Compositional Verification of PLC Software using Horn Clauses and Mode Abstraction

Dimitri Bohlender | Stefan Kowalewski

WODES 2018, June 1, 2018

Outline

Introduction

CHC-based Verification

- Constrained Horn Clauses
- Modelling of PLC Software with CHCs
- Mode-Space as Call Summary
- Experiments

Conclusion

Outline

Introduction

CHC-based Verification

- Constrained Horn Clauses
- Modelling of PLC Software with CHCs
- Mode-Space as Call Summary
- Experiments

Conclusion

Outline

Introduction

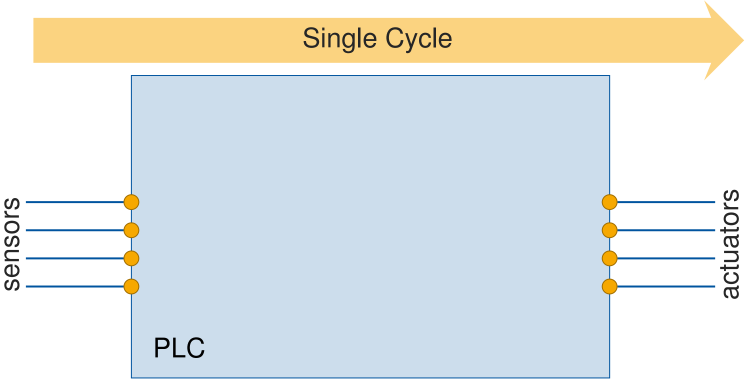
CHC-based Verification

- Constrained Horn Clauses
- Modelling of PLC Software with CHCs
- Mode-Space as Call Summary
- Experiments

Conclusion

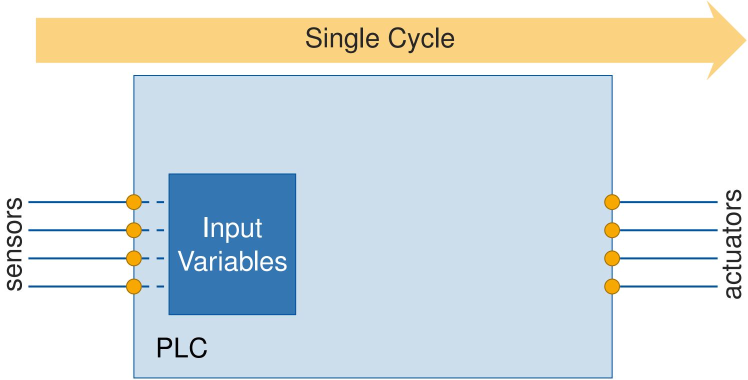
Programmable Logic Controllers (PLCs)

- 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



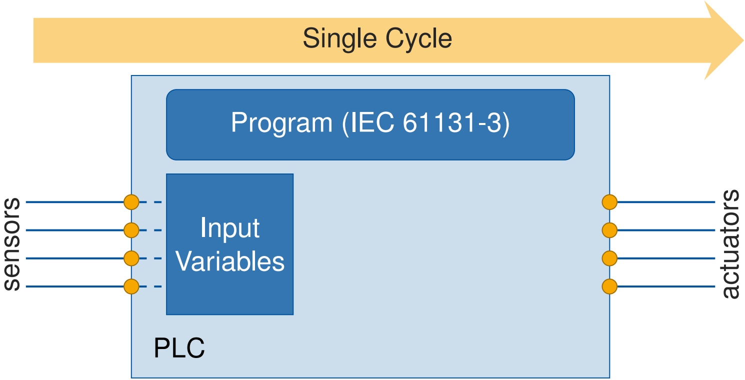
Programmable Logic Controllers (PLCs)

- 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



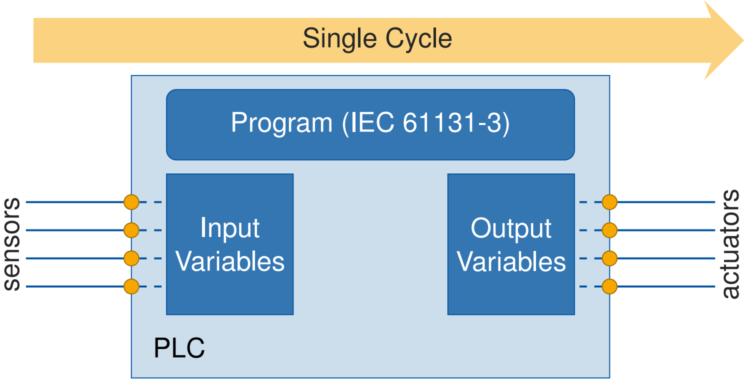
Programmable Logic Controllers (PLCs)

- 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



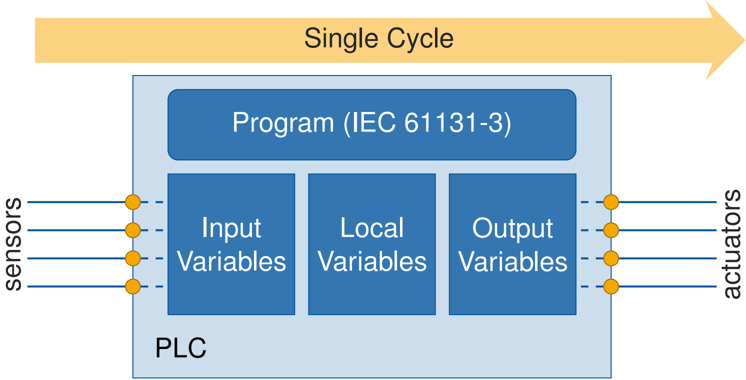
Programmable Logic Controllers (PLCs)

- 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



Running Example

- Implements delegation of requests
- Input in forwarded on rising edge for req
- ReqHandler h is polled in every cycle
- Exhibits state-machine semantics via DiagCode

```
PROGRAM Main
  VAR_INPUT  req:BOOL; in:WORD;      END_VAR
  VAR        m  :BOOL; h :ReqHandler;END_VAR
  VAR_OUTPUT out:WORD;               END_VAR
  // Forward data on rising edge
  IF (req AND NOT m) THEN
    h(data:=in, res=>out);
  ELSE
    h(res=>out);
  END_IF
  m:=req;
END_PROGRAM

FUNCTION_BLOCK ReqHandler
  VAR_INPUT  data    :WORD; END_VAR
  VAR        DiagCode:WORD; END_VAR
  VAR_OUTPUT res     :WORD; END_VAR
  // Body omitted ...
END_FUNCTION_BLOCK
```

