ARCADE.PLC

Symbolic Methods for Formal Verification of Industrial Control Software

Dimitri Bohlender, M. Sc. RWTH

PhD thesis defence, Aachen, 23rd September 2021

Informatik 11
Embedded Software
RWTH AACHEN UNIVERSITY

Outline

Introduction

- Formal Methods
- Setting
- Contributions & Related Work

CHC-based Safety 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

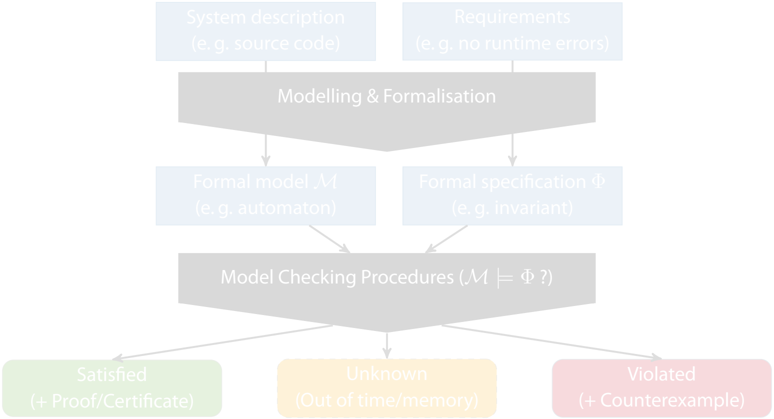
- Software drives the systems we rely on – hardware often off-the-shelf
- While many software bugs are not grave, some may be catastrophic:
  - Misinterpretation & no input validation led to radiation fatalities [Bor06]
  - Blackout after race condition affected 50 million people [Pow04]
- Writing “correct”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

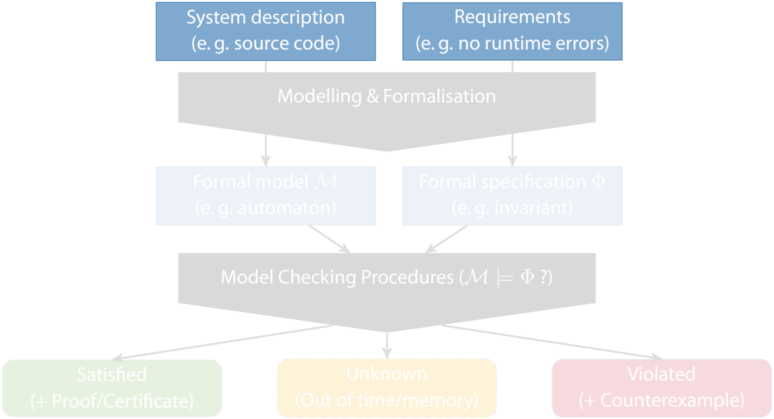
Formal Methods

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



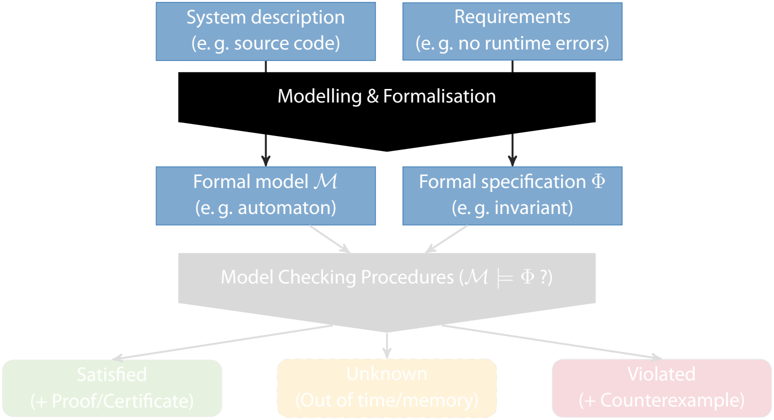
Formal Methods

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



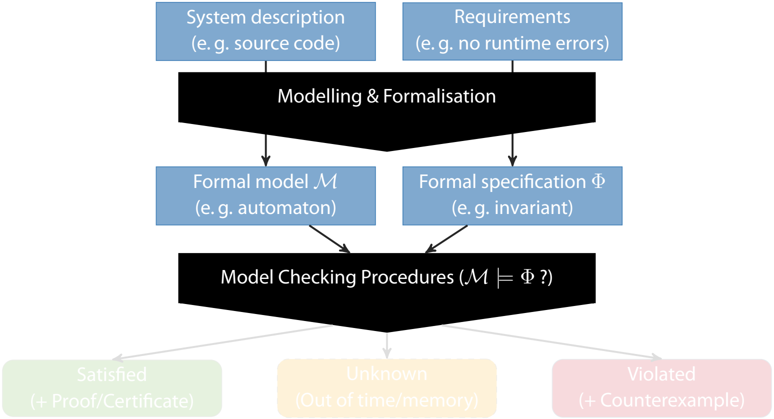
Formal Methods

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



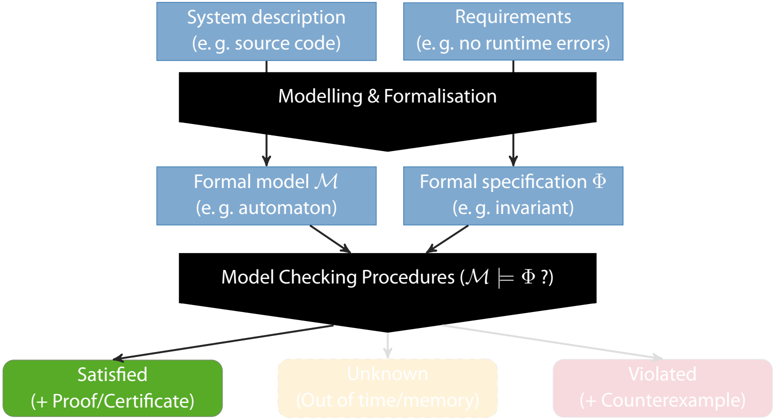
Formal Methods

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



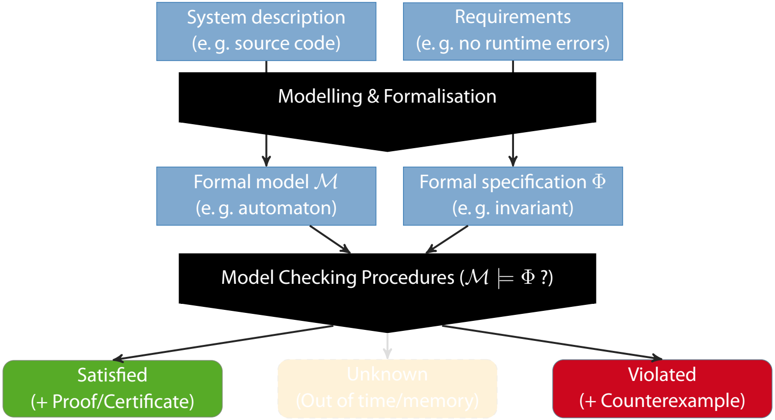
Formal Methods

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



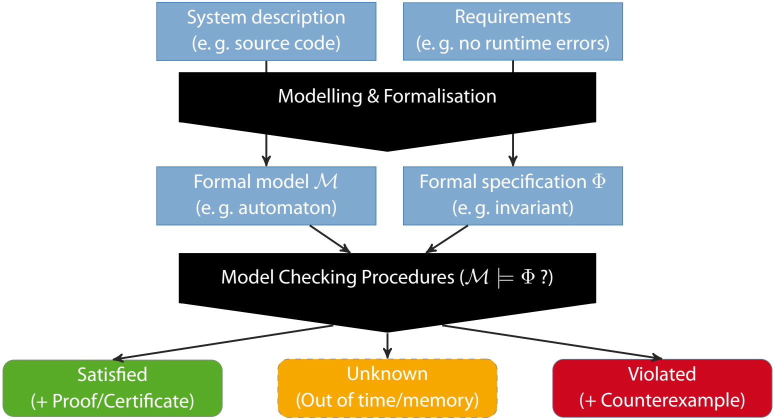
Formal Methods

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

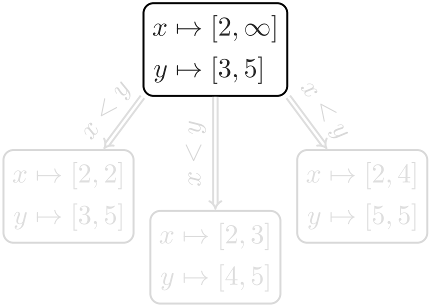


Formal Methods

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



Explicit vs. Symbolic Methods

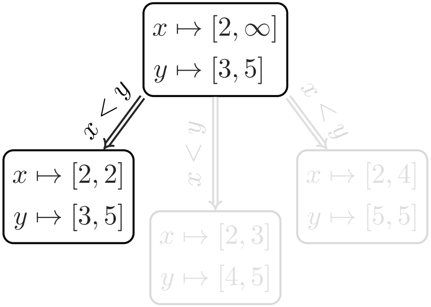


$$
\begin{aligned}
src(x, y) &:= 2 \leq x \\
&\quad \wedge 3 \leq y \leq 5 \\
T(x, y, x', y') &:= x < y \\
&\quad \wedge x' = x \wedge y' = y \\
bad(x', y') &:= x = 4
\end{aligned}
$$

- Explicit construction & search
- Precise representation needs space

- Implicit, lazy reasoning via SAT
- Exploits structure & symmetry

Explicit vs. Symbolic Methods

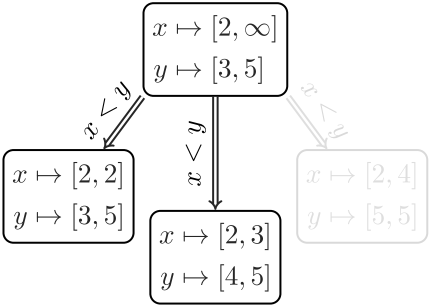


$$\begin{aligned} src(x, y) &:= 2 \leq x \\ &\quad \wedge 3 \leq y \leq 5 \\ T(x, y, x', y') &:= x < y \\ &\quad \wedge x' = x \wedge y' = y \\ bad(x', y') &:= x = 4 \end{aligned}$$

- Explicit construction & search
- Precise representation needs space

- Implicit, lazy reasoning via SAT
- Exploits structure & symmetry

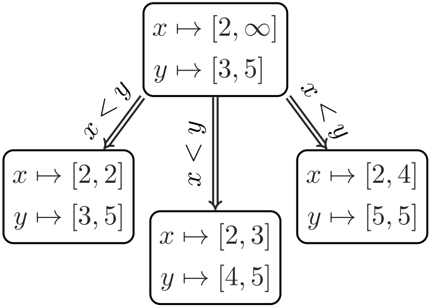
Explicit vs. Symbolic Methods



$$\begin{aligned} src(x, y) &:= 2 \leq x \\ &\quad \wedge 3 \leq y \leq 5 \\ T(x, y, x', y') &:= x < y \\ &\quad \wedge x' = x \wedge y' = y \\ bad(x', y') &:= x = 4 \end{aligned}$$

- Explicit construction & search
- Precise representation needs space
- Implicit, lazy reasoning via SAT
- Exploits structure & symmetry

Explicit vs. Symbolic Methods


$$\begin{tikzcd}
 & \boxed{\begin{array}{l} x \mapsto [2,\infty] \\ y \mapsto [3,5] \end{array}} \arrow[dl, Rightarrow, "x<y"'] \arrow[d, Rightarrow, "x<y"] \arrow[dr, Rightarrow, "x<y"] & \\
\boxed{\begin{array}{l} x \mapsto [2,2] \\ y \mapsto [3,5] \end{array}} & \boxed{\begin{array}{l} x \mapsto [2,3] \\ y \mapsto [4,5] \end{array}} & \boxed{\begin{array}{l} x \mapsto [2,4] \\ y \mapsto [5,5] \end{array}}
\end{tikzcd}$$

$$\begin{aligned} src(x,y) &:= 2 \leq x \\ &\quad \wedge 3 \leq y \leq 5 \\ T(x,y,x',y') &:= x < y \\ &\quad \wedge x' = x \wedge y' = y \\ bad(x',y') &:= x = 4 \end{aligned}$$

