Design & Verification of Restart-robust Industrial Control Software

Dimitri Bohlender | Stefan Kowalewski

iFM 2018, Maynooth, 6 September 2018

Informatik 11
Embedded Software
RWTH AACHEN UNIVERSITY

Outline

Introduction
- Model Checking Industrial Control Software
- Motivation

Modelling the Restart Semantics
- Verification of Restart-robustness
- Synthesis of Safe Retain Configurations

Counterexample-Guided Parameter Synthesis

Outline

Introduction

- Model Checking Industrial Control Software
- Motivation

Modelling the Restart Semantics

- Verification of Restart-robustness
- Synthesis of Safe Retain Configurations

Counterexample-Guided Parameter Synthesis

Outline

Introduction

- Model Checking Industrial Control Software
- Motivation

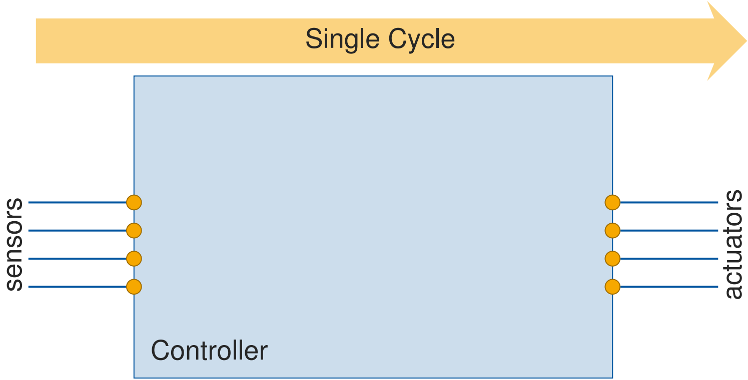
Modelling the Restart Semantics

- Verification of Restart-robustness
- Synthesis of Safe Retain Configurations

Counterexample-Guided Parameter Synthesis

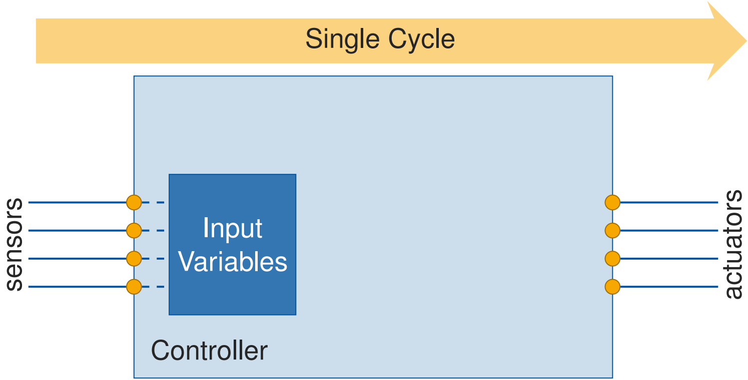
Mode of Operation

- Industrial controllers operate in program execution cycles
- Realise reactive systems



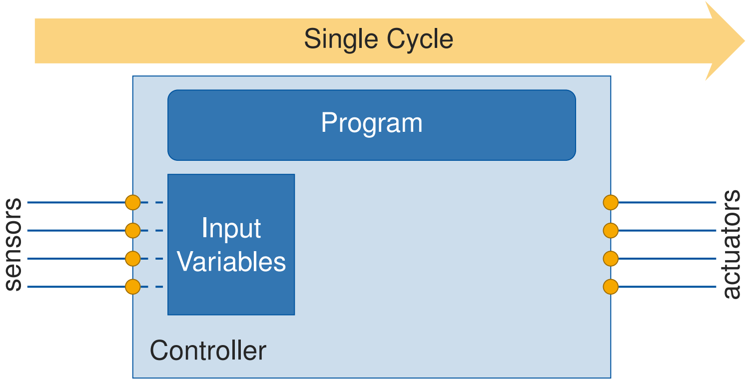
Mode of Operation

- Industrial controllers operate in program execution cycles
- Realise reactive systems



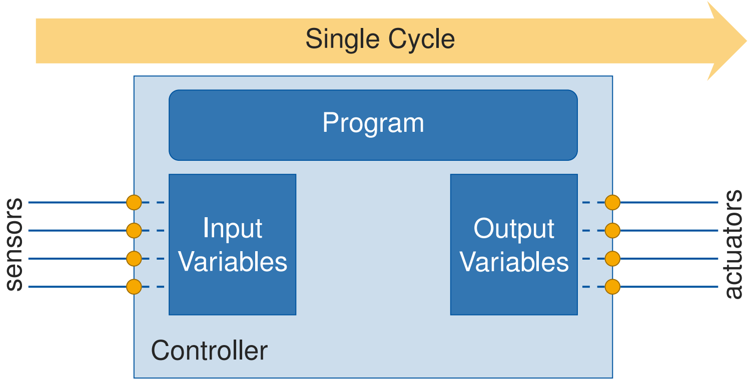
Mode of Operation

- Industrial controllers operate in program execution cycles
- Realise reactive systems



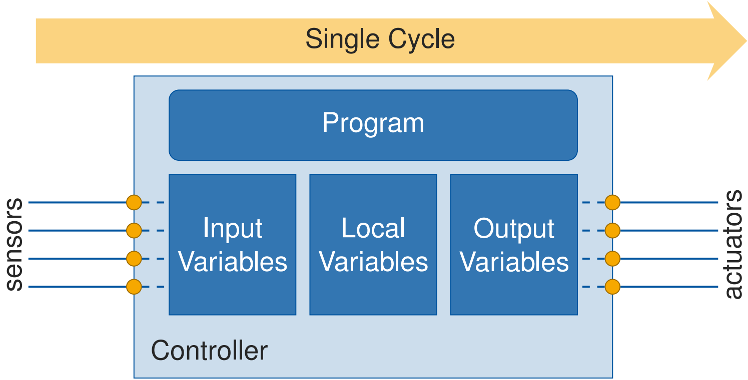
Mode of Operation

- Industrial controllers operate in program execution cycles
- Realise reactive systems



