Counterexample-Guided Abstraction Refinement for PLCs

Sebastian Biallas, Jörg Brauer & Stefan Kowalewski

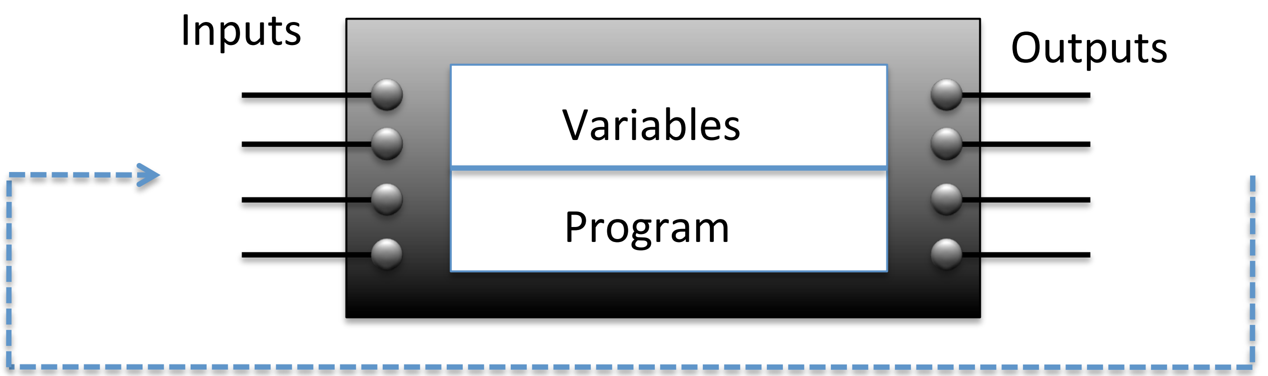
RWTH Aachen University
{lastname}@embedded.rwth-aachen.de

SSV 2010-10-06

Overview

- Introduction & Motivation
  - PLCs
  - Model checking PLC programs
- Abstract simulation with refinement
  - Use constraint solving for refinements
  - Different refinement step
    - local variables
    - global variables
- Case studies & implementation
- Conclusion & future work

Programmable Logic Controllers



- Used in the automation industry
- Controlling many safety-critical systems
- Operating in cyclic scanning mode (sensing inputs, processing, writing outputs)
- No non-determinism during cycle
- Different programming languages, here *instruction list*

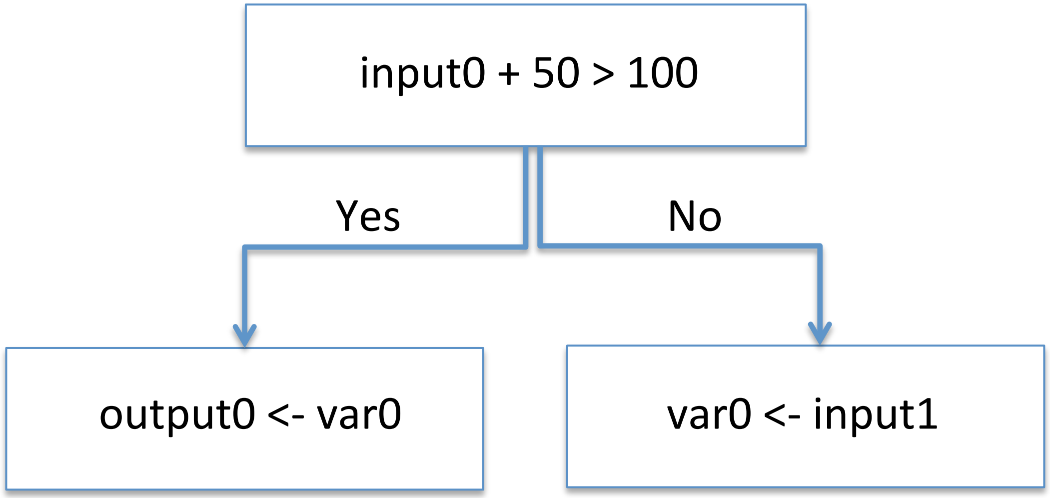
Example IL Program

input0, input1	*INPUT*
output0	*OUTPUT*
var0	*GLOBAL*
Type BYTE	0..255

```
      LD    input0
      ADD   50
      GT    100
      JMPC  Label

      LD    input1
      ST    var0
      RET

Label:
      LD    var0
      ST    output0
      RET
```



