasible for DDs
 - memory via theory of arrays
- ⇒ Software verification machinery is distinctly SAT/SMT-based

Why SAT Modulo Theories (SMT)?

- ▶ Explicit, relational information leads to state space explosion
- ▶ Both **decision diagrams** (DDs) and **SMT** enable symbolic reasoning
- ▶ Practice shows that DDs **do not scale to software** verification problems

- ▶ SMT theories enable modelling and reasoning about software, e. g.
 - datatypes – domains like \mathbb{R} are infeasible for DDs
 - memory via theory of arrays
- ⇒ Software verification machinery is distinctly SAT/SMT-based

Why SAT Modulo Theories (SMT)?

- ▶ Explicit, relational information leads to state space explosion
 - ▶ Both **decision diagrams** (DDs) and **SMT** enable symbolic reasoning
 - ▶ Practice shows that DDs **do not scale to software** verification problems
 - ▶ SMT theories enable modelling and reasoning about software, e.g.
 - datatypes – domains like \mathbb{R} are infeasible for DDs
 - memory via theory of arrays
- ⇒ Software verification machinery is distinctly SAT/SMT-based

Why SAT Modulo Theories (SMT)?

- ▶ Explicit, relational information leads to state space explosion
- ▶ Both **decision diagrams** (DDs) and **SMT** enable symbolic reasoning
- ▶ Practice shows that DDs **do not scale to software** verification problems
- ▶ SMT theories enable modelling and reasoning about software, e.g.
 - datatypes – domains like \mathbb{R} are infeasible for DDs
 - memory via theory of arrays
- ⇒ Software verification machinery is distinctly SAT/SMT-based

Intuition for Logical Characterisation

- ▶ Reason about program semantics by leveraging SMT-solving
 - ⇒ Characterise semantics via first order logic formulae

Intuition for Logical Characterisation

- ▶ Reason about program semantics by leveraging SMT-solving
- ⇒ Characterise semantics via first order logic formulae

Condition Formula over the program's variables

$$\llbracket x > y + 1 \rrbracket = x > y + 1$$

Statement Formula over pre- & post variables instances

$$\llbracket x := y + 1 \rrbracket = (x' = y + 1) \wedge (y' = y)$$

Procedure Predicate over pre- & post variables instances

$$\llbracket \text{abs}(x, x') \rrbracket = (x \geq 0 \rightarrow x' = x) \vee (x < 0 \rightarrow x' = -x)$$

Loops Unclear – no closed form

Intuition for Logical Characterisation

- ▶ Reason about program semantics by leveraging SMT-solving
- ⇒ Characterise semantics via first order logic formulae

Condition Formula over the program's variables

$$\llbracket x > y + 1 \rrbracket = x > y + 1$$

Statement Formula over pre- & post variables instances

$$\llbracket x := y + 1 \rrbracket = (x' = y + 1) \wedge (y' = y)$$

Procedure Predicate over pre- & post variables instances

$$\llbracket \text{abs}(x, x') \rrbracket = (x \geq 0 \rightarrow x' = x) \vee (x < 0 \rightarrow x' = -x)$$

Loops Unclear – no closed form

Intuition for Logical Characterisation

- ▶ Reason about program semantics by leveraging SMT-solving
 - ⇒ Characterise semantics via first order logic formulae

Condition Formula over the program's variables

$$\llbracket x > y + 1 \rrbracket = x > y + 1$$

Statement Formula over pre- & post variables instances

$$[\![x := y + 1]\!] = (x' = y + 1) \wedge (y' = y)$$

Intuition for Logical Characterisation

- ▶ Reason about program semantics by leveraging SMT-solving
- ⇒ Characterise semantics via first order logic formulae

Condition Formula over the program's variables

$$\llbracket x > y + 1 \rrbracket = x > y + 1$$

Statement Formula over pre- & post variables instances

$$\llbracket x := y + 1 \rrbracket = (x' = y + 1) \wedge (y' = y)$$

Procedure Predicate over pre- & post variables instances

$$\llbracket \text{abs}(x, x') \rrbracket = (x \geq 0 \rightarrow x' = x) \vee (x < 0 \rightarrow x' = -x)$$

Loops Unclear – no closed form

Intuition for Logical Characterisation

- ▶ Reason about program semantics by leveraging SMT-solving
- ⇒ Characterise semantics via first order logic formulae

Condition Formula over the program's variables

$$\llbracket x > y + 1 \rrbracket = x > y + 1$$

Statement Formula over pre- & post variables instances

$$\llbracket x := y + 1 \rrbracket = (x' = y + 1) \wedge (y' = y)$$

Procedure Predicate over pre- & post variables instances

$$\llbracket \text{abs}(x, x') \rrbracket = (x \geq 0 \rightarrow x' = x) \vee (x < 0 \rightarrow x' = -x)$$

Loops Unclear – no closed form

Constrained Horn Clauses (CHCs)

- A reactive system is safe if an **inductive invariant** $\text{reach}(\vec{X})$ exists, s.t. the following is SAT [MP95]:

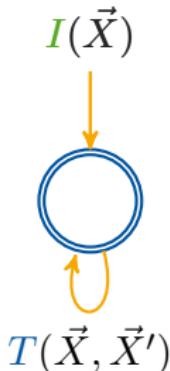
$$I(\vec{X}) \rightarrow \text{reach}(\vec{X})$$

$$\wedge \text{reach}(\vec{X}) \wedge T(\vec{X}, \vec{X}') \rightarrow \text{reach}(\vec{X}')$$

$$\wedge \text{reach}(\vec{X}) \rightarrow \text{safe}(\vec{X})$$

- Given sets of **variables** \mathcal{V} , **function symbols** \mathcal{F} , and **predicates** \mathcal{P} , a CHC is a formula

$$\forall \mathcal{V} \underbrace{p_1(\vec{X}_1) \wedge \cdots \wedge p_k(\vec{X}_k)}_{\text{body}} \wedge \varphi \rightarrow \underbrace{h(\vec{X})}_{\text{head}}, k \geq 0,$$



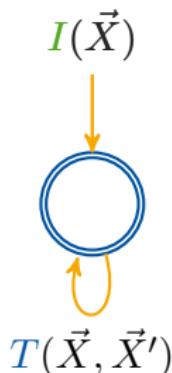
Constrained Horn Clauses (CHCs)

- ▶ A reactive system is safe if an **inductive invariant** $\text{reach}(\vec{X})$ exists, s.t. the following is SAT [MP95]:

$$\begin{aligned} & \textcolor{green}{I}(\vec{X}) \rightarrow \text{reach}(\vec{X}) \\ & \wedge \text{reach}(\vec{X}) \wedge T(\vec{X}, \vec{X}') \rightarrow \text{reach}(\vec{X}') \\ & \wedge \text{reach}(\vec{X}) \rightarrow \text{safe}(\vec{X}) \end{aligned}$$

- ▶ Given sets of **variables** \mathcal{V} , **function symbols** \mathcal{F} , and **predicates** \mathcal{P} , a CHC is a formula

$$\underbrace{\forall \mathcal{V} p_1(\vec{X}_1) \wedge \cdots \wedge p_k(\vec{X}_k)}_{\text{body}} \wedge \varphi \rightarrow \underbrace{h(\vec{X})}_{\text{head}}, \quad k \geq 0,$$



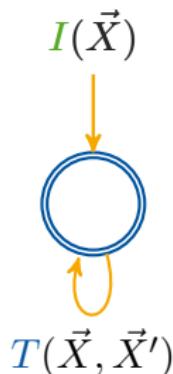
Constrained Horn Clauses (CHCs)

- ▶ A reactive system is safe if an **inductive invariant** $\text{reach}(\vec{X})$ exists, s.t. the following is SAT [MP95]:

$$\begin{aligned} & \textcolor{green}{I}(\vec{X}) \rightarrow \text{reach}(\vec{X}) \\ & \wedge \text{reach}(\vec{X}) \wedge \textcolor{blue}{T}(\vec{X}, \vec{X}') \rightarrow \text{reach}(\vec{X}') \\ & \wedge \text{reach}(\vec{X}) \rightarrow \text{safe}(\vec{X}) \end{aligned}$$

- ▶ Given sets of **variables** \mathcal{V} , **function symbols** \mathcal{F} , and **predicates** \mathcal{P} , a CHC is a formula

$$\underbrace{\forall \mathcal{V} p_1(\vec{X}_1) \wedge \cdots \wedge p_k(\vec{X}_k)}_{\text{body}} \wedge \varphi \rightarrow \underbrace{h(\vec{X})}_{\text{head}}, \quad k \geq 0,$$



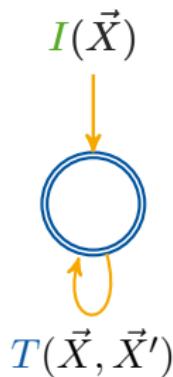
Constrained Horn Clauses (CHCs)

- ▶ A reactive system is safe if an **inductive invariant** $\text{reach}(\vec{X})$ exists, s.t. the following is SAT [MP95]:

$$\begin{aligned} & \textcolor{green}{I}(\vec{X}) \rightarrow \text{reach}(\vec{X}) \\ \wedge & \text{reach}(\vec{X}) \wedge \textcolor{blue}{T}(\vec{X}, \vec{X}') \rightarrow \text{reach}(\vec{X}') \\ \wedge & \text{reach}(\vec{X}) \rightarrow \textcolor{green}{safe}(\vec{X}) \end{aligned}$$

- ▶ Given sets of **variables** \mathcal{V} , **function symbols** \mathcal{F} , and **predicates** \mathcal{P} , a CHC is a formula

$$\underbrace{\forall \mathcal{V} p_1(\vec{X}_1) \wedge \cdots \wedge p_k(\vec{X}_k)}_{\text{body}} \wedge \varphi \rightarrow \underbrace{h(\vec{X})}_{\text{head}}, \quad k \geq 0,$$



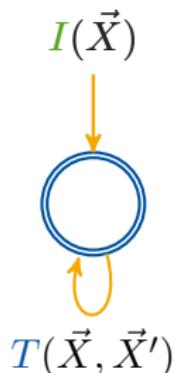
Constrained Horn Clauses (CHCs)

- ▶ A reactive system is safe if an **inductive invariant** $\text{reach}(\vec{X})$ exists, s.t. the following is SAT [MP95]:

$$\begin{aligned} & \text{I}(\vec{X}) \rightarrow \text{reach}(\vec{X}) \\ \wedge & \text{reach}(\vec{X}) \wedge \text{T}(\vec{X}, \vec{X}') \rightarrow \text{reach}(\vec{X}') \\ \wedge & \text{reach}(\vec{X}) \rightarrow \text{safe}(\vec{X}) \end{aligned}$$

- ▶ Given sets of **variables** \mathcal{V} , **function symbols** \mathcal{F} , and **predicates** \mathcal{P} , a CHC is a formula

$$\underbrace{\forall \mathcal{V} p_1(\vec{X}_1) \wedge \cdots \wedge p_k(\vec{X}_k)}_{\text{body}} \wedge \varphi \rightarrow \underbrace{h(\vec{X})}_{\text{head}}, \quad k \geq 0,$$



CHCs as Intermediate Verification Language

- ▶ In 2010, Bradley proposed a novel **hardware model checking algorithm**
 - **PDR** constructs inductive invariants incrementally
 - Was competitive with highly tuned solvers – **3rd place at HWMCC'10**
- ⇒ Incentive for **lifting it to software verification** – no approach prevailed
- ▶ However, PDR was generalised to CHCs
- ▶ CHCs are a logical match for Hoare logic and correspond to proof rules
- ⇒ Using CHC-solving, emerging tools were **competitive at SV-COMP'15**

Practical advantages:

- ▶ CHC solving is just a case of SMT – keeping its flexibility and techniques
- ▶ Both PDR and its extension **SPACER** are in the open SMT-solver **Z3** (only)

CHCs as Intermediate Verification Language

- ▶ In 2010, Bradley proposed a novel **hardware model checking algorithm**
 - PDR constructs inductive invariants incrementally
 - Was competitive with highly tuned solvers – 3rd place at HWMCC'10
- ⇒ Incentive for **lifting it to software verification** – no approach prevailed
- ▶ However, PDR was generalised to CHCs
- ▶ CHCs are a logical match for Hoare logic and correspond to proof rules
- ⇒ Using CHC-solving, emerging tools were **competitive** at SV-COMP'15

Practical advantages:

- ▶ CHC solving is just a case of SMT – keeping its flexibility and techniques
- ▶ Both PDR and its extension SPACER are in the open SMT-solver Z3 (only)

CHCs as Intermediate Verification Language

- ▶ In 2010, Bradley proposed a novel **hardware model checking algorithm**
 - PDR constructs inductive invariants incrementally
 - Was competitive with highly tuned solvers – 3rd place at HWMCC'10
- ⇒ Incentive for **lifting it to software verification** – no approach prevailed
- ▶ However, PDR was **generalised to CHCs**
- ▶ CHCs are a **logical match** for Hoare logic and correspond to proof rules
- ⇒ Using CHC-solving, emerging tools were **competitive** at SV-COMP'15

Practical advantages:

- ▶ CHC solving is just a case of SMT – keeping its flexibility and techniques
- ▶ Both PDR and its extension SPACER are in the open SMT-solver Z3 (only)

CHCs as Intermediate Verification Language

- ▶ In 2010, Bradley proposed a novel **hardware model checking** algorithm
 - **PDR** constructs inductive invariants incrementally
 - Was competitive with highly tuned solvers – **3rd place at HWMCC'10**
- ⇒ Incentive for **lifting it to software verification** – no approach prevailed
- ▶ However, PDR was **generalised to CHCs**
- ▶ CHCs are a **logical match for Hoare logic** and correspond to proof rules
- ⇒ Using CHC-solving, emerging tools were **competitive at SV-COMP'15**

Practical advantages:

- ▶ CHC solving is just a case of SMT – keeping its flexibility and techniques
- ▶ Both PDR and its extension **SPACER** are in the open SMT-solver **Z3** (only)

CHCs as Intermediate Verification Language

- ▶ In 2010, Bradley proposed a novel **hardware model checking** algorithm
 - **PDR** constructs inductive invariants incrementally
 - Was competitive with highly tuned solvers – **3rd place at HWMCC'10**
- ⇒ Incentive for **lifting it to software verification** – no approach prevailed
- ▶ However, PDR was **generalised to CHCs**
- ▶ CHCs are a **logical match for Hoare logic** and correspond to proof rules
- ⇒ Using CHC-solving, emerging tools were **competitive at SV-COMP'15**

Practical advantages:

- ▶ CHC solving is just a case of SMT – keeping its flexibility and techniques
- ▶ Both PDR and its extension SPACER are in the open SMT-solver Z3 (only)

CHCs as Intermediate Verification Language

- ▶ In 2010, Bradley proposed a novel **hardware model checking** algorithm
 - **PDR** constructs inductive invariants incrementally
 - Was competitive with highly tuned solvers – **3rd place at HWMCC'10**
- ⇒ Incentive for **lifting it to software verification** – no approach prevailed
- ▶ However, PDR was **generalised to CHCs**
- ▶ CHCs are a **logical match for Hoare logic** and correspond to proof rules
- ⇒ Using CHC-solving, emerging tools were **competitive at SV-COMP'15**

Practical advantages:

- ▶ CHC solving is just a case of SMT – keeping its flexibility and techniques
- ▶ Both PDR and its extension **SPACER** are **in the open SMT-solver Z3** (only)

Running Example

- ▶ Implements delegation of requests on rising edge
- ▶ ReqHandler is polled in every cycle
- ▶ Input is forwarded on rising edge for req

```
FUNCTION_BLOCK ReqHandler
    VAR_INPUT data      :WORD; END_VAR
    VAR          DiagCode:WORD; END_VAR
    VAR_OUTPUT res       :WORD; END_VAR
    // Body omitted ...
END_FUNCTION_BLOCK

PROGRAM Main
    VAR_INPUT req:BOOL; in:WORD;           END_VAR
    VAR          m :BOOL; h :ReqHandler;END_VAR
    VAR_OUTPUT out:WORD;                  END_VAR
    // Forward data on rising edge
    IF (req AND NOT m) THEN
        h(data:=in, res=>out);
    ELSE
        h(res=>out);
    END_IF
    m:=req;
END_PROGRAM
```