Mode of Operation

- Industrial controllers operate in program execution cycles
- Realise reactive systems

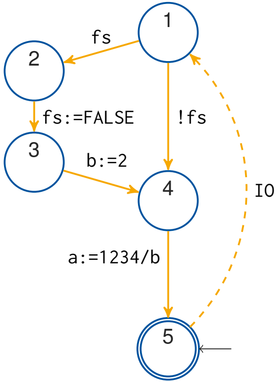


Software & Specifications

- Program as Control Flow Automaton
- Intermediate states are not observable

⇒ Automation engineers and specs always refer to the observable state

- Most specifications can be formalised via invariants or temporal logics
- Off-the-shelf verifier backend checks formalised program w.r.t. the specification

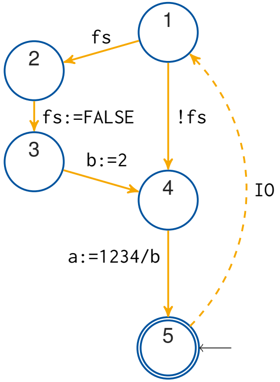


Software & Specifications

- Program as Control Flow Automaton
- Intermediate states are not observable

⇒ Automation engineers and specs always refer to the observable state

- Most specifications can be formalised via invariants or temporal logics
- Off-the-shelf verifier backend checks formalised program w.r.t. the specification

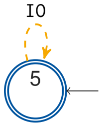


Software & Specifications

- Program as Control Flow Automaton
- Intermediate states are not observable

⇒ Automation engineers and specs always refer to the observable state

- Most specifications can be formalised via invariants or temporal logics
- Off-the-shelf verifier backend checks formalised program w.r.t. the specification

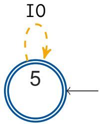


Software & Specifications

- Program as Control Flow Automaton
- Intermediate states are not observable

⇒ Automation engineers and specs always refer to the observable state

- Most specifications can be formalised via invariants or temporal logics
- Off-the-shelf verifier backend checks formalised program w.r.t. the specification

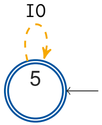


Software & Specifications

- Program as Control Flow Automaton
- Intermediate states are not observable

⇒ Automation engineers and specs always refer to the observable state

- Most specifications can be formalised via invariants or temporal logics
- Off-the-shelf verifier backend checks formalised program w.r.t. the specification



Symbolic Model Verifier (SMV)

- Uses symbolic transition system
- Transition relation

$$T \subseteq \underbrace{Loc \times \vec{V}}_{\text{pre}} \times \underbrace{Loc \times \vec{V}}_{\text{post}}$$

Constrained Horn Clauses (CHC)

- Uses formulas of the form

$$\forall \vec{V}\ \underbrace{p_1(\vec{V}) \wedge \ldots \wedge p_k(\vec{V}) \wedge \varphi}_{\text{body}} \rightarrow h(\vec{V})$$

- Predicates p_i typically characterise values at locations

Example: Invariant $a \geq 0$ at cycle end

- Check invariant $pc = 5 \rightarrow a \geq 0$ (SMV)
- Check satisfiability with $p_5(\vec{V}) \rightarrow a \geq 0$ added (CHC)

Symbolic Model Verifier (SMV)

- Uses symbolic transition system
- Transition relation

$$T \subseteq \underbrace{Loc \times \vec{V}}_{\text{pre}} \times \underbrace{Loc \times \vec{V}}_{\text{post}}$$

Constrained Horn Clauses (CHC)

- Uses formulas of the form

$$\forall \vec{V}\ \underbrace{p_1(\vec{V}) \wedge \ldots \wedge p_k(\vec{V}) \wedge \varphi}_{\text{body}} \rightarrow h(\vec{V})$$

- Predicates p_i typically characterise values at locations

Example: Invariant $a \geq 0$ at cycle end

- Check invariant $pc = 5 \rightarrow a \geq 0$ (SMV)
- Check satisfiability with $p_5(\vec{V}) \rightarrow a \geq 0$ added (CHC)

Symbolic Model Verifier (SMV)

- Uses symbolic transition system
- Transition relation

$$T \subseteq \underbrace{Loc \times \vec{V}}_{\text{pre}} \times \underbrace{Loc \times \vec{V}}_{\text{post}}$$

Constrained Horn Clauses (CHC)

- Uses formulas of the form

$$\forall \vec{V}\; \underbrace{p_1(\vec{V}) \wedge \ldots \wedge p_k(\vec{V}) \wedge \varphi}_{\text{body}} \rightarrow h(\vec{V})$$

- Predicates p_i typically characterise values at locations

Example: Invariant $a \geq 0$ at cycle end

- Check invariant $pc = 5 \rightarrow a \geq 0$ (SMV)
- Check satisfiability with $p_5(\vec{V}) \rightarrow a \geq 0$ added (CHC)

Symbolic Model Verifier (SMV)

- Uses symbolic transition system
- Transition relation

$$T \subseteq \underbrace{Loc \times \vec{V}}_{\text{pre}} \times \underbrace{Loc \times \vec{V}}_{\text{post}}$$

Constrained Horn Clauses (CHC)

- Uses formulas of the form

$$\forall \vec{V}\ \underbrace{p_1(\vec{V}) \land \ldots \land p_k(\vec{V}) \land \varphi}_{\text{body}} \rightarrow h(\vec{V})$$

- Predicates p_i typically characterise values at locations

Example: Invariant $a \geq 0$ at cycle end

- Check invariant $pc = 5 \rightarrow a \geq 0$ (SMV)
- Check satisfiability with $p_5(\vec{V}) \rightarrow a \geq 0$ added (CHC)

Retain Variables

- Applications are often safety critical
- Power outage or manual restart should not affect correctness