Running Example

- ▸ Implements delegation of requests
- ▸ Input `in` forwarded on rising edge for `req`
- ▸ `ReqHandler h` is polled in every cycle
- ▸ Exhibits state-machine semantics via `DiagCode`

```
PROGRAM Main
  VAR_INPUT  req:BOOL; in:WORD;         END_VAR
  VAR        m  :BOOL; h :ReqHandler;END_VAR
  VAR_OUTPUT out:WORD;                  END_VAR
  // Forward data on rising edge
  IF (req AND NOT m) THEN
    h(data:=in, res=>out);
  ELSE
    h(res=>out);
  END_IF
  m:=req;
END_PROGRAM

FUNCTION_BLOCK ReqHandler
  VAR_INPUT  data    :WORD; END_VAR
  VAR        DiagCode:WORD; END_VAR
  VAR_OUTPUT res     :WORD; END_VAR
  // Body omitted ...
END_FUNCTION_BLOCK
```

Running Example

- Implements delegation of requests
- Input `in` forwarded on rising edge for `req`
- `ReqHandler h` is polled in every cycle
- Exhibits state-machine semantics via `DiagCode`

```
PROGRAM Main
  VAR_INPUT  req:BOOL; in:WORD;       END_VAR
  VAR        m  :BOOL; h :ReqHandler;END_VAR
  VAR_OUTPUT out:WORD;                END_VAR
  // Forward data on rising edge
  IF (req AND NOT m) THEN
    h(data:=in, res=>out);
  ELSE
    h(res=>out);
  END_IF
  m:=req;
END_PROGRAM

FUNCTION_BLOCK ReqHandler
  VAR_INPUT  data    :WORD; END_VAR
  VAR        DiagCode:WORD; END_VAR
  VAR_OUTPUT res     :WORD; END_VAR
  // Body omitted ...
END_FUNCTION_BLOCK
```

Running Example

- Implements delegation of requests
- Input `in` forwarded on rising edge for `req`
- `ReqHandler h` is polled in every cycle
- Exhibits state-machine semantics via `DiagCode`

```
PROGRAM Main
  VAR_INPUT   req:BOOL; in:WORD;          END_VAR
  VAR         m  :BOOL; h :ReqHandler;END_VAR
  VAR_OUTPUT  out:WORD;                   END_VAR
  // Forward data on rising edge
  IF (req AND NOT m) THEN
    h(data:=in, res=>out);
  ELSE
    h(res=>out);
  END_IF
  m:=req;
END_PROGRAM

FUNCTION_BLOCK ReqHandler
  VAR_INPUT   data    :WORD; END_VAR
  VAR         DiagCode:WORD; END_VAR
  VAR_OUTPUT  res     :WORD; END_VAR
  // Body omitted ...
END_FUNCTION_BLOCK
```

Mode Spaces in Model-Checking

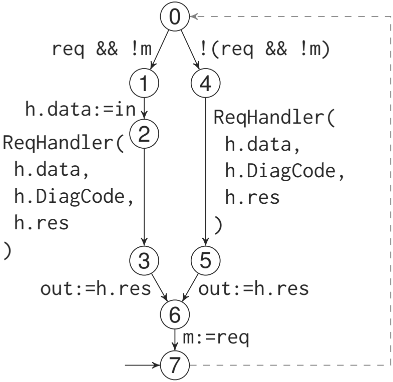


Figure: Main CFA of example

"Idle"
DiagCode
=0x0000

"Processing"
DiagCode
=0x8000

"Error"
DiagCode
=0xC001

Figure: Mode space of ReqHandler

- Requests are processed in $\leq$ two execution cycles?

$\Rightarrow$ On request, "Processing" mode is reached in a single cycle

Mode Spaces in Model-Checking

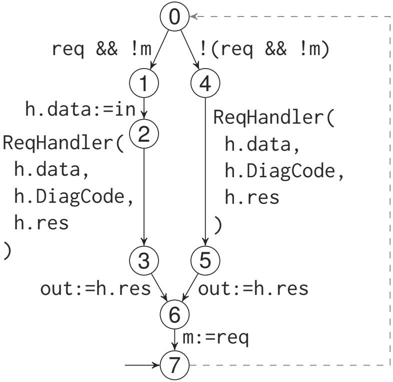


Figure: Main CFA of example

“Idle”
`DiagCode`
`=0x0000`

“Processing”
`DiagCode`
`=0x8000`

“Error”
`DiagCode`
`=0xC001`

Figure: Mode space of `ReqHandler`

- Requests are processed in $\leq$ two execution cycles?

$\Rightarrow$ On request, “Processing” mode is reached in a single cycle

Mode Spaces in Model-Checking

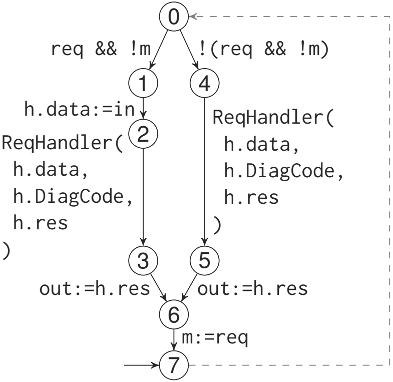


Figure: Main CFA of example

“Idle”
DiagCode
=0x0000

“Processing”
DiagCode
=0x8000

“Error”
DiagCode
=0xC001

Figure: Mode space of `ReqHandler`

- Requests are processed in $\leq$ two execution cycles?

⇒ On request, “Processing” mode is reached in a single cycle

Mode Spaces in Model-Checking

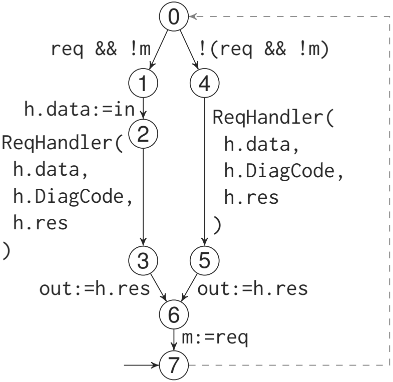


Figure: Main CFA of example

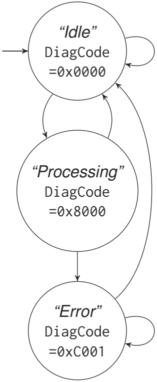


Figure: Mode space of `ReqHandler`

- Requests are processed in $\leq$ two execution cycles?

