Design & Verification of Restart-robust Industrial Control Software

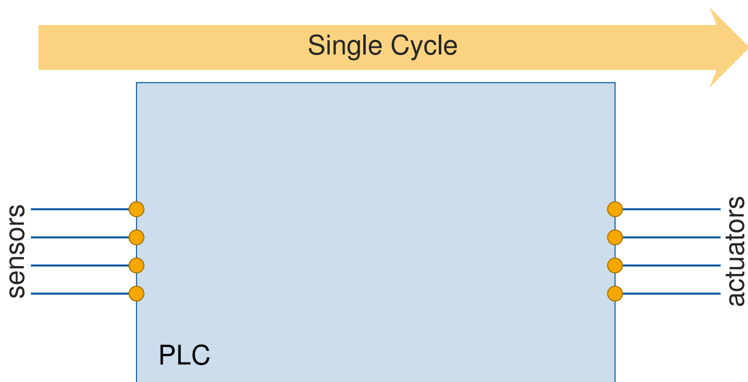
Dimitri Bohlender

VTSA'18, Inria Nancy, 27 August 2018

Informatik 11
Embedded Software
RWTH AACHEN UNIVERSITY

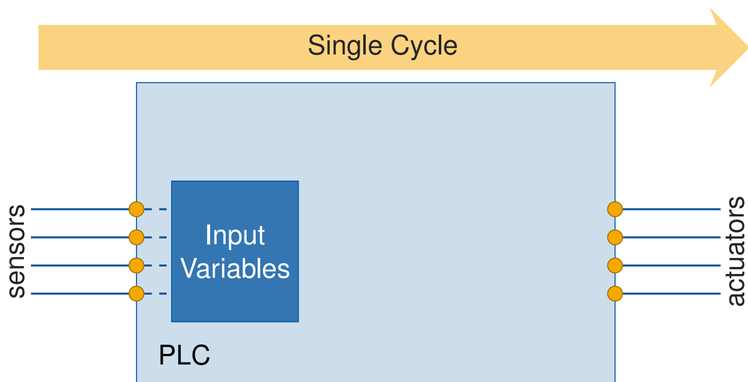
Programmable Logic Controllers (PLCs)

- PLCs are devices tailored to the domain of industrial automation, e.g. for actuating valves of a tank
- Realise reactive systems, repeatedly executing the same task



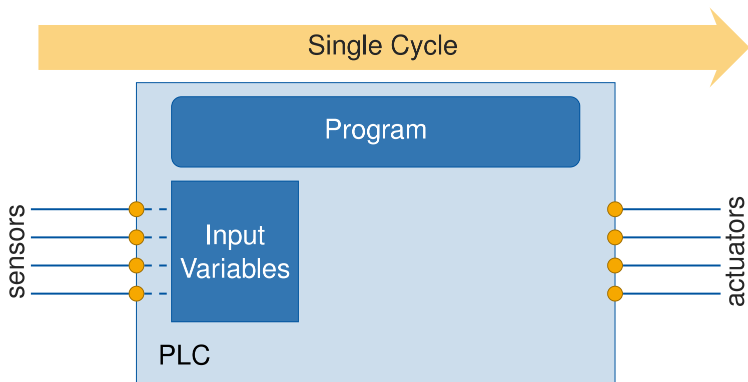
Programmable Logic Controllers (PLCs)

- PLCs are devices tailored to the domain of industrial automation, e.g. for actuating valves of a tank
- Realise reactive systems, repeatedly executing the same task



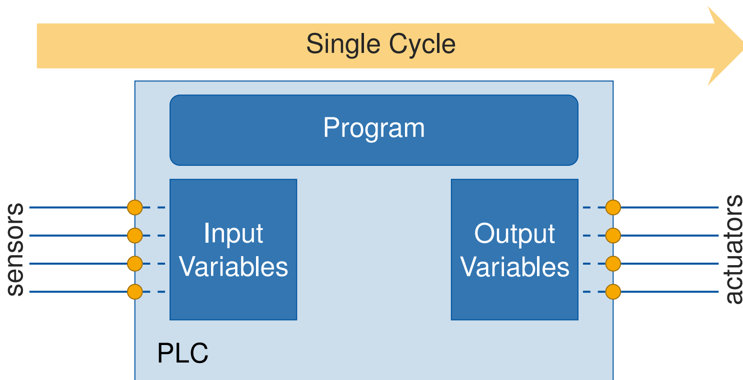
Programmable Logic Controllers (PLCs)