⇒ Controllers feature battery-backed memory to retain variables

Example

Retain drill's position in automated processing of workpieces

- Assignments to such variables have unspecified semantics
- Prominent semantics:
  - Immediate writing to the battery-backed memory
  - Delayed writing at the current execution cycle's end

Informatik 11 Embedded Software
RWTH AACHEN UNIVERSITY

Retain Variables

- Applications are often safety critical
- Power outage or manual restart should not affect correctness

⇒ Controllers feature battery-backed memory to retain variables

Example

Retain drill's position in automated processing of workpieces

- Assignments to such variables have unspecified semantics
- Prominent semantics:
  - Immediate writing to the battery-backed memory
  - Delayed writing at the current execution cycle's end

Informatik 11 Embedded Software

RWTH AACHEN UNIVERSITY

Retain Variables

- Applications are often safety critical
- Power outage or manual restart should not affect correctness

⇒ Controllers feature battery-backed memory to retain variables

Example

Retain drill's position in automated processing of workpieces

- Assignments to such variables have unspecified semantics
- Prominent semantics:
  - Immediate writing to the battery-backed memory
  - Delayed writing at the current execution cycle's end

Retain Variables

- Applications are often safety critical
- Power outage or manual restart should not affect correctness

⇒ Controllers feature battery-backed memory to retain variables

Example

Retain drill's position in automated processing of workpieces

- Assignments to such variables have unspecified semantics
- Prominent semantics:
  - Immediate writing to the battery-backed memory
  - Delayed writing at the current execution cycle's end

Retain Variables

- Applications are often safety critical
- Power outage or manual restart should not affect correctness

⇒ Controllers feature battery-backed memory to retain variables

Example

Retain drill's position in automated processing of workpieces

- Assignments to such variables have unspecified semantics
- Prominent semantics:
  - Immediate writing to the battery-backed memory
  - Delayed writing at the current execution cycle's end

Retain Variables

- Applications are often safety critical
- Power outage or manual restart should not affect correctness

⇒ Controllers feature battery-backed memory to retain variables

Example

Retain drill's position in automated processing of workpieces

- Assignments to such variables have unspecified semantics
- Prominent semantics:
  - Immediate writing to the battery-backed memory
  - Delayed writing at the current execution cycle's end

Informatik 11 Embedded Software
RWTH AACHEN UNIVERSITY

Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

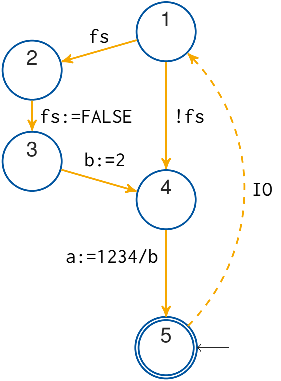
- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant?
- Robust with delayed writes?
- Robust with immediate writes?
- Fixable for delayed writes?
- Fixable for immediate writes?

Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant?
- Robust with delayed writes?
- Robust with immediate writes?
- Fixable for delayed writes?
- Fixable for immediate writes?

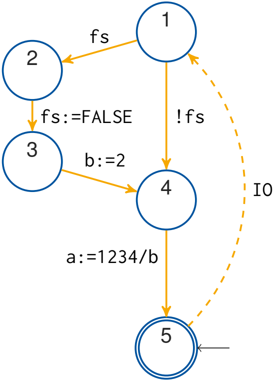


Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

- Initialised with $[\mathit{fs} \mapsto \mathit{true}, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant?
- Robust with delayed writes?
- Robust with immediate writes?
- Fixable for delayed writes?
- Fixable for immediate writes?

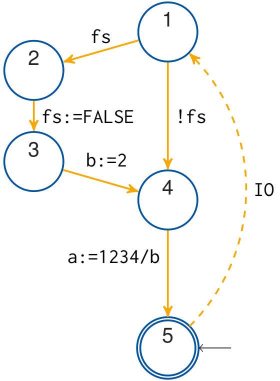


Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag fs is retained
- Nominal behaviour compliant? ✓
- Robust with delayed writes?
- Robust with immediate writes?
- Fixable for delayed writes?
- Fixable for immediate writes?

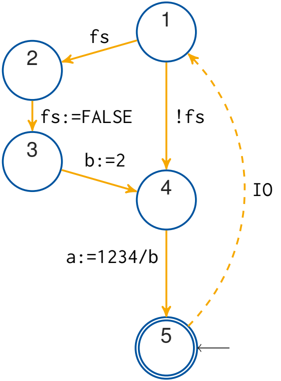


Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant? ✓
- Robust with delayed writes?
- Robust with immediate writes?
- Fixable for delayed writes?
- Fixable for immediate writes?

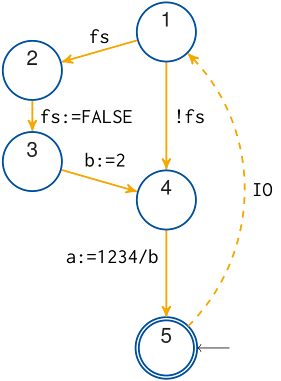


Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant? ✓
- Robust with delayed writes? `a:=1234/0`
- Robust with immediate writes?
- Fixable for delayed writes?
- Fixable for immediate writes?

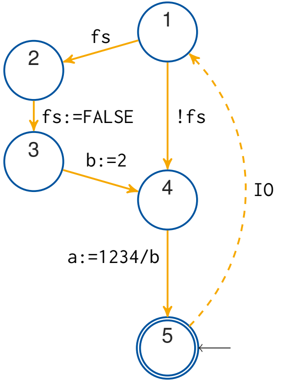


Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant? ✓
- Robust with delayed writes? `a:=1234/0`
- Robust with immediate writes?
- Fixable for delayed writes?
- Fixable for immediate writes?