Running Example

- ▶ Implements delegation of requests on rising edge
- ▶ ReqHandler h is polled in every cycle
- ▶ Input is forwarded on rising edge for req

```
FUNCTION_BLOCK ReqHandler
    VAR_INPUT data      :WORD; END_VAR
    VAR          DiagCode:WORD; END_VAR
    VAR_OUTPUT res       :WORD; END_VAR
    // Body omitted ...
END_FUNCTION_BLOCK

PROGRAM Main
    VAR_INPUT req:BOOL; in:WORD;           END_VAR
    VAR          m :BOOL; h :ReqHandler;END_VAR
    VAR_OUTPUT out:WORD;                  END_VAR
    // Forward data on rising edge
    IF (req AND NOT m) THEN
        h(data:=in, res=>out);
    ELSE
        h(res=>out);
    END_IF
    m:=req;
END_PROGRAM
```

Running Example

- ▶ Implements delegation of requests on rising edge
- ▶ ReqHandler h is polled in every cycle
- ▶ Input in forwarded on rising edge for req

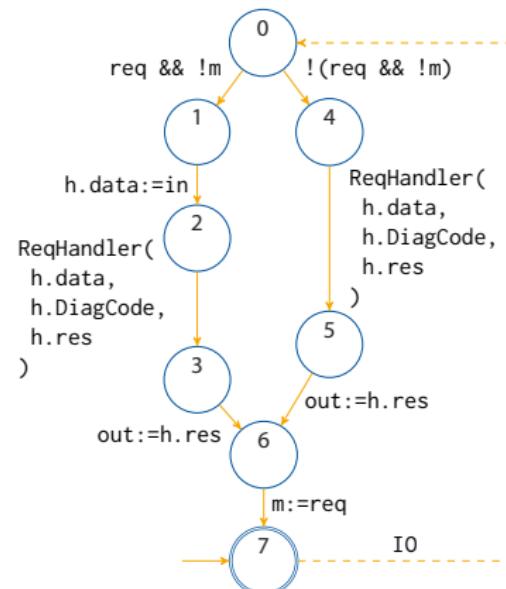
```
FUNCTION_BLOCK ReqHandler
    VAR_INPUT data      :WORD; END_VAR
    VAR          DiagCode:WORD; END_VAR
    VAR_OUTPUT res       :WORD; END_VAR
    // Body omitted ...
END_FUNCTION_BLOCK

PROGRAM Main
    VAR_INPUT req:BOOL; in:WORD;           END_VAR
    VAR          m :BOOL; h :ReqHandler;END_VAR
    VAR_OUTPUT out:WORD;                  END_VAR
    // Forward data on rising edge
    IF (req AND NOT m) THEN
        h(data:=in, res=>out);
    ELSE
        h(res=>out);
    END_IF
    m:=req;
END_PROGRAM
```

Characterisation

- ▶ Introduced DES community to CHCs
- ⇒ Extended reactive system formulation

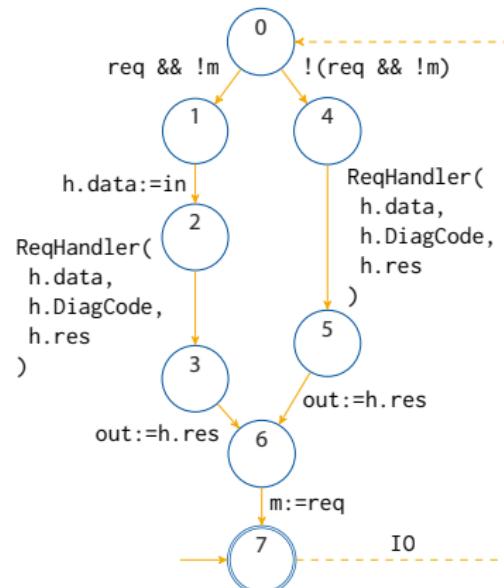
$$\begin{aligned}I(\vec{X}) &\rightarrow p_7(\vec{X}) \\ p_7(\vec{X}) \wedge \llbracket \text{IO} \rrbracket &\rightarrow p_0(\vec{X}') \\ p_0(\vec{X}) \wedge \text{req} \wedge \neg m \wedge \vec{X}' = \vec{X} &\rightarrow p_1(\vec{X}') \\ &\vdots \\ p_7(\vec{X}) &\rightarrow \text{safe}(\vec{X})\end{aligned}$$



Characterisation

- ▶ Introduced DES community to CHCs
- ⇒ Extended reactive system formulation

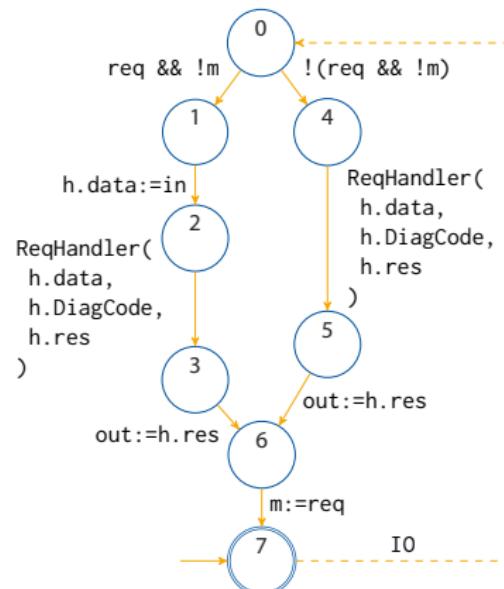
$$\begin{aligned}I(\vec{X}) &\rightarrow p_7(\vec{X}) \\ p_7(\vec{X}) \wedge \llbracket \text{IO} \rrbracket &\rightarrow p_0(\vec{X}') \\ p_0(\vec{X}) \wedge \text{req} \wedge \neg m \wedge \vec{X}' = \vec{X} &\rightarrow p_1(\vec{X}') \\ &\vdots \\ p_7(\vec{X}) &\rightarrow \text{safe}(\vec{X})\end{aligned}$$



Characterisation

- ▶ Introduced DES community to CHCs
- ⇒ Extended reactive system formulation

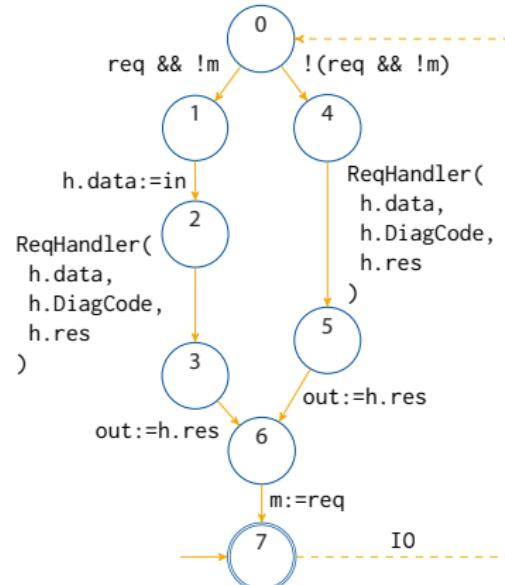
$$\begin{aligned}I(\vec{X}) &\rightarrow p_7(\vec{X}) \\ p_7(\vec{X}) \wedge \llbracket \text{IO} \rrbracket &\rightarrow p_0(\vec{X}') \\ p_0(\vec{X}) \wedge \text{req} \wedge \neg m \wedge \vec{X}' = \vec{X} &\rightarrow p_1(\vec{X}') \\ &\vdots \\ p_7(\vec{X}) &\rightarrow \text{safe}(\vec{X})\end{aligned}$$



Characterisation

- ▶ Introduced DES community to CHCs
- ⇒ Extended reactive system formulation

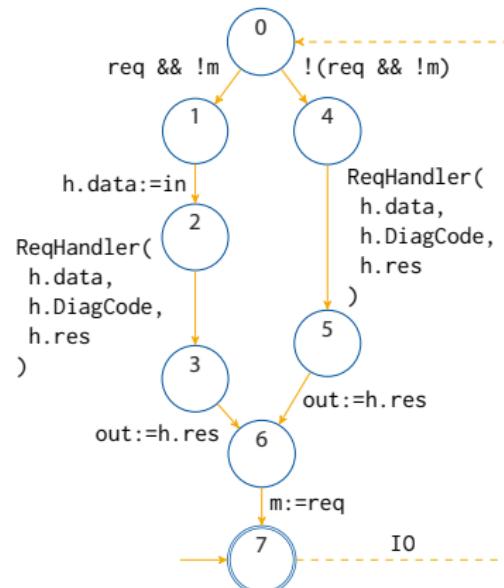
$$\begin{aligned}I(\vec{X}) &\rightarrow p_7(\vec{X}) \\ p_7(\vec{X}) \wedge \llbracket \text{IO} \rrbracket &\rightarrow p_0(\vec{X}') \\ p_0(\vec{X}) \wedge \text{req} \wedge \neg m \wedge \vec{X}' = \vec{X} &\rightarrow p_1(\vec{X}') \\ &\vdots \\ p_7(\vec{X}) &\rightarrow \text{safe}(\vec{X})\end{aligned}$$



Characterisation

- ▶ Introduced DES community to CHCs
- ⇒ Extended reactive system formulation

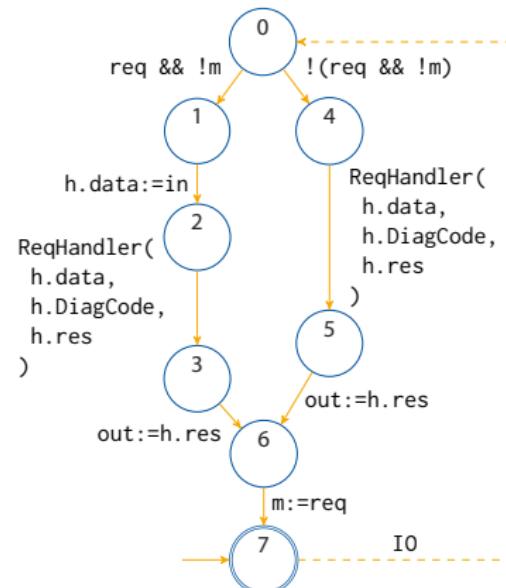
$$\begin{aligned}I(\vec{X}) &\rightarrow p_7(\vec{X}) \\ p_7(\vec{X}) \wedge \llbracket \text{IO} \rrbracket &\rightarrow p_0(\vec{X}') \\ p_0(\vec{X}) \wedge \text{req} \wedge \neg m \wedge \vec{X}' = \vec{X} &\rightarrow p_1(\vec{X}') \\ &\vdots \\ p_7(\vec{X}) &\rightarrow \text{safe}(\vec{X})\end{aligned}$$



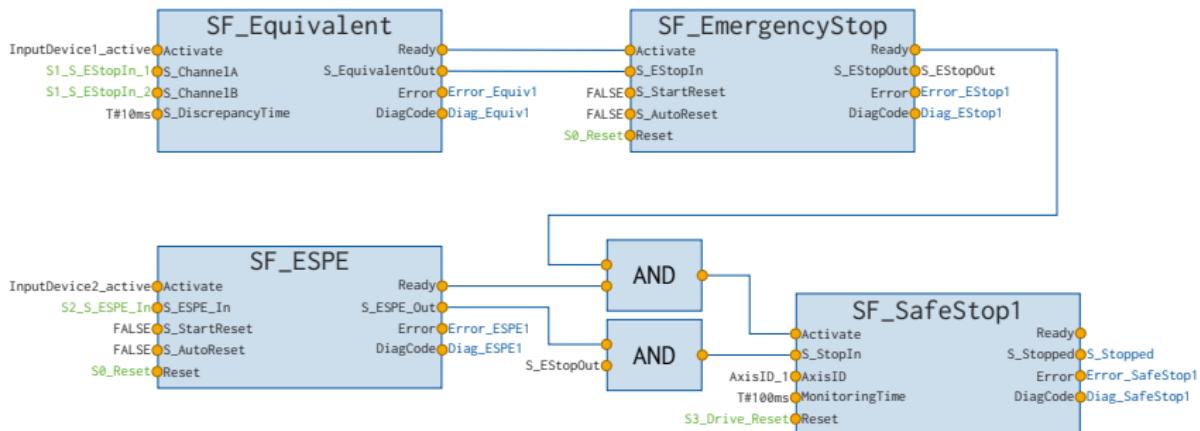
Characterisation

- ▶ Introduced DES community to CHCs
- ⇒ Extended reactive system formulation

$$\begin{aligned}I(\vec{X}) &\rightarrow p_7(\vec{X}) \\ p_7(\vec{X}) \wedge \llbracket \text{IO} \rrbracket &\rightarrow p_0(\vec{X}') \\ p_0(\vec{X}) \wedge \text{req} \wedge \neg m \wedge \vec{X}' = \vec{X} &\rightarrow p_1(\vec{X}') \\ &\vdots \\ p_7(\vec{X}) &\rightarrow \text{safe}(\vec{X})\end{aligned}$$

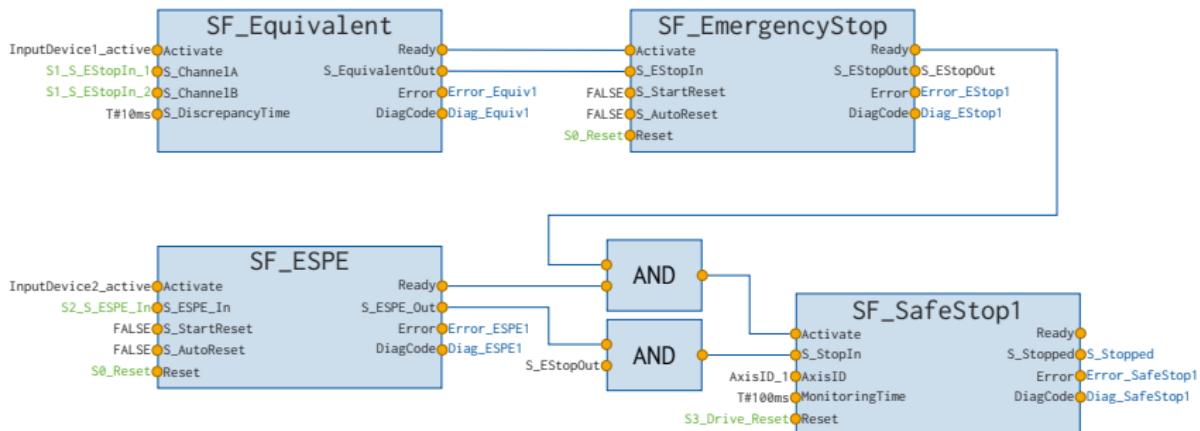


PLCopen Safety Application



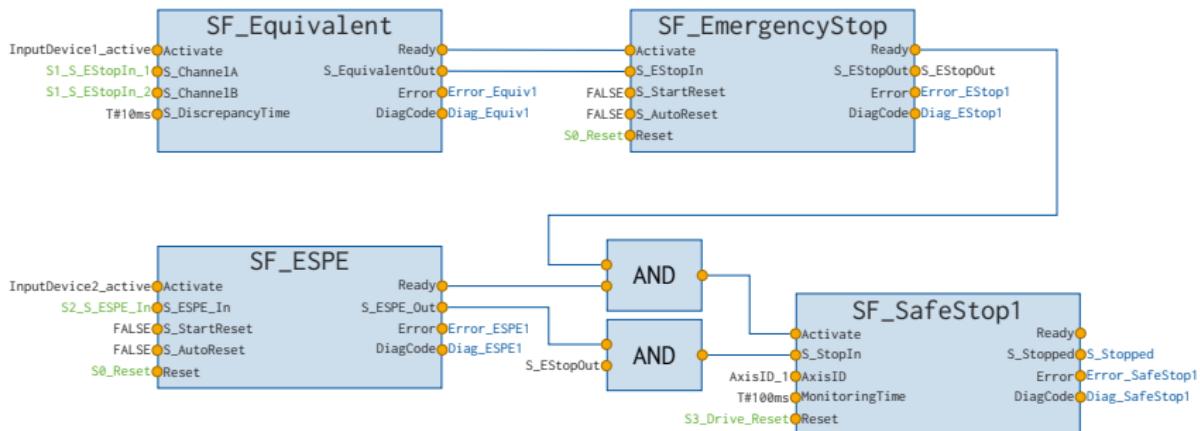
- ▶ Real-world software consists of many blocks – potentially same ones
- ▶ However, existing approaches are non-compositional or BDD-based
- ⇒ Effectively model checking a flattened all-encompassing block
- ▶ Verification condition generation should preserve compositionality