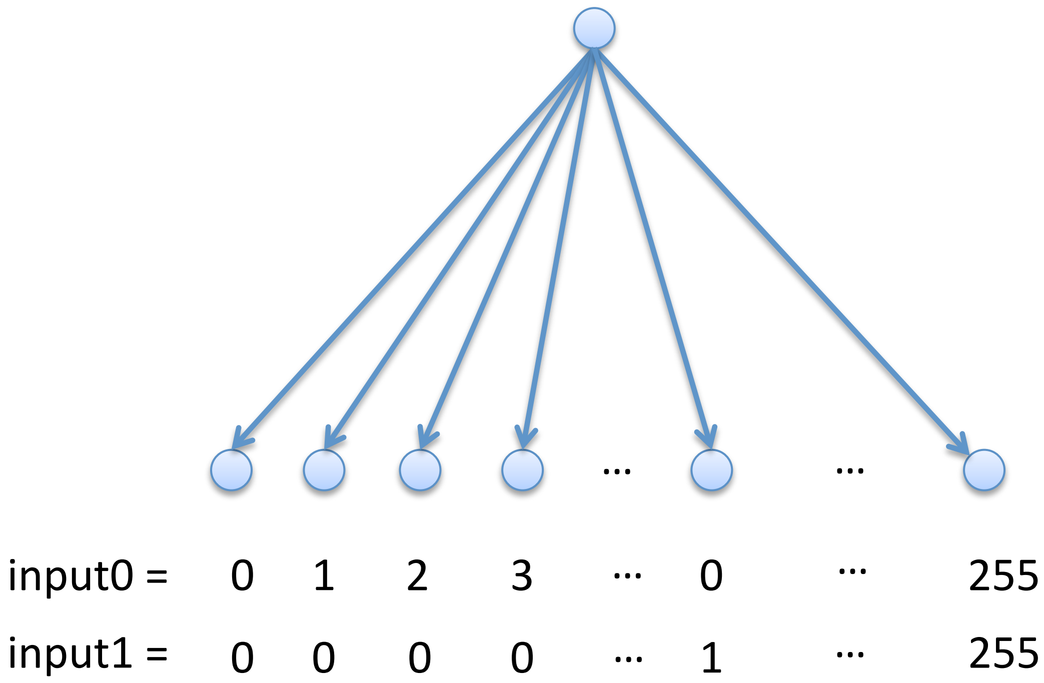
Building the State Space

input0, input1	*INPUT*
output0	*OUTPUT*
var0	*GLOBAL*
Type BYTE	0..255

```
        LD      input0
        ADD     50
        GT      100
        JMPC    Label

        LD      input1
        ST      var0
        RET

Label:
        LD      var0
        ST      output0
        RET
```



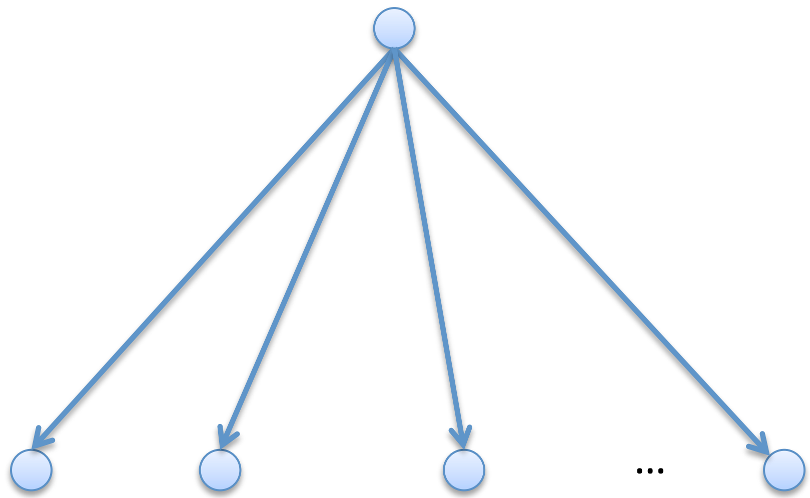
Building the Abstract State Space

input0, input1	*INPUT*
output0	*OUTPUT*
var0	*GLOBAL*
Type BYTE	0..255

```
        LD      input0
        ADD     50
        GT      100
        JMPC    Label

        LD      input1
        ST      var0
        RET

Label:
        LD      var0
        ST      output0
        RET
```



input0 =	[0, 49]	[50, 255]	[50, 255]	…	[50, 255]
input1 =	[0, 255]	[0, 0]	[1, 1]	…	[255, 255]