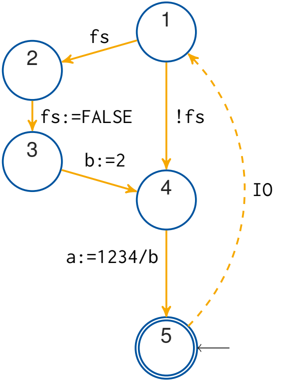


Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant? ✓
- Robust with delayed writes? `a:=1234/0`
- Robust with immediate writes? ✗
- Fixable for delayed writes?
- Fixable for immediate writes?



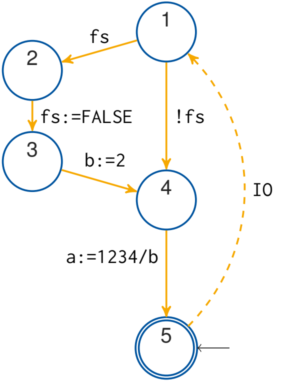
Informatik 11 Embedded Software
RWTH AACHEN UNIVERSITY

Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant? ✓
- Robust with delayed writes? `a:=1234/0`
- Robust with immediate writes? ✗
- Fixable for delayed writes?
- Fixable for immediate writes?

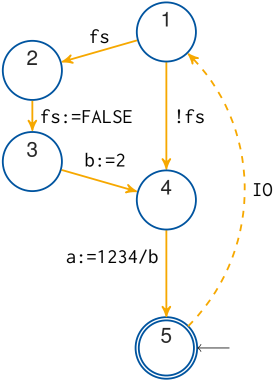


Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant? ✓
- Robust with delayed writes? `a:=1234/0`
- Robust with immediate writes? ✗
- Fixable for delayed writes? Retain b
- Fixable for immediate writes?

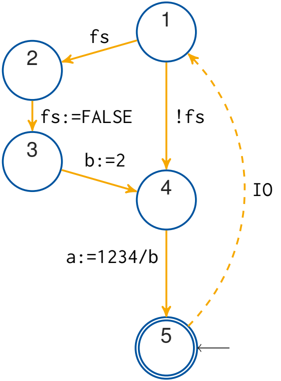


Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant? ✓
- Robust with delayed writes? `a:=1234/0`
- Robust with immediate writes? ✗
- Fixable for delayed writes? Retain b
- Fixable for immediate writes?

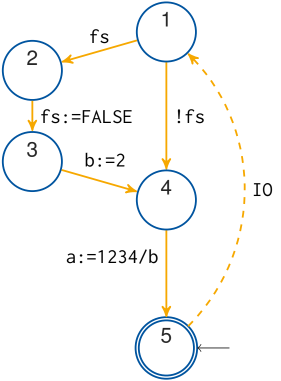


Restart-robustness

Program is restart-robust w.r.t. a spec, if it complies with the spec in the context of restarts

Restart-robustness w.r.t. invariant $a \geq 0$

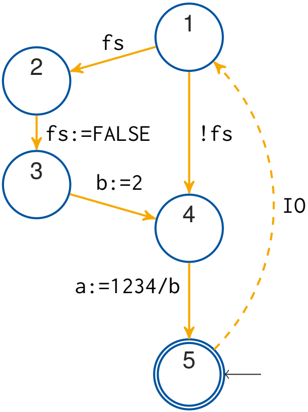
- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant? ✓
- Robust with delayed writes? `a:=1234/0`
- Robust with immediate writes? ✗
- Fixable for delayed writes? Retain b
- Fixable for immediate writes? ✗



Delayed Write Semantics

- Approach by instrumenting the CFA with restart-behaviour
- Observation: In case of restart, operations since last cycle are irrelevant

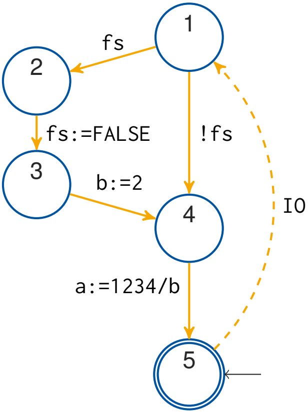
⇒ Model as nondeterministic choice: restart in next cycle?



Delayed Write Semantics

- Approach by instrumenting the CFA with restart-behaviour
- Observation: In case of restart, operations since last cycle are irrelevant

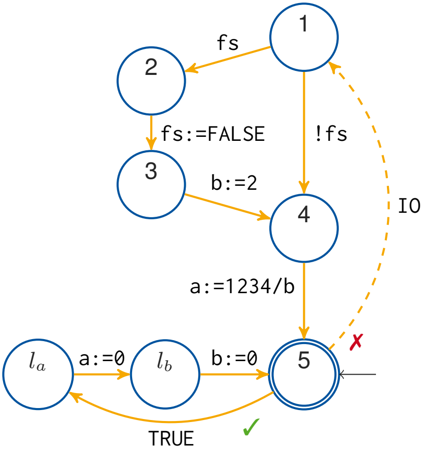
⇒ Model as nondeterministic choice: restart in next cycle?



Delayed Write Semantics

- Approach by instrumenting the CFA with restart-behaviour
- Observation: In case of restart, operations since last cycle are irrelevant

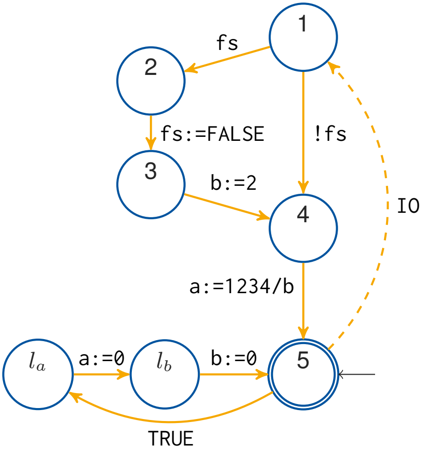
⇒ Model as nondeterministic choice: restart in next cycle?