PLCopen Safety Application



- Real-world software consists of many blocks – potentially same ones
- However, existing approaches are non-compositional or BDD-based
- Effectively model checking a flattened all-encompassing block
- Verification condition generation should preserve compositionality

PLCopen Safety Application



- Real-world software consists of many blocks – potentially same ones
- However, existing approaches are non-compositional or BDD-based
- Effectively model checking a flattened all-encompassing block
- Verification condition generation should preserve compositionality

Compositional Characterisation

- Extend the p_i to characterise a state at i , reachable from a block's **entry state**, e.g.

$$\text{Main}_0(\vec{X}, \vec{X}') \wedge \text{req}' \wedge \neg m' \wedge \vec{X}'' = \vec{X}'$$

$$\rightarrow \text{Main}_1(\vec{X}, \vec{X}'')$$

- Gives a way to capture a block's I/O:

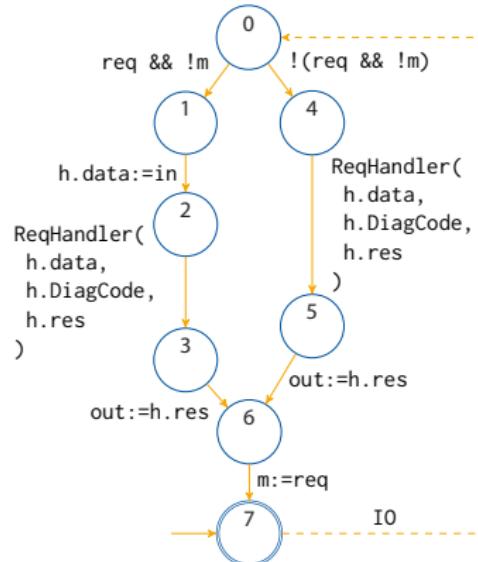
$$\text{Main}_7(\vec{X}, \vec{X}')$$

- Enables characterisation of **calls**, e.g.

$$\text{Main}_2(\vec{X}, \vec{X}') \wedge S(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\wedge \text{ReqHandler}_{\text{exit}}(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\rightarrow \text{Main}_3(\vec{X}, \vec{X}'')$$



Compositional Characterisation

- Extend the p_i to characterise a state at i , reachable from a block's **entry state**, e.g.

$$\text{Main}_0(\vec{X}, \vec{X}') \wedge \text{req}' \wedge \neg m' \wedge \vec{X}'' = \vec{X}'$$

$$\rightarrow \text{Main}_1(\vec{X}, \vec{X}'')$$

- Gives a way to capture a block's I/O:

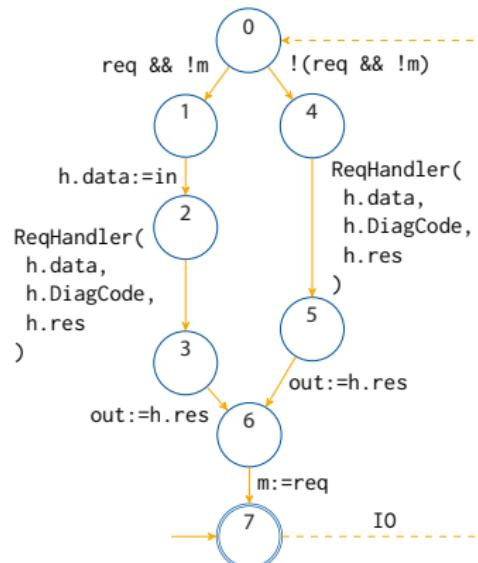
$$\text{Main}_7(\vec{X}, \vec{X}')$$

- Enables characterisation of **calls**, e.g.

$$\text{Main}_2(\vec{X}, \vec{X}') \wedge S(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\wedge \text{ReqHandler}_{\text{exit}}(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\rightarrow \text{Main}_3(\vec{X}, \vec{X}'')$$



Compositional Characterisation

- Extend the p_i to characterise a state at i , reachable from a block's **entry state**, e.g.

$$\text{Main}_0(\vec{X}, \vec{X}') \wedge \text{req}' \wedge \neg m' \wedge \vec{X}'' = \vec{X}'$$

$$\rightarrow \text{Main}_1(\vec{X}, \vec{X}'')$$

- Gives a way to capture a block's I/O:

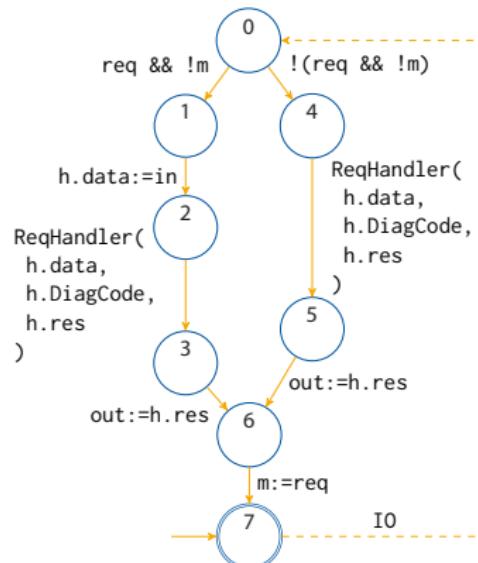
$$\text{Main}_7(\vec{X}, \vec{X}')$$

- Enables characterisation of **calls**, e.g.

$$\text{Main}_2(\vec{X}, \vec{X}') \wedge S(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\wedge \text{ReqHandler}_{\text{exit}}(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\rightarrow \text{Main}_3(\vec{X}, \vec{X}'')$$



Compositional Characterisation

- Extend the p_i to characterise a state at i , reachable from a block's **entry state**, e.g.

$$\text{Main}_0(\vec{X}, \vec{X}') \wedge \text{req}' \wedge \neg m' \wedge \vec{X}'' = \vec{X}'$$

$$\rightarrow \text{Main}_1(\vec{X}, \vec{X}'')$$

- Gives a way to capture a **block's I/O**:

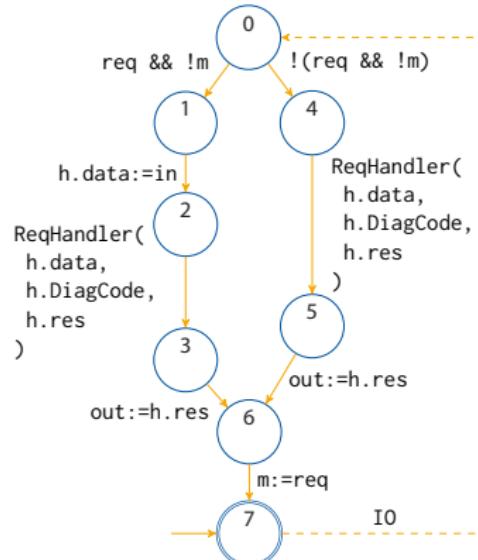
$$\text{Main}_7(\vec{X}, \vec{X}')$$

- Enables characterisation of **calls**, e.g.

$$\text{Main}_2(\vec{X}, \vec{X}') \wedge S(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\wedge \text{ReqHandler}_{\text{exit}}(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\rightarrow \text{Main}_3(\vec{X}, \vec{X}'')$$



Compositional Characterisation

- Extend the p_i to characterise a state at i , reachable from a block's **entry state**, e.g.

$$\text{Main}_0(\vec{X}, \vec{X}') \wedge \text{req}' \wedge \neg m' \wedge \vec{X}'' = \vec{X}'$$

$$\rightarrow \text{Main}_1(\vec{X}, \vec{X}'')$$

- Gives a way to capture a **block's I/O**:

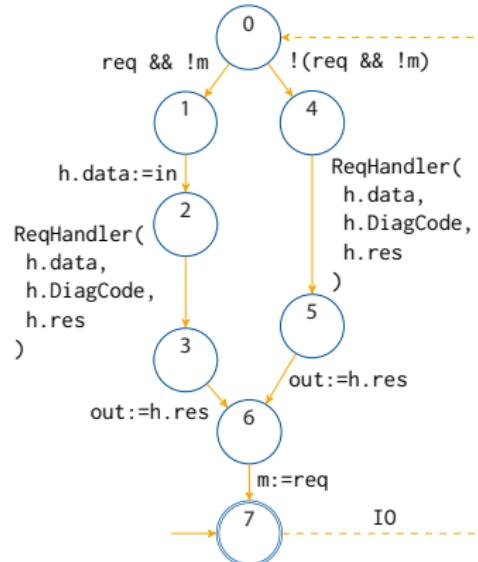
$$\text{Main}_7(\vec{X}, \vec{X}')$$

- Enables characterisation of **calls**, e.g.

$$\text{Main}_2(\vec{X}, \vec{X}') \wedge S(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\wedge \text{ReqHandler}_{\text{exit}}(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\rightarrow \text{Main}_3(\vec{X}, \vec{X}'')$$



Compositional Characterisation

- Extend the p_i to characterise a state at i , reachable from a block's **entry state**, e.g.

$$\text{Main}_0(\vec{X}, \vec{X}') \wedge \text{req}' \wedge \neg m' \wedge \vec{X}'' = \vec{X}'$$

$$\rightarrow \text{Main}_1(\vec{X}, \vec{X}'')$$

- Gives a way to capture a **block's I/O**:

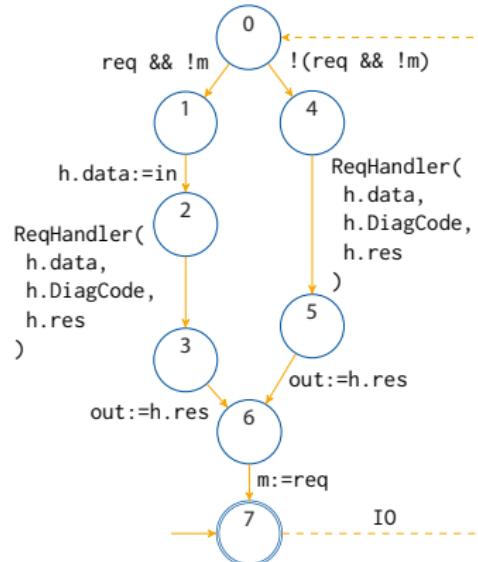
$$\text{Main}_7(\vec{X}, \vec{X}')$$

- Enables characterisation of **calls**, e.g.

$$\text{Main}_2(\vec{X}, \vec{X}') \wedge S(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\wedge \text{ReqHandler}_{\text{exit}}(\vec{X}'_{\text{ReqHandler}}, \vec{X}''_{\text{ReqHandler}})$$

$$\rightarrow \text{Main}_3(\vec{X}, \vec{X}'')$$



Experiments

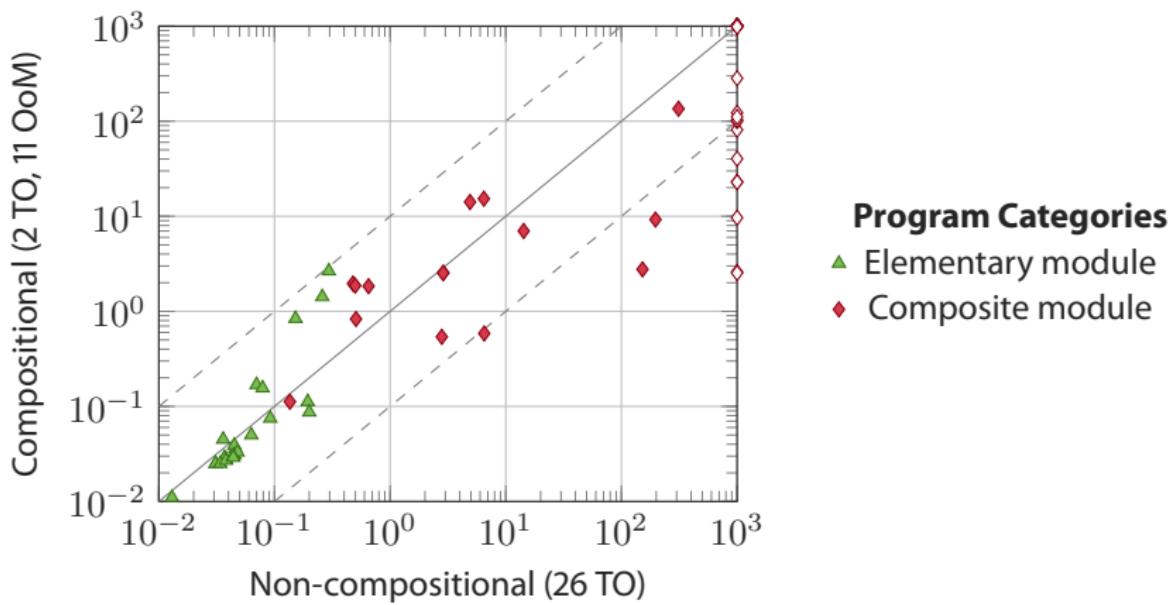
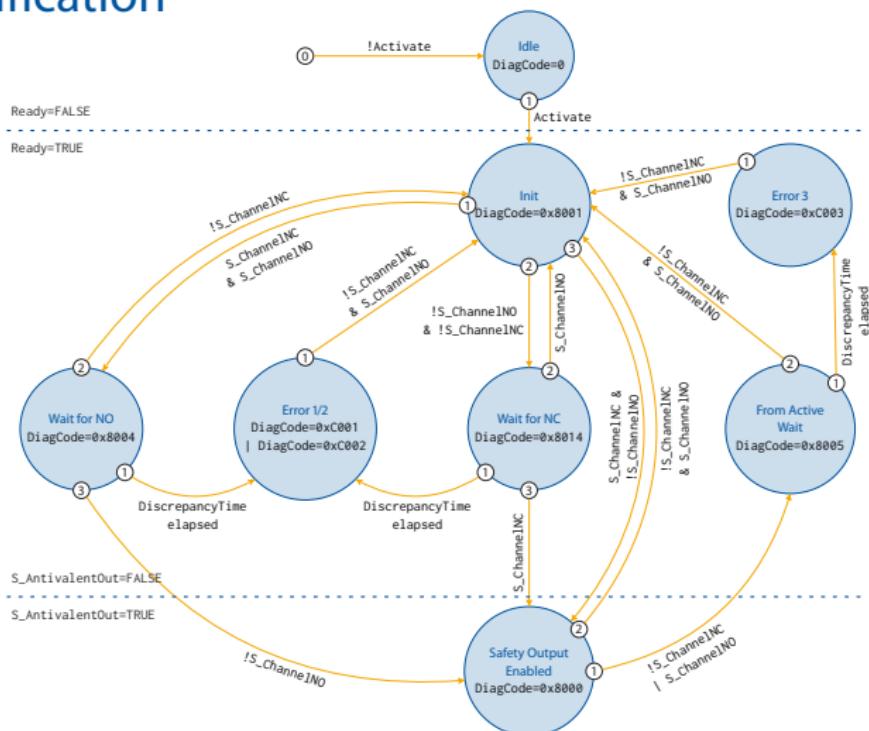


Figure: Time [s] spent on each verification task

Mode Abstraction

PLCopen Block Specification

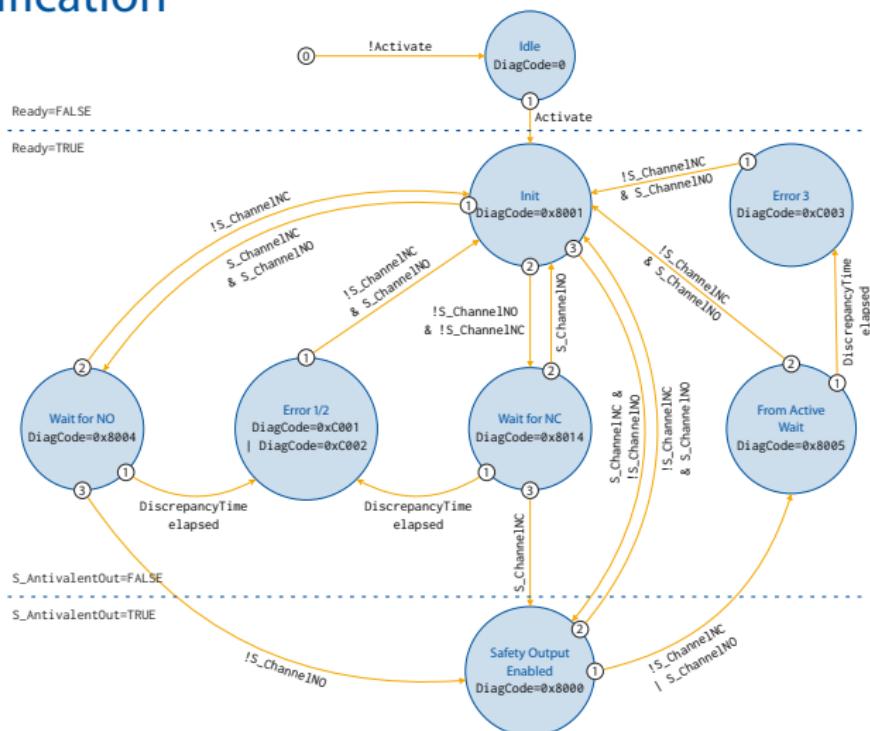
- ▶ An engineer's **mental model** of a block often has **state-machine semantics**
- ▶ Such partitioning into different modes reduces the complexity
- ▶ May simplify reasoning for a verifier, too