⇒ On request, "Processing" mode is reached in a single cycle

Mode Spaces in Model-Checking

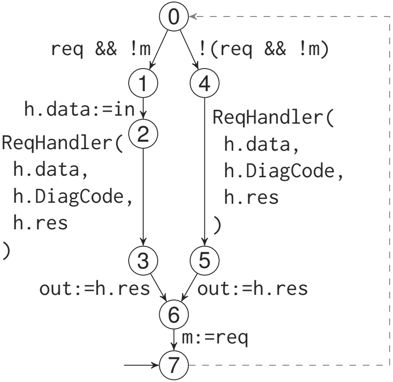


Figure: Main CFA of example

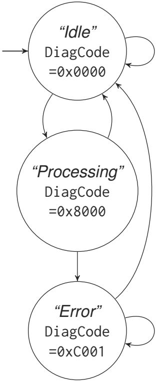


Figure: Mode space of `ReqHandler`

- Requests are processed in $\leq$ two execution cycles?

$\Rightarrow$ On request, "Processing" mode is reached in a single cycle

Requirements Towards Formalism & Model Checker

- Compositional reasoning (no cloning or inlining)
- Integrates with abstraction of calls (call summaries)

Example

- Let `f1:FB; f2:FB;`
- Reason about `FB(...)`
- Avoid `f1(...)`, `f2(...)` and instruction cloning

Example

- Consider $\mathrm{abs}(x) = y$
- Use summary

$$y \geq 0$$

- Until details needed

- Constrained Horn Clauses (CHCs) meet these
- Uniform formalism for symbolic model checking of software

Requirements Towards Formalism & Model Checker

- Compositional reasoning (no cloning or inlining)

Example

- Let `f1:FB; f2:FB;`
- Reason about `FB(...)`
- Avoid `f1(...)`, `f2(...)` and instruction cloning

- Integrates with abstraction of calls (call summaries)

Example

- Consider $\mathrm{abs}(x) = y$
- Use summary

$$y \geq 0$$

- Until details needed

- Constrained Horn Clauses (CHCs) meet these
- Uniform formalism for symbolic model checking of software

Requirements Towards Formalism & Model Checker

- Compositional reasoning (no cloning or inlining)

Example

- Let `f1:FB; f2:FB;`
- Reason about `FB(...)`
- Avoid `f1(...)`, `f2(...)` and instruction cloning

- Integrates with abstraction of calls (call summaries)

Example

- Consider $\mathrm{abs}(x) = y$
- Use summary

$$y \geq 0$$

- Until details needed

- Constrained Horn Clauses (CHCs) meet these
- Uniform formalism for symbolic model checking of software

Requirements Towards Formalism & Model Checker

- Compositional reasoning (no cloning or inlining)
- Integrates with abstraction of calls (call summaries)

Example

- Let `f1:FB; f2:FB;`
- Reason about `FB(...)`
- Avoid `f1(...)`, `f2(...)` and instruction cloning

Example

- Consider $\mathrm{abs}(x) = y$
- Use summary

$$y \geq 0$$

- Until details needed

- Constrained Horn Clauses (CHCs) meet these
- Uniform formalism for symbolic model checking of software

Requirements Towards Formalism & Model Checker

- Compositional reasoning (no cloning or inlining)

Example

- Let `f1:FB; f2:FB;`
- Reason about `FB(...)`
- Avoid `f1(...)`, `f2(...)` and instruction cloning

- Integrates with abstraction of calls (call summaries)

Example

- Consider $\mathrm{abs}(x) = y$
- Use summary

$$y \geq 0$$

- Until details needed

- Constrained Horn Clauses (CHCs) meet these
- Uniform formalism for symbolic model checking of software

Requirements Towards Formalism & Model Checker

- Compositional reasoning (no cloning or inlining)

Example

- Let `f1:FB; f2:FB;`
- Reason about `FB(...)`
- Avoid `f1(...)`, `f2(...)` and instruction cloning

- Integrates with abstraction of calls (call summaries)

Example

- Consider $\mathrm{abs}(x) = y$
- Use summary

$$y \geq 0$$

- Until details needed

- Constrained Horn Clauses (CHCs) meet these
- Uniform formalism for symbolic model checking of software

Requirements Towards Formalism & Model Checker

- Compositional reasoning (no cloning or inlining)

Example

- Let `f1:FB; f2:FB;`
- Reason about `FB(...)`
- Avoid `f1(...)`, `f2(...)` and instruction cloning

- Integrates with abstraction of calls (call summaries)

Example

- Consider $\mathrm{abs}(x) = y$
- Use summary
$$y \geq 0$$
- Until details needed

- Constrained Horn Clauses (CHCs) meet these
- Uniform formalism for symbolic model checking of software

Requirements Towards Formalism & Model Checker

- Compositional reasoning (no cloning or inlining)
- Integrates with abstraction of calls (call summaries)

Example

- Let `f1:FB; f2:FB;`
- Reason about `FB(...)`
- Avoid `f1(...)`, `f2(...)` and instruction cloning

Example

- Consider $\text{abs}(x) = y$
- Use summary

$$y \geq 0$$

- Until details needed

- Constrained Horn Clauses (CHCs) meet these
- Uniform formalism for symbolic model checking of software

Requirements Towards Formalism & Model Checker

- Compositional reasoning (no cloning or inlining)

Example

- Let `f1:FB; f2:FB;`
- Reason about `FB(...)`
- Avoid `f1(...)`, `f2(...)` and instruction cloning

- Integrates with abstraction of calls (call summaries)

Example

- Consider $\mathrm{abs}(x) = y$
- Use summary

$$y \geq 0$$

- Until details needed

- Constrained Horn Clauses (CHCs) meet these
- Uniform formalism for symbolic model checking of software

Requirements Towards Formalism & Model Checker

- Compositional reasoning (no cloning or inlining)

Example

- Let `f1:FB; f2:FB;`
- Reason about `FB(...)`
- Avoid `f1(...)`, `f2(...)` and instruction cloning

- Integrates with abstraction of calls (call summaries)

Example

- Consider $\mathrm{abs}(x) = y$
- Use summary

$$y \geq 0$$

- Until details needed

- Constrained Horn Clauses (CHCs) meet these
- Uniform formalism for symbolic model checking of software

Constrained Horn Clauses

- Special case of Satisfiability Modulo Theories (SMT)

Definition

Given sets of variables $\mathcal{V}$, function symbols $\mathcal{F}$, and predicates $\mathcal{P}$, a *Constrained Horn Clause* (CHC) is a formula

$$\forall \mathcal{V}\ \underbrace{p_1(\vec{X}_1) \wedge \cdots \wedge p_k(\vec{X}_k) \wedge \varphi}_{\text{body}} \rightarrow \underbrace{h(\vec{X})}_{\text{head}},\ k \geq 0,$$

where

- $\vec{X}_i, \vec{X} \subseteq \mathcal{V}$ are possibly empty vectors of variables
- φ is a constraint over $\mathcal{F}$ and $\mathcal{V}$

