Cycle-Bounded Model Checking of PLC Software via Dynamic LBE

Dimitri Bohlender | Daniel Hamm | Stefan Kowalewski

SAC SVT 2018, April 12, 2018

Informatik 11
Embedded Software
RWTH AACHEN UNIVERSITY

Outline

Introduction

- Model Checking PLC Software
- Motivation

Dynamic LBE-based Cycle-BMC

- Concept
- Experiments

Conclusion

Outline

Introduction

- Model Checking PLC Software
- Motivation

Dynamic LBE-based Cycle-BMC

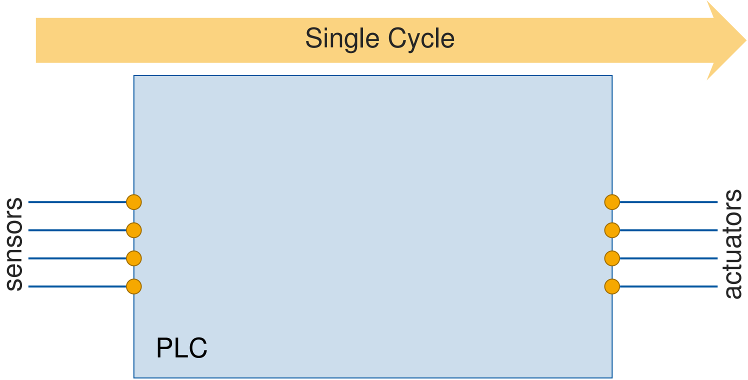
- Concept
- Experiments

Conclusion

Outline

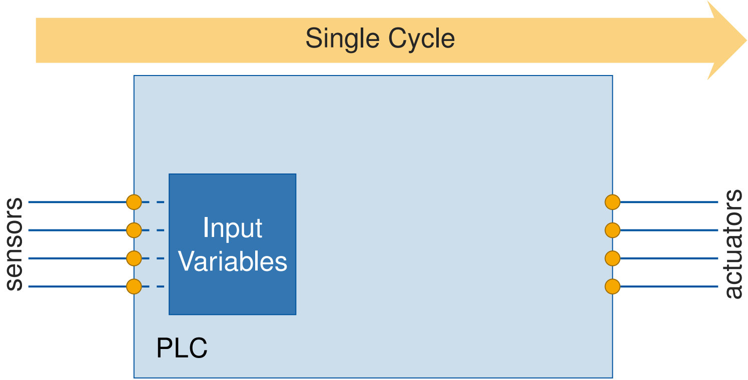
Programmable Logic Controllers (PLCs)

- 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



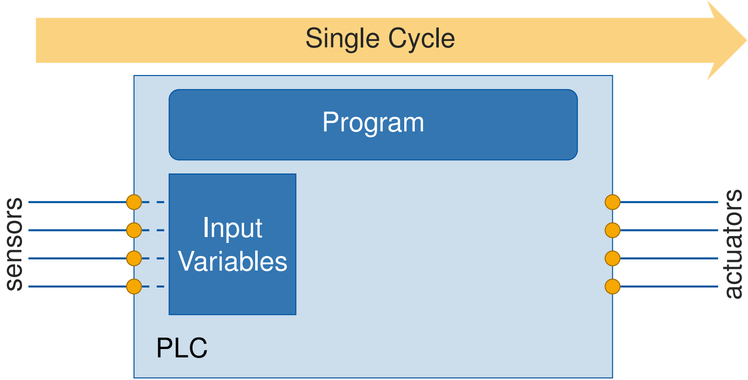
Programmable Logic Controllers (PLCs)

- 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



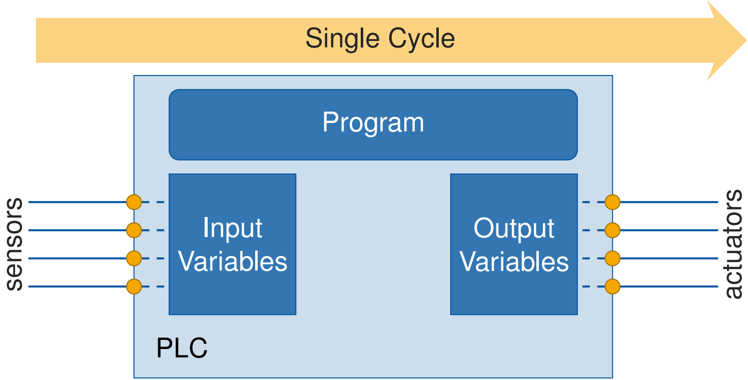
Programmable Logic Controllers (PLCs)

- 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



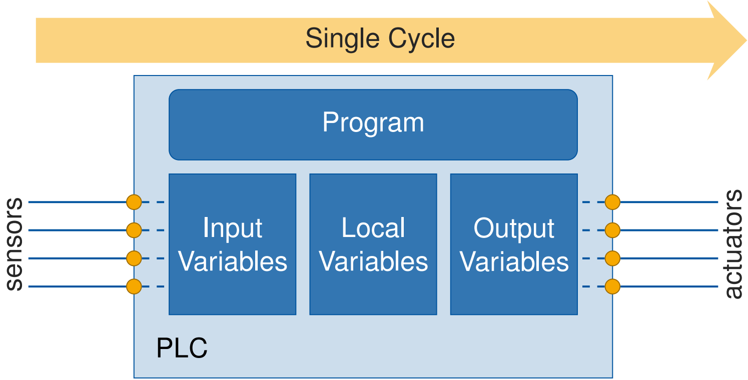
Programmable Logic Controllers (PLCs)

- 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

- Programmed via textual and graphical languages from the IEC 61131-3
- Feature no recursion

⇒ Formalised as Control Flow Automaton

- Program may be unstructured
- Program semantics may translate to guards with intersecting conditions

PLC Software

- Programmed via textual and graphical languages from the IEC 61131-3
- Feature no recursion

⇒ Formalised as Control Flow Automaton

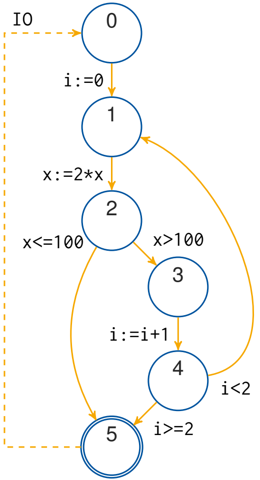
- Program may be unstructured
- Program semantics may translate to guards with intersecting conditions

