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

<table>
<thead>
<tr>
<th>input0, input1</th>
<th>INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>output0</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>var0</td>
<td>GLOBAL</td>
</tr>
<tr>
<td>Type BYTE</td>
<td>0..255</td>
</tr>
</tbody>
</table>

LD input0
ADD 50
GT 100
JMPC Label

LD input1
ST var0
RET

Label:
LD var0
ST output0
RET

input0 + 50 > 100

Yes
output0 <- var0

No
var0 <- input1

© 2010 Sebastian Biallas, Embedded Software Laboratory, RWTH Aachen University
Building the State Space

<table>
<thead>
<tr>
<th>input0, input1</th>
<th>INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>output0</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>var0</td>
<td>GLOBAL</td>
</tr>
<tr>
<td>Type BYTE</td>
<td>0..255</td>
</tr>
</tbody>
</table>

LD input0
ADD 50
GT 100
JMPC Label

LD input1
ST var0
RET

Label:
LD var0
ST output0
RET

input0 = 0 1 2 3 ... 0 ... 255
input1 = 0 0 0 0 ... 1 ... 255
Building the Abstract State Space

<table>
<thead>
<tr>
<th>input0, input1</th>
<th>INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>output0</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>var0</td>
<td>GLOBAL</td>
</tr>
<tr>
<td>Type BYTE</td>
<td>0..255</td>
</tr>
</tbody>
</table>

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 \(\bot\)
    – \(010 \bot \bot 1\) \& \(010010 = 0100 \bot 0\)

• We use the *reduced product* of intervals and bit sets
Example (cont.)

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD input0</td>
<td>[0, 255]</td>
</tr>
<tr>
<td>ADD 50</td>
<td>[50, 305]</td>
</tr>
<tr>
<td>GT 100</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>JMPC Label</td>
<td></td>
</tr>
</tbody>
</table>

[LD input0]
[ADD 50]
[GT 100]
[JMPC Label]
Constraints on Abstract Values

- $\text{cs}_f(v) : \iff \text{Abstract value } v \text{ is } \textit{consistent} \text{ under condition } f$
  - $\text{cs}_{>50}([0, 255])$ is $\textit{false}$
  - $\text{cs}_{>50}([51, 101]), \text{cs}_{>50}([3, 7])$ are $\textit{true}$

- $\text{cs}_{\text{sing}}(v) : \iff v \text{ represents a single value}$

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

© 2010 Sebastian Biallas, Embedded Software Laboratory, RWTH Aachen University
SSA Form

Program

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>input0</td>
</tr>
<tr>
<td>ADD</td>
<td>50</td>
</tr>
<tr>
<td>GT</td>
<td>100</td>
</tr>
<tr>
<td>JMPCLabel</td>
<td>...</td>
</tr>
</tbody>
</table>

SSA form

| acc(0) := input0(0) |
| acc(1) := acc(0) + 50 |
| acc(2) := acc(1) > 100 |
| guard(cs \text{sing}(acc(2)) |
| ... |

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

- $c_{s_{f_1}}(e_1) \vdash c_{s_{f_2}}(e_2) : \iff c_{s_{f_2}}(e_2)$ implies the consistency of $c_{s_{f_2}}(e_2)$
- E.g. $c_{>50}(a + 5) \vdash c_{>45}(a)$

\begin{align*}
acc^0 & := \text{input}_0^0 \\
acc^1 & := acc^0 + 50 \\
acc^2 & := acc^1 > 100 \\
guard(c_{s_{sing}}(acc^2)) & \\
... &
\end{align*}

$c_{s_{sing}}(acc^2)$
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 ⊥ 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)
Lemma: $cs_{<50}(\text{var0})$

$AG\ \text{output0} < 50$
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
constraint

- Input variable
- Global variable

- Split abstract value
- Store lemma

- Restart cycle
- Rebuild state space
### Case Studies

<table>
<thead>
<tr>
<th>Abstraction technique</th>
<th># stored states</th>
<th># created states</th>
<th>State space size [MB]</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without</td>
<td>780 172</td>
<td>199 724 033</td>
<td>1704</td>
<td>5 633</td>
</tr>
<tr>
<td>Only inputs</td>
<td>132 242</td>
<td>3 155 467</td>
<td>351</td>
<td>326</td>
</tr>
<tr>
<td>All variables</td>
<td>75 203</td>
<td>1 098 220</td>
<td>163</td>
<td>99</td>
</tr>
</tbody>
</table>

- 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