- PLCs are devices tailored to the domain of industrial automation, e.g. for actuating valves of a tank
- Realise reactive systems, repeatedly executing the same task



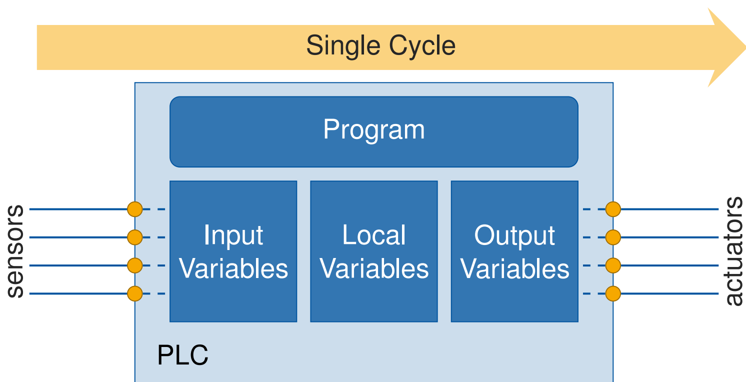
Programmable Logic Controllers (PLCs)

- PLCs are devices tailored to the domain of industrial automation, e.g. for actuating valves of a tank
- Realise reactive systems, repeatedly executing the same task



Programmable Logic Controllers (PLCs)

- PLCs are devices tailored to the domain of industrial automation, e.g. for actuating valves of a tank
- Realise reactive systems, repeatedly executing the same task

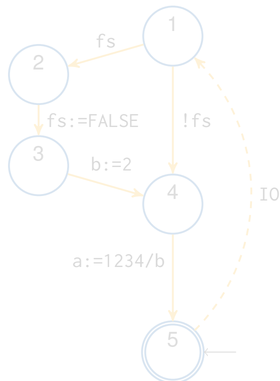


PLC Software

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

⇒ Formalised as Control Flow Automaton (CFA)

```
1  PROGRAM RunningExample
2    VAR RETAIN
3      fs:BOOL := TRUE;
4    END_VAR
5    VAR
6      a:INT := 0;
7      b:INT := 0;
8    END_VAR
9    IF fs THEN
10     fs := FALSE;
11     b := 2;
12   END_IF
13   a := 1234/b;
14 END_PROGRAM
```

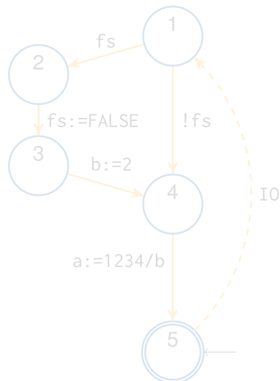


PLC Software

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

⇒ Formalised as Control Flow Automaton (CFA)

```
1  PROGRAM RunningExample
2    VAR RETAIN
3      fs:BOOL := TRUE;
4    END_VAR
5    VAR
6      a:INT := 0;
7      b:INT := 0;
8    END_VAR
9    IF fs THEN
10     fs := FALSE;
11     b := 2;
12   END_IF
13   a := 1234/b;
14 END_PROGRAM
```

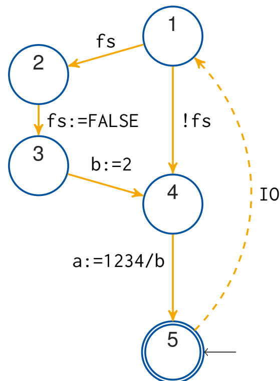


PLC Software

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

⇒ Formalised as Control Flow Automaton (CFA)

```
1  PROGRAM RunningExample
2    VAR RETAIN
3      fs:BOOL := TRUE;
4    END_VAR
5    VAR
6      a:INT := 0;
7      b:INT := 0;
8    END_VAR
9    IF fs THEN
10     fs := FALSE;
11     b := 2;
12   END_IF
13   a := 1234/b;
14 END_PROGRAM
```