PLC Software

- Programmed via textual and graphical languages from the IEC 61131-3
- Feature no recursion

⇒ Formalised as Control Flow Automaton

- Program may be unstructured
- Program semantics may translate to guards with intersecting conditions

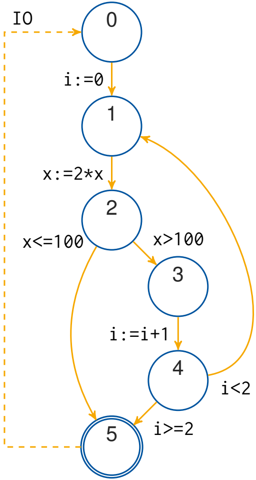


PLC Software

- Programmed via textual and graphical languages from the IEC 61131-3
- Feature no recursion

⇒ Formalised as Control Flow Automaton

- Program may be unstructured
- Program semantics may translate to guards with intersecting conditions

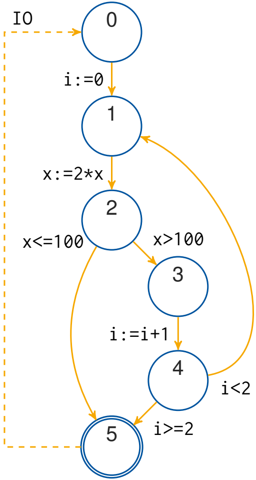


PLC Software

- ▸ Programmed via textual and graphical languages from the IEC 61131-3
- ▸ Feature no recursion

⇒ Formalised as Control Flow Automaton

- ▸ Program may be unstructured
- ▸ Program semantics may translate to guards with intersecting conditions

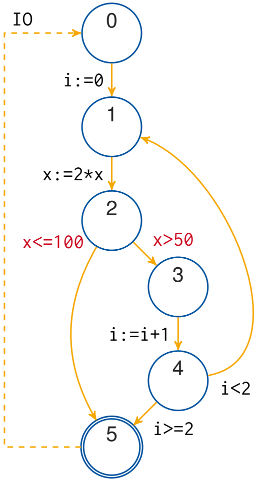


PLC Software

- Programmed via textual and graphical languages from the IEC 61131-3
- Feature no recursion

⇒ Formalised as Control Flow Automaton

- Program may be unstructured
- Program semantics may translate to guards with intersecting conditions



Specifications

- Intermediate states are not observable

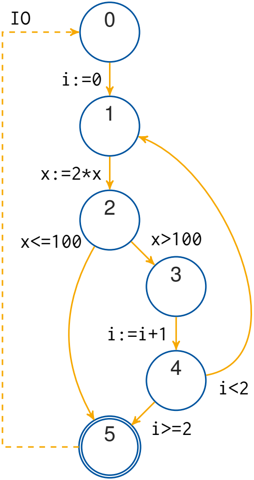
⇒ Automation engineers and specs always refer to the observable state

- Most specifications can be formalised via (bounded) temporal logics, e.g. LTL:

$$\Box(\texttt{reset} \rightarrow \bigcirc(\texttt{mode} = 0 \vee \bigcirc\,\texttt{mode} = 0))$$

- Verifier must operate on cycle-step semantics or specs must be adapted:

$$\Box(\texttt{pc} = 5 \wedge \texttt{reset} \rightarrow \bigcirc$$
$$(\texttt{pc} \neq 5 \,\mathsf{U}(\texttt{pc} = 5 \wedge (\texttt{mode} = 0 \vee \bigcirc(\texttt{pc} \neq 5 \,\mathsf{U}(\texttt{pc} = 5 \wedge \texttt{mode} = 0))))))$$



Specifications

- Intermediate states are not observable

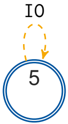
⇒ Automation engineers and specs always refer to the observable state

- Most specifications can be formalised via (bounded) temporal logics, e.g. LTL:

$$\Box(\mathtt{reset} \rightarrow \bigcirc(\mathtt{mode} = 0 \vee \bigcirc\,\mathtt{mode} = 0))$$

- Verifier must operate on cycle-step semantics or specs must be adapted:

$$\Box(\mathtt{pc} = 5 \wedge \mathtt{reset} \rightarrow \bigcirc (\mathtt{pc} \neq 5\, \mathrm{U}(\mathtt{pc} = 5 \wedge (\mathtt{mode} = 0 \vee \bigcirc(\mathtt{pc} \neq 5\, \mathrm{U}(\mathtt{pc} = 5 \wedge \mathtt{mode} = 0))))))$$



Specifications

- Intermediate states are not observable

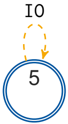
⇒ Automation engineers and specs always refer to the observable state

- Most specifications can be formalised via (bounded) temporal logics, e.g. LTL:

$$\Box(\mathtt{reset} \rightarrow \bigcirc(\mathtt{mode} = 0 \vee \bigcirc \mathtt{mode} = 0))$$

- Verifier must operate on cycle-step semantics or specs must be adapted:

$$\Box(\mathtt{pc} = 5 \wedge \mathtt{reset} \rightarrow \bigcirc$$
$$(\mathtt{pc} \neq 5 \,\mathrm{U}(\mathtt{pc} = 5 \wedge (\mathtt{mode} = 0 \vee \bigcirc(\mathtt{pc} \neq 5 \,\mathrm{U}(\mathtt{pc} = 5 \wedge \mathtt{mode} = 0))))))$$



Specifications

- Intermediate states are not observable

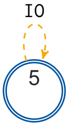
⇒ Automation engineers and specs always refer to the observable state

- Most specifications can be formalised via (bounded) temporal logics, e.g. LTL:

$$\Box(\mathtt{reset} \rightarrow \bigcirc(\mathtt{mode} = 0 \vee \bigcirc\,\mathtt{mode} = 0))$$

- Verifier must operate on cycle-step semantics or specs must be adapted:

$$\Box(\mathtt{pc} = 5 \wedge \mathtt{reset} \rightarrow \bigcirc$$
$$(\mathtt{pc} \neq 5\,\mathsf{U}(\mathtt{pc} = 5 \wedge (\mathtt{mode} = 0 \vee \bigcirc(\mathtt{pc} \neq 5\,\mathsf{U}(\mathtt{pc} = 5 \wedge \mathtt{mode} = 0))))))$$



Specifications

- Intermediate states are not observable

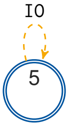
⇒ Automation engineers and specs always refer to the observable state