- CHCs satisfiable if satisfying interpretation of p_i exists

Constrained Horn Clauses

- Special case of Satisfiability Modulo Theories (SMT)

Definition

Given sets of variables $\mathcal{V}$, function symbols $\mathcal{F}$, and predicates $\mathcal{P}$, a *Constrained Horn Clause* (CHC) is a formula

$$\forall \mathcal{V}\ \underbrace{p_1(\vec{X}_1) \land \cdots \land p_k(\vec{X}_k) \land \varphi}_{\text{body}} \rightarrow \underbrace{h(\vec{X})}_{\text{head}},\ \ k \geq 0,$$

where

- $\vec{X}_i, \vec{X} \subseteq \mathcal{V}$ are possibly empty vectors of variables
- φ is a constraint over $\mathcal{F}$ and $\mathcal{V}$

- CHCs satisfiable if satisfying interpretation of p_i exists

Constrained Horn Clauses

- Special case of Satisfiability Modulo Theories (SMT)

Definition

Given sets of variables $\mathcal{V}$, function symbols $\mathcal{F}$, and predicates $\mathcal{P}$, a *Constrained Horn Clause* (CHC) is a formula

$$\forall \mathcal{V}\ \underbrace{p_1(\vec{X}_1) \land \cdots \land p_k(\vec{X}_k) \land \varphi}_{\text{body}} \rightarrow \underbrace{h(\vec{X})}_{\text{head}},\ k \geq 0,$$

where

- $\vec{X}_i, \vec{X} \subseteq \mathcal{V}$ are possibly empty vectors of variables
- φ is a constraint over $\mathcal{F}$ and $\mathcal{V}$

- CHCs satisfiable if satisfying interpretation of p_i exists

Constrained Horn Clauses

- Special case of Satisfiability Modulo Theories (SMT)

Definition

Given sets of variables $\mathcal{V}$, function symbols $\mathcal{F}$, and predicates $\mathcal{P}$, a *Constrained Horn Clause* (CHC) is a formula

$$\forall \mathcal{V}\ \underbrace{p_1(\vec{X}_1) \land \cdots \land p_k(\vec{X}_k) \land \varphi}_{\text{body}} \rightarrow \underbrace{h(\vec{X})}_{\text{head}},\ k \geq 0,$$

where

- $\vec{X}_i, \vec{X} \subseteq \mathcal{V}$ are possibly empty vectors of variables
- φ is a constraint over $\mathcal{F}$ and $\mathcal{V}$

- CHCs satisfiable if satisfying interpretation of p_i exists

Constrained Horn Clauses

- Special case of Satisfiability Modulo Theories (SMT)

Definition

Given sets of variables $\mathcal{V}$, function symbols $\mathcal{F}$, and predicates $\mathcal{P}$, a *Constrained Horn Clause* (CHC) is a formula

$$\forall\mathcal{V}\ \underbrace{p_1(\vec{X}_1) \wedge \cdots \wedge p_k(\vec{X}_k) \wedge \varphi}_{\text{body}} \rightarrow \underbrace{h(\vec{X})}_{\text{head}},\ k \geq 0,$$

where

- $\vec{X}_i, \vec{X} \subseteq \mathcal{V}$ are possibly empty vectors of variables
- φ is a constraint over $\mathcal{F}$ and $\mathcal{V}$

- CHCs satisfiable if satisfying interpretation of p_i exists

Concept

For each block type A with signature $VarTypes$, add a predicate

$$p_A : \underbrace{Loc \times VarTypes}_{source} \times \underbrace{Loc \times VarTypes}_{target} \to \mathbb{B}$$

- Will define transitive reachability within A

⇒ Observable semantics of procedure captured by

$$p_A(\underbrace{l_{entry}}_{const}, \vec{X}, \underbrace{l_{exit}}_{const}, \vec{X}')$$

- Solving CHC $\triangleq$ finding over-approximating summary of A

Concept

For each block type A with signature $\mathit{VarTypes}$, add a predicate

$$p_A : \underbrace{\mathit{Loc} \times \mathit{VarTypes}}_{source} \times \underbrace{\mathit{Loc} \times \mathit{VarTypes}}_{target} \to \mathbb{B}$$

- Will define transitive reachability within A

⇒ Observable semantics of procedure captured by

$$p_A(\underbrace{l_{entry}}_{const}, \vec{X}, \underbrace{l_{exit}}_{const}, \vec{X}')$$

- Solving CHC $\triangleq$ finding over-approximating summary of A

Concept

For each block type A with signature $\mathit{VarTypes}$, add a predicate

$$p_A : \underbrace{\mathit{Loc} \times \mathit{VarTypes}}_{\mathit{source}} \times \underbrace{\mathit{Loc} \times \mathit{VarTypes}}_{\mathit{target}} \to \mathbb{B}$$

- Will define transitive reachability within A

⇒ Observable semantics of procedure captured by

$$p_A(\underbrace{l_{entry}}_{const}, \vec{X}, \underbrace{l_{exit}}_{const}, \vec{X}')$$

- Solving CHC $\triangleq$ finding over-approximating summary of A

Concept

For each block type A with signature $\mathit{VarTypes}$, add a predicate

$$p_A : \underbrace{\mathit{Loc} \times \mathit{VarTypes}}_{\mathit{source}} \times \underbrace{\mathit{Loc} \times \mathit{VarTypes}}_{\mathit{target}} \to \mathbb{B}$$

- Will define transitive reachability within A

⇒ Observable semantics of procedure captured by

$$p_A(\underbrace{l_{entry}}_{const}, \vec{X}, \underbrace{l_{exit}}_{const}, \vec{X}')$$

- Solving CHC $\triangleq$ finding over-approximating summary of A

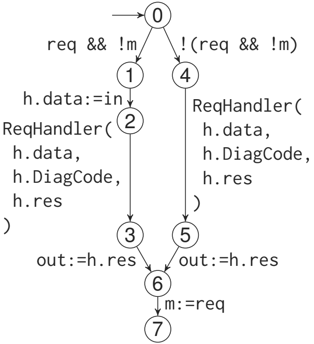
Encoding a Transition

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = (req', in', m', h.data', h.DiagCode', h.res', out')$

Transitive Reachability:

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\ h.data'' = in' \wedge id(X' \setminus \{h.data'\})$$
$$\rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



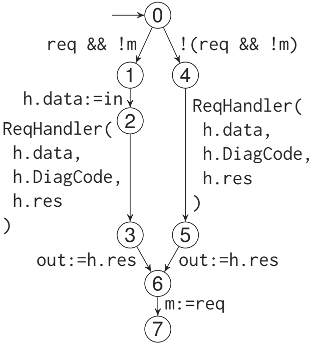
Encoding a Transition

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right)$

Transitive Reachability:

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\ h.data'' = in' \wedge id(X' \setminus \{h.data'\})$$
$$\rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



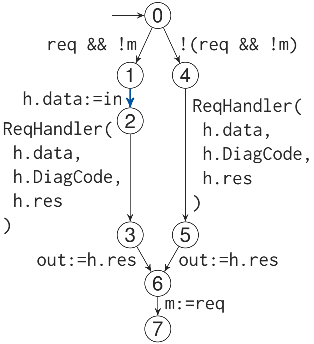
Encoding a Transition

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right)$

Transitive Reachability:

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\ h.data'' = in' \wedge id(X' \setminus \{h.data'\})$$
$$\rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



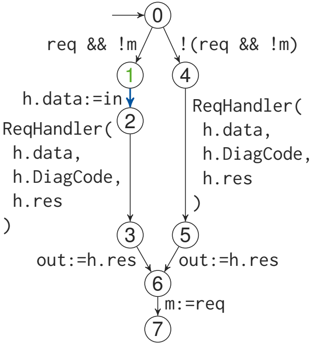
Encoding a Transition

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right)$

Transitive Reachability:

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\ h.data'' = in' \wedge id(X' \setminus \{h.data'\})$$
$$\rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



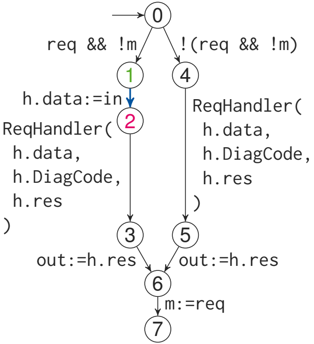
Encoding a Transition

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right)$

Transitive Reachability:

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\ h.data'' = in' \wedge id(X' \setminus \{h.data'\})$$
$$\rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



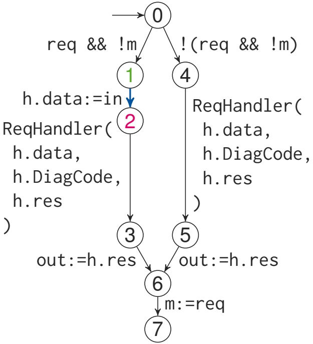
Encoding a Transition

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right)$

Transitive Reachability:

$$p_{Main}(l, \vec{X}, 1, \vec{X}') \\ \wedge\ h.data'' = in' \wedge id(X' \setminus \left\{h.data'\right\}) \\ \rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



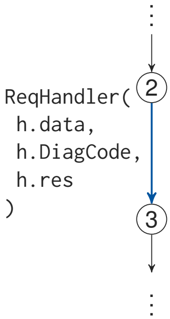
Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

Transitive Reachability (& call summary):

$$
\begin{aligned}
& p_{Main}(l, \vec{X}, 2, \vec{X}') \\
\wedge\ & p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \wedge id(\vec{X}' \setminus \vec{X}'_h) \\
\wedge\ & S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h) \\
\rightarrow\ & p_{Main}(l, \vec{X}, 3, \vec{X}'')
\end{aligned}
$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

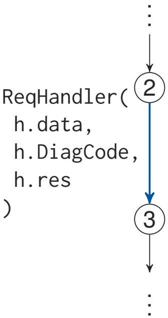
Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

Transitive Reachability (& call summary):

$$\begin{aligned} & p_{Main}(l, \vec{X}, 2, \vec{X}') \\ \land\ & p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \land id(\vec{X}' \setminus \vec{X}'_h) \\ \land\ & S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h) \\ \rightarrow\ & p_{Main}(l, \vec{X}, 3, \vec{X}'') \end{aligned}$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

Encoding a Call

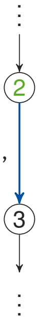
Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

Transitive Reachability (& call summary):