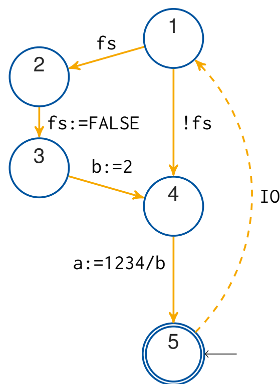


Specifications

- 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
- Domain-specific specifications may require dedicated procedures:
  - PLCopen-/Specification automata
  - Cycle-bounded temporal logics

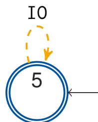


Specifications

- 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
- Domain-specific specifications may require dedicated procedures:
  - PLCopen-/Specification automata
  - Cycle-bounded temporal logics

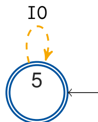


Specifications

- 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
- Domain-specific specifications may require dedicated procedures:
  - PLCopen-/Specification automata
  - Cycle-bounded temporal logics

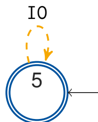


Specifications

- 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
- Domain-specific specifications may require dedicated procedures:
  - PLCopen-/Specification automata
  - Cycle-bounded temporal logics

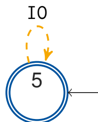


Specifications

- 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
- Domain-specific specifications may require dedicated procedures:
  - PLCopen-/Specification automata
  - Cycle-bounded temporal logics



Retain Variables

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

⇒ PLCs feature battery-backed memory for retain variables

Example

Retain drill's position in automated processing of workpieces

- Assignments to such variables have unspecified semantics
- Prominent: delayed writing at the current PLC cycle's end

Informatik 11
Embedded Software
RWTH AACHEN UNIVERSITY

Retain Variables

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

⇒ PLCs feature battery-backed memory for retain variables

Example

Retain drill's position in automated processing of workpieces

- Assignments to such variables have unspecified semantics
- Prominent: delayed writing at the current PLC cycle's end

Retain Variables

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

⇒ PLCs feature battery-backed memory for retain variables

Example

Retain drill's position in automated processing of workpieces

- Assignments to such variables have unspecified semantics
- Prominent: delayed writing at the current PLC cycle's end

Informatik 11
Embedded Software
RWTH AACHEN UNIVERSITY

Retain Variables

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

$\Rightarrow$ PLCs feature battery-backed memory for retain variables

Example

Retain drill's position in automated processing of workpieces

- Assignments to such variables have unspecified semantics
- Prominent: delayed writing at the current PLC cycle's end

Retain Variables

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

⇒ PLCs feature battery-backed memory for retain variables

Example

Retain drill's position in automated processing of workpieces

- Assignments to such variables have unspecified semantics
- Prominent: delayed writing at the current PLC cycle's end

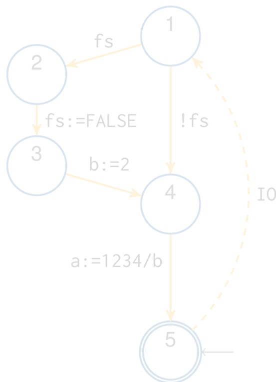
Informatik 11
Embedded Software
RWTH AACHEN UNIVERSITY

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?
- Fixable for delayed writes?

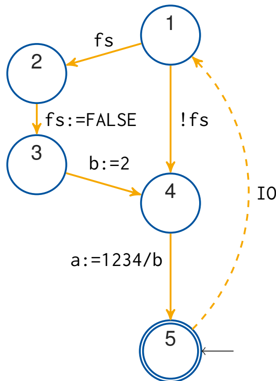


Restart-robustness

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?
- Fixable for delayed writes?