Immediate Write Semantics

- Observation: In case of restart, operations since last write to a retain variable are irrelevant

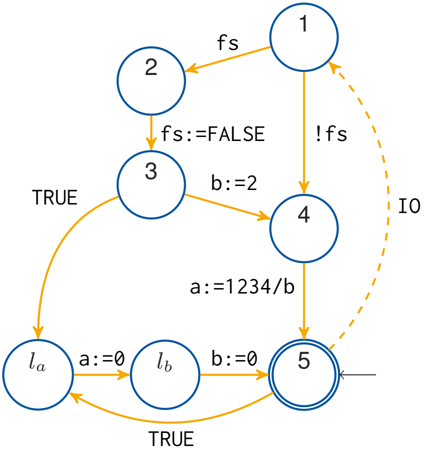
⇒ Each adds the option to restart



Immediate Write Semantics

- **Observation**: In case of restart, operations since last write to a retain variable are irrelevant

⇒ Each adds the option to restart



Parametrisation of Retains

- Instrumentation enables checking restart-robustness w.r.t. a spec via common verifier backends
- Doesn't help with finding safe configuration of retain variables

⇒ Parametrise the CFA with the configuration of retain variables

Parametrisation of Retains

- Instrumentation enables checking restart-robustness w.r.t. a spec via common verifier backends
- Doesn't help with finding safe configuration of retain variables

$\Rightarrow$ Parametrise the CFA with the configuration of retain variables

Parametrisation of Retains

- Instrumentation enables checking restart-robustness w.r.t. a spec via common verifier backends
- Doesn't help with finding safe configuration of retain variables

$\Rightarrow$ Parametrise the CFA with the configuration of retain variables

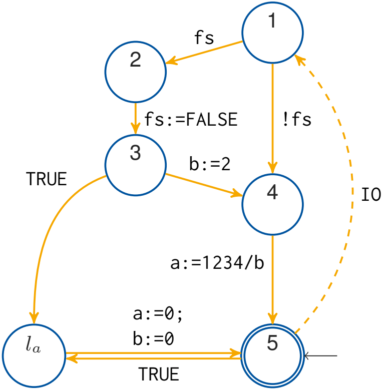
CHC-based Parameter Synthesis

- Add Boolean parameter `ret_v` for each non-retain variable `v`
- Add guarded options to restart
- Derived CHCs check whether all configurations are robust

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

- Parameter synthesis is

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



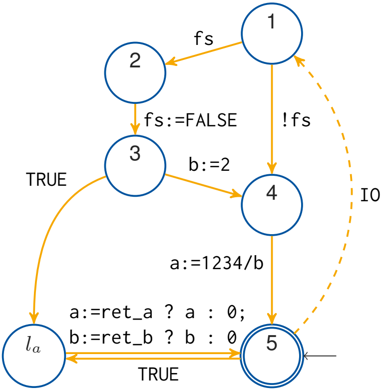
CHC-based Parameter Synthesis

- Add Boolean parameter `ret_v` for each non-retain variable `v`
- Add guarded options to restart
- Derived CHCs check whether all configurations are robust

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

- Parameter synthesis is

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



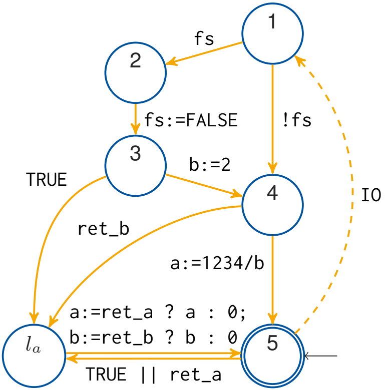
CHC-based Parameter Synthesis

- Add Boolean parameter `ret_v` for each non-retain variable `v`
- Add guarded options to restart
- Derived CHCs check whether all configurations are robust

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

- Parameter synthesis is

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



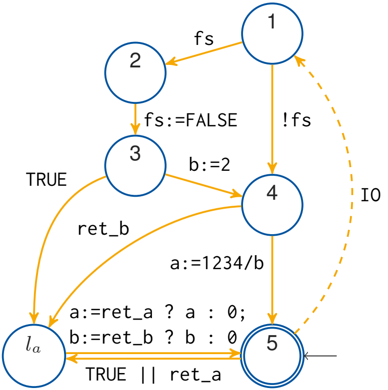
CHC-based Parameter Synthesis

- Add Boolean parameter `ret_v` for each non-retain variable `v`
- Add guarded options to restart
- Derived CHCs check whether all configurations are robust

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

- Parameter synthesis is

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



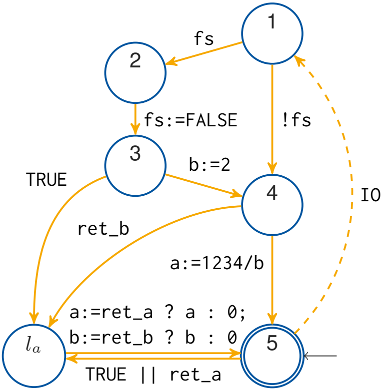
CHC-based Parameter Synthesis

- Add Boolean parameter `ret_v` for each non-retain variable `v`
- Add guarded options to restart
- Derived CHCs check whether all configurations are robust

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

- Parameter synthesis is

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



SMV-based Parameter Synthesis

- No quantifiers for variables

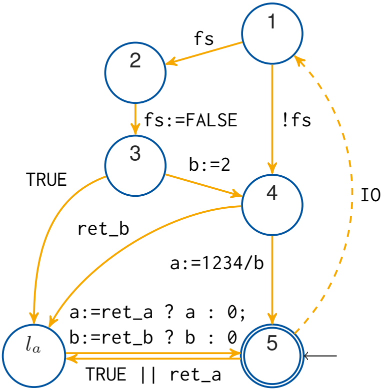
⇒ Integrate parameter choice prepended to entry