- Explicit construction & search
- Precise representation needs space

- Implicit, lazy reasoning via SAT
- Exploits structure & symmetry

Explicit vs. Symbolic Methods

$$\begin{tikzcd}
& \boxed{\begin{matrix} x \mapsto [2,\infty] \\ y \mapsto [3,5] \end{matrix}} \arrow[dl, Rightarrow, "x<y"'] \arrow[d, Rightarrow, "x<y"] \arrow[dr, Rightarrow, "x<y"] & \\
\boxed{\begin{matrix} x \mapsto [2,2] \\ y \mapsto [3,5] \end{matrix}} & \boxed{\begin{matrix} x \mapsto [2,3] \\ y \mapsto [4,5] \end{matrix}} & \boxed{\begin{matrix} x \mapsto [2,4] \\ y \mapsto [5,5] \end{matrix}}
\end{tikzcd}$$

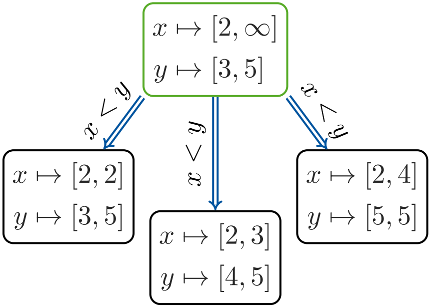
$$src(x,y) := 2 \le x \land 3 \le y \le 5$$

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

$$bad(x',y') := x = 4$$

- Explicit construction & search
- Precise representation needs space
- Implicit, lazy reasoning via SAT
- Exploits structure & symmetry

Explicit vs. Symbolic Methods

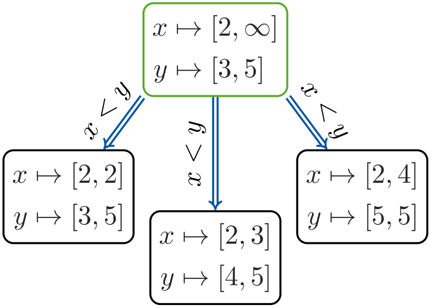


$$
\begin{aligned}
src(x, y) &:= 2 \leq x \\
&\quad \wedge 3 \leq y \leq 5 \\
T(x, y, x', y') &:= x < y \\
&\quad \wedge x' = x \wedge y' = y \\
bad(x', y') &:= x = 4
\end{aligned}
$$

- Explicit construction & search
- Precise representation needs space

- Implicit, lazy reasoning via SAT
- Exploits structure & symmetry

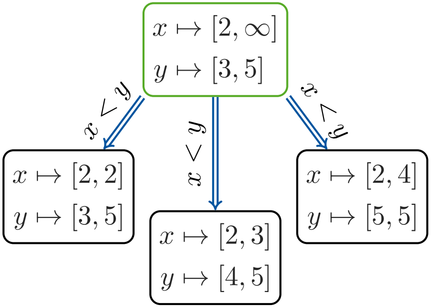
Explicit vs. Symbolic Methods



$$\begin{aligned} src(x, y) &:= 2 \leq x \\ &\quad \wedge 3 \leq y \leq 5 \\ T(x, y, x', y') &:= x < y \\ &\quad \wedge x' = x \wedge y' = y \\ bad(x', y') &:= x = 4 \end{aligned}$$

- Explicit construction & search
- Precise representation needs space
- Implicit, lazy reasoning via SAT
- Exploits structure & symmetry

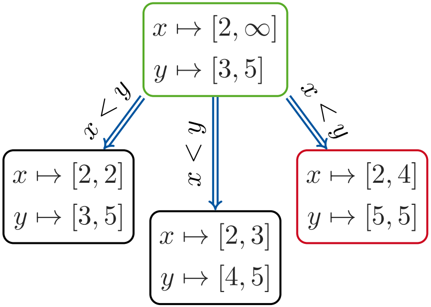
Explicit vs. Symbolic Methods



$$src(x, y) \wedge T(x, y, x', y') \wedge bad(x', y')$$

- Explicit construction & search
- Precise representation needs space
- Implicit, lazy reasoning via SAT
- Exploits structure & symmetry

Explicit vs. Symbolic Methods

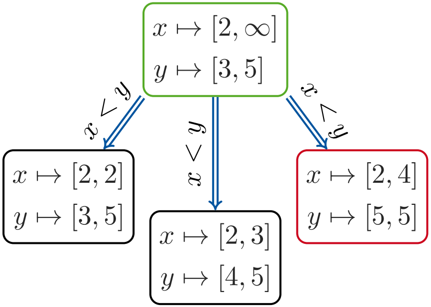


$$\begin{aligned} src(x,y) &:= 2 \leq x \\ &\quad \wedge 3 \leq y \leq 5 \\ T(x,y,x',y') &:= x < y \\ &\quad \wedge x' = x \wedge y' = y \\ bad(x',y') &:= x = 4 \end{aligned}$$

- Explicit construction & search
- Precise representation needs space

- Implicit, lazy reasoning via SAT
- Exploits structure & symmetry

Explicit vs. Symbolic Methods



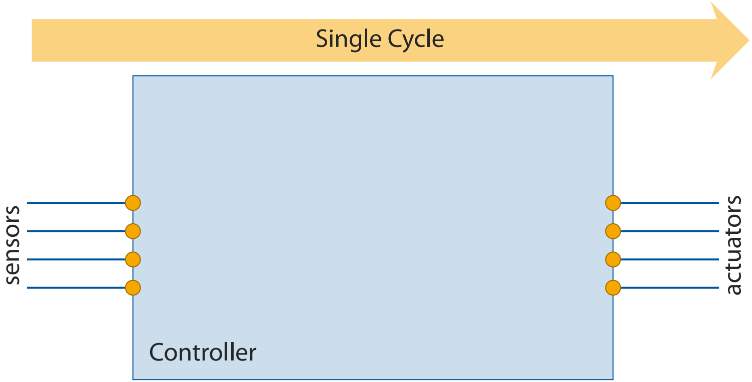
$$\begin{aligned} src(x, y) &:= 2 \leq x \\ &\quad \wedge 3 \leq y \leq 5 \\ T(x, y, x', y') &:= x < y \\ &\quad \wedge x' = x \wedge y' = y \\ bad(x', y') &:= x = 4 \end{aligned}$$

- Explicit construction & search
- Precise representation needs space

- Implicit, lazy reasoning via SAT
- Exploits structure & symmetry

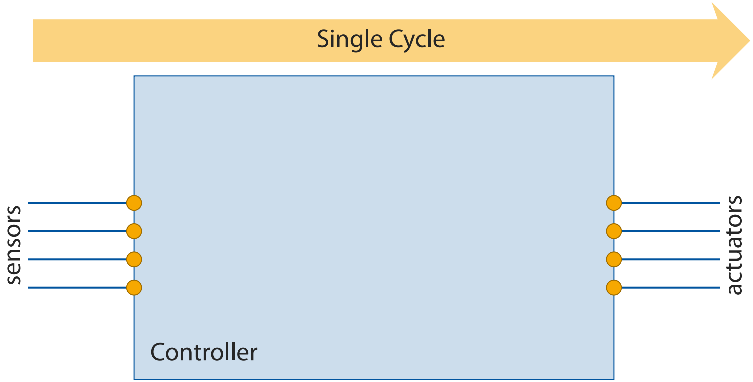
Programmable Logic Controllers (PLCs)

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



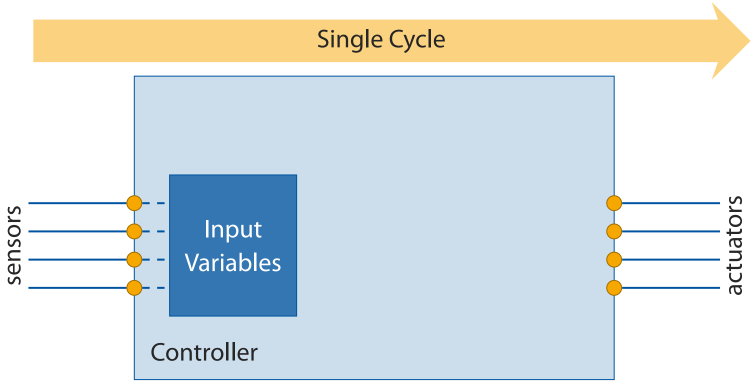
Programmable Logic Controllers (PLCs)

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



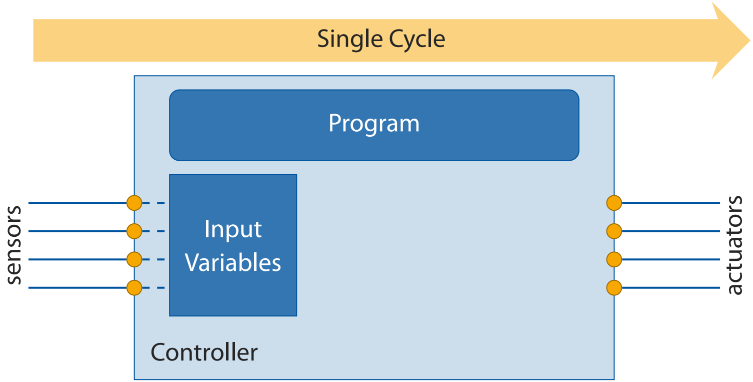
Programmable Logic Controllers (PLCs)

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



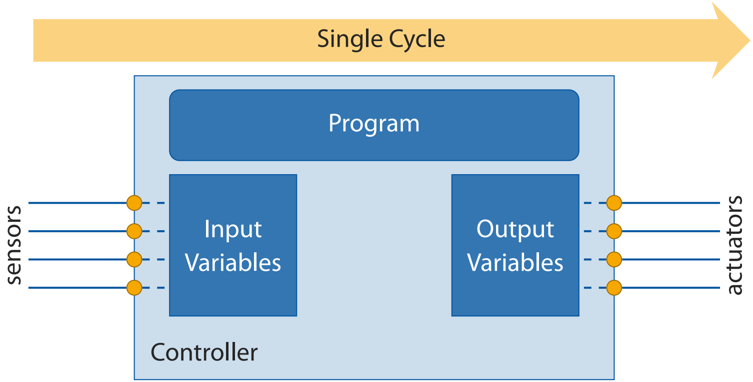
Programmable Logic Controllers (PLCs)

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



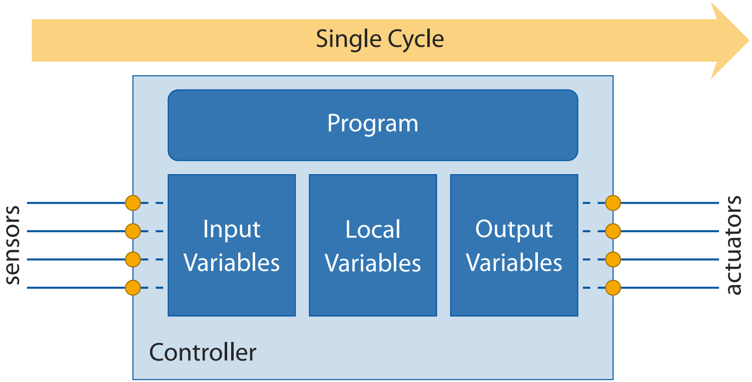
Programmable Logic Controllers (PLCs)

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



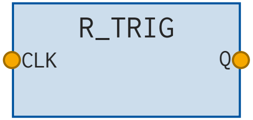
Programmable Logic Controllers (PLCs)

- 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

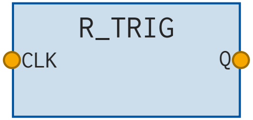
```
PROGRAM R_TRIG
  VAR_INPUT
    CLK:BOOL;
  END_VAR
  VAR
    M:BOOL;
  END_VAR
  VAR_OUTPUT
    Q:BOOL;
  END_VAR

  IF CLK AND NOT M THEN
    Q:=TRUE;
  ELSE
    Q:=FALSE;
  END_IF;
  M:=CLK;
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

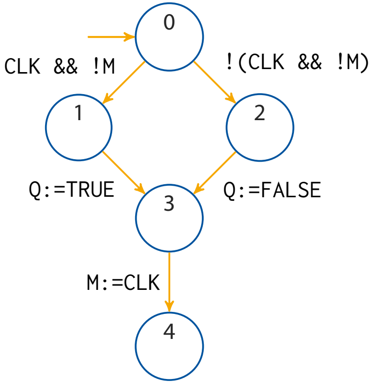
```
FUNCTION_BLOCK R_TRIG
  VAR_INPUT
    CLK:BOOL;
  END_VAR
  VAR
    M:BOOL;
  END_VAR
  VAR_OUTPUT
    Q:BOOL;
  END_VAR