Abstract Domains

- Intervals
  - [1, 50] + [2, 3] = [3, 53]
- Bit sets
  - Each bit is 0, 1 or ⊥
  - 010⊥⊥1 & *0*10010 = 0100⊥0
- We use the *reduced product* of intervals and bit sets

Example (cont.)

Program	Accumulator
LD input0	[0, 255]
ADD 50	[50, 305]
GT 100	[0, 1]
JMPC Label	
...	

- Let's start with input0 = [0, 255]
- Condition jump (JMPC) demands a concrete value in accumulator
- This poses a constraint on the abstract value in the accumulator
- Intuitively: Restart cycle with abstract values [0, 49] and [50, 255] for input0 to obey constraint

Constraints on Abstract Values

- $cs_f(v) :\Leftrightarrow$ Abstract value v is *consistent* under condition f

  - $cs_{>50}([0, 255])$ is *false*
  - $cs_{>50}([51, 101])$, $cs_{>50}([3, 7])$ are *true*

- $cs_{sing}(v) :\Leftrightarrow$ v represents a single value

- Idea:
  - Extend constraints to expressions
  - To guard conditional jumps, etc
  - Next: Formal model for IL programs

SSA Form

Program	SSA form
LD input0 ADD 50 GT 100 JMPCLabel ...	$acc^{(0)} := input_0^{(0)}$ $acc^{(1)} := acc^{(0)} + 50$ $acc^{(2)} := acc^{(1)} > 100$ $guard(cs_{sing}(acc^{(2)}))$ ...

- If $cs_{sing}(acc^{(2)})$ is not fulfilled, $input0^{(0)}$ should be split
- Next step: Transform $cs_{sing}(acc^{(2)})$ into a constraint on $input0^{(0)}$

Transforming Constraints

- $cs_{f1}(e_1) \vdash cs_{f2}(e_2) :\Leftrightarrow cs_{f2}(e_2)$ implies the consistency of $cs_{f2}(e_2)$
- E.g. $cs_{>50}(a + 5) \vdash cs_{>45}(a)$

```
acc⁰ := input₀⁰
acc¹ := acc⁰ + 50
acc² := acc¹ > 100
guard(cs_sing(acc²))
...
```

$cs_{sing}(acc^2)$

Constraint Guards

- Constraint guards are needed
  - for deterministic control flow
  - for some hardware function blocks (e.g. timers) that require concrete values
  - to guarantee that the atomic propositions of the model-checker have a consistent truth value
- If those constraints are not fulfilled they are transform into constraints on variables