- Requires adaptation of spec

Invariant $a \geq 0$

Quantification via CTL:

$$\text{EX AG}(pc = 5 \rightarrow a \geq 0)$$



SMV-based Parameter Synthesis

- No quantifiers for variables

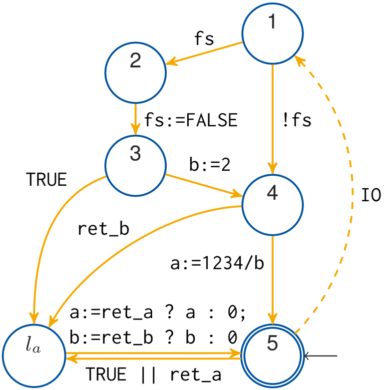
⇒ Integrate parameter choice prepended to entry

- Requires adaptation of spec

Invariant $a \geq 0$

Quantification via CTL:

EX AG$(pc = 5 \rightarrow a \geq 0)$



SMV-based Parameter Synthesis

- No quantifiers for variables

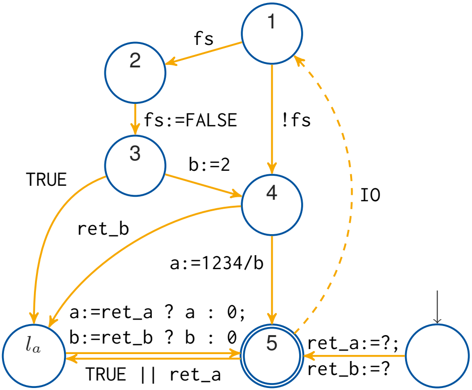
⇒ Integrate parameter choice prepended to entry

- Requires adaptation of spec

Invariant $a \geq 0$

Quantification via CTL:

$$\text{EX AG}(pc = 5 \rightarrow a \geq 0)$$



Benchmarks

Experiments on implementation of PLCopen Safety library:

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

Specifications:

- Only invariants are natively supported by all backends
- Compare runtime for nominal and instrumented semantics

Backends:

- NUXMV for SMV formalism
- Z3 for CHC formalism

Benchmarks

Experiments on implementation of PLCopen Safety library:

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

Specifications:

- Only invariants are natively supported by all backends
- Compare runtime for nominal and instrumented semantics

Backends:

- nuXmv for SMV formalism
- Z3 for CHC formalism

Benchmarks

Experiments on implementation of PLCopen Safety library:

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

Specifications:

- Only invariants are natively supported by all backends
- Compare runtime for nominal and instrumented semantics

Backends:

- NUXMV for SMV formalism
- Z3 for CHC formalism

Restart-robustness Checking

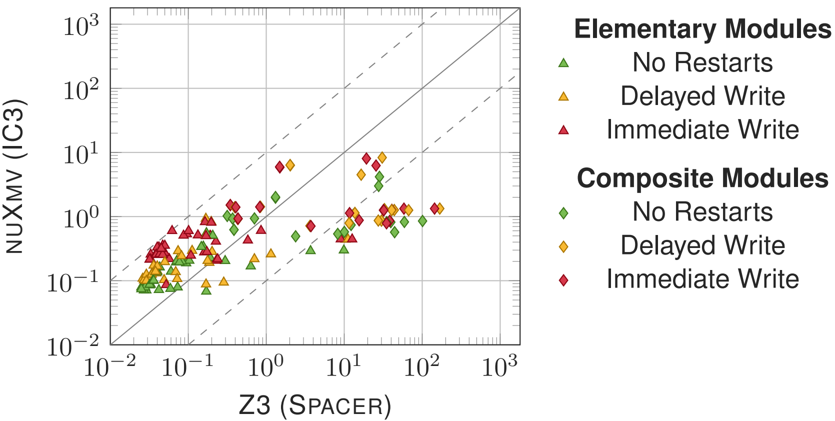


Figure: Time [s] spent checking restart-robustness w.r.t. each spec

Parameter Synthesis

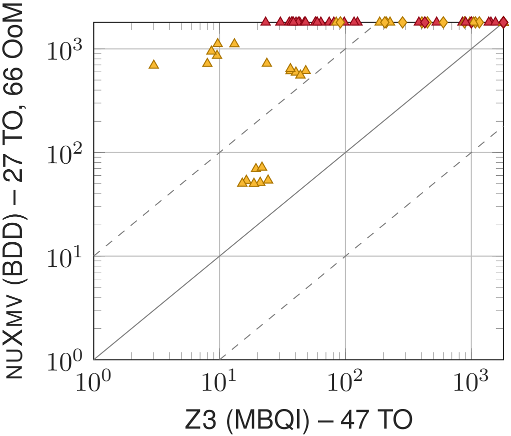


Figure: Time [s] spent on synthesis of restart-robust configurations

Can we do better?

Observations:

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

Idea:

- Manage choice of parameters and reuse efficient procedures for reasoning about restart-robustness for fixed parameters
- Over-approximate set of "safe" parameters and refine it while counterexamples exist

Can we do better?

Observations:

- ∃∀-quantified Horn clauses harder than regular CHCs
- Our special case: existential quantification over Booleans

Idea:

- Manage choice of parameters and reuse efficient procedures for reasoning about restart-robustness for fixed parameters
- Over-approximate set of "safe" parameters and refine it while counterexamples exist

Can we do better?

Observations:

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

Idea:

- Manage choice of parameters and reuse efficient procedures for reasoning about restart-robustness for fixed parameters
- Over-approximate set of “safe” parameters and refine it while counterexamples exist

Can we do better?