Mode Abstraction

PLCopen Block Specification

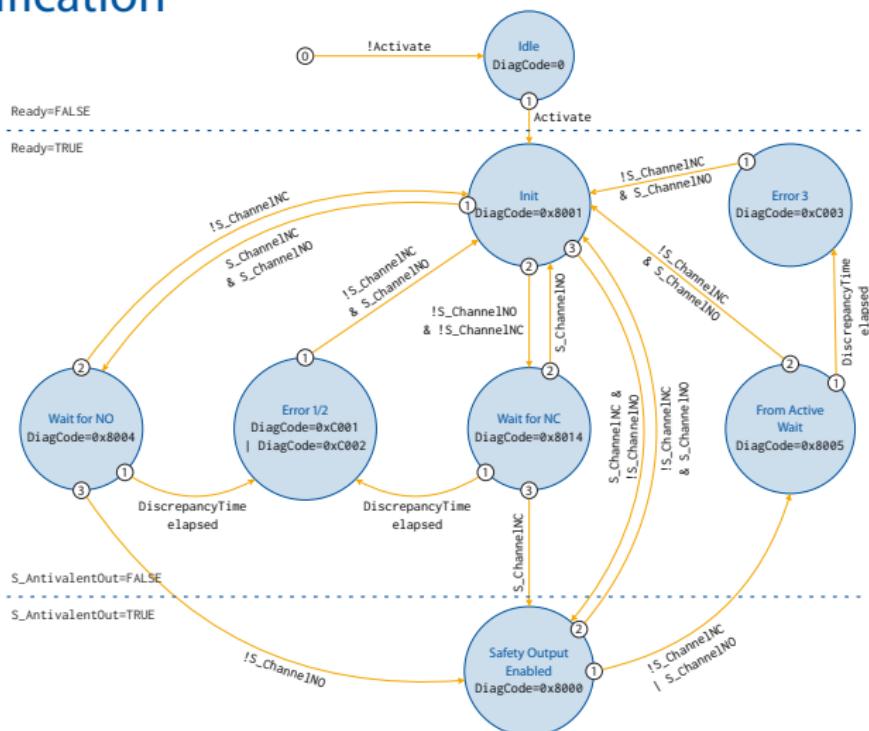
- ▶ An engineer's **mental model** of a block often has **state-machine semantics**
- ▶ Such partitioning into different **modes** reduces the complexity
- ▶ May **simplify reasoning** for a verifier, too



Mode Abstraction

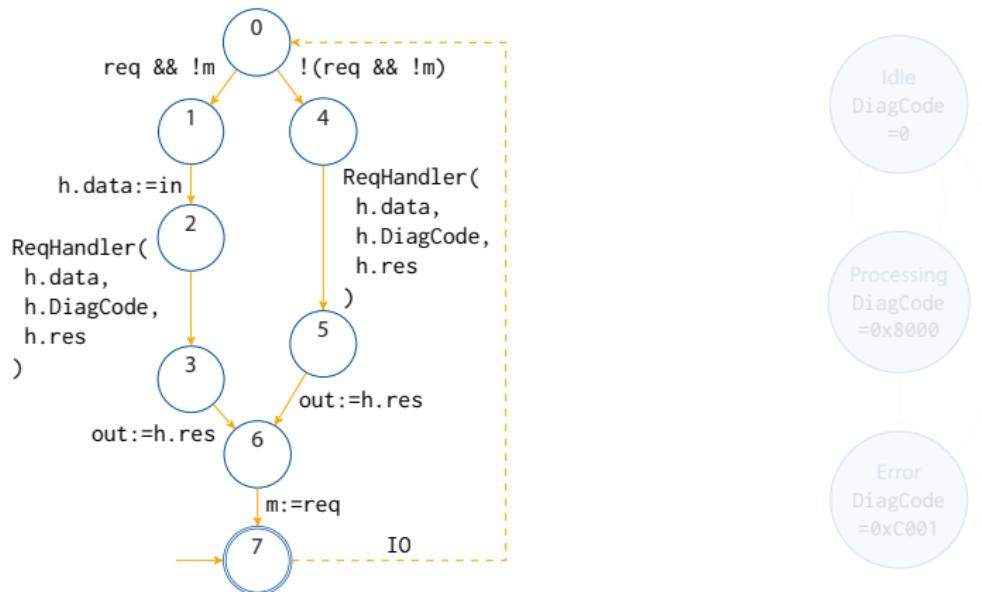
PLCopen Block Specification

- ▶ An engineer's **mental model** of a block often has **state-machine semantics**
- ▶ Such partitioning into different **modes** reduces the complexity
- ▶ May **simplify reasoning** for a verifier, too



Mode Abstraction

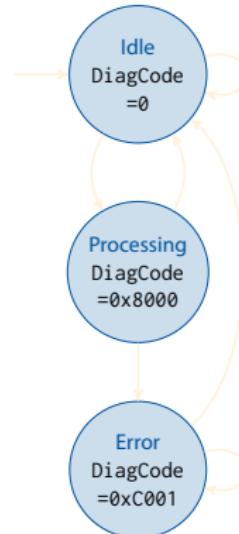
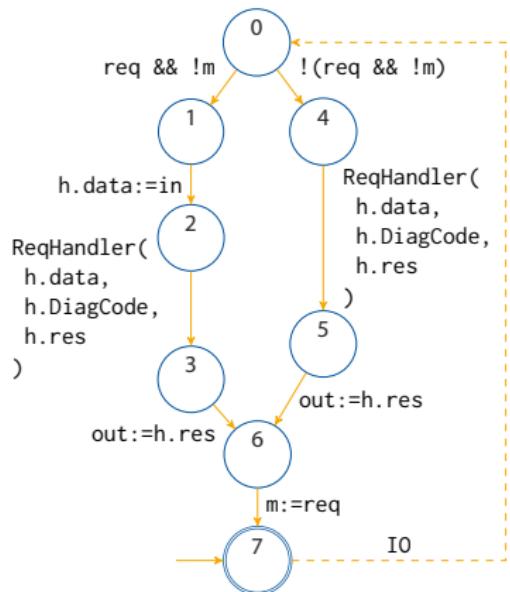
Mode Spaces in Model Checking



- ▶ Are requests processed in \leq two execution cycles?
- ⇒ On request, is "Processing" mode reached in a single cycle?

Mode Abstraction

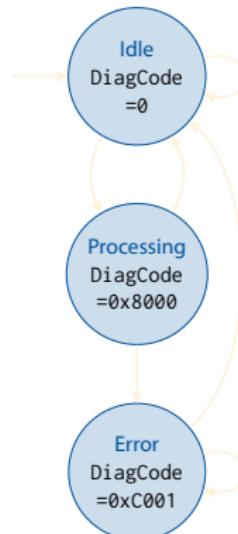
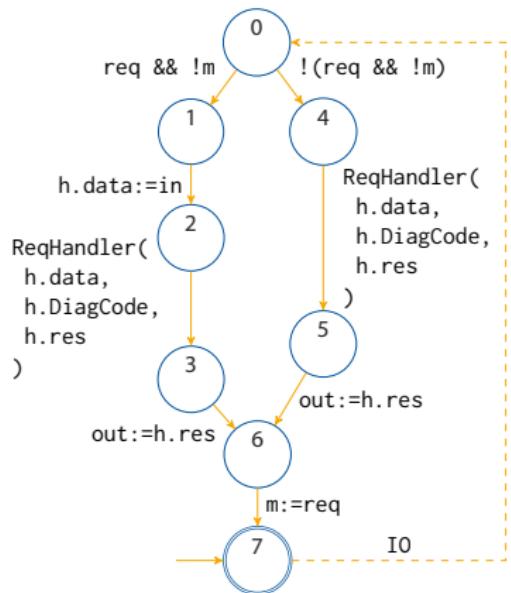
Mode Spaces in Model Checking



- ▶ Are requests processed in \leq two execution cycles?
- ⇒ On request, is "Processing" mode reached in a single cycle?

Mode Abstraction

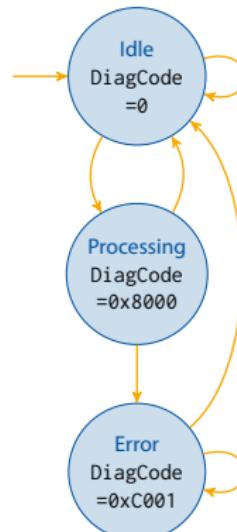
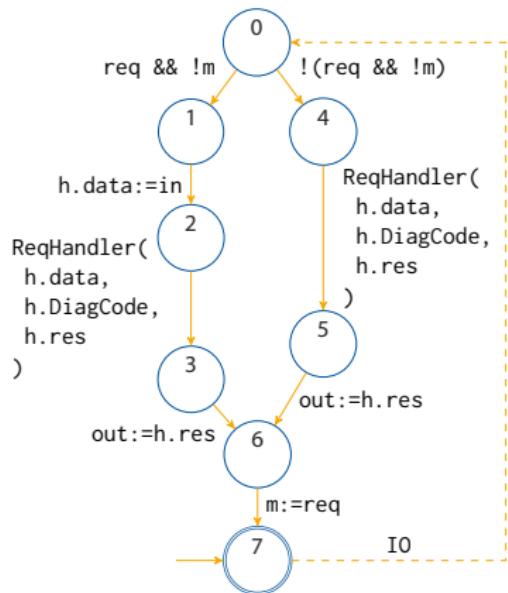
Mode Spaces in Model Checking



- Are requests processed in \leq two execution cycles?
- ⇒ On request, is "Processing" mode reached in a single cycle?

Mode Abstraction

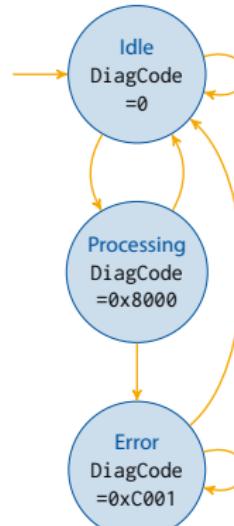
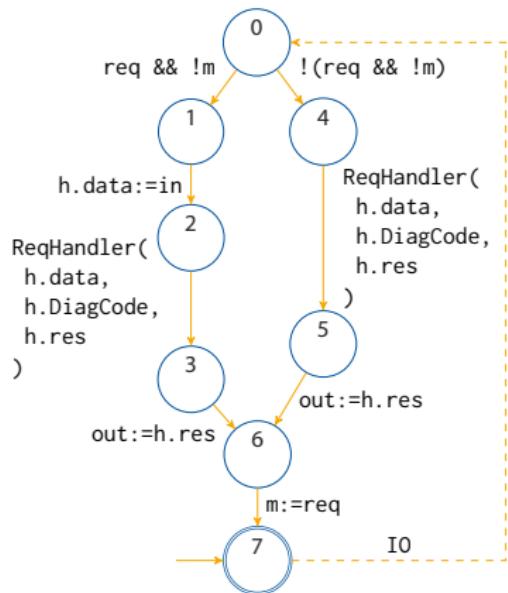
Mode Spaces in Model Checking



- ▶ Are requests processed in \leq two execution cycles?
- ⇒ On request, is “Processing” mode reached in a single cycle?

Mode Abstraction

Mode Spaces in Model Checking



- ▶ Are requests processed in \leq two execution cycles?
- ⇒ On request, is “Processing” mode reached in a single cycle?

Mode Abstraction

Idea: Procedure's complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA)



1. Perform VSA on main CFA to approximate all variables' values
2. For each block type and mode, e. g. ReqHandler and 0x8000
 - 2.1 Keep VSA's values but fix source mode
 - 2.2 Perform VSA on block and interpret resulting mode-values as targets

Mode Abstraction

Idea: Procedure's complexity needs to be low w.r.t. CHC-solving
⇒ Adapt value-set analysis (VSA)

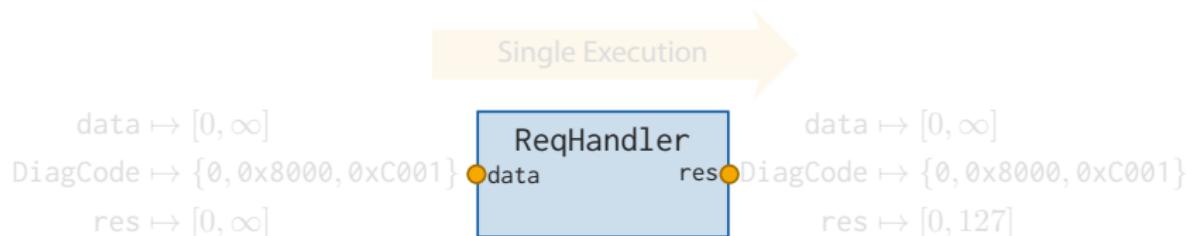


1. Perform VSA on main CFA to approximate all variables' values
2. For each block type and mode, e. g. ReqHandler and 0x8000
 - 2.1 Keep VSA's values but fix source mode
 - 2.2 Perform VSA on block and interpret resulting mode-values as targets

Mode Abstraction

Idea: Procedure's complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA)

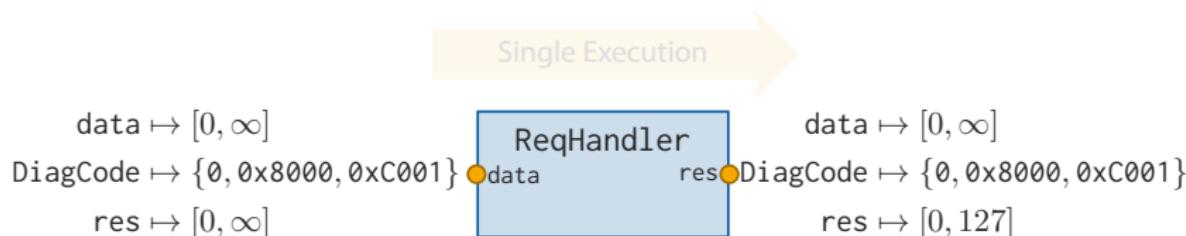


1. Perform VSA on main CFA to approximate all variables' values
2. For each block type and mode, e.g. ReqHandler and 0x8000
 - 2.1 Keep VSA's values but fix source mode
 - 2.2 Perform VSA on block and interpret resulting mode-values as targets

Mode Abstraction

Idea: Procedure's complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA)

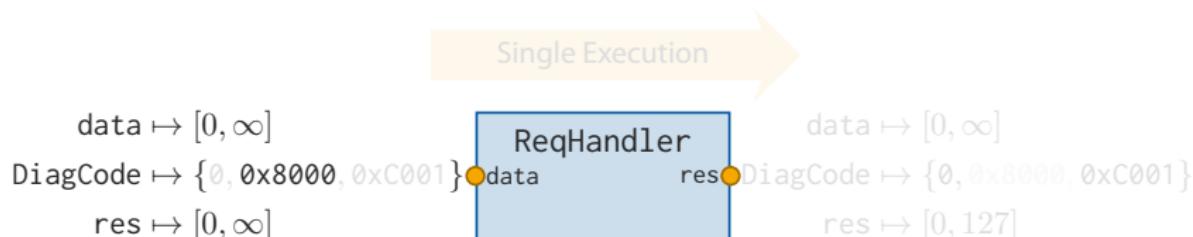


1. Perform VSA on main CFA to approximate all variables' values
2. For each block type and mode, e.g. ReqHandler and 0x8000
 - 2.1 Keep VSA's values but fix source mode
 - 2.2 Perform VSA on block and interpret resulting mode-values as targets