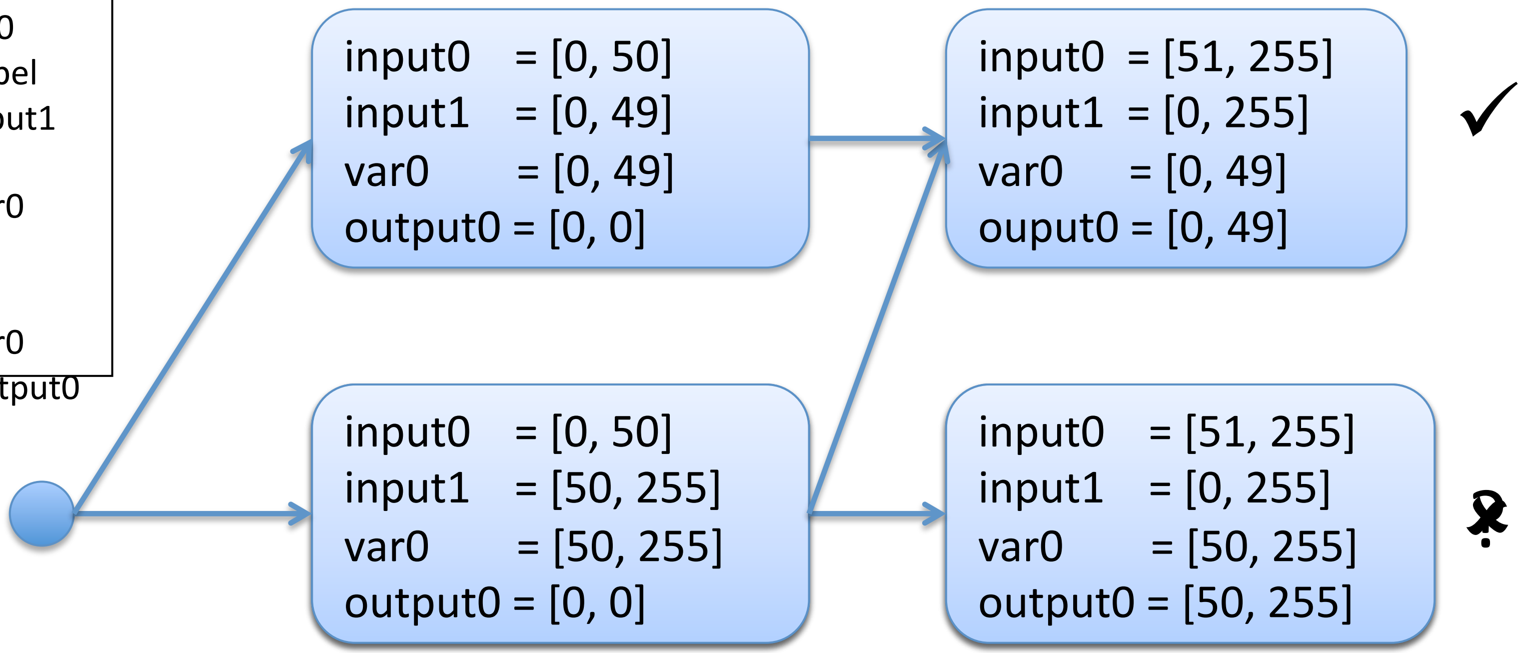
Refinements of Local Variables

- Refinement loop: Begin with $\bot$ for all inputs
- Transform constraints to constraints on inputs
- Refine inputs and restart cycle
- Each restart refines an abstract value, so the refinement process terminates
- Protect all global variables with single value constraints (no non-determinism in the state space)

Refinements of Global Variables

```
        LD      input0
        ADD     50
        GT      100
        JMPC    Label
        LD      input1

        ST      var0
        RET
Label:
        LD      var0
        ST      output0
```



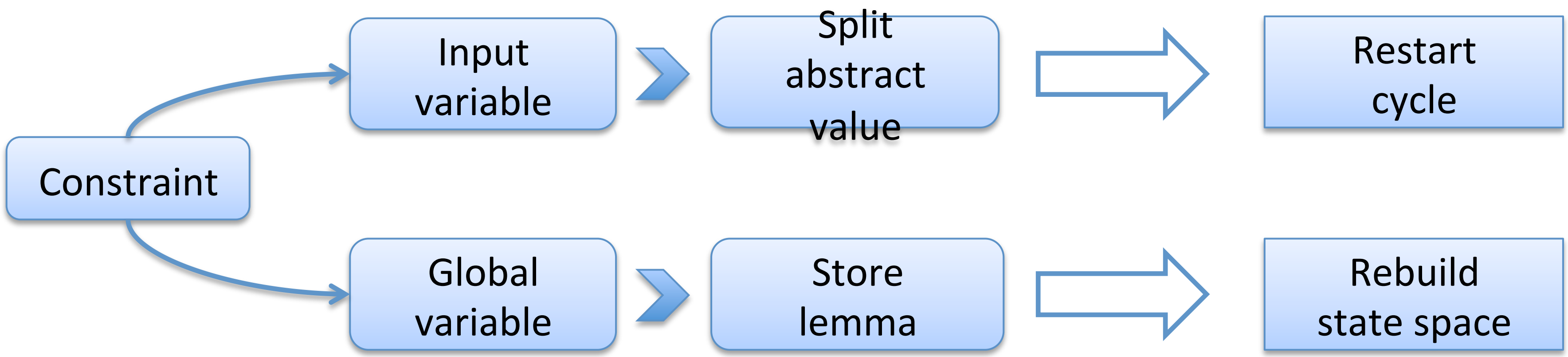
Lemma: $cs_{< 50}$(var0)