$$p_{Main}(l, \vec{X}, 2, \vec{X}')$$
$$\wedge\; p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \wedge id(\vec{X}' \setminus \vec{X}'_h)$$
$$\wedge\; S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h)$$
$$\rightarrow p_{Main}(l, \vec{X}, 3, \vec{X}'')$$

```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```



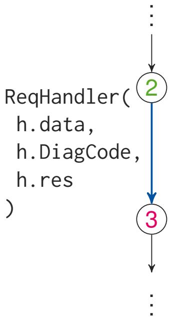
Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

Transitive Reachability (& call summary):

$$\begin{aligned} & p_{Main}(l, \vec{X}, 2, \vec{X}') \\ \wedge\ & p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \wedge id(\vec{X}' \setminus \vec{X}'_h) \\ \wedge\ & S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h) \\ \rightarrow\ & p_{Main}(l, \vec{X}, 3, \vec{X}'') \end{aligned}$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

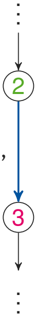
Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

Transitive Reachability (& call summary):

$$\begin{aligned} & p_{Main}(l, \vec{X}, 2, \vec{X}') \\ \wedge\ & p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \wedge id(\vec{X}' \setminus \vec{X}'_h) \\ \wedge\ & S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h) \\ \rightarrow\ & p_{Main}(l, \vec{X}, 3, \vec{X}'') \end{aligned}$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

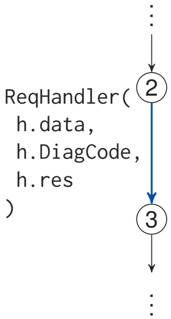
Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

Transitive Reachability (& call summary):

$$\begin{aligned} & p_{Main}(l, \vec{X}, 2, \vec{X}') \\ \wedge\ & p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \wedge id(\vec{X}' \setminus \vec{X}'_h) \\ \wedge\ & S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h) \\ \rightarrow\ & p_{Main}(l, \vec{X}, 3, \vec{X}'') \end{aligned}$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

Checking Safety Specifications

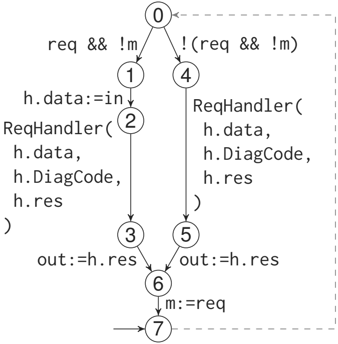
- Does safety property $safe(\vec{X})$ hold at the end of every cycle?

⇒ Yes, if adding

$$p_{Main}(0, \vec{X}, 7, \vec{X}') \rightarrow safe(\vec{X}')$$

keeps CHCs satisfiable

- Violated, if unsatisfiable



Checking Safety Specifications

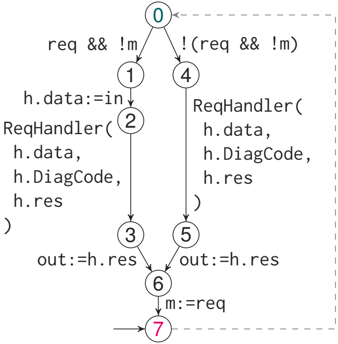
- Does safety property $safe(\vec{X})$ hold at the end of every cycle?

⇒ Yes, if adding

$$p_{Main}(0, \vec{X}, 7, \vec{X}') \rightarrow safe(\vec{X}')$$

keeps CHCs satisfiable

- Violated, if unsatisfiable



Checking Safety Specifications

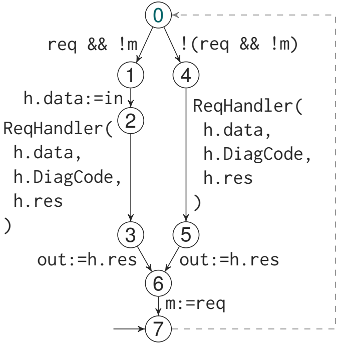
- Does safety property $safe(\vec{X})$ hold at the end of every cycle?

⇒ Yes, if adding

$$p_{Main}(0, \vec{X}, 7, \vec{X}') \rightarrow safe(\vec{X}')$$

keeps CHCs satisfiable

- Violated, if unsatisfiable



Approximating the Mode-Space

Idea:

- Procedure's complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA) to determine mode-transitions

Procedure:

1. Use global VSA to approximate all variables' values
2. For each function block type, e.g. `ReqHandler`
   2.1 For each computed mode-value, e.g. $\{$`0, 0x8000, 0xC001`$\}$
      - Keep global VSA values but fix mode, e.g. set `DiagCode=0`
      - Compute block's single-cycle VSA
      - Interpret resulting mode-values as targets, e.g. $\{$`0, 0x8000`$\}$

Approximating the Mode-Space

Idea:

- Procedure’s complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA) to determine mode-transitions

Procedure:

1. Use global VSA to approximate all variables’ values
2. For each function block type, e.g. `ReqHandler`
    2.1 For each computed mode-value, e.g. `{0,0x8000,0xC001}`
        - Keep global VSA values but fix mode, e.g. set `DiagCode=0`
        - Compute block’s single-cycle VSA
        - Interpret resulting mode-values as targets, e.g. `{0,0x8000}`

Approximating the Mode-Space

Idea:

- Procedure’s complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA) to determine mode-transitions

Procedure:

1. Use global VSA to approximate all variables’ values
2. For each function block type, e.g. `ReqHandler`
   2.1 For each computed mode-value, e.g. $\{$`0`, `0x8000`, `0xC001`$\}$
      - Keep global VSA values but fix mode, e.g. set `DiagCode=0`
      - Compute block’s single-cycle VSA
      - Interpret resulting mode-values as targets, e.g. $\{$`0`, `0x8000`$\}$

Approximating the Mode-Space

Idea:

- Procedure's complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA) to determine mode-transitions

Procedure:

1. Use global VSA to approximate all variables' values
2. For each function block type, e.g. `ReqHandler`

   2.1 For each computed mode-value, e.g. $\{$`0`, `0x8000`, `0xC001`$\}$

   - Keep global VSA values but fix mode, e.g. set `DiagCode=0`
   - Compute block's single-cycle VSA
   - Interpret resulting mode-values as targets, e.g. $\{$`0`, `0x8000`$\}$

Approximating the Mode-Space

Idea:

- Procedure's complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA) to determine mode-transitions

Procedure:

1. Use global VSA to approximate all variables' values
2. For each function block type, e.g. `ReqHandler`
   - 2.1 For each computed mode-value, e.g. $\{$`0`, `0x8000`, `0xC001`$\}$
     - Keep global VSA values but fix mode, e.g. set `DiagCode=0`
     - Compute block's single-cycle VSA
     - Interpret resulting mode-values as targets, e.g. $\{$`0`, `0x8000`$\}$

Approximating the Mode-Space

Idea:

- Procedure's complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA) to determine mode-transitions

Procedure:

1. Use global VSA to approximate all variables' values
2. For each function block type, e.g. `ReqHandler`
   2.1 For each computed mode-value, e.g. {`0`, `0x8000`, `0xC001`}
   - Keep global VSA values but fix mode, e.g. set `DiagCode=0`
   - Compute block's single-cycle VSA
   - Interpret resulting mode-values as targets, e.g. {`0`, `0x8000`}

Approximating the Mode-Space

Idea:

- Procedure's complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA) to determine mode-transitions

Procedure:

1. Use global VSA to approximate all variables' values
2. For each function block type, e.g. `ReqHandler`
   2.1 For each computed mode-value, e.g. `{0, 0x8000, 0xC001}`