  IF CLK AND NOT M THEN
    Q:=TRUE;
  ELSE
    Q:=FALSE;
  END_IF;
  M:=CLK;
END_FUNCTION_BLOCK
```



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

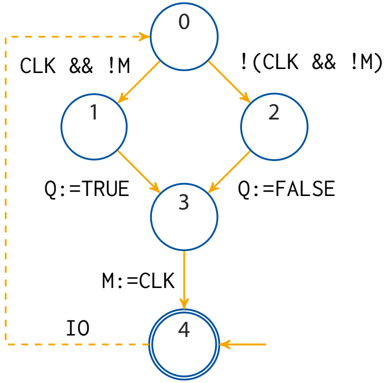
Control Flow Automaton (CFA)

```
FUNCTION_BLOCK R_TRIG
  VAR_INPUT
    CLK:BOOL;
  END_VAR
  VAR
    M:BOOL;
  END_VAR
  VAR_OUTPUT
    Q:BOOL;
  END_VAR

  IF CLK AND NOT M THEN
    Q:=TRUE;
  ELSE
    Q:=FALSE;
  END_IF;
  M:=CLK;
END_FUNCTION_BLOCK
```



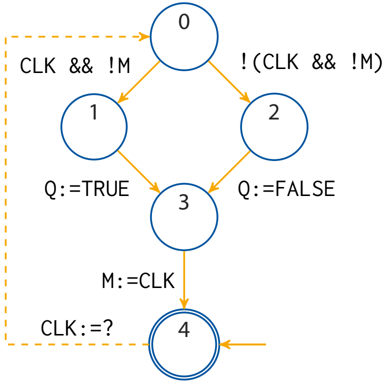
Control Flow Automaton (CFA)

```
PROGRAM R_TRIG
  VAR_INPUT
    CLK:BOOL;
  END_VAR
  VAR
    M:BOOL;
  END_VAR
  VAR_OUTPUT
    Q:BOOL;
  END_VAR

  IF CLK AND NOT M THEN
    Q:=TRUE;
  ELSE
    Q:=FALSE;
  END_IF;
  M:=CLK;
END_PROGRAM
```



Control Flow Automaton (CFA)