- Most specifications can be formalised via (bounded) temporal logics, e.g. LTL:

$$\square(\mathtt{reset} \rightarrow \bigcirc(\mathtt{mode} = 0 \vee \bigcirc\, \mathtt{mode} = 0))$$

- Verifier must operate on cycle-step semantics or specs must be adapted:

$$\square(\mathtt{pc} = 5 \wedge \mathtt{reset} \rightarrow \bigcirc (\mathtt{pc} \neq 5\, \mathsf{U}(\mathtt{pc} = 5 \wedge (\mathtt{mode} = 0 \vee \bigcirc(\mathtt{pc} \neq 5\, \mathsf{U}(\mathtt{pc} = 5 \wedge \mathtt{mode} = 0))))))$$



Bounded Model Checking (BMC)

Checks reachability of bad states within k steps from I

$$I(\vec{x}_0) \wedge \bigwedge_{0 \leq i < k} T(\vec{x}_i, \vec{x}_{i+1}) \wedge \bigvee_{0 \leq i \leq k} B(\vec{x}_i)$$

where T characterises the program's step semantics

- Basis of unbounded techniques, e.g. k-Induction
- Unbounded verification might be too hard
- Often used to complement testing, e.g.

  "The first 100 execution cycles are safe"

  helps an engineer with estimating safety

Note: *"The first 100 instructions are safe"* does not

Bounded Model Checking (BMC)

Checks reachability of bad states within k steps from I

$$I(\vec{x}_0) \wedge \bigwedge_{0 \leq i < k} T(\vec{x}_i, \vec{x}_{i+1}) \wedge \bigvee_{0 \leq i \leq k} B(\vec{x}_i)$$

where T characterises the program's step semantics

- Basis of unbounded techniques, e.g. k-Induction
- Unbounded verification might be too hard
- Often used to complement testing, e.g.

  "The first 100 execution cycles are safe"

  helps an engineer with estimating safety

Note: *"The first 100 instructions are safe"* does not

Bounded Model Checking (BMC)

Checks reachability of bad states within k steps from I

$$I(\vec{x}_0) \wedge \bigwedge_{0 \leq i < k} T(\vec{x}_i, \vec{x}_{i+1}) \wedge \bigvee_{0 \leq i \leq k} B(\vec{x}_i)$$

where T characterises the program's step semantics

- Basis of unbounded techniques, e.g. k-Induction
- Unbounded verification might be too hard
- Often used to complement testing, e.g.

  "The first 100 execution cycles are safe"

  helps an engineer with estimating safety

Note: *"The first 100 instructions are safe"* does not

Bounded Model Checking (BMC)

Checks reachability of bad states within k steps from I

$$I(\vec{x}_0) \wedge \bigwedge_{0 \leq i < k} T(\vec{x}_i, \vec{x}_{i+1}) \wedge \bigvee_{0 \leq i \leq k} B(\vec{x}_i)$$

where T characterises the program's step semantics

- Basis of unbounded techniques, e.g. k-Induction
- Unbounded verification might be too hard
- Often used to complement testing, e.g.

  "The first 100 execution cycles are safe"

  helps an engineer with estimating safety

Note: *"The first 100 instructions are safe"* does not

Bounded Model Checking (BMC)

Checks reachability of bad states within k steps from I

$$I(\vec{x}_0) \wedge \bigwedge_{0 \leq i < k} T(\vec{x}_i, \vec{x}_{i+1}) \wedge \bigvee_{0 \leq i \leq k} B(\vec{x}_i)$$

where T characterises the program's step semantics

- Basis of unbounded techniques, e.g. k-Induction
- Unbounded verification might be too hard
- Often used to complement testing, e.g.

  "The first 100 execution cycles are safe"

  helps an engineer with estimating safety

Note: *"The first 100 instructions are safe"* does not

Bounded Model Checking (BMC)

Checks reachability of bad states within k steps from I

$$I(\vec{x}_0) \wedge \bigwedge_{0 \leq i < k} T(\vec{x}_i, \vec{x}_{i+1}) \wedge \bigvee_{0 \leq i \leq k} B(\vec{x}_i)$$

where T characterises the program's step semantics

- Basis of unbounded techniques, e.g. k-Induction
- Unbounded verification might be too hard
- Often used to complement testing, e.g.

  "The first 100 execution cycles are safe"

  helps an engineer with estimating safety

Note: *"The first 100 instructions are safe"* does not

Bounded Model Checking (BMC)

Checks reachability of bad states within k steps from I

$$I(\vec{x}_0) \wedge \bigwedge_{0 \leq i < k} T(\vec{x}_i, \vec{x}_{i+1}) \wedge \bigvee_{0 \leq i \leq k} B(\vec{x}_i)$$

where T characterises the program's step semantics

- Basis of unbounded techniques, e.g. k-Induction
- Unbounded verification might be too hard
- Often used to complement testing, e.g.

  "The first 100 execution cycles are safe"

  helps an engineer with estimating safety

Note: *"The first 100 instructions are safe"* does not

Bounded Model Checking (BMC)

Checks reachability of bad states within k steps from I

$$I(\vec{x}_0) \wedge \bigwedge_{0 \leq i < k} T(\vec{x}_i, \vec{x}_{i+1}) \wedge \bigvee_{0 \leq i \leq k} B(\vec{x}_i)$$

where T characterises the program's step semantics

- Basis of unbounded techniques, e.g. k-Induction
- Unbounded verification might be too hard
- Often used to complement testing, e.g.

  "The first 100 execution cycles are safe"

  helps an engineer with estimating safety

Note: *"The first 100 instructions are safe"* does not

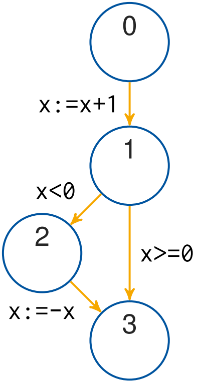
Large-Block Encoding (LBE)

- Combines the semantics of loop-free fragments of a CFA statically

⇒ Less but more complex transitions

- Exponential formula growth of original formulation is avoidable:

$$T(x, x') := \exists_{x_1}\ x_1 = x + 1 \\ \wedge\ (x_1 < 0 \rightarrow x' = -x_1 \\ \vee\ \ x_1 \geq 0 \rightarrow x' = x)$$



- Doesn't help with loops or unreachable code in cycle
- Merge via disjunction (incremental SAT needs $\wedge$ at top level)