   - Keep global VSA values but fix mode, e.g. set `DiagCode=0`
   - Compute block's single-cycle VSA
   - Interpret resulting mode-values as targets, e.g. `{0, 0x8000}`

Approximating the Mode-Space

Idea:

- Procedure's complexity needs to be low w.r.t. CHC-solving

⇒ Adapt value-set analysis (VSA) to determine mode-transitions

Procedure:

1. Use global VSA to approximate all variables' values
2. For each function block type, e.g. `ReqHandler`
   2.1 For each computed mode-value, e.g. $\{$`0`$,$`0x8000`$,$`0xC001`$\}$
      - Keep global VSA values but fix mode, e.g. set `DiagCode=0`
      - Compute block's single-cycle VSA
      - Interpret resulting mode-values as targets, e.g. $\{$`0`$,$`0x8000`$\}$

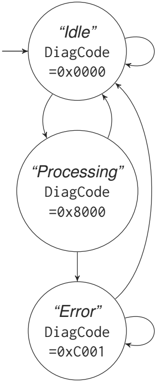
Mode-Space as Call Summary

- Mode-space constrains possible transitions

⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$\begin{aligned}
&(h.DiagCode = \mathtt{0} \rightarrow h.DiagCode' = \mathtt{0} \\
&\qquad \vee\ h.DiagCode' = \mathtt{0x8000}) \\
\wedge\ &(h.DiagCode = \mathtt{0x8000} \rightarrow h.DiagCode' = \mathtt{0} \\
&\qquad \vee\ h.DiagCode' = \mathtt{0xC001}) \\
\wedge\ &(h.DiagCode = \mathtt{0xC001} \rightarrow h.DiagCode' = \mathtt{0} \\
&\qquad \vee\ h.DiagCode' = \mathtt{0xC001})
\end{aligned}$$

- Add to encoding of each call of `ReqHandler`

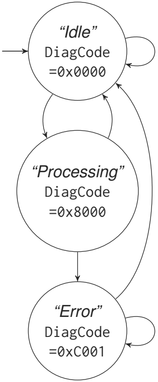


Mode-Space as Call Summary

- Mode-space constrains possible transitions

⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$\begin{aligned}
&(h.DiagCode = \texttt{0} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0x8000}) \\
\wedge\ &(h.DiagCode = \texttt{0x8000} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0xC001}) \\
\wedge\ &(h.DiagCode = \texttt{0xC001} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0xC001})
\end{aligned}$$



- Add to encoding of each call of `ReqHandler`

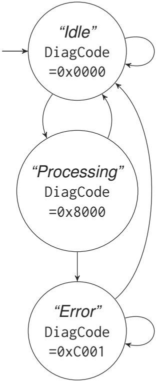
Mode-Space as Call Summary

- ▸ Mode-space constrains possible transitions

⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$
\begin{aligned}
&(h.DiagCode = \mathtt{0} \rightarrow h.DiagCode' = \mathtt{0} \\
&\qquad\qquad\qquad\qquad \vee\ h.DiagCode' = \mathtt{0x8000}) \\
\wedge\ &(h.DiagCode = \mathtt{0x8000} \rightarrow h.DiagCode' = \mathtt{0} \\
&\qquad\qquad\qquad\qquad \vee\ h.DiagCode' = \mathtt{0xC001}) \\
\wedge\ &(h.DiagCode = \mathtt{0xC001} \rightarrow h.DiagCode' = \mathtt{0} \\
&\qquad\qquad\qquad\qquad \vee\ h.DiagCode' = \mathtt{0xC001})
\end{aligned}
$$

- ▸ Add to encoding of each call of `ReqHandler`



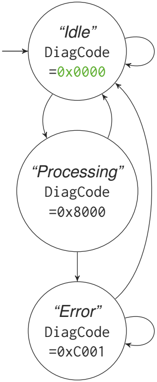
Mode-Space as Call Summary

- Mode-space constrains possible transitions

⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$\begin{aligned}
&(h.DiagCode = \texttt{0} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad\qquad\qquad\quad \vee\, h.DiagCode' = \texttt{0x8000}) \\
\wedge\ &(h.DiagCode = \texttt{0x8000} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad\qquad\qquad\quad \vee\, h.DiagCode' = \texttt{0xC001}) \\
\wedge\ &(h.DiagCode = \texttt{0xC001} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad\qquad\qquad\quad \vee\, h.DiagCode' = \texttt{0xC001})
\end{aligned}$$

- Add to encoding of each call of `ReqHandler`



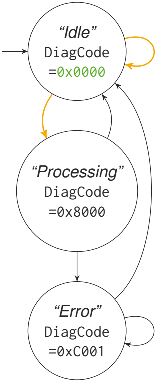
Mode-Space as Call Summary

- Mode-space constrains possible transitions

⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$\begin{aligned}(h.DiagCode = \texttt{0} \rightarrow h.DiagCode' &= \texttt{0} \\ \vee\ h.DiagCode' &= \texttt{0x8000}) \\ \wedge\ (h.DiagCode = \texttt{0x8000} \rightarrow h.DiagCode' &= \texttt{0} \\ \vee\ h.DiagCode' &= \texttt{0xC001}) \\ \wedge\ (h.DiagCode = \texttt{0xC001} \rightarrow h.DiagCode' &= \texttt{0} \\ \vee\ h.DiagCode' &= \texttt{0xC001})\end{aligned}$$

- Add to encoding of each call of `ReqHandler`



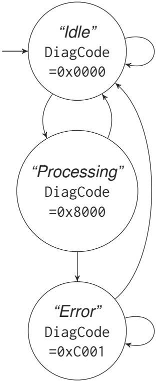
Mode-Space as Call Summary

- Mode-space constrains possible transitions

⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$\begin{aligned}
&(h.DiagCode = \texttt{0} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \lor h.DiagCode' = \texttt{0x8000}) \\
\land\ &(h.DiagCode = \texttt{0x8000} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \lor h.DiagCode' = \texttt{0xC001}) \\
\land\ &(h.DiagCode = \texttt{0xC001} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \lor h.DiagCode' = \texttt{0xC001})
\end{aligned}$$

- Add to encoding of each call of `ReqHandler`



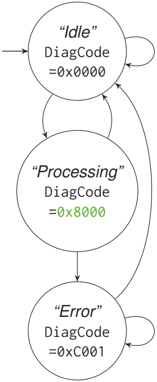
Mode-Space as Call Summary

- Mode-space constrains possible transitions

⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$
\begin{aligned}
&(h.DiagCode = \texttt{0} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0x8000}) \\
\wedge\ &(h.DiagCode = \texttt{0x8000} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0xC001}) \\
\wedge\ &(h.DiagCode = \texttt{0xC001} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0xC001})
\end{aligned}
$$

- Add to encoding of each call of ReqHandler



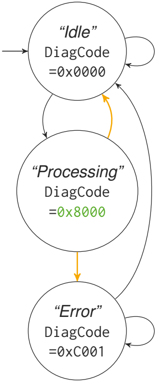
Mode-Space as Call Summary

- Mode-space constrains possible transitions

⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$
\begin{aligned}
&(h.DiagCode = \mathtt{0} \rightarrow h.DiagCode' = \mathtt{0} \\
&\qquad \lor h.DiagCode' = \mathtt{0x8000}) \\
\land\ &(h.DiagCode = \mathtt{0x8000} \rightarrow h.DiagCode' = \mathtt{0} \\
&\qquad \lor h.DiagCode' = \mathtt{0xC001}) \\
\land\ &(h.DiagCode = \mathtt{0xC001} \rightarrow h.DiagCode' = \mathtt{0} \\
&\qquad \lor h.DiagCode' = \mathtt{0xC001})
\end{aligned}
$$

- Add to encoding of each call of ReqHandler

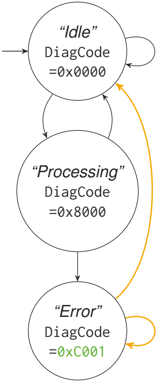


Mode-Space as Call Summary

- Mode-space constrains possible transitions

⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$\begin{aligned}
&(h.DiagCode = \texttt{0} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad\qquad\qquad \vee\ h.DiagCode' = \texttt{0x8000}) \\
\wedge\ &(h.DiagCode = \texttt{0x8000} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad\qquad\qquad \vee\ h.DiagCode' = \texttt{0xC001}) \\