Mode Abstraction

Idea: Procedure's complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA)

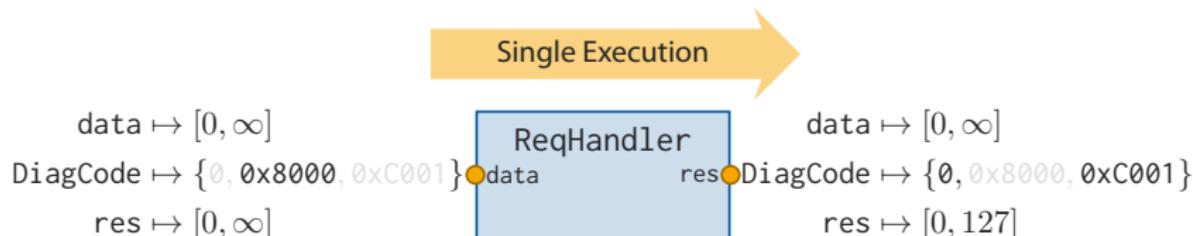


1. Perform VSA on main CFA to approximate all variables' values
2. For each block type and mode, e.g. **ReqHandler** and **0x8000**
 - 2.1 Keep VSA's values but fix source mode
 - 2.2 Perform VSA on block and interpret resulting mode-values as targets

Mode Abstraction

Idea: Procedure's complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA)



1. Perform VSA on main CFA to approximate all variables' values
2. For each block type and mode, e.g. ReqHandler and 0x8000
 - 2.1 Keep VSA's values but fix source mode
 - 2.2 Perform VSA on block and interpret resulting mode-values as targets

Mode Space as Call Summary

- Mode space **constrains** possible transitions

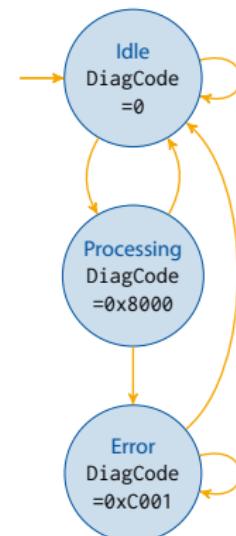
⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$\begin{aligned} h.\text{DiagCode} = 0 &\rightarrow h.\text{DiagCode}' = 0 \\ &\vee h.\text{DiagCode}' = 0x8000 \end{aligned}$$

$$\begin{aligned} \wedge h.\text{DiagCode} = 0x8000 &\rightarrow h.\text{DiagCode}' = 0 \\ &\vee h.\text{DiagCode}' = 0xC001 \end{aligned}$$

$$\wedge (h.\text{DiagCode} = 0xC001 \rightarrow h.\text{DiagCode}' = 0 \\ \vee h.\text{DiagCode}' = 0xC001)$$

- Add to encoding of each call of ReqHandler



Mode Space as Call Summary

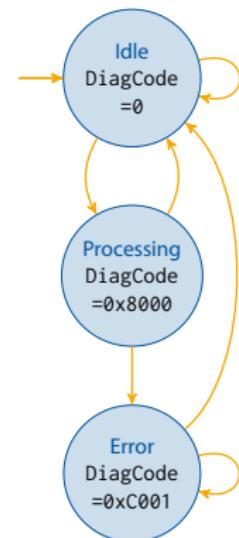
- ▶ Mode space **constrains** possible transitions
- ⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$(h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0x8000)$$

$$\wedge (h.DiagCode = 0x8000 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

$$\wedge (h.DiagCode = 0xC001 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

- ▶ Add to encoding of each call of ReqHandler



Mode Space as Call Summary

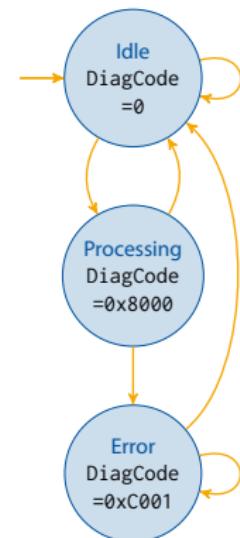
- ▶ Mode space **constrains** possible transitions
- ⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$(h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0x8000)$$

$$\wedge (h.DiagCode = 0x8000 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

$$\wedge (h.DiagCode = 0xC001 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

- ▶ Add to encoding of each call of ReqHandler



Mode Space as Call Summary

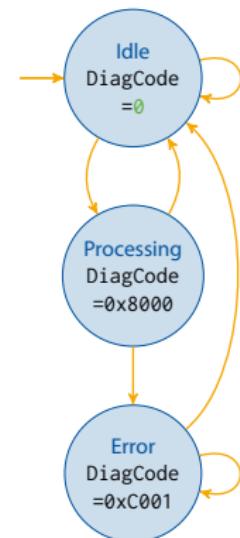
- ▶ Mode space **constrains** possible transitions
- ⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$(h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0x8000)$$

$$\wedge (h.DiagCode = 0x8000 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

$$\wedge (h.DiagCode = 0xC001 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

- ▶ Add to encoding of each call of ReqHandler



Mode Space as Call Summary

- Mode space **constrains** possible transitions

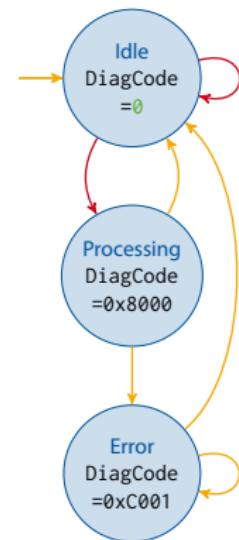
⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$(h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0x8000)$$

$$\wedge (h.DiagCode = 0x8000 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

$$\wedge (h.DiagCode = 0xC001 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

- Add to encoding of each call of ReqHandler



Mode Space as Call Summary

► Mode space **constrains** possible transitions

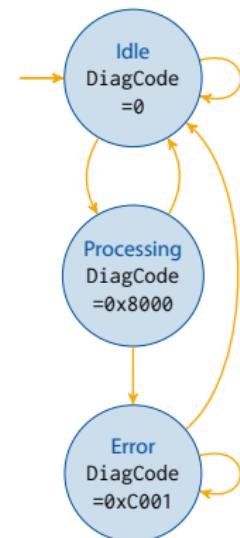
⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$(h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0x8000)$$

$$\wedge (h.DiagCode = 0x8000 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

$$\wedge (h.DiagCode = 0xC001 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

► Add to encoding of each call of ReqHandler



Mode Space as Call Summary

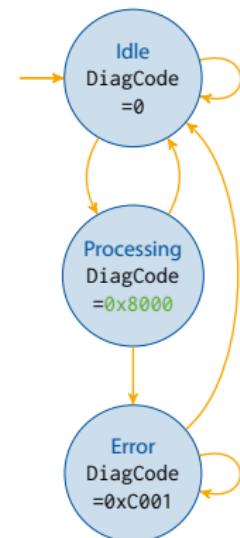
- ▶ Mode space **constrains** possible transitions
- ⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$(h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0x8000)$$

$$\wedge (h.DiagCode = 0x8000 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

$$\wedge (h.DiagCode = 0xC001 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

- ▶ Add to encoding of each call of ReqHandler



Mode Space as Call Summary

► Mode space **constrains** possible transitions

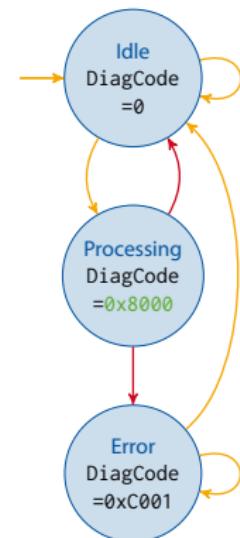
⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$(h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0x8000)$$

$$\wedge (h.DiagCode = 0x8000 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

$$\wedge (h.DiagCode = 0xC001 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

► Add to encoding of each call of ReqHandler



Mode Space as Call Summary

► Mode space **constrains** possible transitions

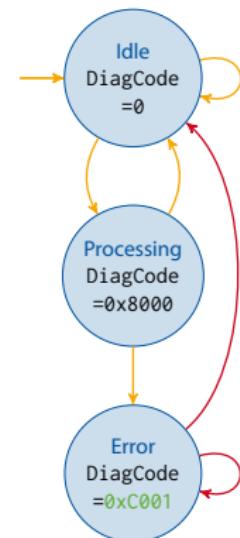
⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$(h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0x8000)$$

$$\wedge (h.DiagCode = 0x8000 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

$$\wedge (h.DiagCode = 0xC001 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0x8000)$$

► Add to encoding of each call of ReqHandler



Mode Space as Call Summary

- Mode space **constrains** possible transitions

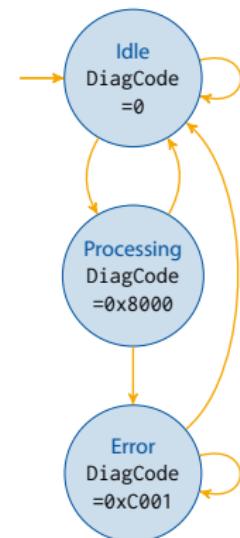
⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$(h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0x8000)$$

$$\wedge (h.DiagCode = 0x8000 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

$$\wedge (h.DiagCode = 0xC001 \rightarrow h.DiagCode' = 0 \\ \vee h.DiagCode' = 0xC001)$$

- Add to encoding of each call of ReqHandler



Mode Abstraction

Experiments

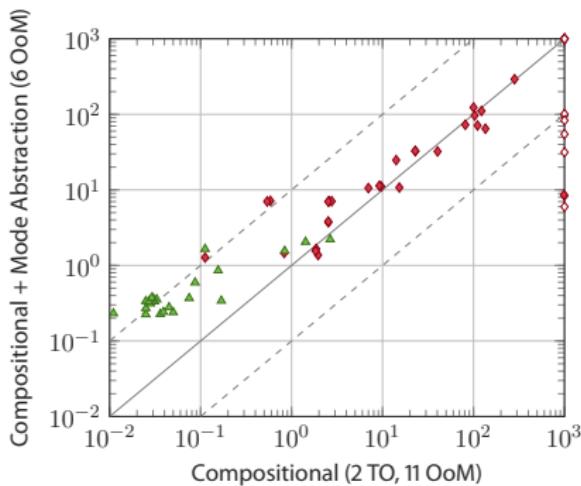
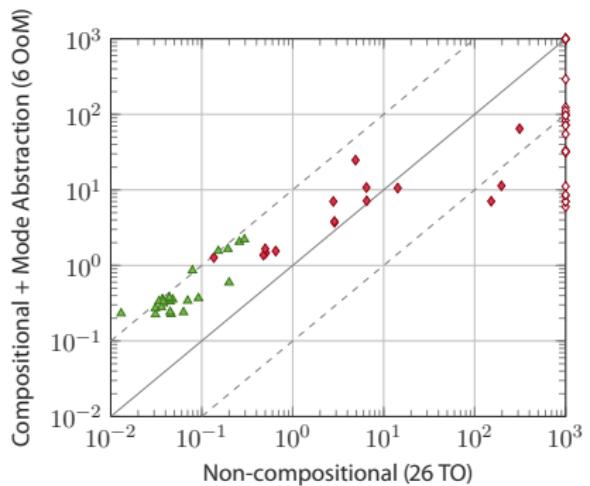


Figure: Time [s] spent with mode abstraction and solving compositional CHCs

Restart-Behaviour

However:

- ▶ A proof holds w.r.t. the formal model – not the real system
- ⇒ Model is usually **missing behaviour** enabled by hardware

Battery-backed memory & restart-functionality are widespread:

- ▶ Non-volatile state variables allow for “restart-robust” designs
- ▶ Restarts may be
 - ▶ triggered by a watchdog timer
 - ▶ triggered by a power failure
 - ▶ triggered by a user command

Restart-Behaviour

However:

- ▶ A proof holds w.r.t. the formal model – not the real system
- ⇒ Model is usually **missing behaviour** enabled by hardware

Battery-backed memory & restart-functionality are widespread:

- ▶ Non-volatile state variables allow for “restart-robust” designs
- ▶ Restarts may be
 - ▶ triggered by a watchdog timer
 - ▶ triggered by a power failure
 - ▶ triggered by a user command

Restart-Behaviour

However:

- ▶ A proof holds w.r.t. the formal model – not the real system
- ⇒ Model is usually **missing behaviour** enabled by hardware

Battery-backed memory & restart-functionality are widespread:

- ▶ Non-volatile state variables allow for “**restart-robust**” designs
- ▶ Restarts may be
 - triggered by a watchdog timer
 - the result of a power outage or voltage fluctuation
 - triggered manually, e.g. during testing

Restart-Behaviour

However:

- ▶ A proof holds w.r.t. the formal model – not the real system
- ⇒ Model is usually **missing behaviour** enabled by hardware

Battery-backed memory & restart-functionality are widespread:

- ▶ Non-volatile state variables allow for “**restart-robust**” designs
- ▶ Restarts may be
 - triggered by a **watchdog timer**
 - the result of a power outage or voltage fluctuation
 - triggered **manually**, e. g. during testing

Restart-Behaviour

However:

- ▶ A proof holds w.r.t. the formal model – not the real system
- ⇒ Model is usually **missing behaviour** enabled by hardware

Battery-backed memory & restart-functionality are widespread:

- ▶ Non-volatile state variables allow for “**restart-robust**” designs
- ▶ Restarts may be
 - triggered by a **watchdog timer**
 - the result of a **power outage** or voltage fluctuation
 - triggered **manually**, e. g. during testing

Restart-Behaviour

However:

- ▶ A proof holds w.r.t. the formal model – not the real system
- ⇒ Model is usually **missing behaviour** enabled by hardware

Battery-backed memory & restart-functionality are widespread:

- ▶ Non-volatile state variables allow for “**restart-robust**” designs
- ▶ Restarts may be
 - triggered by a **watchdog timer**
 - the result of a **power outage** or voltage fluctuation
 - triggered **manually**, e. g. during testing

Managing Restart-Behaviour

- ▶ Restarts significantly increase the number of **corner cases**
- ▶ Different semantics of writing to battery-backed memory exist
 - ▶ Immediate writing to the battery-backed memory
 - ▶ Delayed writing at the current execution cycle's end
- ▶ Choice of retain-variables and **reasoning left to developer**

Example (Automated drilling of holes in workpieces)

- ▶ Let drill's position be volatile
- ▶ A restart may result in unintended movement and **damage**
- ▶ Even though restart-free operation might be as expected

Managing Restart-Behaviour

- ▶ Restarts significantly increase the number of **corner cases**
- ▶ **Different semantics** of writing to battery-backed memory exist
 - Immediate writing to the battery-backed memory
 - Delayed writing at the current execution cycle's end
- ▶ Choice of retain-variables and **reasoning left to developer**

Example (Automated drilling of holes in workpieces)

- ▶ Let drill's position be volatile
- ▶ A restart may result in unintended movement and **damage**
- ▶ Even though restart-free operation might be as expected

Managing Restart-Behaviour

- ▶ Restarts significantly increase the number of **corner cases**
- ▶ **Different semantics** of writing to battery-backed memory exist
 - Immediate writing to the battery-backed memory
 - Delayed writing at the current execution cycle's end
- ▶ Choice of retain-variables and **reasoning left to developer**

Example (Automated drilling of holes in workpieces)

- ▶ Let drill's position be volatile
- ▶ A restart may result in unintended movement and damage
- ▶ Even though restart-free operation might be as expected

Managing Restart-Behaviour

- ▶ Restarts significantly increase the number of **corner cases**
- ▶ **Different semantics** of writing to battery-backed memory exist
 - Immediate writing to the battery-backed memory
 - Delayed writing at the current execution cycle's end
- ▶ Choice of retain-variables and **reasoning left to developer**

Example (Automated drilling of holes in workpieces)

- ▶ Let drill's position be volatile
- ▶ A restart may result in unintended movement and damage
- ▶ Even though restart-free operation might be as expected

Managing Restart-Behaviour

- ▶ Restarts significantly increase the number of **corner cases**
- ▶ **Different semantics** of writing to battery-backed memory exist
 - Immediate writing to the battery-backed memory
 - Delayed writing at the current execution cycle's end
- ▶ Choice of retain-variables and **reasoning left to developer**