```
PROGRAM R_TRIG
  VAR_INPUT
    CLK:BOOL;
  END_VAR
  VAR
    M:BOOL;
  END_VAR
  VAR_OUTPUT
    Q:BOOL;
  END_VAR

  IF CLK AND NOT M THEN
    Q:=TRUE;
  ELSE
    Q:=FALSE;
  END_IF;
  M:=CLK;
END_PROGRAM
```


$$\begin{tikzcd}
 & 0 \arrow[dl, "CLK \&\& !M"'] \arrow[dr, "!(CLK \&\& !M)"] & \\
1 \arrow[dr, "Q:=TRUE"'] & & 2 \arrow[dl, "Q:=FALSE"] \\
 & 3 \arrow[d, "M:=CLK"'] & \\
 & 4 \arrow[uuu, dashed, bend left=90, "CLK:=?"] & \arrow[l]
\end{tikzcd}$$

Specifications

- Intermediate states are not observable

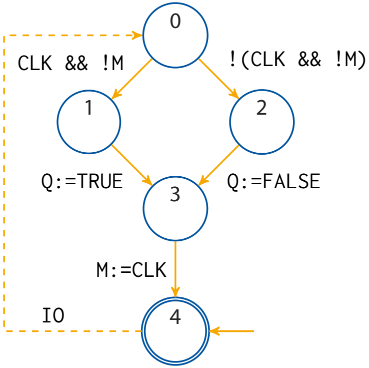
⇒ Automation engineers and specifications always refer to the observable state

- Common specifications can be adapted to such cyle-step semantics, e. g.

$\Box(\mathtt{M} = \mathtt{CLK}) \rightsquigarrow \Box(pc = 4 \rightarrow \mathtt{M} = \mathtt{CLK})$

and checked with off-the-shelf backends

- Unique specs need dedicated procedures



Specifications

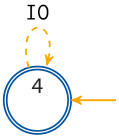
- Intermediate states are not observable

⇒ Automation engineers and specifications always refer to the observable state

- Common specifications can be adapted to such cyle-step semantics, e. g.

  $\Box(\mathtt{M} = \mathtt{CLK}) \rightsquigarrow \Box(\mathtt{pc} = 4 \rightarrow \mathtt{M} = \mathtt{CLK})$

  and checked with off-the-shelf backends
- Unique specs need dedicated procedures



Specifications

- Intermediate states are not observable

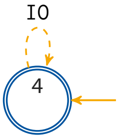
⇒ Automation engineers and specifications always refer to the observable state

- Common specifications can be adapted to such cyle-step semantics, e. g.

  $\Box(\mathtt{M} = \mathtt{CLK}) \rightsquigarrow \Box(pc = 4 \to \mathtt{M} = \mathtt{CLK})$

  and checked with off-the-shelf backends

- Unique specs need dedicated procedures



Specifications

- Intermediate states are not observable

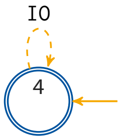
⇒ Automation engineers and specifications always refer to the observable state

- Common specifications can be adapted to such cyle-step semantics, e. g.

  $\Box(\mathtt{M} = \mathtt{CLK}) \rightsquigarrow \Box(\mathtt{pc} = 4 \rightarrow \mathtt{M} = \mathtt{CLK})$

  and checked with off-the-shelf backends

- Unique specs need dedicated procedures



Specifications

- ▶ Intermediate states are not observable

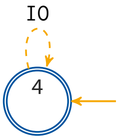
$\Rightarrow$ Automation engineers and specifications always refer to the observable state

- ▶ Common specifications can be adapted to such cyle-step semantics, e. g.

  $\Box(\mathtt{M} = \mathtt{CLK}) \rightsquigarrow \Box(\mathtt{pc} = 4 \rightarrow \mathtt{M} = \mathtt{CLK})$

  and checked with off-the-shelf backends

- ▶ Unique specs need dedicated procedures



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

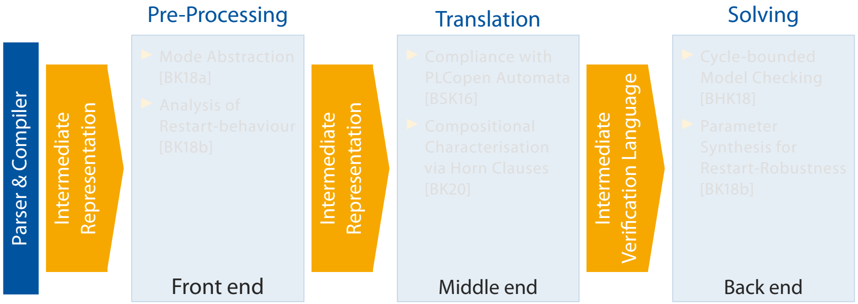
Standard modules

Performed most experiments on implementation of PLCopen Safety library:

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

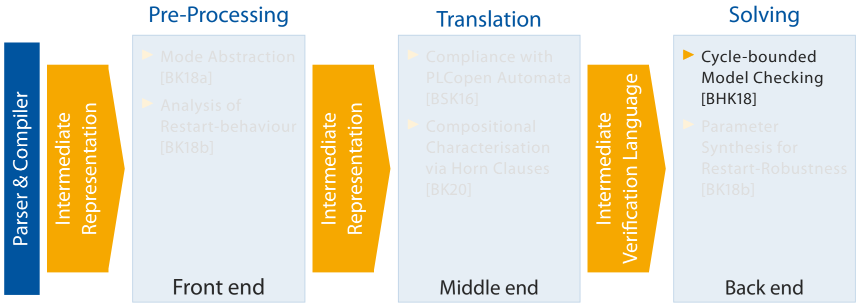
Contributions

- 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]


Parser & Compiler
Intermediate Representation
Pre-Processing
Mode Abstraction [BK18a]
Analysis of Restart-behaviour [BK18b]
Front end
Intermediate Representation
Translation
Compliance with PLCopen Automata [BSK16]
Compositional Characterisation via Horn Clauses [BK20]
Middle end
Intermediate Verification Language
Solving
Cycle-bounded Model Checking [BHK18]
Parameter Synthesis for Restart-Robustness [BK18b]
Back end

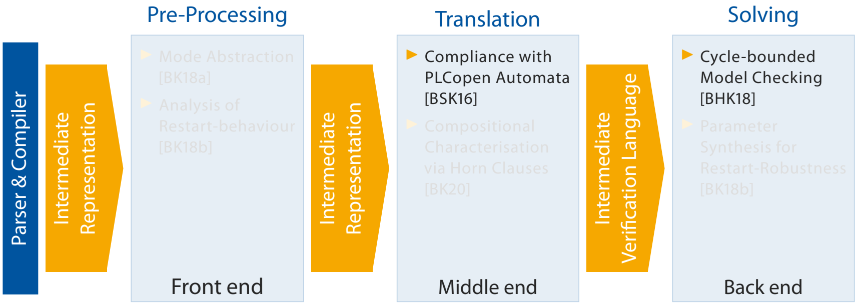
Contributions

- 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]


Pre-Processing
Translation
Solving
Parser & Compiler
Intermediate Representation
Mode Abstraction [BK18a]
Analysis of Restart-behaviour [BK18b]
Front end
Intermediate Representation
Compliance with PLCopen Automata [BSK16]
Compositional Characterisation via Horn Clauses [BK20]
Middle end
Intermediate Verification Language
Cycle-bounded Model Checking [BHK18]
Parameter Synthesis for Restart-Robustness [BK18b]
Back end

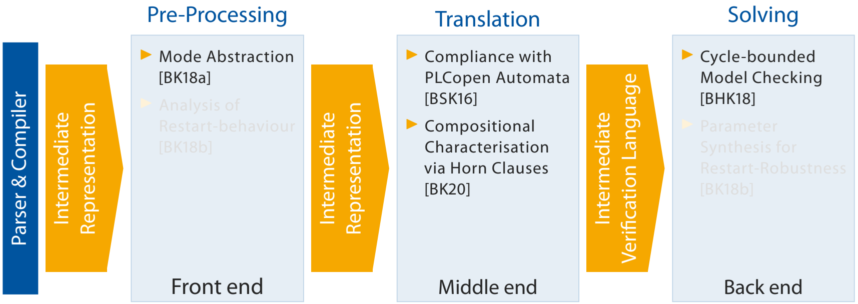
Contributions

- 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]


Parser & Compiler
Intermediate Representation
Pre-Processing
Mode Abstraction [BK18a]
Analysis of Restart-behaviour [BK18b]
Front end
Intermediate Representation
Translation
Compliance with PLCopen Automata [BSK16]
Compositional Characterisation via Horn Clauses [BK20]
Middle end
Intermediate Verification Language
Solving
Cycle-bounded Model Checking [BHK18]
Parameter Synthesis for Restart-Robustness [BK18b]
Back end

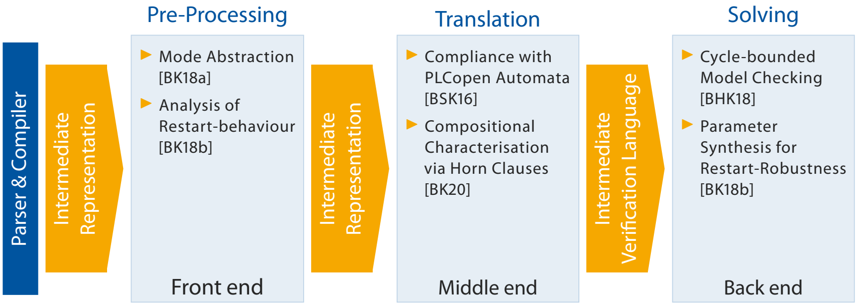
Contributions

- 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]


Parser & Compiler
Intermediate Representation
Pre-Processing
Mode Abstraction [BK18a]
Analysis of Restart-behaviour [BK18b]
Front end
Intermediate Representation
Translation
Compliance with PLCopen Automata [BSK16]
Compositional Characterisation via Horn Clauses [BK20]
Middle end
Intermediate Verification Language
Solving
Cycle-bounded Model Checking [BHK18]
Parameter Synthesis for Restart-Robustness [BK18b]
Back end

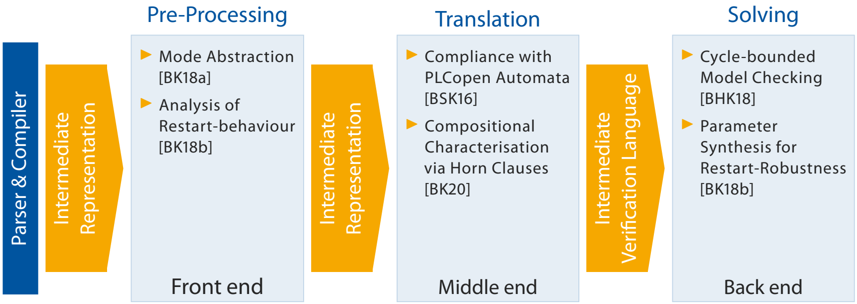
Contributions

- 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]


Parser & Compiler
Intermediate Representation
Pre-Processing
Mode Abstraction [BK18a]
Analysis of Restart-behaviour [BK18b]
Front end
Intermediate Representation
Translation
Compliance with PLCopen Automata [BSK16]
Compositional Characterisation via Horn Clauses [BK20]
Middle end
Intermediate Verification Language
Solving
Cycle-bounded Model Checking [BHK18]
Parameter Synthesis for Restart-Robustness [BK18b]
Back end

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]


Parser & Compiler
Intermediate Representation
Pre-Processing
Mode Abstraction [BK18a]
Analysis of Restart-behaviour [BK18b]
Front end
Intermediate Representation
Translation
Compliance with PLCopen Automata [BSK16]
Compositional Characterisation via Horn Clauses [BK20]
Middle end
Intermediate Verification Language
Solving
Cycle-bounded Model Checking [BHK18]
Parameter Synthesis for Restart-Robustness [BK18b]
Back end

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) for 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) for 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) for 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

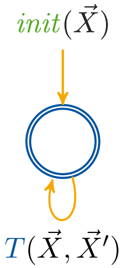
Constrained Horn Clauses (CHCs)

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

$$init(\vec{X}) \rightarrow Reach(\vec{X})$$
$$\wedge\ Reach(\vec{X}) \wedge T(\vec{X}, \vec{X}') \rightarrow Reach(\vec{X}')$$
$$\wedge\ Reach(\vec{X}) \rightarrow safe(\vec{X})$$

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

$$\forall \mathcal{V}\ \underbrace{P_1(\vec{X}_1) \wedge \cdots \wedge P_k(\vec{X}_k) \wedge \varphi}_{\text{body}} \rightarrow \underbrace{h(\vec{X})}_{\text{head}},\ k \geq 0,$$