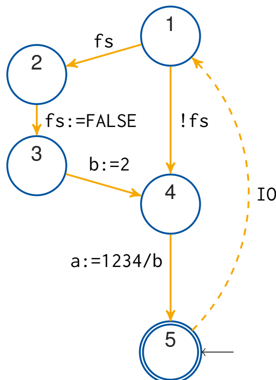
Informatik 11 Embedded Software
RWTH AACHEN UNIVERSITY

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?
- Fixable for delayed writes?

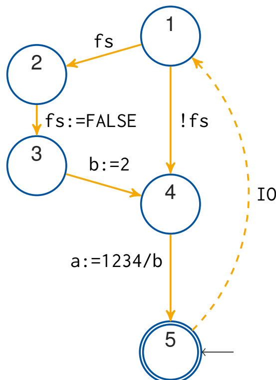


Restart-robustness

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

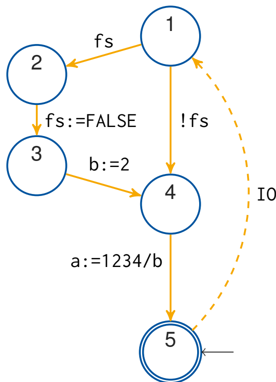
Restart-robustness w.r.t. invariant $a \geq 0$

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



Restart-robustness

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

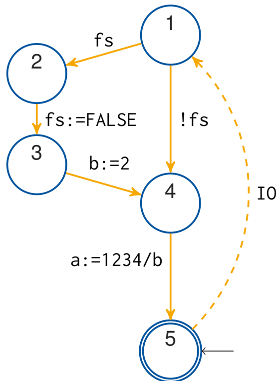


Restart-robustness w.r.t. invariant $a \geq 0$

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

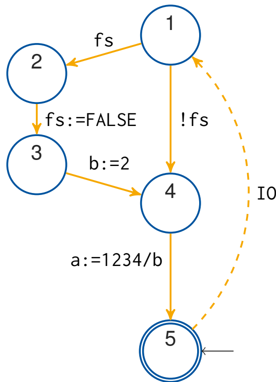
- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant? ✓
- Robust with delayed writes? `a:=1234/0`
- Fixable for delayed writes?

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`
- Fixable for delayed writes?

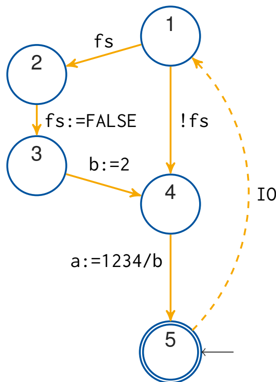


Restart-robustness

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 $a \geq 0$

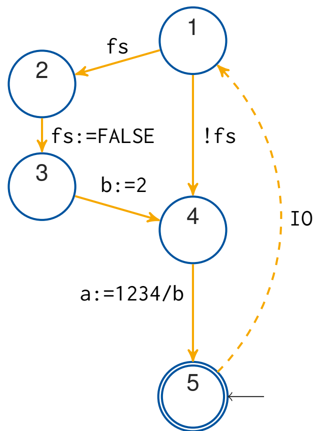
- Initialised with $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$
- The flag `fs` is retained
- Nominal behaviour compliant? ✓
- Robust with delayed writes? `a:=1234/0`
- Fixable for delayed writes? Retain b



Delayed Write Semantics

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

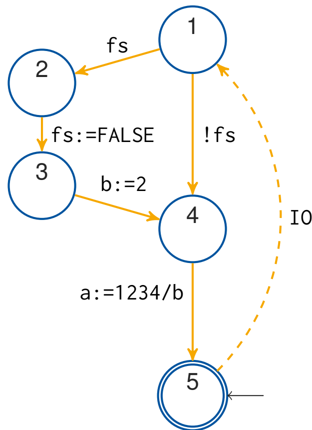
⇒ Model as nondeterministic choice: restart in next cycle?



Delayed Write Semantics

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

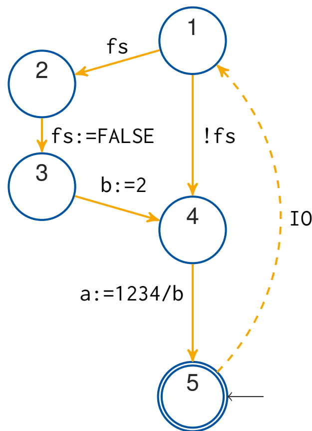
⇒ Model as nondeterministic choice: restart in next cycle?