Example (Automated drilling of holes in workpieces)

- ▶ Let drill's position be volatile
- ▶ A restart may result in unintended movement and damage
- ▶ Even though restart-free operation might be as expected

Managing Restart-Behaviour

- ▶ Restarts significantly increase the number of **corner cases**
- ▶ **Different semantics** of writing to battery-backed memory exist
 - Immediate writing to the battery-backed memory
 - Delayed writing at the current execution cycle's end
- ▶ Choice of retain-variables and **reasoning left to developer**

Example (Automated drilling of holes in workpieces)

- ▶ Let drill's position be volatile
- ▶ A restart may result in **unintended movement** and **damage**
- ▶ Even though restart-free operation might be as expected

Managing Restart-Behaviour

- ▶ Restarts significantly increase the number of **corner cases**
- ▶ **Different semantics** of writing to battery-backed memory exist
 - Immediate writing to the battery-backed memory
 - Delayed writing at the current execution cycle's end
- ▶ Choice of retain-variables and **reasoning left to developer**

Example (Automated drilling of holes in workpieces)

- ▶ Let drill's position be volatile
- ▶ A restart may result in **unintended movement** and **damage**
- ▶ Even though restart-free operation might be as expected

Managing Restart-Behaviour

- ▶ Restarts significantly increase the number of **corner cases**
- ▶ **Different semantics** of writing to battery-backed memory exist
 - Immediate writing to the battery-backed memory
 - Delayed writing at the current execution cycle's end
- ▶ Choice of retain-variables and **reasoning left to developer**

Example (Automated drilling of holes in workpieces)

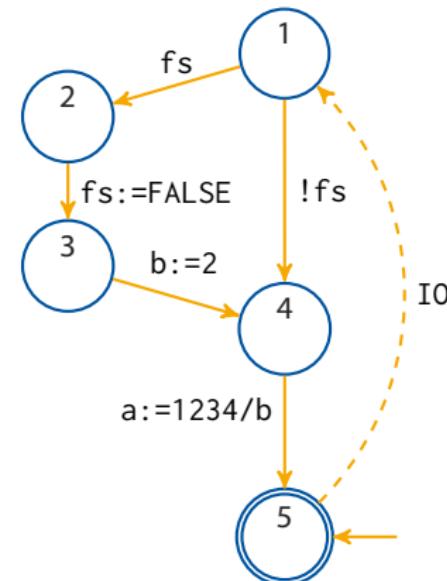
- ▶ Let drill's position be volatile
- ▶ A restart may result in **unintended movement** and **damage**
- ▶ Even though **restart-free** operation might be as expected

Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant?

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes?
- ▶ Fixable for delayed writes?
- ▶ Robust with immediate writes?
- ▶ Fixable for immediate writes?

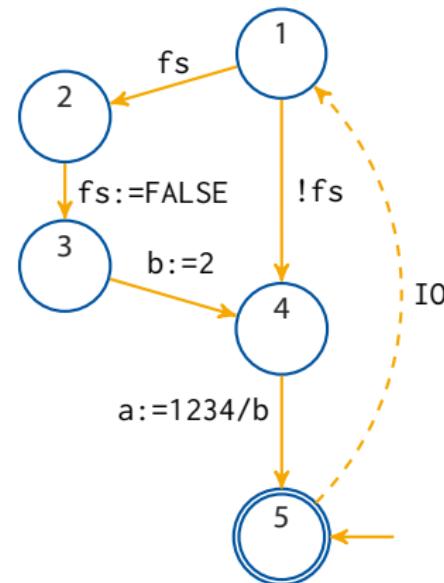


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant?

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes?
- ▶ Fixable for delayed writes?
- ▶ Robust with immediate writes?
- ▶ Fixable for immediate writes?

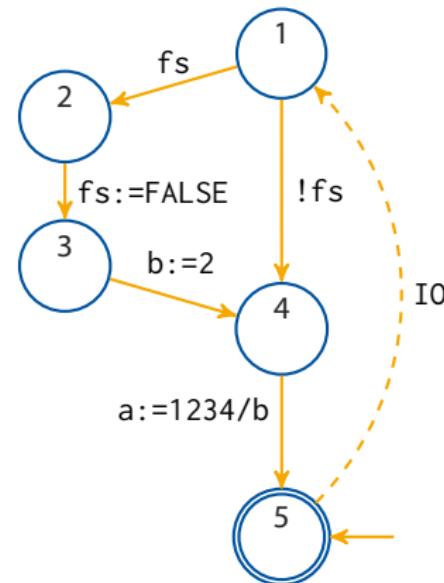


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant?

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes?
- ▶ Fixable for delayed writes?
- ▶ Robust with immediate writes?
- ▶ Fixable for immediate writes?

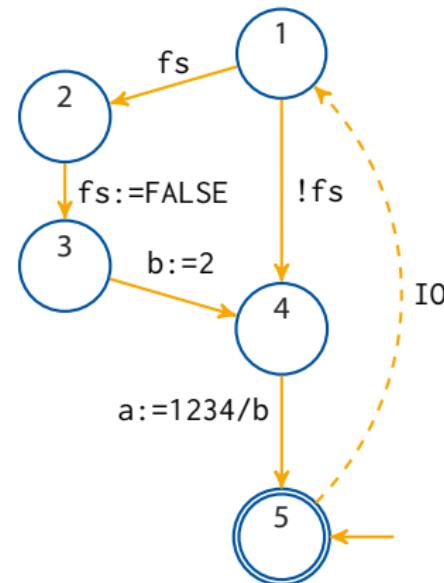


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant? ✓

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes?
- ▶ Fixable for delayed writes?
- ▶ Robust with immediate writes?
- ▶ Fixable for immediate writes?

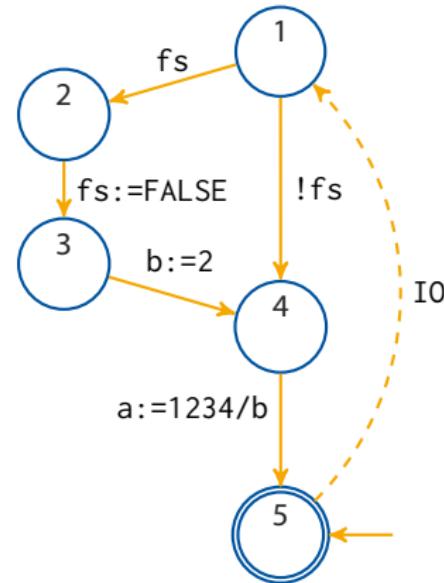


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant? ✓

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes?
- ▶ Fixable for delayed writes?
- ▶ Robust with immediate writes?
- ▶ Fixable for immediate writes?

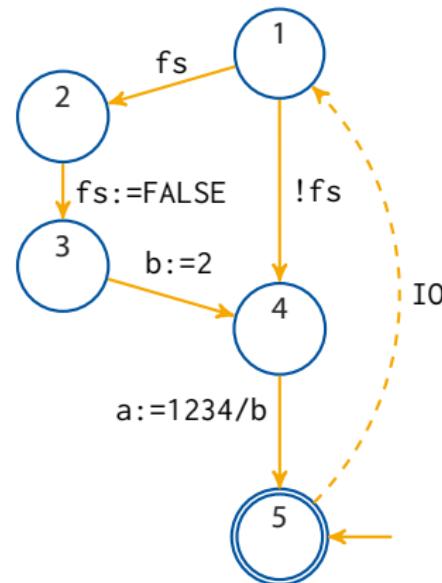


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant? ✓

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes?
- ▶ Fixable for delayed writes?
- ▶ Robust with immediate writes?
- ▶ Fixable for immediate writes?

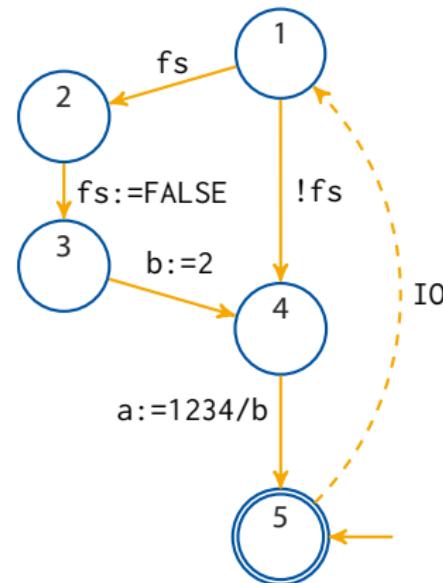


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant? ✓

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes? $a := 1234/0$
- ▶ Fixable for delayed writes?
- ▶ Robust with immediate writes?
- ▶ Fixable for immediate writes?

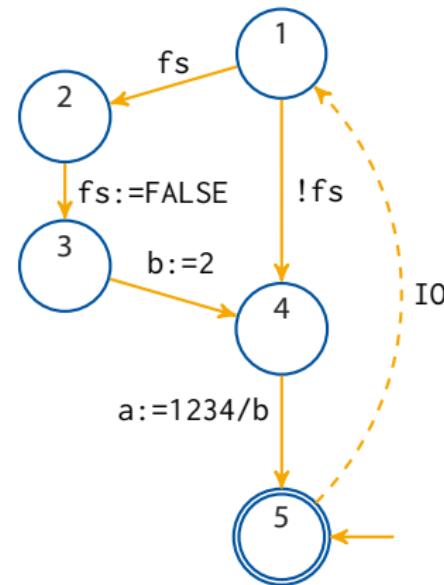


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant? ✓

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes? $a := 1234/0$
- ▶ Fixable for delayed writes?
- ▶ Robust with immediate writes?
- ▶ Fixable for immediate writes?

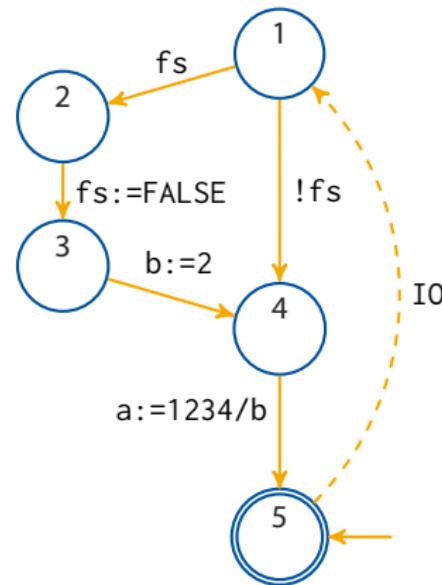


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant? ✓

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes? $a := 1234/0$
- ▶ Fixable for delayed writes? Retain b
- ▶ Robust with immediate writes?
- ▶ Fixable for immediate writes?

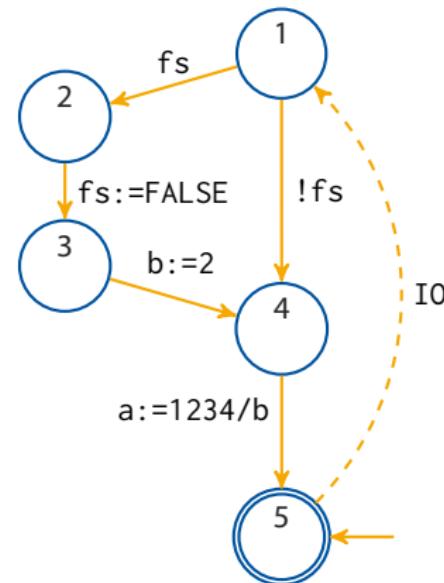


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant? ✓

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes? $a := 1234/0$
- ▶ Fixable for delayed writes? Retain b
- ▶ Robust with immediate writes?
- ▶ Fixable for immediate writes?

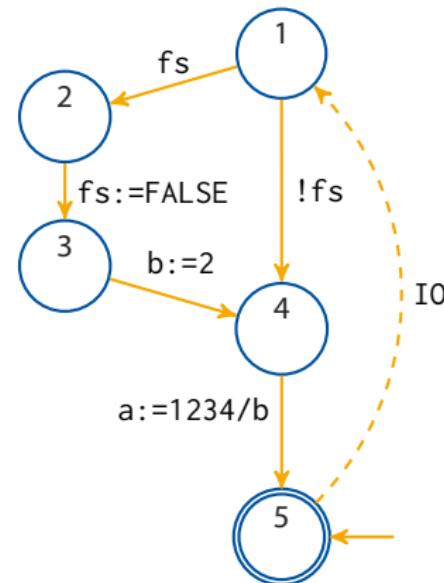


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant? ✓

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes? $a := 1234/0$
- ▶ Fixable for delayed writes? Retain b
- ▶ Robust with immediate writes? ✗
- ▶ Fixable for immediate writes?

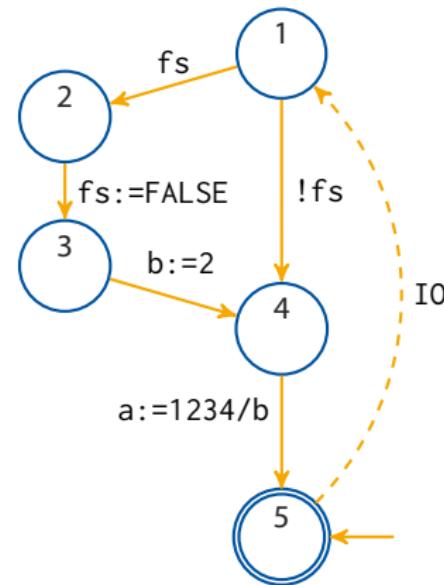


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant? ✓

In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes? $a := 1234/0$
- ▶ Fixable for delayed writes? Retain b
- ▶ Robust with immediate writes? ✗
- ▶ Fixable for immediate writes?

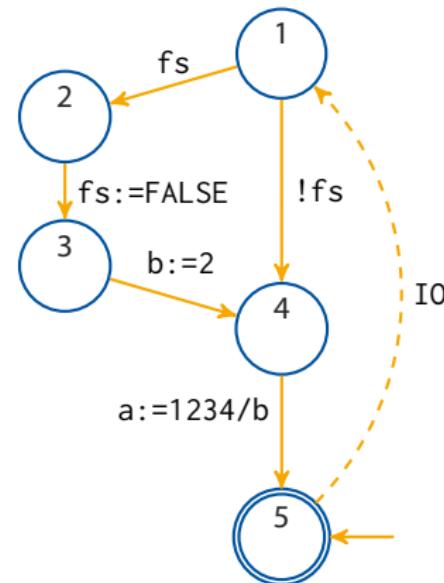


Toy Example: Invariant $a \geq 0$

- ▶ Initially $fs \mapsto true, a \mapsto 0, b \mapsto 0$
- ▶ Nominal behaviour compliant? ✓

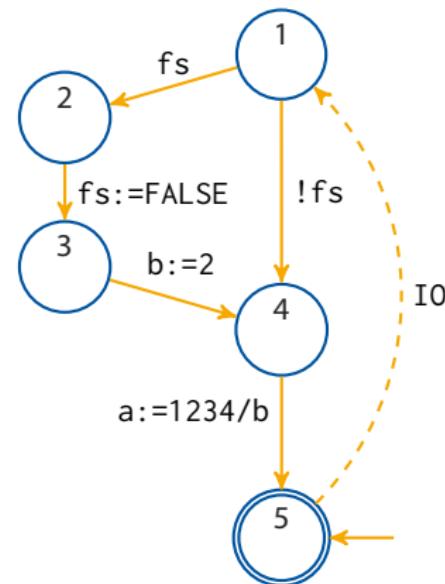
In context of restarts

- ▶ Let the flag fs be retained
- ▶ Robust with delayed writes? $a := 1234/0$
- ▶ Fixable for delayed writes? Retain b
- ▶ Robust with immediate writes? ✗
- ▶ Fixable for immediate writes? ✗