Observations:

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

Idea:

- Manage choice of parameters and reuse efficient procedures for reasoning about restart-robustness for fixed parameters
- Over-approximate set of “safe” parameters and 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:

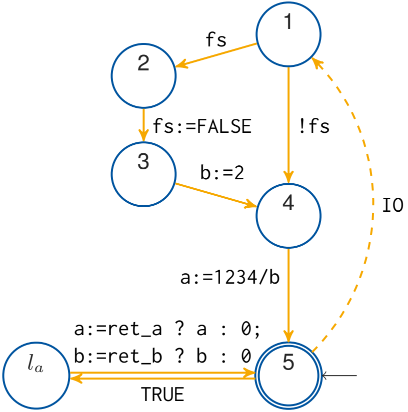
1. Start with $safe(\vec{V}_{par}) = true$
2. Backend finds counterexample

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

3. Find subset of violating params

$$c_g = \neg ret_b$$

4. Refine $safe(\vec{V}_{par}) = true \wedge \neg c_g$
5. Backend finds no violations

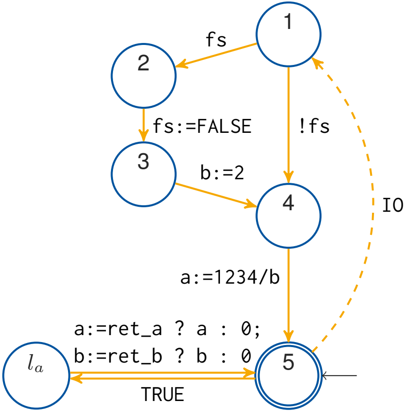


Example

- 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 params
$$c_g = \neg ret_b$$
4. Refine $safe(\vec{V}_{par}) = true \wedge \neg c_g$
5. Backend finds no violations

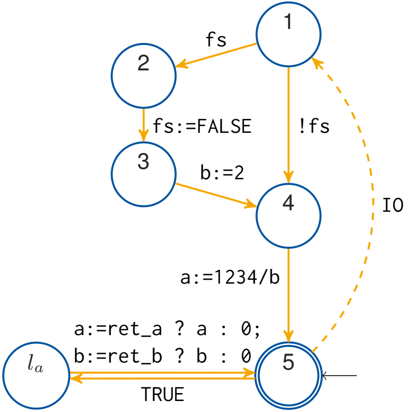


Example

- 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 params
$$c_g = \neg ret_b$$
4. Refine $safe(\vec{V}_{par}) = true \wedge \neg c_g$
5. Backend finds no violations

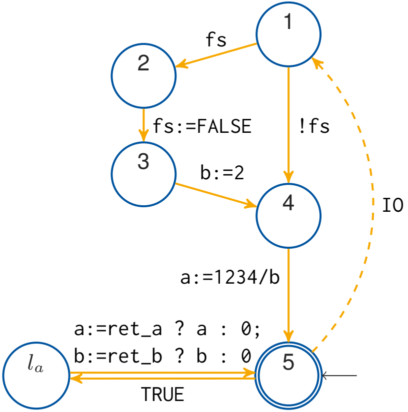


Example

- 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 params
$$c_g = \neg ret_b$$
4. Refine $safe(\vec{V}_{par}) = true \wedge \neg c_g$
5. Backend finds no violations

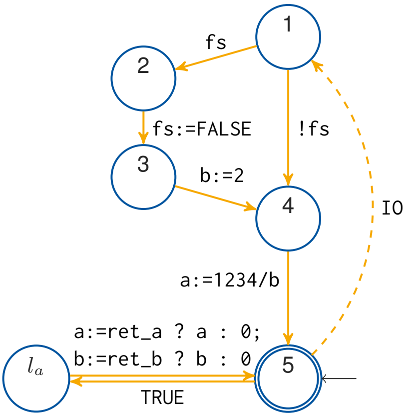


Example

- 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 params
$$c_g = \neg ret_b$$
4. Refine $safe(\vec{V}_{par}) = true \wedge \neg c_g$
5. Backend finds no violations

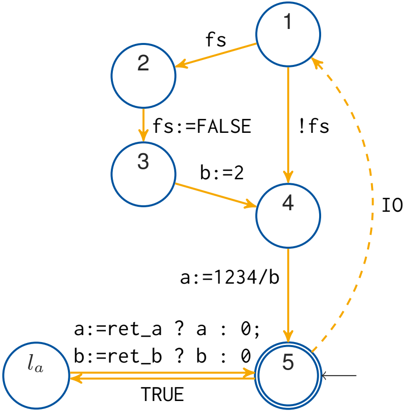


Example

- 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 params
$$c_g = \neg ret_b$$
4. Refine $safe(\vec{V}_{par}) = true \wedge \neg c_g$
5. Backend finds no violations



Improved Results

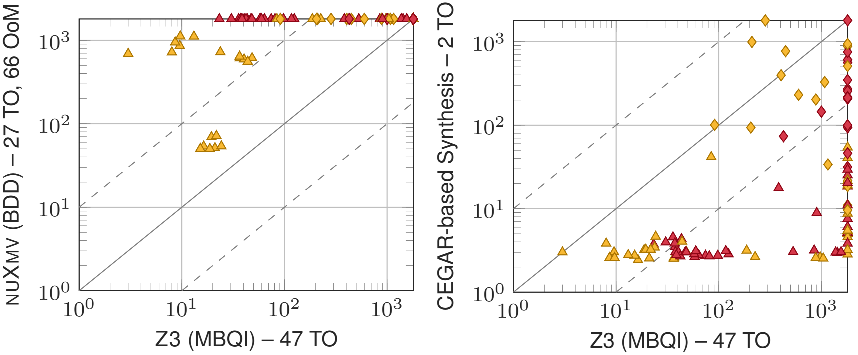


Figure: Time [s] spent on synthesis of restart-robust configurations

Summary

- Retain variables were introduced with better safety in mind but allow for subtle corner cases and unexpected behaviour
- Restart-robustness checking and synthesis of "safe" retain configurations can be accomplished with existing tooling
- However, parameter synthesis only feasible with our counterexample-guided approach

Future work will investigate restart-robustness as a relational property between the nominal and restart-behaviour.

Summary

- Retain variables were introduced with better safety in mind but allow for subtle corner cases and unexpected behaviour
- Restart-robustness checking and synthesis of “safe” retain configurations can be accomplished with existing tooling
- However, parameter synthesis only feasible with our counterexample-guided approach

Future work will investigate restart-robustness as a relational property between the nominal and restart-behaviour.