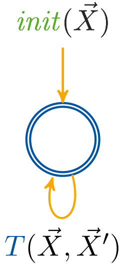
Constrained Horn Clauses (CHCs)

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

$$init(\vec{X}) \rightarrow Reach(\vec{X})$$
$$\wedge\ Reach(\vec{X}) \wedge T(\vec{X}, \vec{X}') \rightarrow Reach(\vec{X}')$$
$$\wedge\ Reach(\vec{X}) \rightarrow safe(\vec{X})$$

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

$$\forall \mathcal{V}\ \underbrace{P_1(\vec{X}_1) \wedge \cdots \wedge P_k(\vec{X}_k) \wedge \varphi}_{\text{body}} \rightarrow \underbrace{h(\vec{X})}_{\text{head}},\ k \geq 0,$$



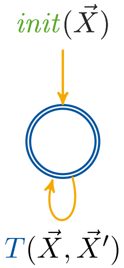
Constrained Horn Clauses (CHCs)

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

$$init(\vec{X}) \rightarrow Reach(\vec{X})$$
$$\wedge\ Reach(\vec{X}) \wedge T(\vec{X},\vec{X}') \rightarrow Reach(\vec{X}')$$
$$\wedge\ Reach(\vec{X}) \rightarrow safe(\vec{X})$$

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

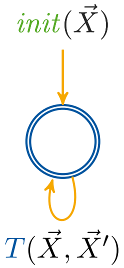
$$\forall\mathcal{V}\ \underbrace{P_1(\vec{X}_1) \wedge \cdots \wedge P_k(\vec{X}_k) \wedge \varphi}_{\text{body}} \rightarrow \underbrace{h(\vec{X})}_{\text{head}},\ k \geq 0,$$



Constrained Horn Clauses (CHCs)

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

$$
\begin{aligned}
& init(\vec{X}) \rightarrow Reach(\vec{X}) \\
\wedge\ & Reach(\vec{X}) \wedge T(\vec{X}, \vec{X}') \rightarrow Reach(\vec{X}') \\
\wedge\ & Reach(\vec{X}) \rightarrow safe(\vec{X})
\end{aligned}
$$



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

$$\forall \mathcal{V}\ \underbrace{P_1(\vec{X}_1) \wedge \cdots \wedge P_k(\vec{X}_k) \wedge \varphi}_{\text{body}} \rightarrow \underbrace{h(\vec{X})}_{\text{head}},\ k \geq 0,$$

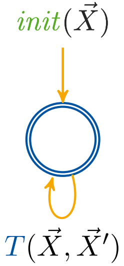
Constrained Horn Clauses (CHCs)

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

$$\begin{aligned} & init(\vec{X}) \rightarrow Reach(\vec{X}) \\ \wedge\ & Reach(\vec{X}) \wedge T(\vec{X}, \vec{X}') \rightarrow Reach(\vec{X}') \\ \wedge\ & Reach(\vec{X}) \rightarrow safe(\vec{X}) \end{aligned}$$

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

$$\forall \mathcal{V}\ \underbrace{P_1(\vec{X}_1) \wedge \cdots \wedge P_k(\vec{X}_k) \wedge \varphi}_{\text{body}} \rightarrow \underbrace{h(\vec{X})}_{\text{head}},\ k \geq 0,$$



CHCs as Intermediate Verification Language

- In 2010, Bradley proposed a novel hardware model checking algorithm
  - IC3/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

- CHCs are a logical match for Hoare logic and correspond to proof rules
- GPDR and SPACER generalised PDR to CHCs

⇒ 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
- SPACER still best-in-class and in the open SMT solver Z3 (CHC-COMP'21)

CHCs as Intermediate Verification Language

- In 2010, Bradley proposed a novel hardware model checking algorithm
  - IC3/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

- CHCs are a logical match for Hoare logic and correspond to proof rules
- GPDR and SPACER generalised PDR to CHCs

⇒ 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
- SPACER still best-in-class and in the open SMT solver Z3 (CHC-COMP'21)

CHCs as Intermediate Verification Language

- In 2010, Bradley proposed a novel hardware model checking algorithm
  - IC3/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

- CHCs are a logical match for Hoare logic and correspond to proof rules
- GPDR and Spacer generalised PDR to CHCs

⇒ 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
- Spacer still best-in-class and in the open SMT solver Z3 (CHC-COMP'21)

CHCs as Intermediate Verification Language

- In 2010, Bradley proposed a novel hardware model checking algorithm
  - IC3/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
- CHCs are a logical match for Hoare logic and correspond to proof rules
- GPDR and SPACER generalised PDR to CHCs
- ⇒ 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
- SPACER still best-in-class and in the open SMT solver Z3 (CHC-COMP'21)

CHCs as Intermediate Verification Language

- In 2010, Bradley proposed a novel hardware model checking algorithm
  - IC3/PDR constructs inductive invariants incrementally
  - Was competitive with highly tuned solvers – 3rd place at HWMCC'10

$\Rightarrow$ Incentive for lifting it to software verification – no approach prevailed

- CHCs are a logical match for Hoare logic and correspond to proof rules
- GPDR and SPACER generalised PDR to CHCs

$\Rightarrow$ 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
- SPACER still best-in-class and in the open SMT solver Z3 (CHC-COMP'21)

CHCs as Intermediate Verification Language

- In 2010, Bradley proposed a novel hardware model checking algorithm
  - IC3/PDR constructs inductive invariants incrementally
  - Was competitive with highly tuned solvers – 3rd place at HWMCC'10

$\Rightarrow$ Incentive for lifting it to software verification – no approach prevailed

- CHCs are a logical match for Hoare logic and correspond to proof rules
- GPDR and SPACER generalised PDR to CHCs

$\Rightarrow$ 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
- SPACER still best-in-class and in the open SMT solver Z3 (CHC-COMP'21)

Characterisation

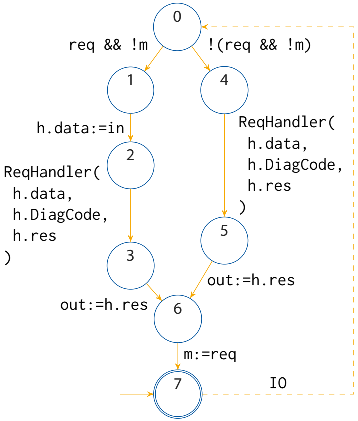
$$init(\vec{X}) \rightarrow P_7(\vec{X})$$

$$P_7(\vec{X}) \wedge [\![\mathtt{IO}]\!] \rightarrow P_0(\vec{X}')$$

$$P_0(\vec{X}) \wedge req \wedge \neg m \wedge \vec{X}' = \vec{X} \rightarrow P_1(\vec{X}')$$

$$\vdots$$

$$P_7(\vec{X}) \rightarrow safe(\vec{X})$$


0
req && !m
!(req && !m)
1
4
h.data:=in
ReqHandler(h.data, h.DiagCode, h.res)
2
ReqHandler(h.data, h.DiagCode, h.res)
5
3
out:=h.res
out:=h.res
6
m:=req
7
IO

Characterisation

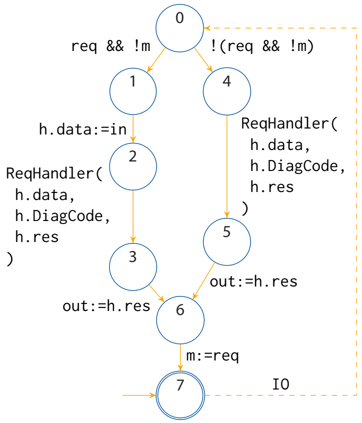
$$init(\vec{X}) \rightarrow P_7(\vec{X})$$

$$P_7(\vec{X}) \wedge [\![\mathtt{IO}]\!] \rightarrow P_0(\vec{X}')$$

$$P_0(\vec{X}) \wedge req \wedge \neg m \wedge \vec{X}' = \vec{X} \rightarrow P_1(\vec{X}')$$

$$\vdots$$

$$P_7(\vec{X}) \rightarrow safe(\vec{X})$$


0
req && !m
!(req && !m)
1
4
h.data:=in
2
ReqHandler(h.data, h.DiagCode, h.res)
ReqHandler(h.data, h.DiagCode, h.res)
3
5
out:=h.res
out:=h.res
6
m:=req
7
IO

Characterisation

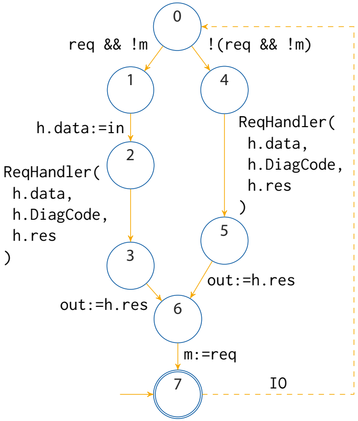
$$init(\vec{X}) \rightarrow P_7(\vec{X})$$

$$P_7(\vec{X}) \wedge [\![\texttt{IO}]\!] \rightarrow P_0(\vec{X}')$$

$$P_0(\vec{X}) \wedge req \wedge \neg m \wedge \vec{X}' = \vec{X} \rightarrow P_1(\vec{X}')$$

$$\vdots$$

$$P_7(\vec{X}) \rightarrow safe(\vec{X})$$


0
req && !m
!(req && !m)
1
4
h.data:=in
ReqHandler(h.data, h.DiagCode, h.res)
2
ReqHandler(h.data, h.DiagCode, h.res)
5
3
out:=h.res
out:=h.res
6
m:=req
7
IO

Characterisation

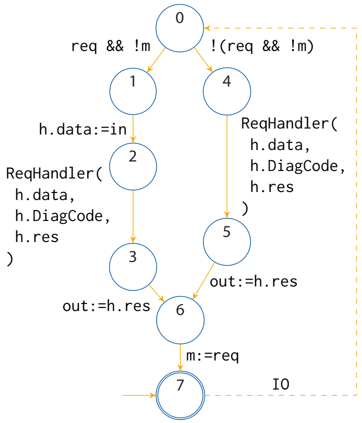
$$init(\vec{X}) \rightarrow P_7(\vec{X})$$

$$P_7(\vec{X}) \wedge [\![\mathtt{IO}]\!] \rightarrow P_0(\vec{X}')$$

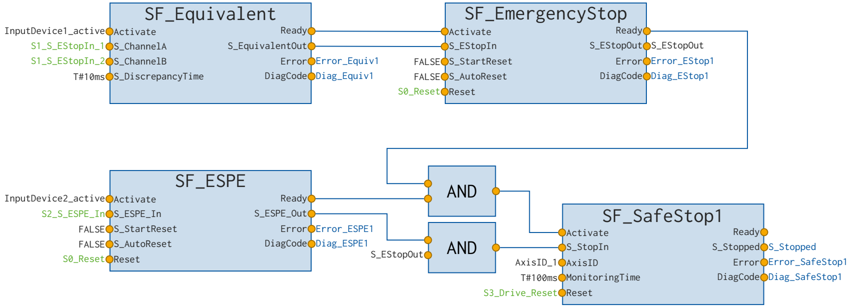