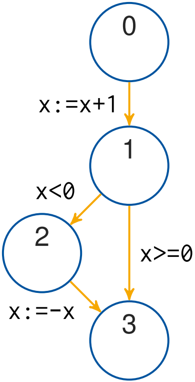
Large-Block Encoding (LBE)

- Combines the semantics of loop-free fragments of a CFA statically

⇒ Less but more complex transitions

- Exponential formula growth of original formulation is avoidable:

$$
\begin{aligned}
T(x, x') := \exists_{x_1}\ & x_1 = x + 1 \\
& \wedge (x_1 < 0 \rightarrow x' = -x_1 \\
& \vee \ \ x_1 \geq 0 \rightarrow x' = x)
\end{aligned}
$$



- Doesn't help with loops or unreachable code in cycle
- Merge via disjunction (incremental SAT needs ∧ at top level)

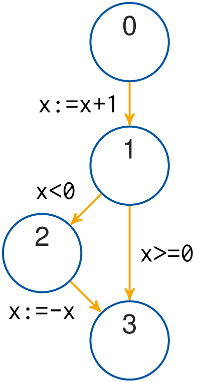
Large-Block Encoding (LBE)

- Combines the semantics of loop-free fragments of a CFA statically

$\Rightarrow$ Less but more complex transitions

- Exponential formula growth of original formulation is avoidable:

$$
\begin{aligned}
T(x, x') := \exists_{x_1}\ x_1 = x + 1 \\
\wedge\ (x_1 < 0 \rightarrow x' = -x_1 \\
\vee\ \ x_1 \geq 0 \rightarrow x' = x)
\end{aligned}
$$



- Doesn't help with loops or unreachable code in cycle
- Merge via disjunction (incremental SAT needs $\wedge$ at top level)

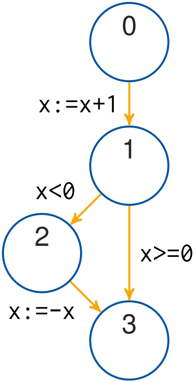
Large-Block Encoding (LBE)

- Combines the semantics of loop-free fragments of a CFA statically

⇒ Less but more complex transitions

- Exponential formula growth of original formulation is avoidable:

$$\begin{aligned} T(x, x') := \exists_{x_1}\ & x_1 = x + 1 \\ & \wedge (x_1 < 0 \rightarrow x' = -x_1 \\ & \vee \ \ x_1 \geq 0 \rightarrow x' = x) \end{aligned}$$



- Doesn't help with loops or unreachable code in cycle
- Merge via disjunction (incremental SAT needs $\wedge$ at top level)

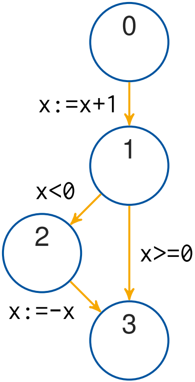
Large-Block Encoding (LBE)

- Combines the semantics of loop-free fragments of a CFA statically

$\Rightarrow$ Less but more complex transitions

- Exponential formula growth of original formulation is avoidable:

$$T(x,x') := \exists_{x_1}\ x_1 = x+1 \\ \wedge (x_1 < 0 \rightarrow x' = -x_1 \\ \vee\ x_1 \geq 0 \rightarrow x' = x)$$



- Doesn't help with loops or unreachable code in cycle
- Merge via disjunction (incremental SAT needs $\wedge$ at top level)

Dynamic LBE

- Combines the semantics of all feasible paths dynamically
- Efficiency based on combination of several traits

Dynamic	Symbolic execution (with merging)
$\Rightarrow$	Loop unrolling & no unreachable code
Compact	LBE-style with auxiliary variables for blocks
$\Rightarrow$	Path(s) selection & no exponential growth
Incremental	Uses one monotonically growing solver instance
$\Rightarrow$	Incremental SAT solving under assumptions

Dynamic LBE

- Combines the semantics of all feasible paths dynamically
- Efficiency based on combination of several traits

Dynamic	Symbolic execution (with merging)
	$\Rightarrow$ Loop unrolling & no unreachable code
Compact	LBE-style with auxiliary variables for blocks
	$\Rightarrow$ Path(s) selection & no exponential growth
Incremental	Uses one monotonically growing solver instance
	$\Rightarrow$ Incremental SAT solving under assumptions

Dynamic LBE

- Combines the semantics of all feasible paths dynamically
- Efficiency based on combination of several traits

Dynamic	Symbolic execution (with merging)
	$\Rightarrow$ Loop unrolling & no unreachable code
Compact	LBE-style with auxiliary variables for blocks
	$\Rightarrow$ Path(s) selection & no exponential growth
Incremental	Uses one monotonically growing solver instance
	$\Rightarrow$ Incremental SAT solving under assumptions

Dynamic LBE

- Combines the semantics of all feasible paths dynamically
- Efficiency based on combination of several traits

Dynamic	Symbolic execution (with merging)
	$\Rightarrow$ Loop unrolling & no unreachable code
Compact	LBE-style with auxiliary variables for blocks
	$\Rightarrow$ Path(s) selection & no exponential growth
Incremental	Uses one monotonically growing solver instance
	$\Rightarrow$ Incremental SAT solving under assumptions

Dynamic LBE

- Combines the semantics of all feasible paths dynamically
- Efficiency based on combination of several traits

Dynamic	Symbolic execution (with merging)
	$\Rightarrow$ Loop unrolling & no unreachable code
Compact	LBE-style with auxiliary variables for blocks
	$\Rightarrow$ Path(s) selection & no exponential growth
Incremental	Uses one monotonically growing solver instance
	$\Rightarrow$ Incremental SAT solving under assumptions

Dynamic LBE

- Combines the semantics of all feasible paths dynamically
- Efficiency based on combination of several traits

Dynamic	Symbolic execution (with merging)
	$\Rightarrow$ Loop unrolling & no unreachable code
Compact	LBE-style with auxiliary variables for blocks
	$\Rightarrow$ Path(s) selection & no exponential growth
Incremental	Uses one monotonically growing solver instance
	$\Rightarrow$ Incremental SAT solving under assumptions

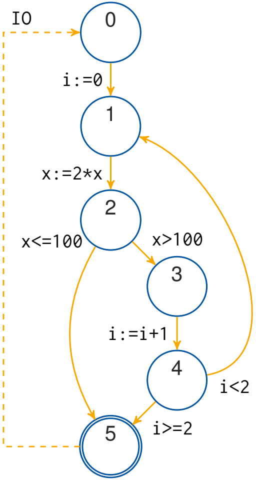
Dynamic LBE

- Combines the semantics of all feasible paths dynamically
- Efficiency based on combination of several traits