Informatik 11 Embedded Software
RWTH AACHEN UNIVERSITY

Summary

- Retain variables were introduced with better safety in mind but allow for subtle corner cases and unexpected behaviour
- Restart-robustness checking and synthesis of “safe” retain configurations can be accomplished with existing tooling
- However, parameter synthesis only feasible with our counterexample-guided approach

Future work will investigate restart-robustness as a relational property between the nominal and restart-behaviour.

Informatik 11 Embedded Software

RWTH AACHEN UNIVERSITY

Summary

- Retain variables were introduced with better safety in mind but allow for subtle corner cases and unexpected behaviour
- Restart-robustness checking and synthesis of "safe" retain configurations can be accomplished with existing tooling
- However, parameter synthesis only feasible with our counterexample-guided approach

Future work will investigate restart-robustness as a relational property between the nominal and restart-behaviour.

Related Work

- [Hau+15] assumes delayed write semantics and adapts static value analysis to distinguish between variables' values before and after a restart
- Crash recoverability of C programs [KY16] is a related problem, using a similar modelling, but differing from restart-robustness in terms of requirements and program transformations
- SMV-based parameter synthesis for models of gene regulatory networks [Bat+10]
- Our counterexample-guided approach is most similar to [Cim+13] but does not require quantifier elimination, is independent of the chosen theory to model values, and works with any CHC-solving algorithm

PLC Software

- Written in textual & graphical languages from IEC 61131-3
- Features no recursion

⇒ Formalised as Control Flow Automaton (CFA)

```
PROGRAM RunningExample
  VAR RETAIN
    fs:BOOL := TRUE;
  END_VAR
  VAR
    a:INT := 0;
    b:INT := 0;
  END_VAR
  IF fs THEN
    fs := FALSE;
    b := 2;
  END_IF
  a := 1234/b;
END_PROGRAM
```

PLC Software

- Written in textual & graphical languages from IEC 61131-3
- Features no recursion

⇒ Formalised as Control Flow Automaton (CFA)

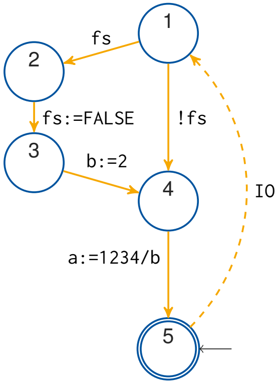
```
PROGRAM RunningExample
  VAR RETAIN
    fs:BOOL := TRUE;
  END_VAR
  VAR
    a:INT := 0;
    b:INT := 0;
  END_VAR
  IF fs THEN
    fs := FALSE;
    b := 2;
  END_IF
  a := 1234/b;
END_PROGRAM
```

PLC Software

- Written in textual & graphical languages from IEC 61131-3
- Features no recursion

⇒ Formalised as Control Flow Automaton (CFA)

```
PROGRAM RunningExample
  VAR RETAIN
    fs:BOOL := TRUE;
  END_VAR
  VAR
    a:INT := 0;
    b:INT := 0;
  END_VAR
  IF fs THEN
    fs := FALSE;
    b := 2;
  END_IF
  a := 1234/b;
END_PROGRAM
```



Informatik 11 Embedded Software
RWTH AACHEN UNIVERSITY

Algorithm 1: SynthRetainConf(P, φ)

Input: Program $P = (\vec{X} \uplus \vec{X}_{\text{par}}, \vec{X}_{\text{in}}, \mathcal{A}, l_{\text{EoC}}, l_{\text{EoC}}, \mathit{def})$ with prametrised retains
Predicate $\varphi(\vec{X})$ characterising safe states

Variables: Predicate $\mathit{safe}(\vec{X}_{\text{par}})$ charactering parameters that do not lead to violations
Universally quantified Horn clauses $\mathcal{H}$

$\mathcal{H} \leftarrow$ toHorn(P) // Represent program as ∀CHCs
$(\vec{V}, I, T) \leftarrow$ toSymTS(P) // and as symbolic transition system
$\mathit{safe}(\vec{X}_{\text{par}}) \leftarrow \mathit{true}$ // All parameters are assumed to be safe
while ¬sat $(\mathcal{H} \cup \{\varphi(\vec{X}) \leftarrow p_{EoC}(\vec{X} \uplus \vec{X}_{par}), \mathit{safe}(\vec{X}_{par})\})$ **do** // ∃ violating run?
  $k \leftarrow$ length of violating run
  $c_{\text{par}} \leftarrow$ cube of chosen (Boolean) parameter values in violating run
  foreach lit *in* c_{par} **do**
    $\bar{c}_{\text{par}} \leftarrow c_{\text{par}}$ with negated lit // Flip literal
    if sat $(I(\vec{V}) \wedge \bigwedge_{0 \leq i < k} T(\vec{V}_i, \vec{V}_{i+1}) \wedge \bar{c}_{par} \wedge \neg\varphi(\vec{X}_k))$ **then** // Still violating?
      $c_{\text{par}} \leftarrow c_{\text{par}} \setminus \mathit{lit}$ // Drop literal
  $\mathit{safe}(\vec{X}_{\text{par}}) \leftarrow \mathit{safe}(\vec{X}_{\text{par}}) \wedge \neg c_{\text{par}}$ // Block unsafe parameters
return $\mathit{safe}(\vec{X}_{\text{par}})$ // (Potentially empty) region of safe parameters

References I

[Bat+10] Grégory Batt et al. "Efficient parameter search for qualitative models of regulatory networks using symbolic model checking". In: *Bioinformatics* 26.18 (2010).

[Cim+13] Alessandro Cimatti et al. "Parameter synthesis with IC3". In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 2013, pp. 165–168.

References II

[Hau+15] Stefan Hauck-Stattelmann et al. "Analyzing the Restart Behavior of Industrial Control Applications". In: *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*. 2015, pp. 585–588.

[KY16] Eric Koskinen and Junfeng Yang. "Reducing crash recoverability to reachability". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, pp. 97–108.