AG output0 < 50

Refinements of Global Variables

- Storing abstract values in states possibly allows new behavior
  - A valid ACTL formula is also valid in the concrete state space
  - For an invalid ACTL formula, we have to check whether we found a real counterexample
  - This is achieved by rebuilding the state space using the lemmas as refinements

Overview

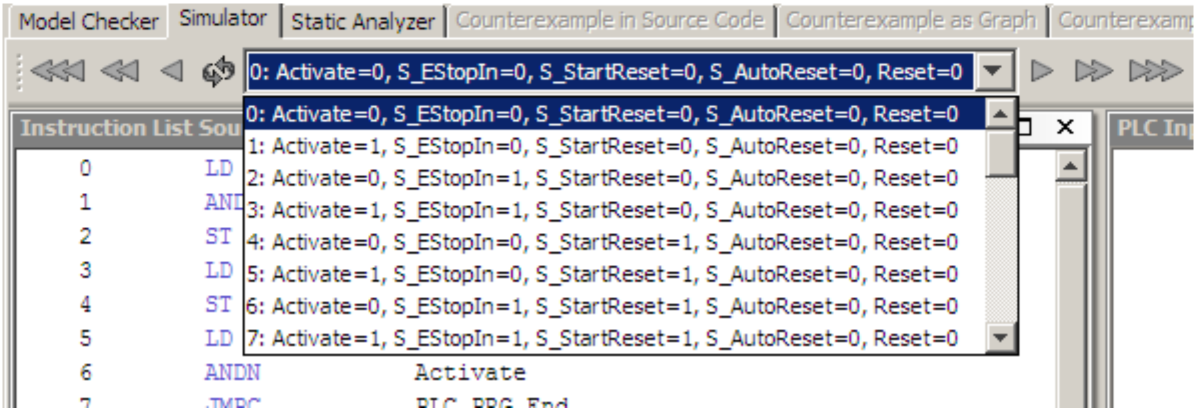

Constraint
Input variable
Split abstract value
Restart cycle
Global variable
Store lemma
Rebuild state space

Case Studies

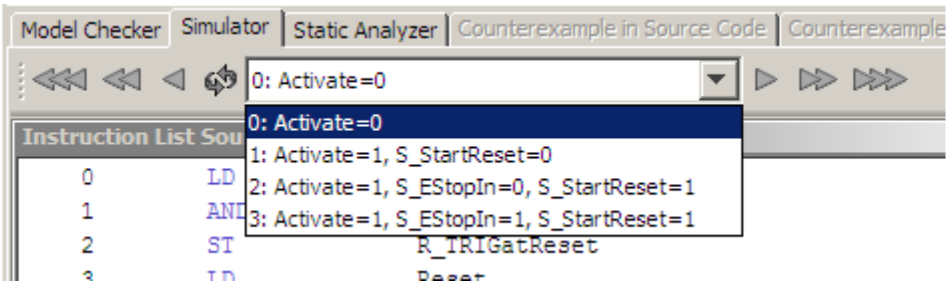
Abstraction technique	# stored states	# created states	State space size [MB]	Time [s]
Without	780 172	199 724 033	1 704	5 633
Only inputs	132 242	3 155 467	351	326
All variables	75 203	1 098 220	163	99

- Function block for monitoring a guard lock (PLCopen)
- 8 Boolean inputs and 5 outputs
- We used an implementation with 300 lines of IL code and 16 internal variables

Implementation in [mc]square



vs.



Conclusion & Future Work

- Conclusion
  - Abstraction refinement for PLC programs
  - Based on constraint resolving
  - Different refinement loop for local and global variables
- Future Work
  - Better constraint resolving using SMT, SAT solvers
  - Incremental rebuild of state space
  - Relational domains constructions