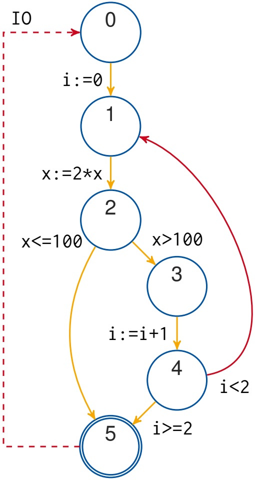
Dynamic	Symbolic execution (with merging)
	$\Rightarrow$ Loop unrolling & no unreachable code
Compact	LBE-style with auxiliary variables for blocks
	$\Rightarrow$ Path(s) selection & no exponential growth
Incremental	Uses one monotonically growing solver instance
	$\Rightarrow$ Incremental SAT solving under assumptions

Informatik 11 Embedded Software
RWTH AACHEN UNIVERSITY

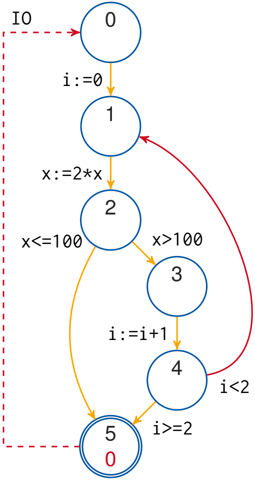
Dynamic LBE Example



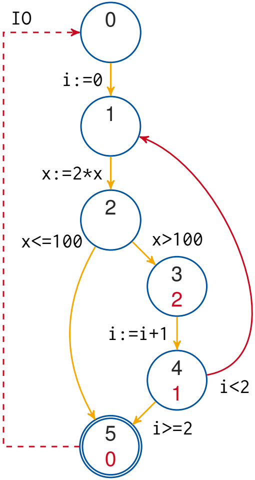
Dynamic LBE Example



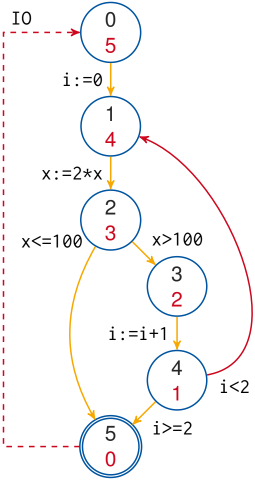
Dynamic LBE Example



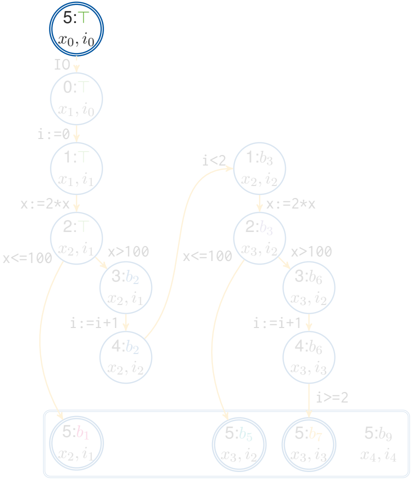
Dynamic LBE Example



Dynamic LBE Example



Dynamic LBE Example



$x_0 = 0, i_0 = 0,$

$i_1 = 0,$

$x_2 = 2 \cdot x_1,$

$b_1 \rightarrow true, b_1 \rightarrow x_2 \leq 100,$

$b_2 \rightarrow true, b_2 \rightarrow x_2 > 100,$

$b_2 \rightarrow i_2 = i_1 + 1,$

$b_3 \rightarrow b_2, b_3 \rightarrow i_2 < 2,$

$\ldots$

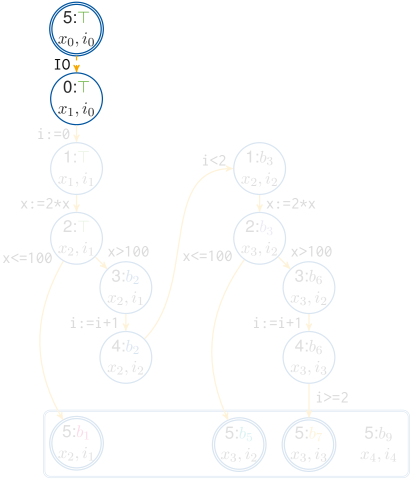
$b_9 \rightarrow b_1 \vee b_5 \vee b_7,$