$$P_0(\vec{X}) \wedge req \wedge \neg m \wedge \vec{X}' = \vec{X} \rightarrow P_1(\vec{X}')$$

$$\vdots$$

$$P_7(\vec{X}) \rightarrow safe(\vec{X})$$


0
req && !m
!(req && !m)
1
4
h.data:=in
2
ReqHandler(h.data, h.DiagCode, h.res)
ReqHandler(h.data, h.DiagCode, h.res)
3
5
out:=h.res
out:=h.res
6
m:=req
7
IO

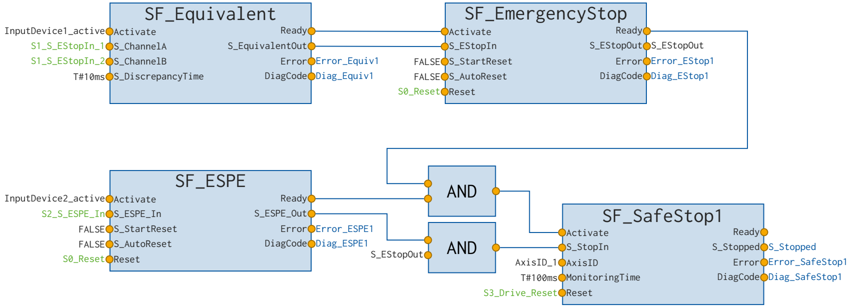
PLCopen Safety Application



- 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

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

Compositional Characterisation

- Let P_i characterise a state $\vec{X}'$ at i, reachable from an entry state $\vec{X}$ of P:

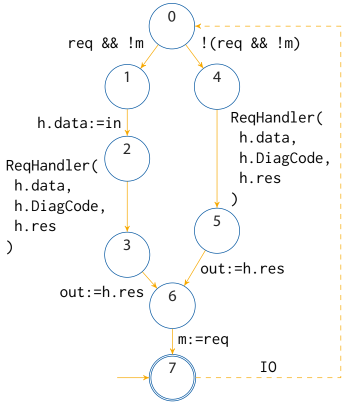
$$Main_0(\vec{X},\vec{X}') \land req' \land \neg m' \land \vec{X}'' = \vec{X}'$$
$$\rightarrow Main_1(\vec{X},\vec{X}'')$$

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

$$Main_7(\vec{X},\vec{X}')$$

- Enables characterisation of calls, e. g.

$$Main_2(\vec{X},\vec{X}') \land S(\vec{X}'_{ReqHandler},\vec{X}''_{ReqHandler})$$
$$\land\ ReqHandler_{exit}(\vec{X}'_{ReqHandler},\vec{X}''_{ReqHandler})$$
$$\rightarrow Main_3(\vec{X},\vec{X}'')$$


0
req && !m
!(req && !m)
1
4
h.data:=in
2
ReqHandler(h.data, h.DiagCode, h.res)
ReqHandler(h.data, h.DiagCode, h.res)
5
3
out:=h.res
out:=h.res
6
m:=req
7
IO

Compositional Characterisation

- Let P_i characterise a state $\vec{X}'$ at i, reachable from an entry state $\vec{X}$ of P:

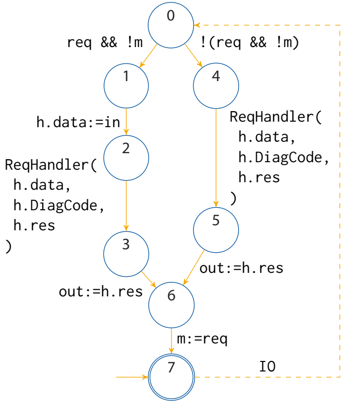
$$Main_0(\vec{X}, \vec{X}') \wedge req' \wedge \neg m' \wedge \vec{X}'' = \vec{X}'$$
$$\rightarrow Main_1(\vec{X}, \vec{X}'')$$

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

$$Main_7(\vec{X}, \vec{X}')$$

- Enables characterisation of calls, e. g.

$$Main_2(\vec{X}, \vec{X}') \wedge S(\vec{X}'_{ReqHandler}, \vec{X}''_{ReqHandler})$$
$$\wedge\ ReqHandler_{exit}(\vec{X}'_{ReqHandler}, \vec{X}''_{ReqHandler})$$
$$\rightarrow Main_3(\vec{X}, \vec{X}'')$$


0
req && !m
!(req && !m)
1
4
h.data:=in
ReqHandler(h.data, h.DiagCode, h.res)
2
ReqHandler(h.data, h.DiagCode, h.res)
5
3
out:=h.res
out:=h.res
6
m:=req
7
IO

Compositional Characterisation

- Let P_i characterise a state $\vec{X}'$ at i, reachable from an entry state $\vec{X}$ of P:

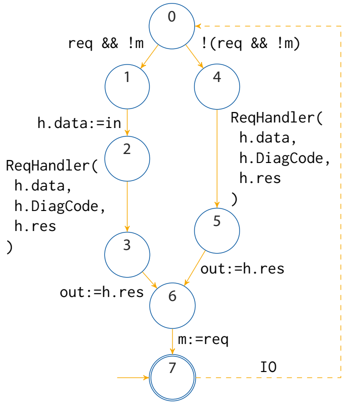
$$Main_0(\vec{X}, \vec{X}') \wedge req' \wedge \neg m' \wedge \vec{X}'' = \vec{X}'$$
$$\rightarrow Main_1(\vec{X}, \vec{X}'')$$

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

$$Main_7(\vec{X}, \vec{X}')$$

- Enables characterisation of calls, e. g.

$$Main_2(\vec{X}, \vec{X}') \wedge S(\vec{X}'_{ReqHandler}, \vec{X}''_{ReqHandler})$$
$$\wedge\ ReqHandler_{exit}(\vec{X}'_{ReqHandler}, \vec{X}''_{ReqHandler})$$
$$\rightarrow Main_3(\vec{X}, \vec{X}'')$$


0
1
2
3
4
5
6
7
req && !m
!(req && !m)
h.data:=in
ReqHandler(h.data, h.DiagCode, h.res)
ReqHandler(h.data, h.DiagCode, h.res)
out:=h.res
out:=h.res
m:=req
IO

Compositional Characterisation

- Let P_i characterise a state $\vec{X}'$ at i, reachable from an entry state $\vec{X}$ of P:

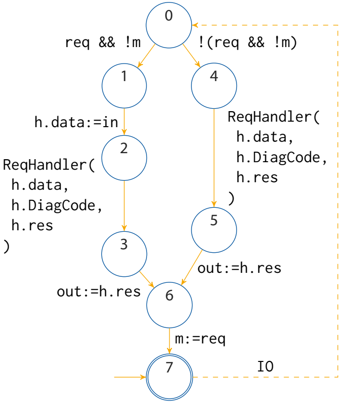
$$Main_0(\vec{X}, \vec{X}') \wedge req' \wedge \neg m' \wedge \vec{X}'' = \vec{X}'$$
$$\rightarrow Main_1(\vec{X}, \vec{X}'')$$

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

$$Main_7(\vec{X}, \vec{X}')$$

- Enables characterisation of calls, e. g.

$$Main_2(\vec{X}, \vec{X}') \wedge S(\vec{X}'_{ReqHandler}, \vec{X}''_{ReqHandler})$$
$$\wedge\, ReqHandler_{exit}(\vec{X}'_{ReqHandler}, \vec{X}''_{ReqHandler})$$
$$\rightarrow Main_3(\vec{X}, \vec{X}'')$$


0
req && !m
!(req && !m)
1
4
h.data:=in
2
ReqHandler(h.data, h.DiagCode, h.res)
ReqHandler(h.data, h.DiagCode, h.res)
3
5
out:=h.res
out:=h.res
6
m:=req
7
IO

Compositional Characterisation

- Let P_i characterise a state $\vec{X}'$ at i, reachable from an entry state $\vec{X}$ of P:

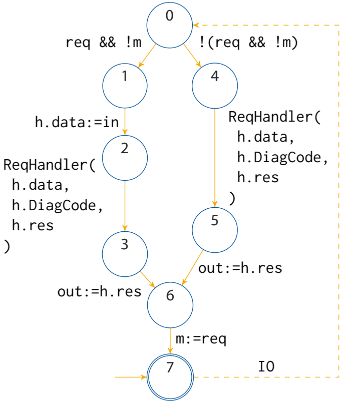
$$Main_0(\vec{X}, \vec{X}') \wedge req' \wedge \neg m' \wedge \vec{X}'' = \vec{X}'$$
$$\rightarrow Main_1(\vec{X}, \vec{X}'')$$

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

$$Main_7(\vec{X}, \vec{X}')$$

- Enables characterisation of calls, e. g.

$$Main_2(\vec{X}, \vec{X}') \wedge S(\vec{X}'_{ReqHandler}, \vec{X}''_{ReqHandler})$$
$$\wedge\ ReqHandler_{exit}(\vec{X}'_{ReqHandler}, \vec{X}''_{ReqHandler})$$
$$\rightarrow Main_3(\vec{X}, \vec{X}'')$$


0
req && !m
!(req && !m)
1
4
h.data:=in
ReqHandler(h.data, h.DiagCode, h.res)
2
ReqHandler(h.data, h.DiagCode, h.res)
5
3
out:=h.res
out:=h.res
6
m:=req
7
IO

Compositional Characterisation

- Let P_i characterise a state $\vec{X}'$ at i, reachable from an entry state $\vec{X}$ of P:

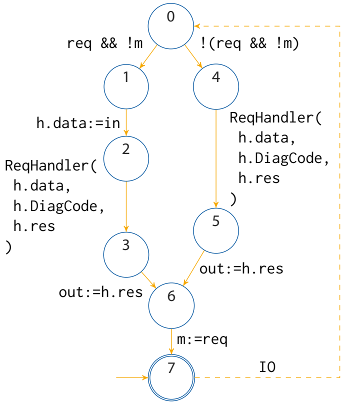
$$Main_0(\vec{X}, \vec{X}') \wedge req' \wedge \neg m' \wedge \vec{X}'' = \vec{X}'$$
$$\rightarrow Main_1(\vec{X}, \vec{X}'')$$

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

$$Main_7(\vec{X}, \vec{X}')$$

- Enables characterisation of calls, e. g.

$$Main_2(\vec{X}, \vec{X}') \wedge S(\vec{X}'_{ReqHandler}, \vec{X}''_{ReqHandler})$$
$$\wedge\ ReqHandler_{exit}(\vec{X}'_{ReqHandler}, \vec{X}''_{ReqHandler})$$
$$\rightarrow Main_3(\vec{X}, \vec{X}'')$$


0
req && !m
!(req && !m)
1
4
h.data:=in
2
ReqHandler(h.data, h.DiagCode, h.res)
ReqHandler(h.data, h.DiagCode, h.res)
3
5
out:=h.res
out:=h.res
6
m:=req
7
IO

Informatik 11 Embedded Software
RWTH AACHEN UNIVERSITY

Experiments

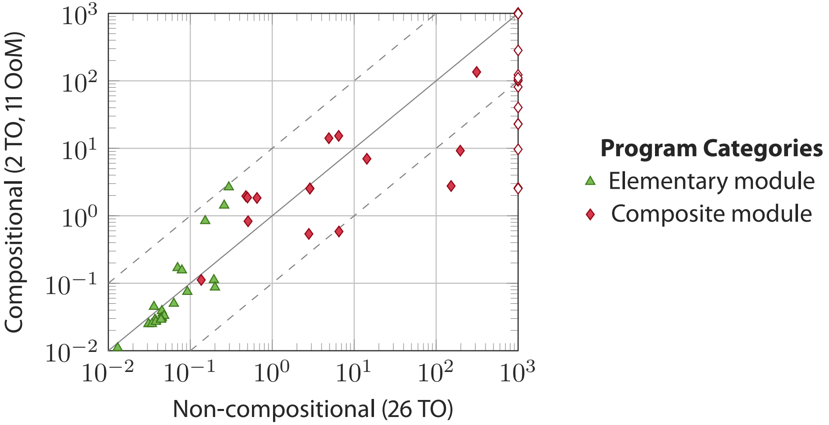
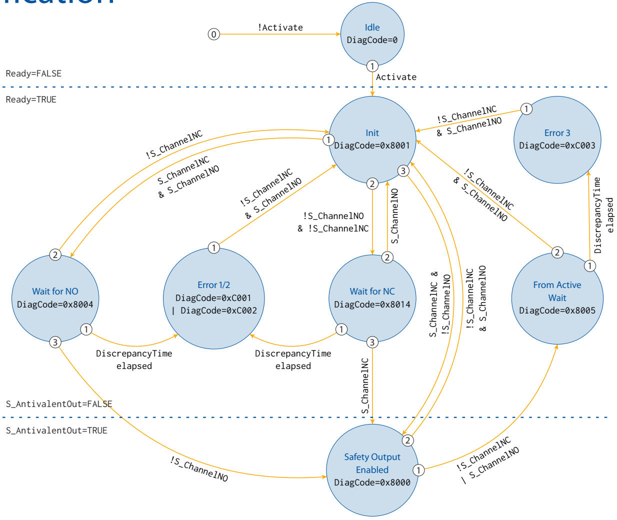


Figure: Time [s] spent on each verification task (n=64)

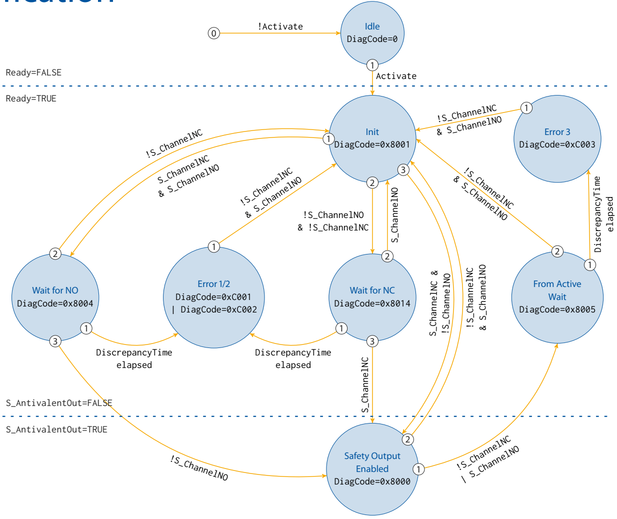
PLCopen Block Specification

- 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



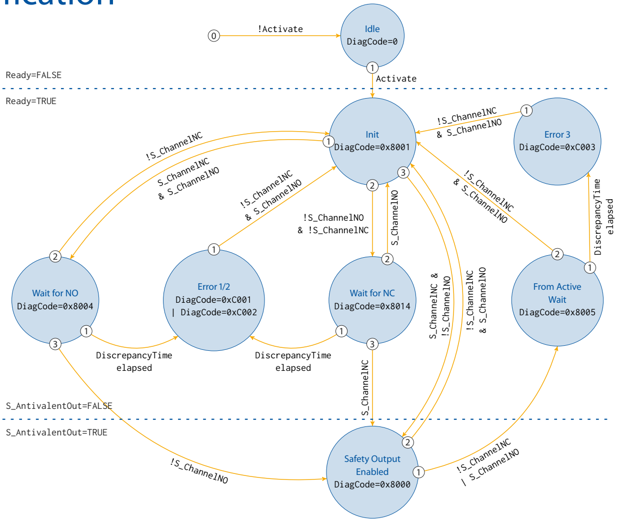
PLCopen Block Specification

- 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

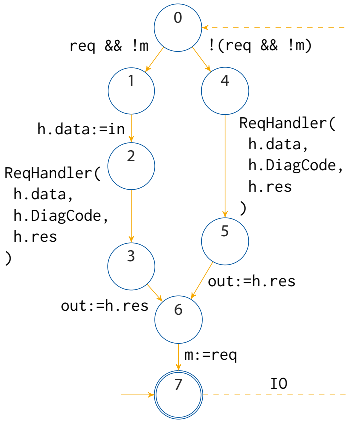


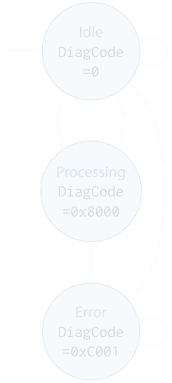
PLCopen Block Specification

- 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 Spaces in Model Checking

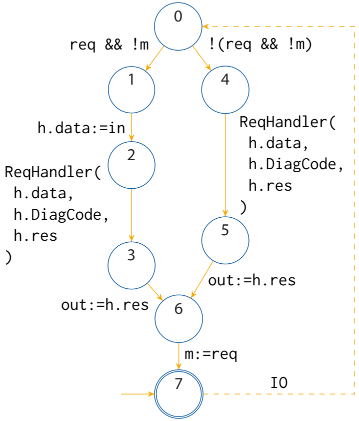

0
req && !m
!(req && !m)
1
4
h.data:=in
2
ReqHandler(h.data, h.DiagCode, h.res)
3
ReqHandler(h.data, h.DiagCode, h.res)
5
out:=h.res
out:=h.res
6
m:=req
7
IO

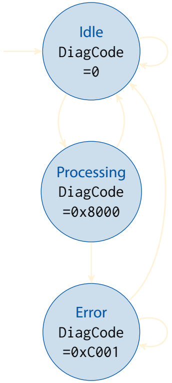


- Are requests processed in $\leq$ two execution cycles?

$\Rightarrow$ On a request, is the "Processing" mode reached in a single cycle?

Mode Spaces in Model Checking

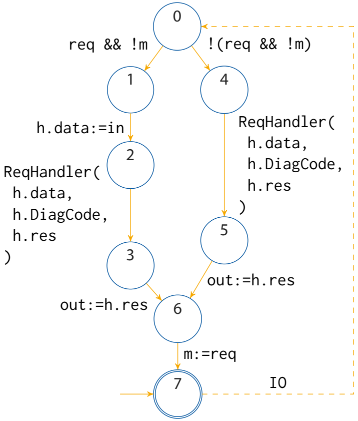


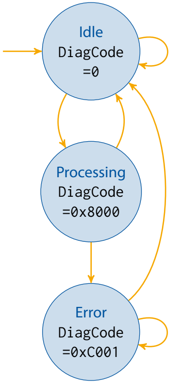


- Are requests processed in $\leq$ two execution cycles?

$\Rightarrow$ On a request, is the “Processing” mode reached in a single cycle?

Mode Spaces in Model Checking

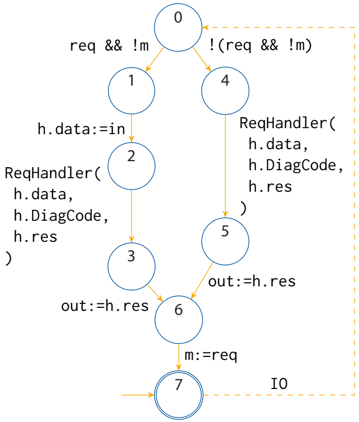

0
req && !m
!(req && !m)
1
4
h.data:=in
2
ReqHandler(h.data, h.DiagCode, h.res)
ReqHandler(h.data, h.DiagCode, h.res)
3
5
out:=h.res
out:=h.res
6
m:=req
7
IO

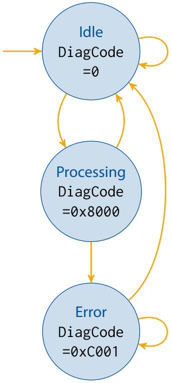


- Are requests processed in $\leq$ two execution cycles?