Delayed Write Semantics

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

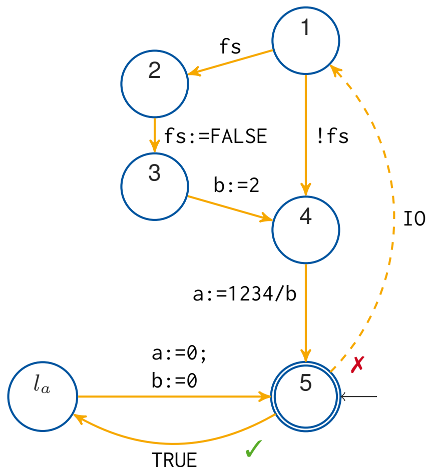
⇒ Model as nondeterministic choice: restart in next cycle?



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?



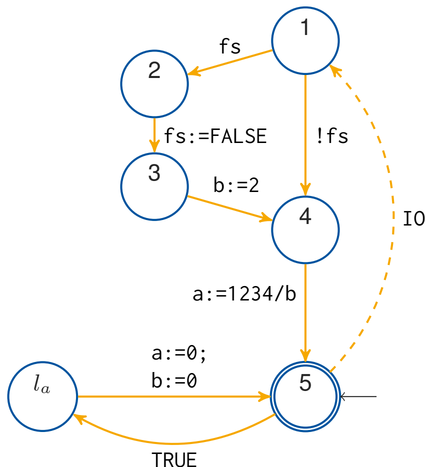
Parameter Synthesis

- Instrumentation enables checking restart-robustness
- Doesn't help with finding safe configuration of retain variables

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

- Synthesis boils down to solving

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \; \dots$$



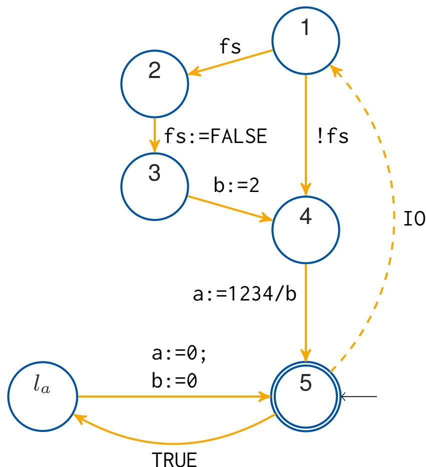
Parameter Synthesis

- Instrumentation enables checking restart-robustness
- Doesn't help with finding safe configuration of retain variables

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

- Synthesis boils down to solving

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \ \ldots$$



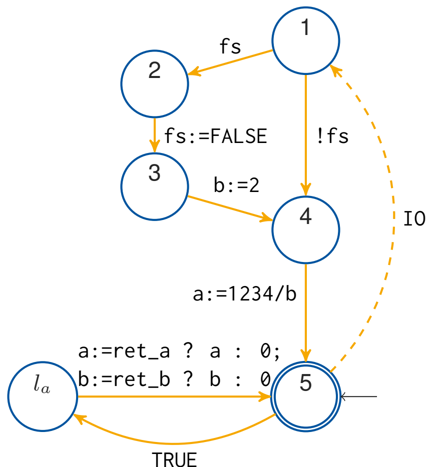
Parameter Synthesis

- Instrumentation enables checking restart-robustness
- Doesn't help with finding safe configuration of retain variables

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

- Synthesis boils down to solving

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \ \ldots$$



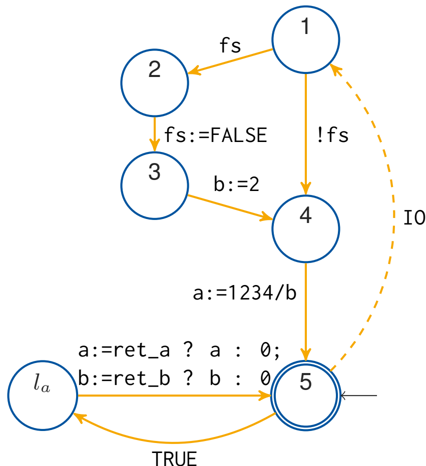
Parameter Synthesis

- Instrumentation enables checking restart-robustness
- Doesn't help with finding safe configuration of retain variables