$b_1 \rightarrow x_4 = x_2 \wedge i_4 = i_1,$

$b_5 \rightarrow x_4 = x_3 \wedge i_4 = i_2,$

$b_7 \rightarrow x_4 = x_3 \wedge i_4 = i_3$

Dynamic LBE Example



$x_0 = 0, i_0 = 0,$

$i_1 = 0,$

$x_2 = 2 \cdot x_1,$

$b_1 \to true, b_1 \to x_2 \leq 100,$

$b_2 \to true, b_2 \to x_2 > 100,$

$b_2 \to i_2 = i_1 + 1,$

$b_3 \to b_2, b_3 \to i_2 < 2,$

$\dots$

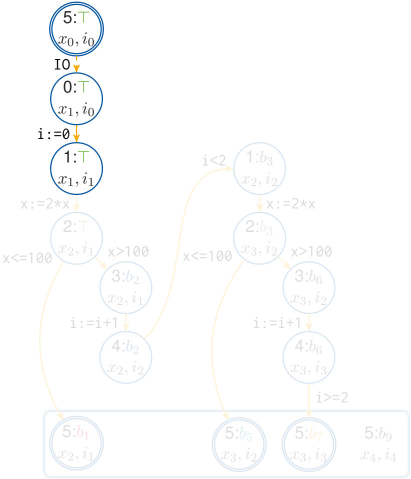
$b_9 \to b_1 \vee b_5 \vee b_7,$

$b_1 \to x_4 = x_2 \wedge i_4 = i_1,$

$b_5 \to x_4 = x_3 \wedge i_4 = i_2,$

$b_7 \to x_4 = x_3 \wedge i_4 = i_3$

Dynamic LBE Example



$x_0 = 0, i_0 = 0,$

$i_1 = 0,$

$x_2 = 2 \cdot x_1,$

$b_1 \to true, b_1 \to x_2 \leq 100,$

$b_2 \to true, b_2 \to x_2 > 100,$

$b_2 \to i_2 = i_1 + 1,$

$b_3 \to b_2, b_3 \to i_2 < 2,$

$\ldots$

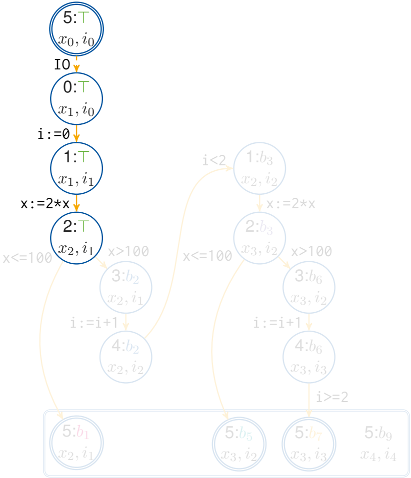
$b_9 \to b_1 \vee b_5 \vee b_7,$

$b_1 \to x_4 = x_2 \wedge i_4 = i_1,$

$b_5 \to x_4 = x_3 \wedge i_4 = i_2,$

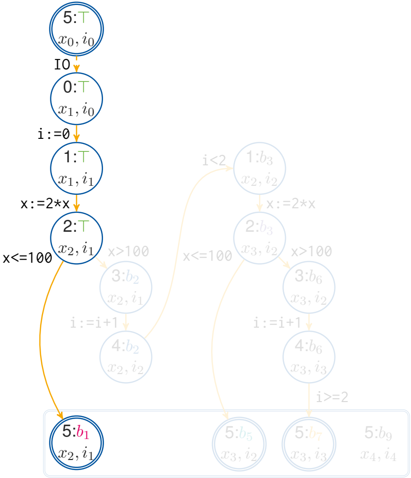
$b_7 \to x_4 = x_3 \wedge i_4 = i_3$

Dynamic LBE Example



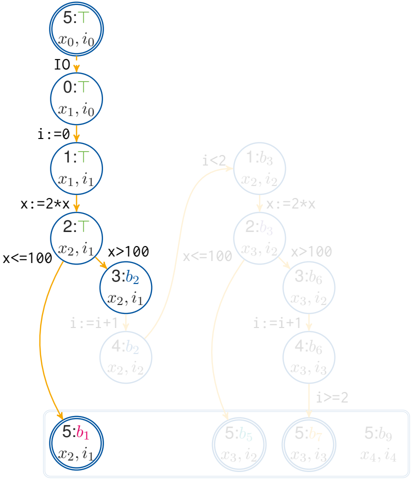
$$x_0 = 0, i_0 = 0,$$
$$i_1 = 0,$$
$$x_2 = 2 \cdot x_1,$$
$$b_1 \rightarrow true, b_1 \rightarrow x_2 \leq 100,$$
$$b_2 \rightarrow true, b_2 \rightarrow x_2 > 100,$$
$$b_2 \rightarrow i_2 = i_1 + 1,$$
$$b_3 \rightarrow b_2, b_3 \rightarrow i_2 < 2,$$
$$\ldots$$
$$b_9 \rightarrow b_1 \vee b_5 \vee b_7,$$
$$b_1 \rightarrow x_4 = x_2 \wedge i_4 = i_1,$$
$$b_5 \rightarrow x_4 = x_3 \wedge i_4 = i_2,$$
$$b_7 \rightarrow x_4 = x_3 \wedge i_4 = i_3$$

Dynamic LBE Example



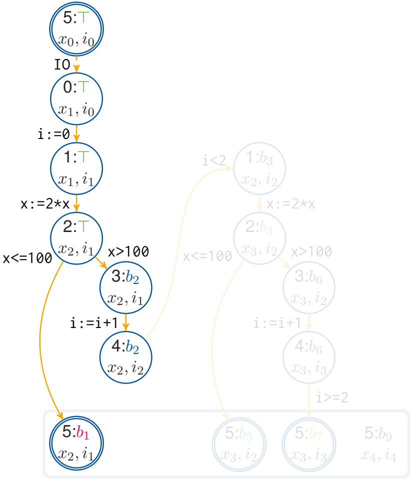
$$x_0 = 0, i_0 = 0,$$
$$i_1 = 0,$$
$$x_2 = 2 \cdot x_1,$$
$$b_1 \rightarrow \mathit{true}, b_1 \rightarrow x_2 \leq 100,$$
$$b_2 \rightarrow \mathit{true}, b_2 \rightarrow x_2 > 100,$$
$$b_2 \rightarrow i_2 = i_1 + 1,$$
$$b_3 \rightarrow b_2, b_3 \rightarrow i_2 < 2,$$
$$\ldots$$
$$b_9 \rightarrow b_1 \vee b_5 \vee b_7,$$
$$b_1 \rightarrow x_4 = x_2 \wedge i_4 = i_1,$$
$$b_5 \rightarrow x_4 = x_3 \wedge i_4 = i_2,$$
$$b_7 \rightarrow x_4 = x_3 \wedge i_4 = i_3$$

Dynamic LBE Example



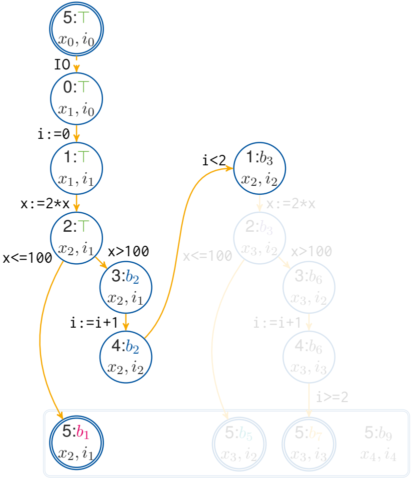
$$
\begin{aligned}
&x_0 = 0, i_0 = 0,\\
&i_1 = 0,\\
&x_2 = 2 \cdot x_1,\\
&b_1 \rightarrow \mathit{true}, b_1 \rightarrow x_2 \leq 100,\\
&b_2 \rightarrow \mathit{true}, b_2 \rightarrow x_2 > 100,\\
&b_2 \rightarrow i_2 = i_1 + 1,\\
&b_3 \rightarrow b_2, b_3 \rightarrow i_2 < 2,\\
&\dots\\
&b_9 \rightarrow b_1 \vee b_5 \vee b_7,\\
&b_1 \rightarrow x_4 = x_2 \wedge i_4 = i_1,\\
&b_5 \rightarrow x_4 = x_3 \wedge i_4 = i_2,\\
&b_7 \rightarrow x_4 = x_3 \wedge i_4 = i_3
\end{aligned}
$$