$\Rightarrow$ On a request, is the “Processing” mode reached in a single cycle?

Mode Spaces in Model Checking





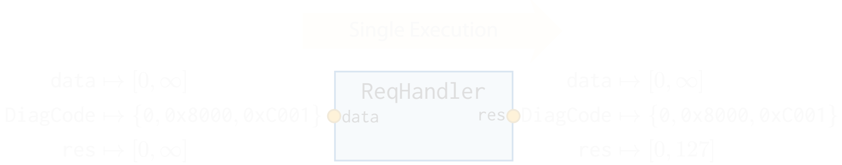
- Are requests processed in $\leq$ two execution cycles?

$\Rightarrow$ On a request, is the “Processing” mode reached in a single cycle?

Mode Abstraction

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

$\Rightarrow$ Adapt value-set analysis (VSA)

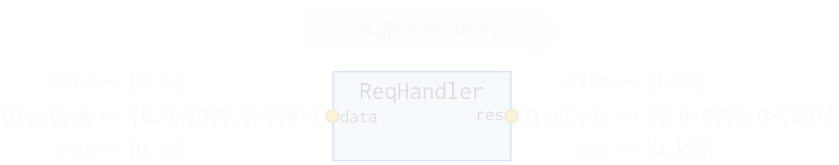

Single Execution
data $\mapsto [0, \infty]$
DiagCode $\mapsto \{0, 0x8000, 0xC001\}$
res $\mapsto [0, \infty]$
ReqHandler
data
res
data $\mapsto [0, \infty]$
DiagCode $\mapsto \{0, 0x8000, 0xC001\}$
res $\mapsto [0, 127]$

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

$\Rightarrow$ Adapt value-set analysis (VSA)

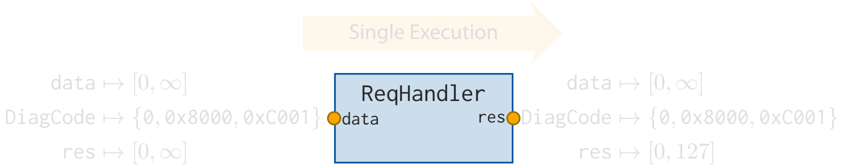


1. Perform VSA on main CFA to approximate all variables' values
2. For each block type and mode, e. g. `ReqHandler` and `0x8000`
   - 2.1 Keep VSA's values but fix source mode
   - 2.2 Perform VSA on block and interpret resulting mode-values as targets

Mode Abstraction

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

$\Rightarrow$ Adapt value-set analysis (VSA)

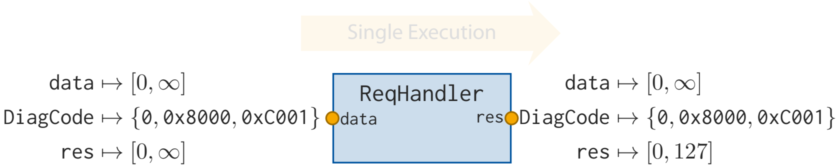


1. Perform VSA on main CFA to approximate all variables' values
2. For each block type and mode, e. g. `ReqHandler` and `0x8000`
   2.1 Keep VSA's values but fix source mode
   2.2 Perform VSA on block and interpret resulting mode-values as targets

Mode Abstraction

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

$\Rightarrow$ Adapt value-set analysis (VSA)

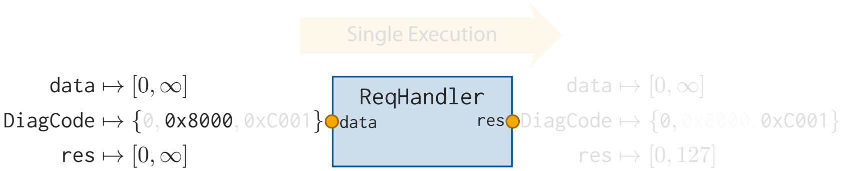

Single Execution
data $\mapsto [0, \infty]$
DiagCode $\mapsto \{$0, 0x8000, 0xC001$\}$
res $\mapsto [0, \infty]$
ReqHandler
data
res
data $\mapsto [0, \infty]$
DiagCode $\mapsto \{$0, 0x8000, 0xC001$\}$
res $\mapsto [0, 127]$

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

$\Rightarrow$ Adapt value-set analysis (VSA)

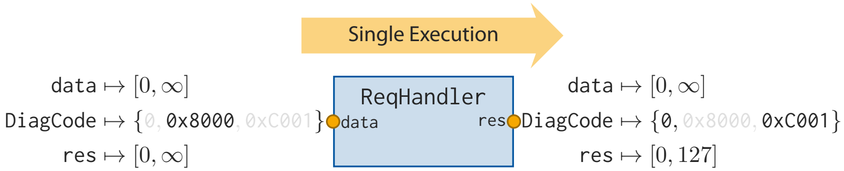


1. Perform VSA on main CFA to approximate all variables' values
2. For each block type and mode, e. g. `ReqHandler` and `0x8000`
   - 2.1 Keep VSA's values but fix source mode
   - 2.2 Perform VSA on block and interpret resulting mode-values as targets

Mode Abstraction

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

⇒ Adapt value-set analysis (VSA)



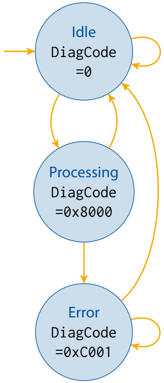
1. Perform VSA on main CFA to approximate all variables' values
2. For each block type and mode, e. g. `ReqHandler` and `0x8000`
   2.1 Keep VSA's values but fix source mode
   2.2 Perform VSA on block and interpret resulting mode-values as targets

Mode Space as Call Summary

- Mode space constrains possible transitions

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

$$
\begin{aligned}
&(h.DiagCode = \texttt{0} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0x8000}) \\
\wedge\ &(h.DiagCode = \texttt{0x8000} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0xC001}) \\
\wedge\ &(h.DiagCode = \texttt{0xC001} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0xC001})
\end{aligned}
$$



- Add to encoding of each call of `ReqHandler`

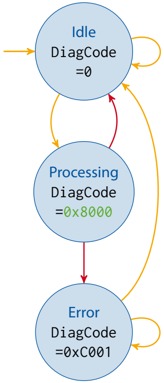
Mode Space as Call Summary

- Mode space constrains possible transitions

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

$$\begin{aligned}
&(h.DiagCode = \texttt{0} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad \vee\, h.DiagCode' = \texttt{0x8000}) \\
\wedge\, &(h.DiagCode = \texttt{0x8000} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad \vee\, h.DiagCode' = \texttt{0xC001}) \\
\wedge\, &(h.DiagCode = \texttt{0xC001} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad \vee\, h.DiagCode' = \texttt{0xC001})
\end{aligned}$$

- Add to encoding of each call of `ReqHandler`



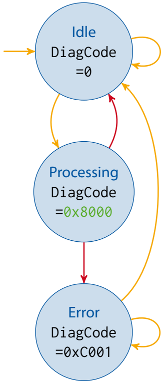
Mode Space as Call Summary

- Mode space constrains possible transitions

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

$$\begin{aligned}
&(h.DiagCode = \texttt{0} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad \lor h.DiagCode' = \texttt{0x8000}) \\
\land\ &(h.DiagCode = \texttt{0x8000} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad \lor h.DiagCode' = \texttt{0xC001}) \\
\land\ &(h.DiagCode = \texttt{0xC001} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad \lor h.DiagCode' = \texttt{0xC001})
\end{aligned}$$

- Add to encoding of each call of `ReqHandler`



Experiments

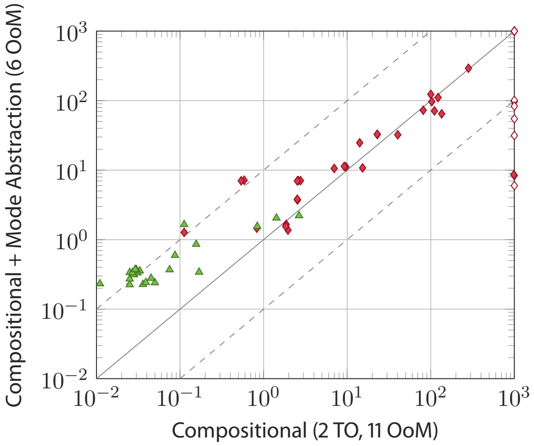


Figure: Time [s] spent on mode abstraction and solving CHCs ($n = 64$)

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:

- Non-volatile state variables allow for "restart-robust" designs
- Restarts may be triggered by a watchdog timer, power surge, …
- Writing to battery-backed memory immediate or delayed to cycle end

⇒ Choice of retain-variables and reasoning left to developer

Restart-Behaviour

However:

- A proof holds w.r.t. the formal model – not the real system

$\Rightarrow$ Model is usually missing behaviour enabled by hardware

Battery-backed memory & restart-functionality:

- Non-volatile state variables allow for “restart-robust” designs
- Restarts may be triggered by a watchdog timer, power surge, …
- Writing to battery-backed memory immediate or delayed to cycle end

$\Rightarrow$ Choice of retain-variables and reasoning left to developer

Restart-Behaviour

However:

- A proof holds w.r.t. the formal model – not the real system

$\Rightarrow$ Model is usually missing behaviour enabled by hardware

Battery-backed memory & restart-functionality:

- Non-volatile state variables allow for "restart-robust" designs
- Restarts may be triggered by a watchdog timer, power surge, …
- Writing to battery-backed memory immediate or delayed to cycle end

$\Rightarrow$ Choice of retain-variables and reasoning left to developer

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:

- ▶ Non-volatile state variables allow for "restart-robust" designs
- ▶ Restarts may be triggered by a watchdog timer, power surge, …
- ▶ Writing to battery-backed memory immediate or delayed to cycle end
- ⇒ Choice of retain-variables and reasoning left to developer

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:

- Non-volatile state variables allow for “restart-robust” designs
- Restarts may be triggered by a watchdog timer, power surge, …
- Writing to battery-backed memory immediate or delayed to cycle end

⇒ Choice of retain-variables and reasoning left to developer

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:

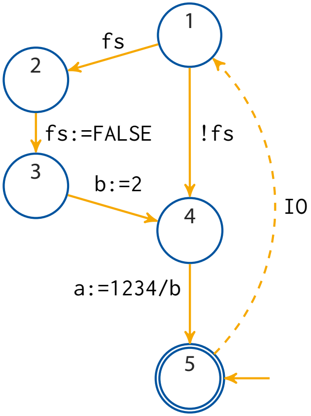
- ▶ Non-volatile state variables allow for "restart-robust" designs
- ▶ Restarts may be triggered by a watchdog timer, power surge, …
- ▶ Writing to battery-backed memory immediate or delayed to cycle end
- ⇒ Choice of retain-variables and reasoning left to developer