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

- Synthesis boils down to solving

$$\exists \vec{V}_{par} \forall \vec{V} \setminus \vec{V}_{par} \; \ldots$$



Approach

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 (CEGAR)

Approach

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 (CEGAR)

Approach

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 (CEGAR)

Approach

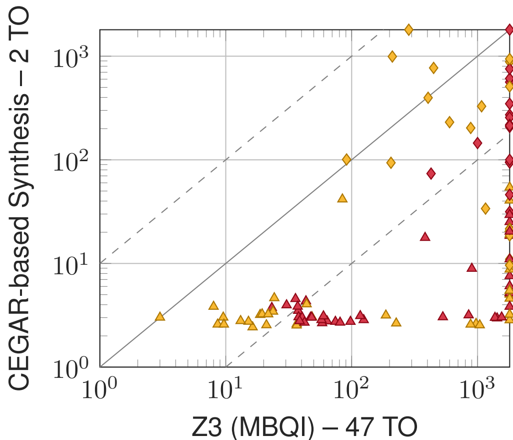
Observations:

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

Idea:

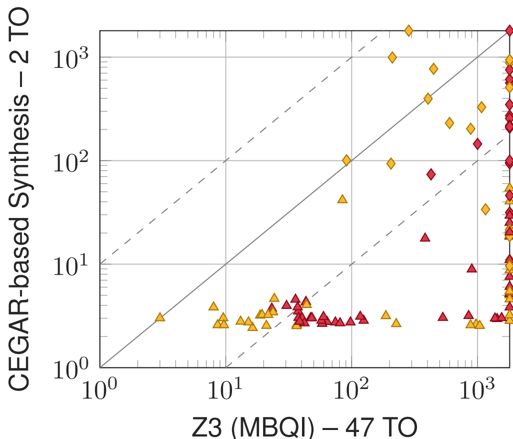
- Manage choice of parameters and reuse efficient procedures for reasoning about restart-robustness for fixed parameters
- Over-approximate set of “safe” parameters and refine it while counterexamples exist (CEGAR)

Experiments – Synthesis Runtime [s]



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

Experiments – Synthesis Runtime [s]



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

Algorithm 1: SynthRetainConf(P, φ)

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

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

1. $\mathcal{H} \leftarrow \text{toHorn}(P)$ // Represent program as ∀CHCs
2. $(\vec{V}, I, T) \leftarrow \text{toSymTS}(P)$ // and as symbolic transition system
3. $safe(\vec{X}_{\text{par}}) \leftarrow true$ // All parameters are assumed to be safe
4. **while** $\neg\text{sat}\ (\mathcal{H} \cup \{\varphi(\vec{X}) \leftarrow p_{EoC}(\vec{X} \uplus \vec{X}_{par}), safe(\vec{X}_{par})\})$ **do** // ∃ violating run?
5. $\quad k \leftarrow$ length of violating run
6. $\quad c_{\text{par}} \leftarrow$ cube of chosen (Boolean) parameter values in violating run
7. $\quad$ **foreach** lit *in* c_{par} **do**
8. $\quad\quad \bar{c}_{\text{par}} \leftarrow c_{\text{par}}$ with negated lit // Flip literal
9. $\quad\quad$ **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?
10. $\quad\quad\quad c_{\text{par}} \leftarrow c_{\text{par}} \setminus lit$ // Drop literal
11. $\quad safe(\vec{X}_{\text{par}}) \leftarrow safe(\vec{X}_{\text{par}}) \wedge \neg c_{\text{par}}$ // Block unsafe parameters
12. **return** $safe(\vec{X}_{\text{par}})$ // (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.

[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.

References II

[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.