Dynamic LBE Example



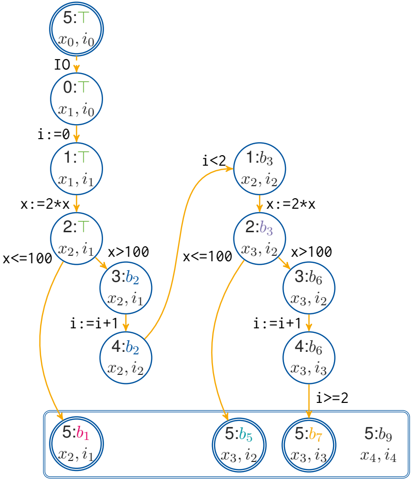
$$
\begin{aligned}
& x_0 = 0, i_0 = 0, \\
& i_1 = 0, \\
& x_2 = 2 \cdot x_1, \\
& b_1 \rightarrow \mathit{true}, b_1 \rightarrow x_2 \leq 100, \\
& b_2 \rightarrow \mathit{true}, b_2 \rightarrow x_2 > 100, \\
& b_2 \rightarrow i_2 = i_1 + 1, \\
& b_3 \rightarrow b_2, b_3 \rightarrow i_2 < 2, \\
& \cdots \\
& b_9 \rightarrow b_1 \vee b_5 \vee b_7, \\
& b_1 \rightarrow x_4 = x_2 \wedge i_4 = i_1, \\
& b_5 \rightarrow x_4 = x_3 \wedge i_4 = i_2, \\
& b_7 \rightarrow x_4 = x_3 \wedge i_4 = i_3
\end{aligned}
$$

Dynamic LBE Example



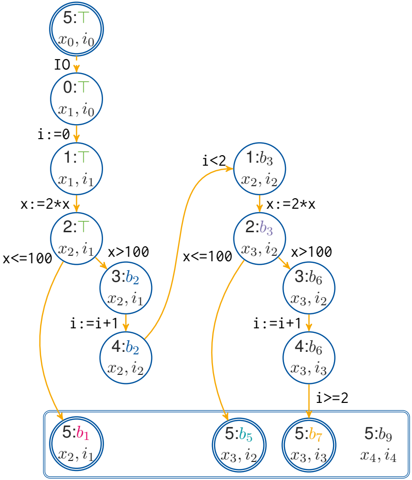
$$x_0 = 0, i_0 = 0,$$
$$i_1 = 0,$$
$$x_2 = 2 \cdot x_1,$$
$$b_1 \to true, b_1 \to x_2 \leq 100,$$
$$b_2 \to true, b_2 \to x_2 > 100,$$
$$b_2 \to i_2 = i_1 + 1,$$
$$b_3 \to b_2, b_3 \to i_2 < 2,$$
$$\ldots$$
$$b_9 \to b_1 \vee b_5 \vee b_7,$$
$$b_1 \to x_4 = x_2 \wedge i_4 = i_1,$$
$$b_5 \to x_4 = x_3 \wedge i_4 = i_2,$$
$$b_7 \to x_4 = x_3 \wedge i_4 = i_3$$

Dynamic LBE Example



$$x_0 = 0, i_0 = 0,$$
$$i_1 = 0,$$
$$x_2 = 2 \cdot x_1,$$
$$b_1 \to \mathit{true}, b_1 \to x_2 \le 100,$$
$$b_2 \to \mathit{true}, b_2 \to x_2 > 100,$$
$$b_2 \to i_2 = i_1 + 1,$$
$$b_3 \to b_2, b_3 \to i_2 < 2,$$
$$\dots$$
$$b_9 \to b_1 \lor b_5 \lor b_7,$$
$$b_1 \to x_4 = x_2 \land i_4 = i_1,$$
$$b_5 \to x_4 = x_3 \land i_4 = i_2,$$
$$b_7 \to x_4 = x_3 \land i_4 = i_3$$

Dynamic LBE Example



$$
\begin{aligned}
&x_0 = 0, i_0 = 0,\\
&i_1 = 0,\\
&x_2 = 2 \cdot x_1,\\
&b_1 \to \mathit{true}, b_1 \to x_2 \leq 100,\\
&b_2 \to \mathit{true}, b_2 \to x_2 > 100,\\
&b_2 \to i_2 = i_1 + 1,\\
&b_3 \to b_2, b_3 \to i_2 < 2,\\
&\ldots\\
&b_9 \to b_1 \vee b_5 \vee b_7,\\
&b_1 \to x_4 = x_2 \wedge i_4 = i_1,\\
&b_5 \to x_4 = x_3 \wedge i_4 = i_2,\\
&b_7 \to x_4 = x_3 \wedge i_4 = i_3
\end{aligned}
$$

Cycle-BMC

- k iterations of DLBE will yield the symbolic contexts:

$$\underbrace{(l_{\text{EoC}}, true, \vec{x}_0)}_{c_0}, \underbrace{(l_{\text{EoC}}, b_1, \vec{x}_1)}_{c_1}, \dots, \underbrace{(l_{\text{EoC}}, b_k, \vec{x}_k)}_{c_k}$$

⇒ At this point the solver will *effectively* contain

$$I(\vec{x}_0) \wedge \begin{pmatrix} b_1 \to T_1(\vec{x}_0, \vec{x}_1) \wedge b_1 \to true \\ \wedge \dots \\ \wedge b_k \to T_{k-1}(\vec{x}_{k-1}, \vec{x}_k) \wedge b_k \to b_{k-1} \end{pmatrix}$$

- Assuming b_k allows checking for bad behaviours, e.g.

$$\bigvee_{0 \leq i \leq k} B(\vec{x}_i)$$

Cycle-BMC

- k iterations of DLBE will yield the symbolic contexts:

$$\underbrace{(l_{\text{EoC}}, \mathit{true}, \vec{x}_0)}_{c_0}, \underbrace{(l_{\text{EoC}}, b_1, \vec{x}_1)}_{c_1}, \ldots, \underbrace{(l_{\text{EoC}}, b_k, \vec{x}_k)}_{c_k}$$

$\Rightarrow$ At this point the solver will *effectively* contain

$$I(\vec{x}_0) \wedge \begin{pmatrix} b_1 \rightarrow T_1(\vec{x}_0, \vec{x}_1) \wedge b_1 \rightarrow \mathit{true} \\ \wedge \ \ldots \\ \wedge\ b_k \rightarrow T_{k-1}(\vec{x}_{k-1}, \vec{x}_k) \wedge b_k \rightarrow b_{k-1} \end{pmatrix}$$