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?

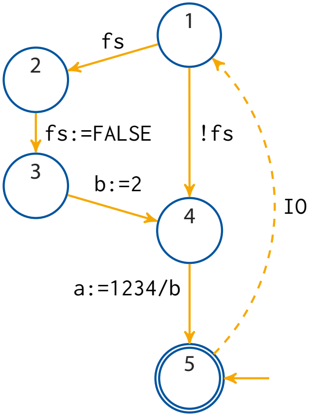


Toy Example: Invariant $a \geq 0$

- 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?

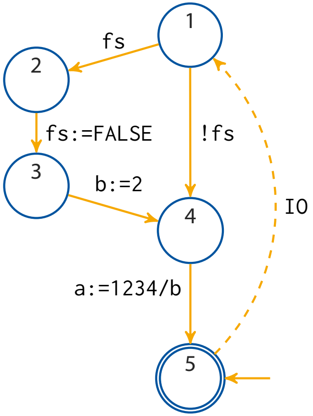


Toy Example: Invariant $a \geq 0$

- 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?

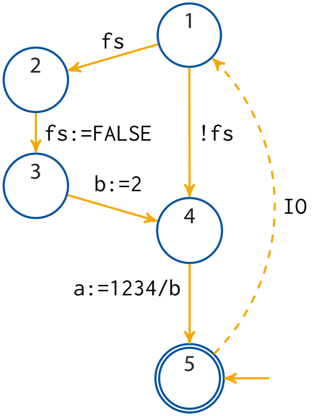


Toy Example: Invariant $a \geq 0$

- 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?

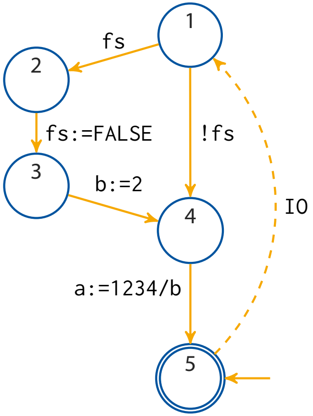


Toy Example: Invariant $a \geq 0$

- 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?

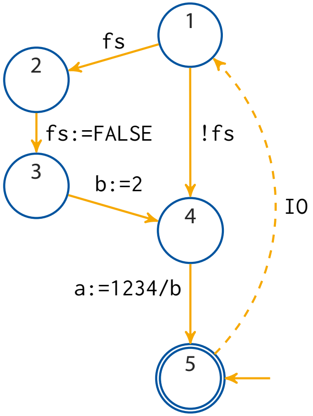


Toy Example: Invariant $a \geq 0$

- 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?

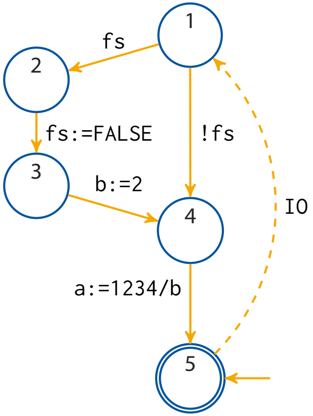


Toy Example: Invariant $a \geq 0$

- 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?

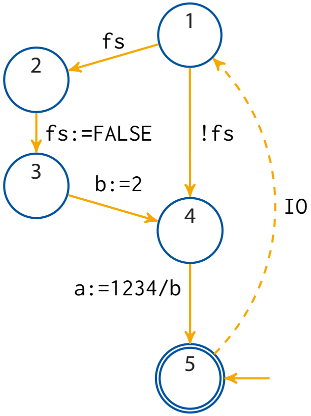


Toy Example: Invariant $a \geq 0$

- 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?

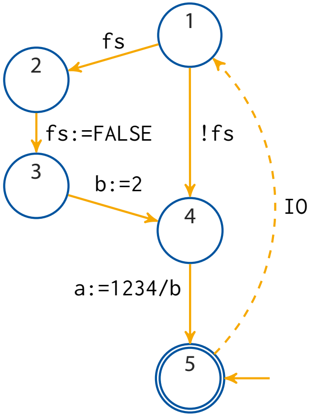


Toy Example: Invariant $a \geq 0$

- 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?

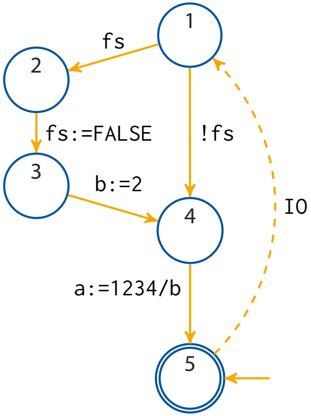


Toy Example: Invariant $a \geq 0$

- 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?

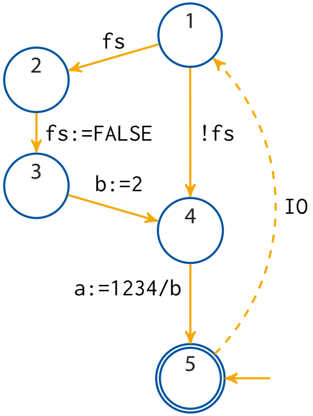


Toy Example: Invariant $a \geq 0$

- 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?

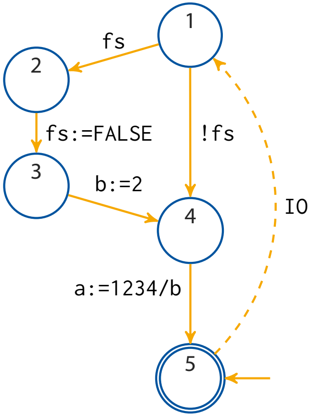


Toy Example: Invariant $a \geq 0$

- 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?

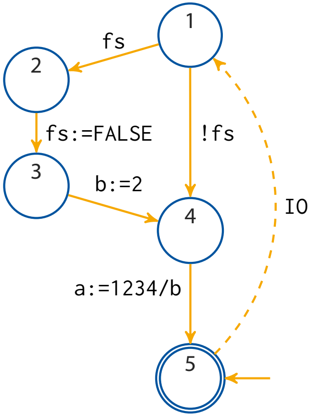


Toy Example: Invariant $a \geq 0$

- 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? ✗

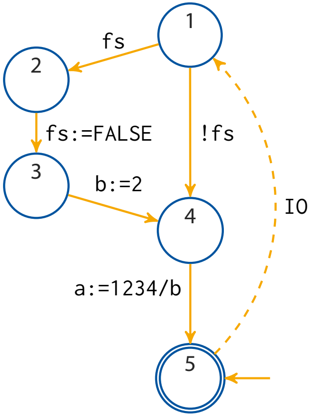


Delayed Write Semantics

- 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

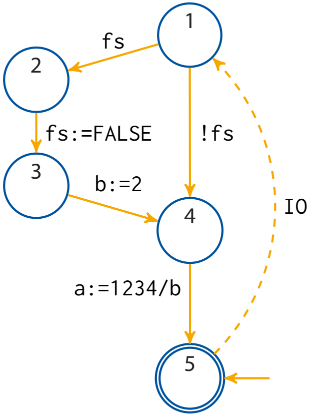


Delayed Write Semantics

- 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

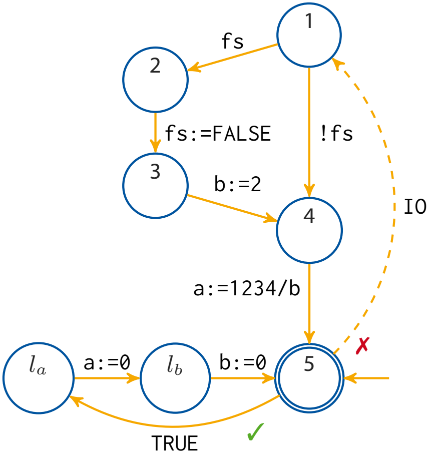


Delayed Write Semantics

- 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

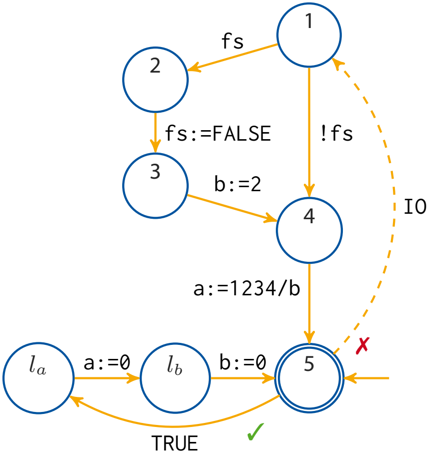


Delayed Write Semantics

- 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

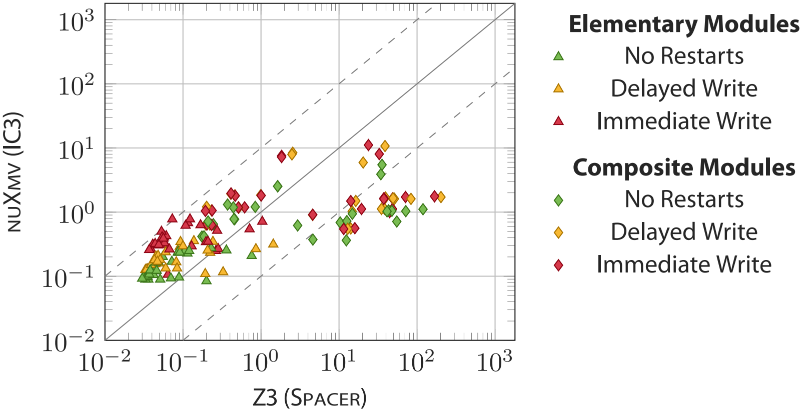


Figure: Time [s] spent checking restart-robustness w.r.t. each spec ($n = 3 \cdot 56$)

CHC-based Parameter Synthesis

- No aid in picking safe configuration of retain variables

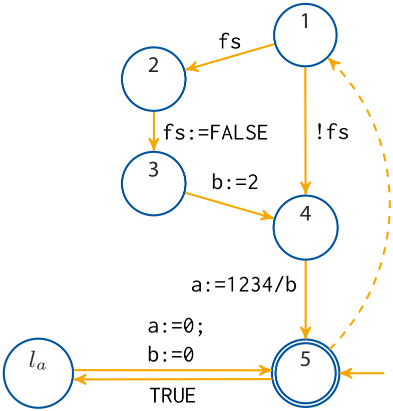
⇒ Add Boolean parameter `ret_v` for each non-retain variable v

- Derived CHCs check whether all configurations are robust

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

- Parameter synthesis is

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



CHC-based Parameter Synthesis

- No aid in picking safe configuration of retain variables

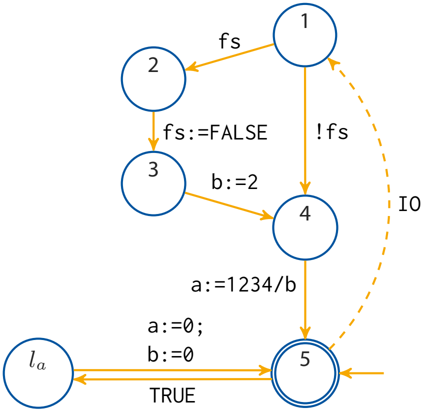
⇒ Add Boolean parameter `ret_v` for each non-retain variable v

- Derived CHCs check whether all configurations are robust

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

- Parameter synthesis is

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



CHC-based Parameter Synthesis

- No aid in picking safe configuration of retain variables

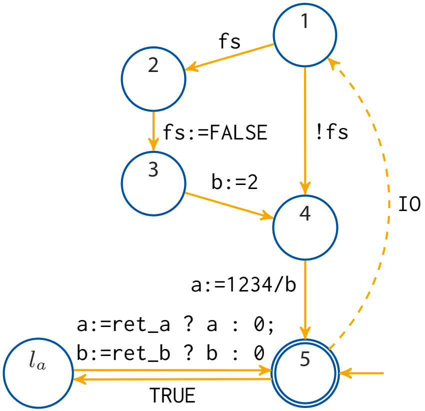
⇒ Add Boolean parameter `ret_v` for each non-retain variable v

- Derived CHCs check whether all configurations are robust

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

- Parameter synthesis is

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \cdots \to h(\vec{V})$$



CHC-based Parameter Synthesis

- No aid in picking safe configuration of retain variables

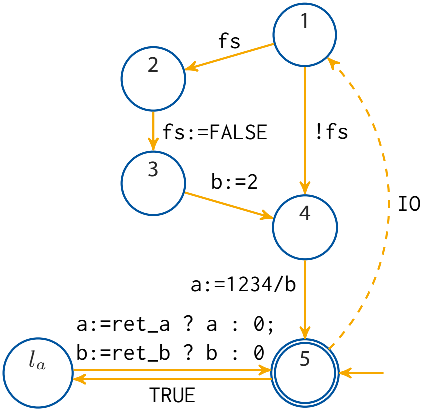
⇒ Add Boolean parameter `ret_v` for each non-retain variable v

- Derived CHCs check whether all configurations are robust

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

- Parameter synthesis is

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



CHC-based Parameter Synthesis

- No aid in picking safe configuration of retain variables

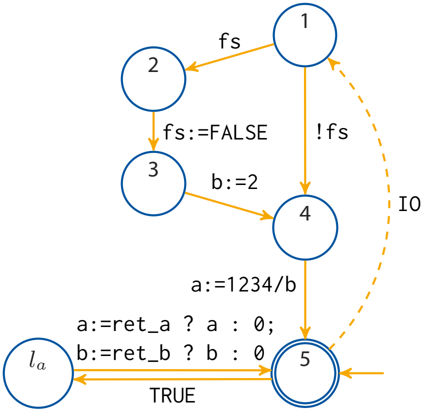
⇒ Add Boolean parameter `ret_v` for each non-retain variable v

- Derived CHCs check whether all configurations are robust

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

- Parameter synthesis is

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



Counterexample-Guided Parameter Synthesis

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

Observations:

- $\exists\forall$-quantified Horn clauses harder than regular CHCs (48 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

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

Observations:

- ∃∀-quantified Horn clauses harder than regular CHCs (48 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

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

Observations:

- ∃∀-quantified Horn clauses harder than regular CHCs (48 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

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \ \cdots \to h(\vec{V})$$

Observations:

- ∃∀-quantified Horn clauses harder than regular CHCs (48 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

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \; \cdots \to h(\vec{V})$$

Observations:

- $\exists\forall$-quantified Horn clauses harder than regular CHCs (48 TO)
- Our special case: existential quantification over Booleans

Idea:

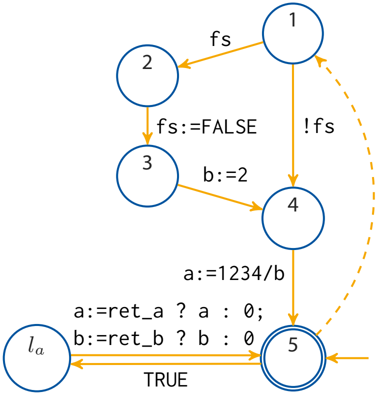
- Manage choice and reuse efficient check for fixed parameters
- Over-approximate set of “safe” parameters
- Refine it while counterexamples exist

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

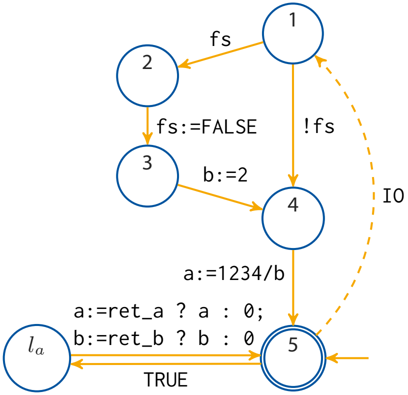


Example

- 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 $safe(\vec{V}_{par}) = true$
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:

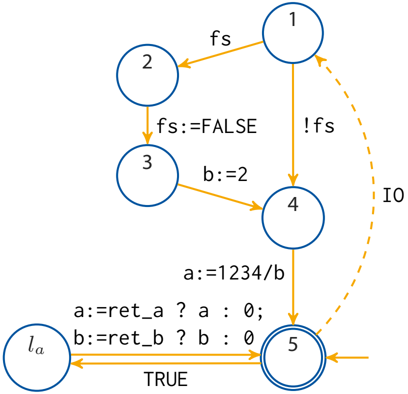
1. Start with $safe(\vec{V}_{par}) = true$
2. Backend finds counterexample

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

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:

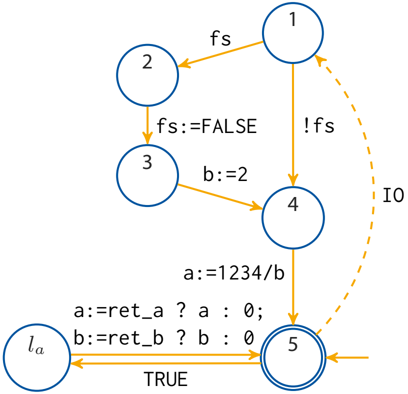
1. Start with $safe(\vec{V}_{par}) = true$
2. Backend finds counterexample

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

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:

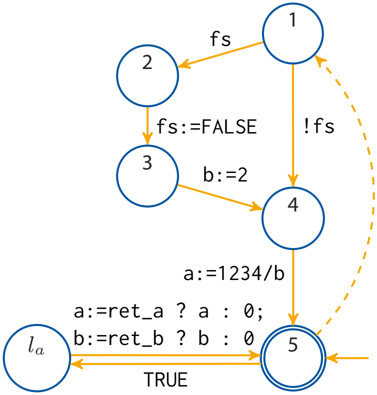
1. Start with $safe(\vec{V}_{par}) = true$
2. Backend finds counterexample

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

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

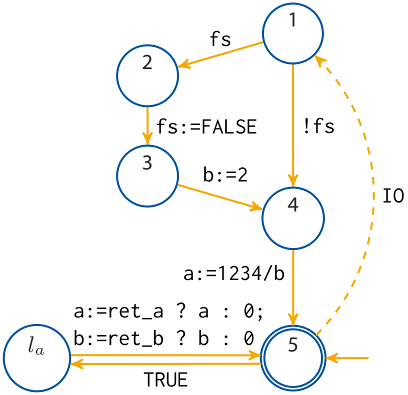


Example

- 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 $safe(\vec{V}_{par}) = true$
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



Experiments

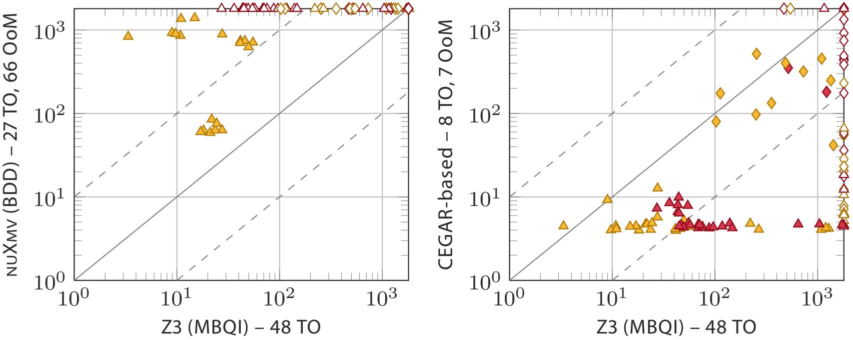


Figure: Time [s] spent on synthesis of restart-robust configurations ($n = 2 \cdot 56$)

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.