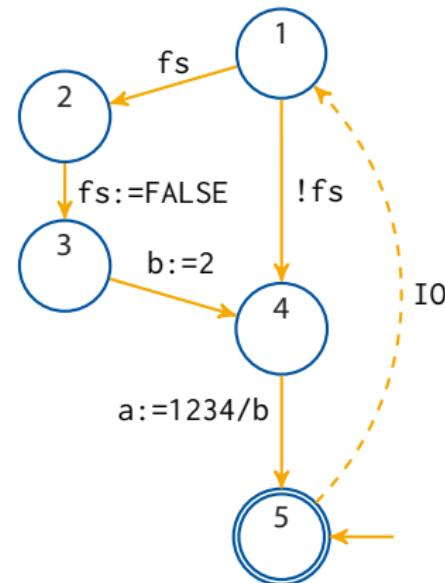
Delayed Write Semantics

- ▶ Approached by **instrumenting the CFA** with restart-behaviour
- ▶ Observation: On restart, operations since last cycle are **irrelevant**
- ⇒ Model as nondeterministic choice:
restart in next cycle?
- ▶ Handle immediate writes similarly



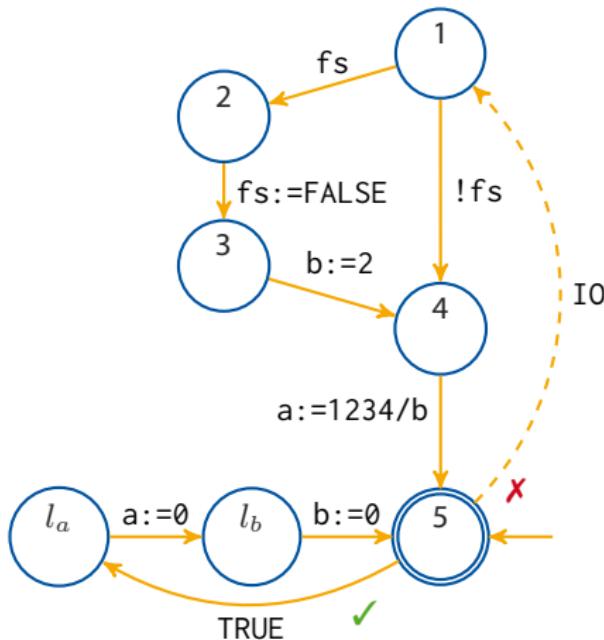
Delayed Write Semantics

- ▶ Approached by **instrumenting the CFA** with restart-behaviour
- ▶ **Observation:** On restart, operations since last cycle are **irrelevant**
- ⇒ Model as nondeterministic choice:
restart in next cycle?
- ▶ Handle immediate writes similarly



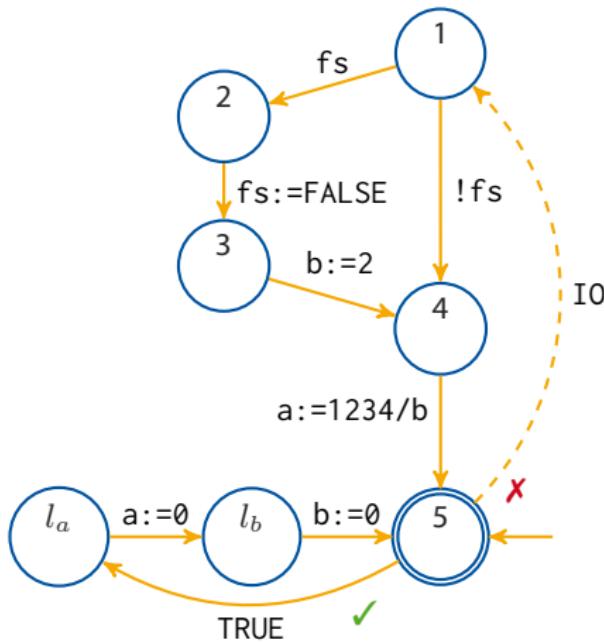
Delayed Write Semantics

- Approached by **instrumenting the CFA** with restart-behaviour
- Observation:** On restart, operations since last cycle are **irrelevant**
- Model as **nondeterministic choice**: restart in next cycle?
- Handle immediate writes similarly



Delayed Write Semantics

- Approached by **instrumenting the CFA** with restart-behaviour
- Observation:** On restart, operations since last cycle are **irrelevant**
- Model as **nondeterministic choice**: restart in next cycle?
- Handle **immediate writes** similarly



Experiments

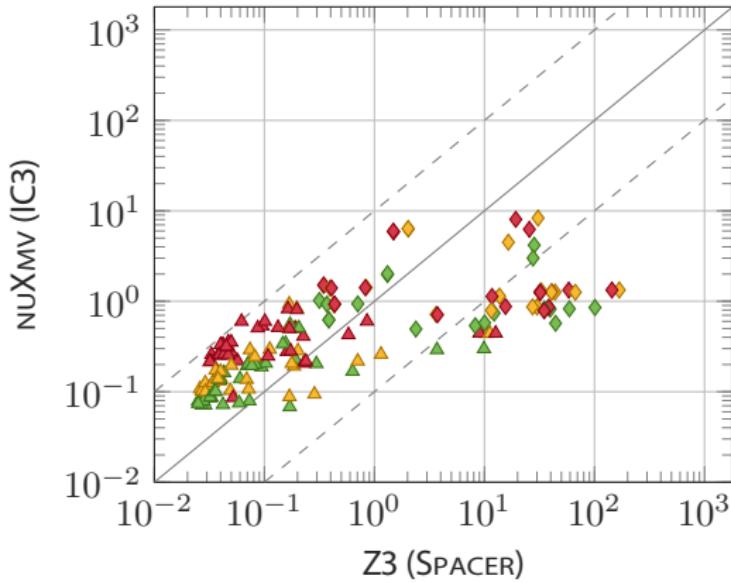


Figure: Time [s] spent checking restart-robustness w.r.t. each specification

Synthesis of Safe Retain Configurations

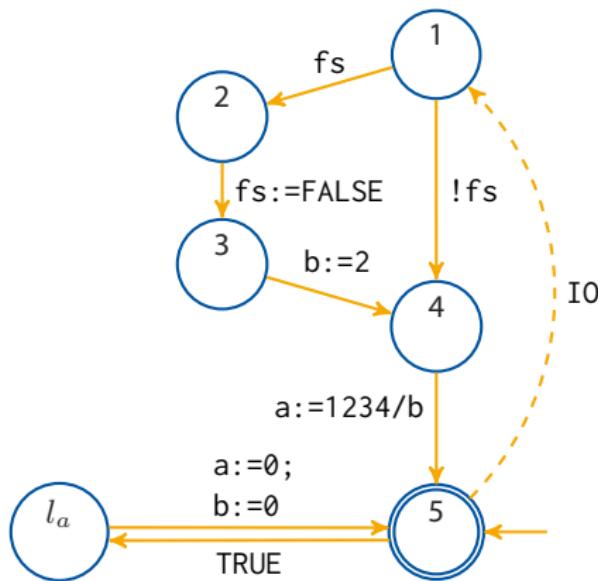
CHC-based Parameter Synthesis

- ▶ No aid in picking safe configuration of retain variables
- ⇒ Add Boolean parameter `ret_v` for each non-retain variable v
- ▶ Derived CHCs check whether all configurations are robust

$$\forall \vec{V} \underbrace{\dots}_{\text{body}} \rightarrow h(\vec{V})$$

- ▶ Parameter synthesis is

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \dots \rightarrow h(\vec{V})$$



Synthesis of Safe Retain Configurations

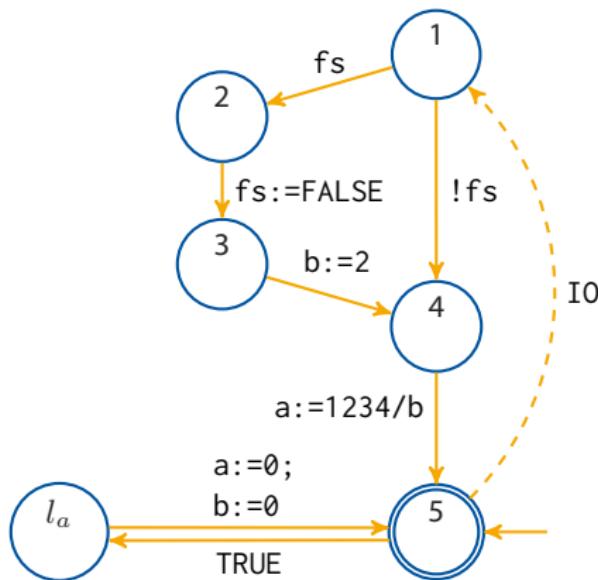
CHC-based Parameter Synthesis

- ▶ No aid in picking safe configuration of retain variables
- ⇒ Add Boolean parameter `ret_v` for each non-retain variable v
- ▶ Derived CHCs check whether all configurations are robust

$$\forall \vec{V} \underbrace{\dots}_{\text{body}} \rightarrow h(\vec{V})$$

- ▶ Parameter synthesis is

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \dots \rightarrow h(\vec{V})$$



CHC-based Parameter Synthesis

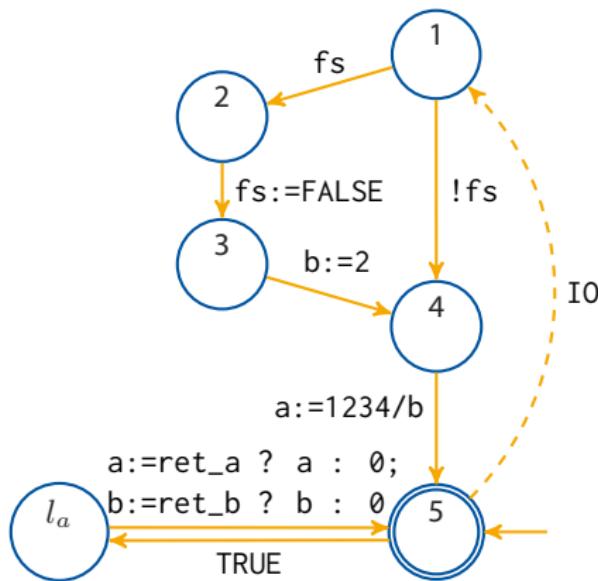
- ▶ No aid in picking safe configuration of retain variables
- ⇒ Add Boolean parameter `ret_v` for each non-retain variable v
- ▶ Derived CHCs check whether all configurations are robust

$$\forall \vec{V} \dots \rightarrow h(\vec{V})$$

body

- ▶ Parameter synthesis is

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \dots \rightarrow h(\vec{V})$$



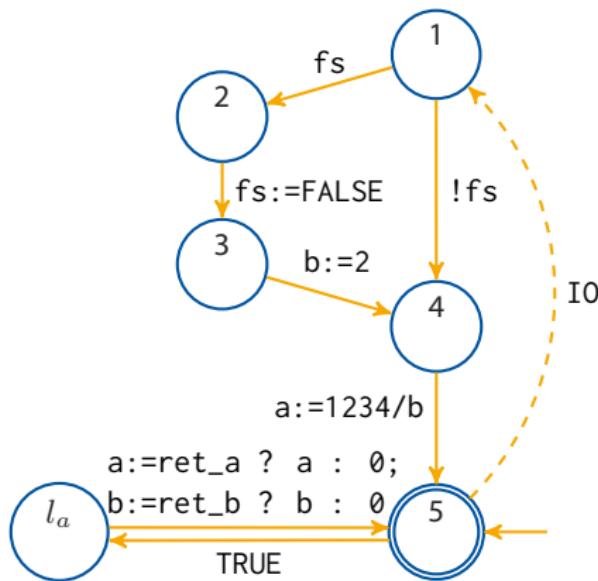
CHC-based Parameter Synthesis

- ▶ No aid in picking safe configuration of retain variables
- ⇒ Add Boolean parameter `ret_v` for each non-retain variable v
- ▶ Derived CHCs check whether all configurations are robust

$$\forall \vec{V} \underbrace{\dots}_{\text{body}} \rightarrow h(\vec{V})$$

- ▶ Parameter synthesis is

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \dots \rightarrow h(\vec{V})$$



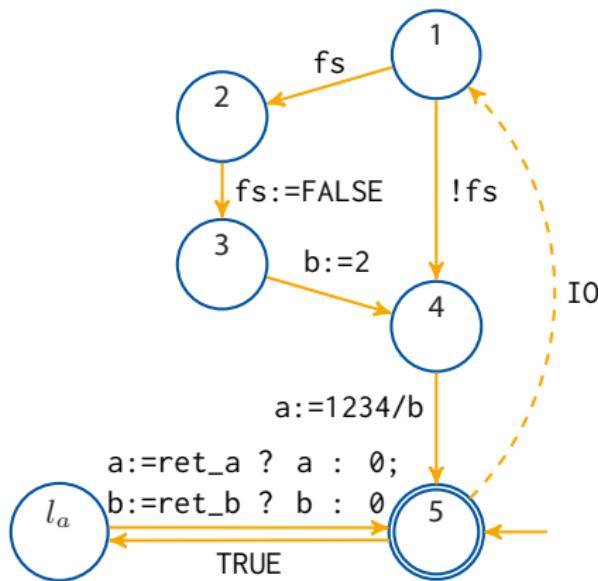
CHC-based Parameter Synthesis

- ▶ No aid in picking safe configuration of retain variables
- ⇒ Add Boolean parameter `ret_v` for each non-retain variable v
- ▶ Derived CHCs check whether all configurations are robust

$$\forall \vec{V} \underbrace{\dots}_{\text{body}} \rightarrow h(\vec{V})$$

- ▶ Parameter synthesis is

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \dots \rightarrow h(\vec{V})$$



Counterexample-Guided Parameter Synthesis

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \dots \rightarrow h(\vec{V})$$

Observations:

- ▶ $\exists \forall$ -quantified Horn clauses **harder** than regular CHCs (47 TO)
- ▶ Our special case: existential quantification over Booleans

Idea:

- ▶ Manage choice and reuse efficient check for fixed parameters
 - Over-approximate set of "safe" parameters
 - Refine it while counterexamples exist

Counterexample-Guided Parameter Synthesis

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \dots \rightarrow h(\vec{V})$$

Observations:

- ▶ $\exists \forall$ -quantified Horn clauses **harder** than regular CHCs (47 TO)
- ▶ Our **special case**: existential quantification over Booleans

Idea:

- ▶ Manage choice and reuse efficient check for fixed parameters
 - Over-approximate set of "safe" parameters
 - Refine it while counterexamples exist

Counterexample-Guided Parameter Synthesis

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \dots \rightarrow h(\vec{V})$$

Observations:

- ▶ $\exists \forall$ -quantified Horn clauses **harder** than regular CHCs (47 TO)
- ▶ Our **special case**: existential quantification over Booleans

Idea:

- ▶ Manage choice and **reuse efficient check** for fixed parameters
- ▶ Over-approximate set of “safe” parameters
- ▶ Refine it while counterexamples exist

Counterexample-Guided Parameter Synthesis

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \dots \rightarrow h(\vec{V})$$

Observations:

- ▶ $\exists \forall$ -quantified Horn clauses **harder** than regular CHCs (47 TO)
- ▶ Our **special case**: existential quantification over Booleans

Idea:

- ▶ Manage choice and **reuse efficient check** for fixed parameters
- ▶ **Over-approximate** set of “safe” parameters
- ▶ Refine it while counterexamples exist

Counterexample-Guided Parameter Synthesis

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \dots \rightarrow h(\vec{V})$$

Observations:

- ▶ $\exists \forall$ -quantified Horn clauses **harder** than regular CHCs (47 TO)
- ▶ Our **special case**: existential quantification over Booleans

Idea:

- ▶ Manage choice and **reuse efficient check** for fixed parameters
- ▶ **Over-approximate** set of “safe” parameters
- ▶ **Refine it** while counterexamples exist

Synthesis of Safe Retain Configurations

Example

- ▶ Make the program restart-robust w.r.t. $a \geq 0$ under delayed writes
- ▶ Let fs be required to be retained

Process:

1. Start with $safe(\vec{V}_{par}) = true$
2. Backend finds counterexample

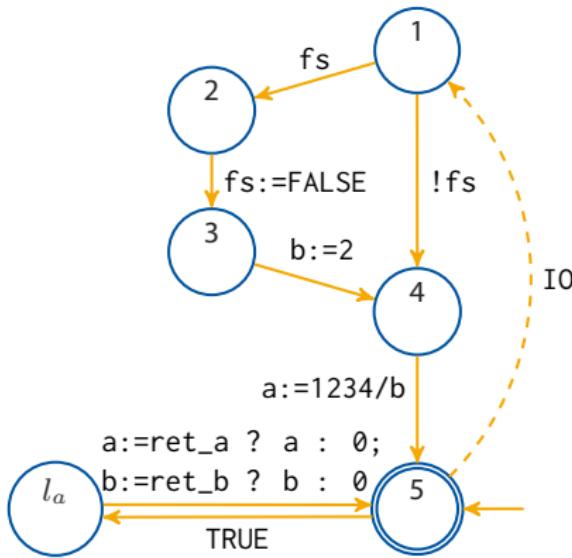
$$c = \neg ret_a \wedge \neg ret_b$$

3. Find subset of violating parameters

$$c_g = \neg ret_b$$

4. Refine $safe(\vec{V}_{par}) = true \wedge \neg c_g$

5. Backend finds no violations



Example

- ▶ Make the program restart-robust w.r.t. $a \geq 0$ under **delayed writes**
- ▶ Let **fs** be required to be **retained**

Process:

1. Start with $\text{safe}(\vec{V}_{\text{par}}) = \text{true}$
2. Backend finds **counterexample**

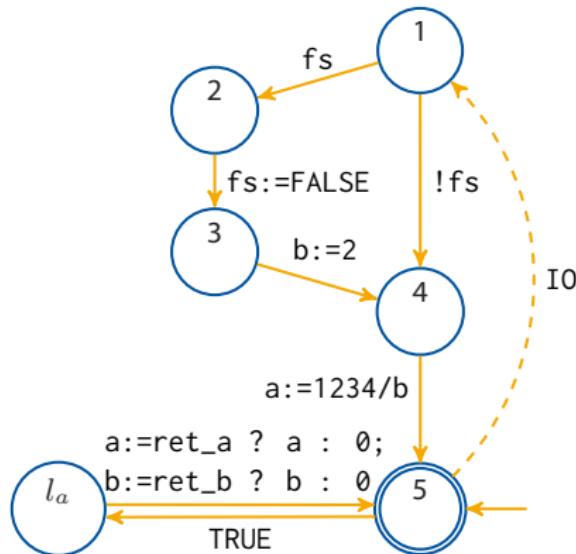
$$c = \neg \text{ret}_a \wedge \neg \text{ret}_b$$

3. Find subset of violating parameters

$$c_g = \neg \text{ret}_b$$

4. Refine $\text{safe}(\vec{V}_{\text{par}}) = \text{true} \wedge \neg c_g$

5. Backend finds **no violations**



Example

- ▶ Make the program restart-robust w.r.t. $a \geq 0$ under **delayed writes**
- ▶ Let **fs** be required to be **retained**

Process:

1. Start with $\text{safe}(\vec{V}_{\text{par}}) = \text{true}$
2. Backend finds **counterexample**

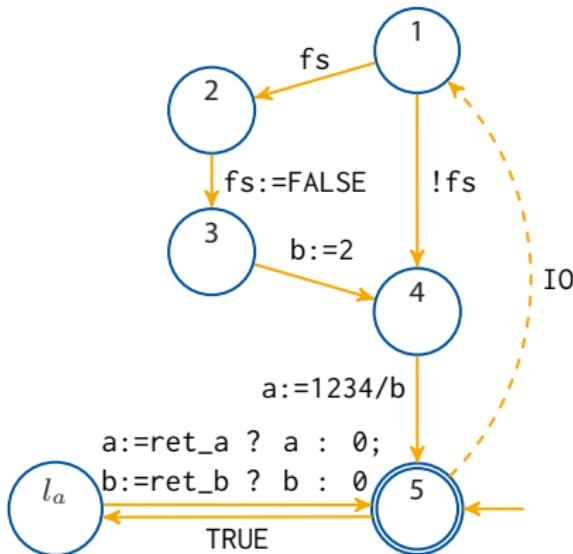
$$c = \neg \text{ret}_a \wedge \neg \text{ret}_b$$

3. Find subset of violating parameters

$$c_g = \neg \text{ret}_b$$

4. Refine $\text{safe}(\vec{V}_{\text{par}}) = \text{true} \wedge \neg c_g$

5. Backend finds **no violations**



Example

- ▶ Make the program restart-robust w.r.t. $a \geq 0$ under **delayed writes**
- ▶ Let **fs** be required to be **retained**

Process:

1. Start with $\text{safe}(\vec{V}_{\text{par}}) = \text{true}$
2. Backend finds **counterexample**

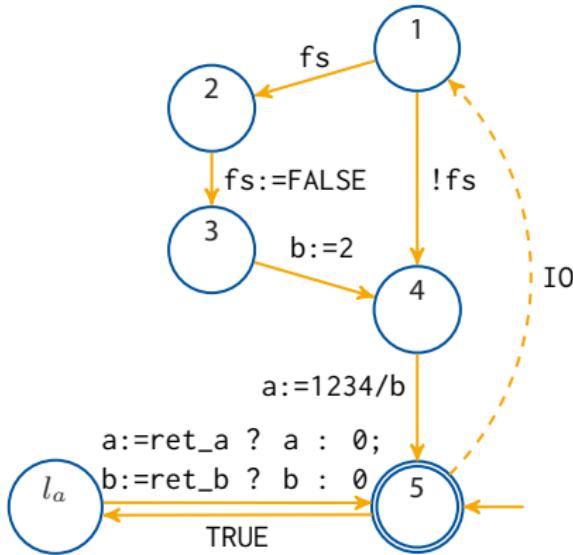
$$c = \neg \text{ret}_a \wedge \neg \text{ret}_b$$

3. Find subset of **violating parameters**

$$c_g = \neg \text{ret}_b$$

4. Refine $\text{safe}(\vec{V}_{\text{par}}) = \text{true} \wedge \neg c_g$

5. Backend finds **no violations**



Example

- ▶ Make the program restart-robust w.r.t. $a \geq 0$ under **delayed writes**
- ▶ Let **fs** be required to be **retained**

Process:

1. Start with $\text{safe}(\vec{V}_{\text{par}}) = \text{true}$
2. Backend finds **counterexample**

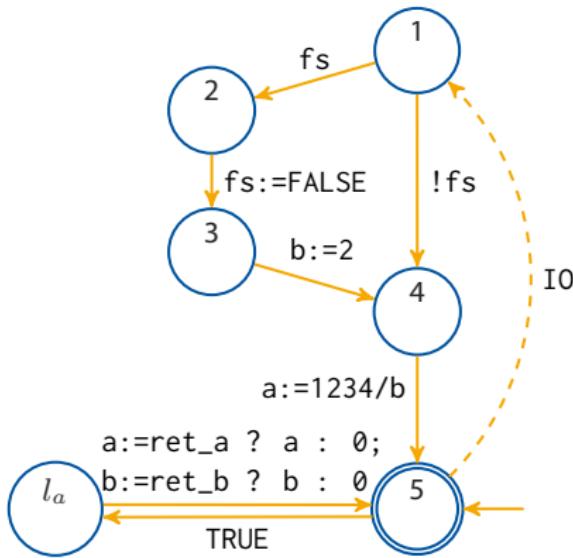
$$c = \neg \text{ret}_a \wedge \neg \text{ret}_b$$

3. Find subset of **violating parameters**

$$c_g = \neg \text{ret}_b$$

4. Refine $\text{safe}(\vec{V}_{\text{par}}) = \text{true} \wedge \neg c_g$

5. Backend finds **no violations**



Example

- ▶ Make the program restart-robust w.r.t. $a \geq 0$ under **delayed writes**
- ▶ Let **fs** be required to be **retained**

Process:

1. Start with $\text{safe}(\vec{V}_{\text{par}}) = \text{true}$
2. Backend finds **counterexample**

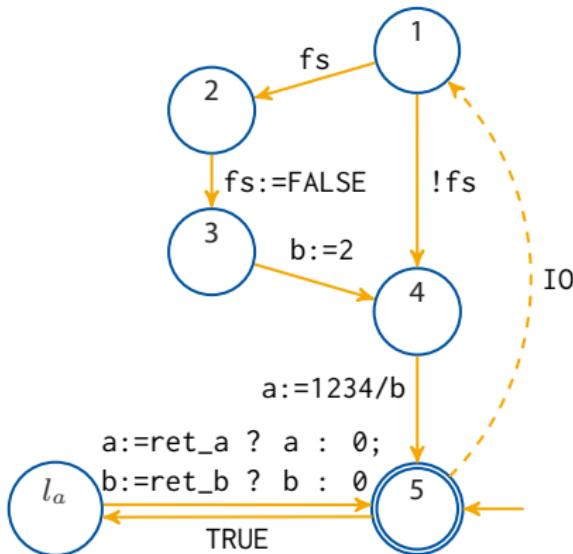
$$c = \neg \text{ret}_a \wedge \neg \text{ret}_b$$

3. Find subset of **violating parameters**

$$c_g = \neg \text{ret}_b$$

4. Refine $\text{safe}(\vec{V}_{\text{par}}) = \text{true} \wedge \neg c_g$

5. Backend finds **no violations**



Synthesis of Safe Retain Configurations

Experiments

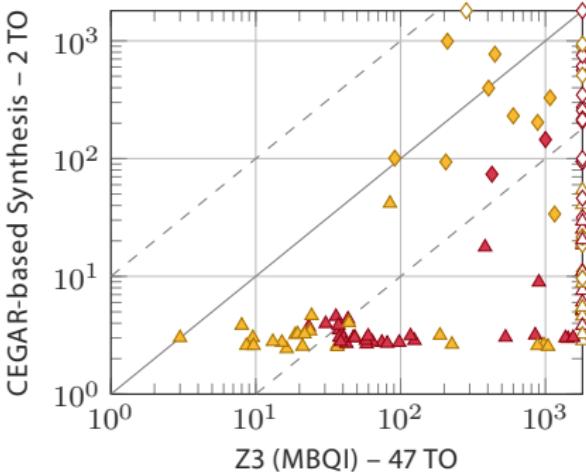
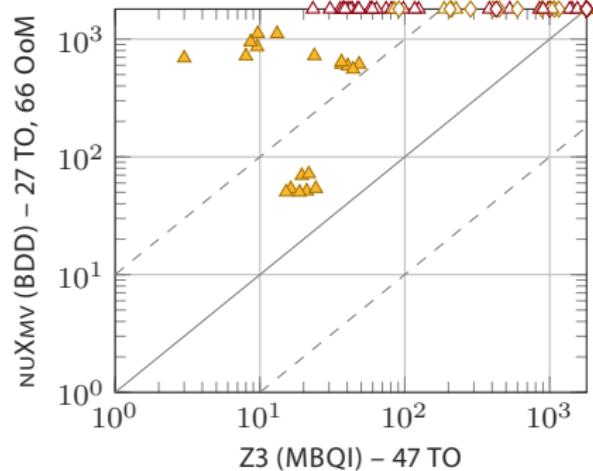


Figure: Time [s] spent on synthesis of restart-robust configurations

Summary

- ▶ Software verification machinery **hardly used in industrial control**
- ▶ Most focus on checking **common specifications** with existing tooling

- ▶ We proposed **SMT-based verification** procedures
- ▶ **Competitive** with existing tooling
- ▶ Enabled verification of previously
 - “**problematic**” tasks
 - unsupported **domain-specific specifications**

References I

- [Bec+15] Bernhard Beckert, Mattias Ulbrich, Birgit Vogel-Heuser and Alexander Weigl. 'Regression Verification for Programmable Logic Controller Software'. In: *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*. Ed. by Michael J. Butler, Sylvain Conchon and Fatiha Zaïdi. Vol. 9407. Lecture Notes in Computer Science. Springer, 2015, pp. 234–251.

References II

- [Bec+17] Bernhard Beckert, Suhyun Cha, Mattias Ulbrich, Birgit Vogel-Heuser and Alexander Weigl. 'Generalised Test Tables: A Practical Specification Language for Reactive Systems'. In: *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings*. Ed. by Nadia Polikarpova and Steve Schneider. Vol. 10510. Lecture Notes in Computer Science. Springer, 2017, pp. 129–144.
- [BHK18] Dimitri Bohlender, Daniel Hamm and Stefan Kowalewski. 'Cycle-Bounded Model Checking of PLC Software via Dynamic Large-Block Encoding'. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*. Ed. by Hisham M. Haddad, Roger L. Wainwright and Richard Chbeir. ACM, 2018, pp. 1891–1898.

References III

- [Bia16] Sebastian Biallas. 'Verification of Programmable Logic Controller Code using Model Checking and Static Analysis'. PhD thesis. RWTH Aachen University, Germany, 2016.
- [BK18a] Dimitri Bohlender and Stefan Kowalewski. 'Compositional Verification of PLC Software using Horn Clauses and Mode Abstraction'. In: *IFAC-PapersOnLine* 51.7 (2018). 14th IFAC Workshop on Discrete Event Systems WODES 2018, pp. 428 –433.

References IV

- [BK18b] Dimitri Bohlender and Stefan Kowalewski. 'Design and Verification of Restart-Robust Industrial Control Software'. In: *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings*. Ed. by Carlo A. Furia and Kirsten Winter. Vol. 11023. Lecture Notes in Computer Science. Springer, 2018, pp. 47–68.
- [BK20] Dimitri Bohlender and Stefan Kowalewski. 'Leveraging Horn Clause Solving for Compositional Verification of PLC Software'. In: *Discrete Event Dynamic Systems* 30 (2020). To appear.

References V

- [Boh+16] Dimitri Bohlender, Hendrik Simon, Nico Friedrich, Stefan Kowalewski and Stefan Hauck-Stattelmann. 'Concolic Test Generation for PLC Programs using Coverage Metrics'. In: *13th International Workshop on Discrete Event Systems, WODES 2016, Xi'an, China, May 30 - June 1, 2016*. Ed. by Christos G. Cassandras, Alessandro Giua and Zhiwu Li. IEEE, 2016, pp. 432–437.
- [Bor06] Caridad Borras. 'Overexposure of radiation therapy patients in Panama: Problem recognition and follow-up measures'. In: *Revista panamericana de salud pública* 20 (Aug. 2006), pp. 173–87.

References VI

- [BSK16] Dimitri Bohlender, Hendrik Simon and Stefan Kowalewski. 'Symbolic Verification of PLC Safety-Applications based on PLCopen Automata'. In: *19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2016, Freiburg im Breisgau, Germany, March 1-2, 2016*. Ed. by Ralf Wimmer. Albert-Ludwigs-Universität Freiburg, 2016, pp. 33–45.
- [Dar17] Daniel Darvas. 'Practice-Oriented Formal Methods to Support the Software Development of Industrial Control Systems. Gyakorlatorientált formális módszerek az ipari vezérlőrendszerek szoftverfejlesztésének támogatására'. Presented 2017. PhD thesis. Budapest University of Technology and Economics, 2017.

References VII

- [DVA15] D. Darvas, E. Blanco Vinuela and B. Fernández Adiego. 'PLCverif: A Tool to Verify PLC Programs Based on Model Checking Techniques'. In: *Proc. of International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15), Melbourne, Australia, 17-23 October 2015* (Melbourne, Australia). International Conference on Accelerator and Large Experimental Physics Control Systems 15.
doi:10.18429/JACoW-ICALEPCS2015-WEPGF092. Geneva, Switzerland: JACoW, 2015, pp. 911–914.

References VIII

- [Gur+15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli and Jorge A. Navas. 'The SeaHorn Verification Framework'. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 343–361.
- [Kö+19] Maximilian Köhl, Dimitri Bohlender, Kevin Baum, Markus Langer, Daniel Oster and Timo Speith. 'Explainability as a Non-Functional Requirement'. In: *27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, South Korea, September 23-27, 2019*. To appear. IEEE Computer Society, 2019.

References IX

- [Lan18] Tim Felix Lange. 'IC3 software model checking'. PhD thesis. RWTH Aachen University, Germany, 2018.
- [MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.
- [Mye12] Glenford J. Myers. *The Art of Software Testing*. Ed. by Glenford J. Myers, Tom Badgett and Corey Sandler. Third Edition. Wiley, Jan. 2012.
- [Ova+16] Tolga Ovatman, Atakan Aral, Davut Polat and Ali Osman Ünver. 'An overview of model checking practices on verification of PLC software'. In: *Software and System Modeling* 15.4 (2016), pp. 937–960.

References X

- [Pow04] U.S.-Canada Power System Outage Task Force. *Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations.* 2004.
- [RE14] Zvonimir Rakamaric and Michael Emmi. 'SMACK: Decoupling Source Language Details from Verifier Implementations'. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings.* Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 106–113.