- Assuming b_k allows checking for bad behaviours, e.g.

$$\bigvee_{0 \leq i \leq k} B(\vec{x}_i)$$

Cycle-BMC

- k iterations of DLBE will yield the symbolic contexts:

$$\underbrace{(l_{\text{EoC}}, \mathit{true}, \vec{x}_0)}_{c_0}, \underbrace{(l_{\text{EoC}}, b_1, \vec{x}_1)}_{c_1}, \ldots, \underbrace{(l_{\text{EoC}}, b_k, \vec{x}_k)}_{c_k}$$

⇒ At this point the solver will *effectively* contain

$$I(\vec{x}_0) \wedge \begin{pmatrix} b_1 \rightarrow T_1(\vec{x}_0, \vec{x}_1) \wedge b_1 \rightarrow \mathit{true} \\ \wedge \ \ldots \\ \wedge\ b_k \rightarrow T_{k-1}(\vec{x}_{k-1}, \vec{x}_k) \wedge b_k \rightarrow b_{k-1} \end{pmatrix}$$

- Assuming b_k allows checking for bad behaviours, e.g.

$$\bigvee_{0 \leq i \leq k} B(\vec{x}_i)$$

Setup

Experiments on implementation of PLCopen Safety library:

- Single modules implementing particular safety concepts
- User examples combine those

Specifications:

- Only invariants are supported by all tools
- Wrapped CFA in bounded for-loop and translated to C / SMV

BMC-based Tools:

- ARCADE.PLC: Built Cycle-BMC prototype upon this frontend
- CPACHECKER: Adjustable-/LBE with predicate abstraction
- ESBMC: Unrolls programs & optimises verification conditions
- NUSMV: BMC required bound $\Rightarrow$ Used BDD-engine

Setup

Experiments on implementation of PLCopen Safety library:

- Single modules implementing particular safety concepts
- User examples combine those

Specifications:

- Only invariants are supported by all tools
- Wrapped CFA in bounded for-loop and translated to C / SMV

BMC-based Tools:

- ARCADE.PLC: Built Cycle-BMC prototype upon this frontend
- CPACHECKER: Adjustable-/LBE with predicate abstraction
- ESBMC: Unrolls programs & optimises verification conditions
- NUSMV: BMC required bound ⇒ Used BDD-engine

Setup

Experiments on implementation of PLCopen Safety library:

- Single modules implementing particular safety concepts
- User examples combine those

Specifications:

- Only invariants are supported by all tools
- Wrapped CFA in bounded for-loop and translated to C / SMV

BMC-based Tools:

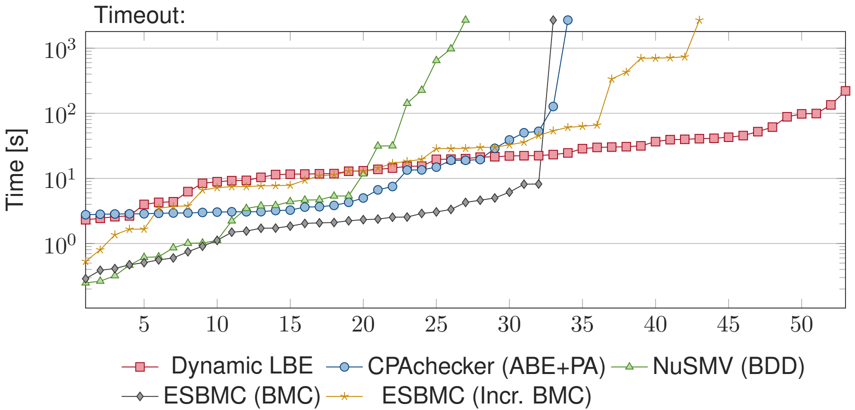
- ARCADE.PLC: Built Cycle-BMC prototype upon this frontend
- CPACHECKER: Adjustable-/LBE with predicate abstraction
- ESBMC: Unrolls programs & optimises verification conditions
- NUSMV: BMC required bound ⇒ Used BDD-engine

Results

- 53 cycle-bounded verification tasks (#locations: 117 – 1450)
- 24 specs for elementary modules and 29 for composite ones

Results

- 53 cycle-bounded verification tasks (#locations: 117 – 1450)
- 24 specs for elementary modules and 29 for composite ones



Summary

- PLC software specs refer to observable states only
- Using existing solutions is not always possible / feasible
- Dynamic LBE is a generic approach to unrolling and encoding CFA semantics in a fully incremental way

⇒ Enabled efficient Cycle-BMC without reformulation of specs

Future Work:

- Encoding is compatible with abstraction refinement

⇒ Unroll loops iteratively & evaluate calls lazily

- Experiment with C benchmarks

Summary

- PLC software specs refer to observable states only
- Using existing solutions is not always possible / feasible
- Dynamic LBE is a generic approach to unrolling and encoding CFA semantics in a fully incremental way

⇒ Enabled efficient Cycle-BMC without reformulation of specs

Future Work:

- Encoding is compatible with abstraction refinement

⇒ Unroll loops iteratively & evaluate calls lazily

- Experiment with C benchmarks

Summary

- PLC software specs refer to observable states only
- Using existing solutions is not always possible / feasible
- Dynamic LBE is a generic approach to unrolling and encoding CFA semantics in a fully incremental way

$\Rightarrow$ Enabled efficient Cycle-BMC without reformulation of specs

Future Work:

- Encoding is compatible with abstraction refinement

$\Rightarrow$ Unroll loops iteratively & evaluate calls lazily

- Experiment with C benchmarks

Summary

- PLC software specs refer to observable states only
- Using existing solutions is not always possible / feasible
- Dynamic LBE is a generic approach to unrolling and encoding CFA semantics in a fully incremental way

⇒ Enabled efficient Cycle-BMC without reformulation of specs

Future Work:

- Encoding is compatible with abstraction refinement

⇒ Unroll loops iteratively & evaluate calls lazily

- Experiment with C benchmarks

Summary

- PLC software specs refer to observable states only
- Using existing solutions is not always possible / feasible
- Dynamic LBE is a generic approach to unrolling and encoding CFA semantics in a fully incremental way

⇒ Enabled efficient Cycle-BMC without reformulation of specs

Future Work:

- Encoding is compatible with abstraction refinement

⇒ Unroll loops iteratively & evaluate calls lazily

- Experiment with C benchmarks