\wedge\ &(h.DiagCode = \texttt{0xC001} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad\qquad\qquad\qquad \vee\ h.DiagCode' = \texttt{0xC001})
\end{aligned}$$



- Add to encoding of each call of `ReqHandler`

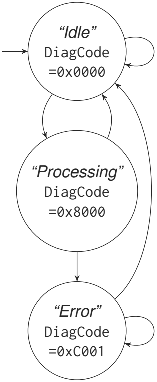
Mode-Space as Call Summary

- Mode-space constrains possible transitions

⇒ Yields call summary $S_{ReqHandler}(\vec{X}_h, \vec{X}'_h)$:

$$
\begin{aligned}
&(h.DiagCode = \texttt{0} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0x8000}) \\
\wedge\ &(h.DiagCode = \texttt{0x8000} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0xC001}) \\
\wedge\ &(h.DiagCode = \texttt{0xC001} \rightarrow h.DiagCode' = \texttt{0} \\
&\qquad \vee\ h.DiagCode' = \texttt{0xC001})
\end{aligned}
$$

- Add to encoding of each call of `ReqHandler`



Benchmarks

- Two groups of PLC programs from PLCopen Safety library
  - elementary modules implementing safety concepts (23 specs)
  - user examples composed of elementary ones (17 specs)
- Elementary modules exhibit mode-semantics via `DiagCode`
- We check invariants taken or derived from PLCopen
- CHC-solving via Z3's Property Directed Reachability (PDR)

Benchmarks

- Two groups of PLC programs from PLCopen Safety library
  - elementary modules implementing safety concepts (23 specs)
  - user examples composed of elementary ones (17 specs)
- Elementary modules exhibit mode-semantics via `DiagCode`
- We check invariants taken or derived from PLCopen
- CHC-solving via Z3's Property Directed Reachability (PDR)

Benchmarks

- Two groups of PLC programs from PLCopen Safety library
  - elementary modules implementing safety concepts (23 specs)
  - user examples composed of elementary ones (17 specs)
- Elementary modules exhibit mode-semantics via `DiagCode`
- We check invariants taken or derived from PLCopen
- CHC-solving via Z3's Property Directed Reachability (PDR)

Benchmarks

- Two groups of PLC programs from PLCopen Safety library
  - elementary modules implementing safety concepts (23 specs)
  - user examples composed of elementary ones (17 specs)
- Elementary modules exhibit mode-semantics via `DiagCode`
- We check invariants taken or derived from PLCopen
- CHC-solving via Z3's Property Directed Reachability (PDR)

Benchmarks

- Two groups of PLC programs from PLCopen Safety library
  - elementary modules implementing safety concepts (23 specs)
  - user examples composed of elementary ones (17 specs)
- Elementary modules exhibit mode-semantics via `DiagCode`
- We check invariants taken or derived from PLCopen
- CHC-solving via Z3's Property Directed Reachability (PDR)

Benchmarks

- Two groups of PLC programs from PLCopen Safety library
  - elementary modules implementing safety concepts (23 specs)
  - user examples composed of elementary ones (17 specs)
- Elementary modules exhibit mode-semantics via `DiagCode`
- We check invariants taken or derived from PLCopen
- CHC-solving via Z3's Property Directed Reachability (PDR)

Results

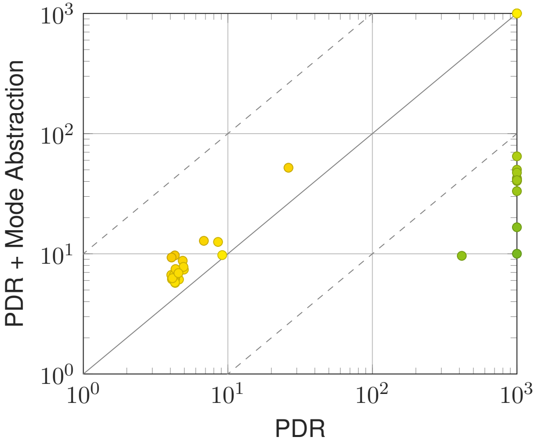


Figure: Time [s] spent on verification of each task

Summary

- Logic control applications exhibit mode-semantics
- Mode-space contains global information
- Approximate mode-abstraction possible via VSA
- CHCs enable
  - compositional characterisation and reasoning
  - uniform characterisation of program, spec & abstraction
- Experiments suggest that
  - mode-abstraction may help significantly
  - overall, mode-abstraction overhead is negligible

Summary

- Logic control applications exhibit mode-semantics
- Mode-space contains global information
- Approximate mode-abstraction possible via VSA
- CHCs enable
  - compositional characterisation and reasoning
  - uniform characterisation of program, spec & abstraction
- Experiments suggest that
  - mode-abstraction may help significantly
  - overall, mode-abstraction overhead is negligible

Summary

- Logic control applications exhibit mode-semantics
- Mode-space contains global information
- Approximate mode-abstraction possible via VSA
- CHCs enable
  - compositional characterisation and reasoning
  - uniform characterisation of program, spec & abstraction
- Experiments suggest that
  - mode-abstraction may help significantly
  - overall, mode-abstraction overhead is negligible

Summary

- Logic control applications exhibit mode-semantics
- Mode-space contains global information
- Approximate mode-abstraction possible via VSA
- CHCs enable
  - compositional characterisation and reasoning
  - uniform characterisation of program, spec & abstraction
- Experiments suggest that
  - mode-abstraction may help significantly
  - overall, mode-abstraction overhead is negligible

Summary

- Logic control applications exhibit mode-semantics
- Mode-space contains global information
- Approximate mode-abstraction possible via VSA
- CHCs enable
  - compositional characterisation and reasoning
  - uniform characterisation of program, spec & abstraction
- Experiments suggest that
  - mode-abstraction may help significantly
  - overall, mode-abstraction overhead is negligible

Summary

- Logic control applications exhibit mode-semantics
- Mode-space contains global information
- Approximate mode-abstraction possible via VSA
- CHCs enable
  - compositional characterisation and reasoning
  - uniform characterisation of program, spec & abstraction
- Experiments suggest that
  - mode-abstraction may help significantly
  - overall, mode-abstraction overhead is negligble

Summary

- Logic control applications exhibit mode-semantics
- Mode-space contains global information
- Approximate mode-abstraction possible via VSA
- CHCs enable
  - compositional characterisation and reasoning
  - uniform characterisation of program, spec & abstraction
- Experiments suggest that
  - mode-abstraction may help significantly
  - overall, mode-abstraction overhead is negligble

Related Tools & Techniques

Mode-Abstraction:

- Predicate abstraction [GS97]
- Abstract-domain selection based on variable usage [Ape+13]

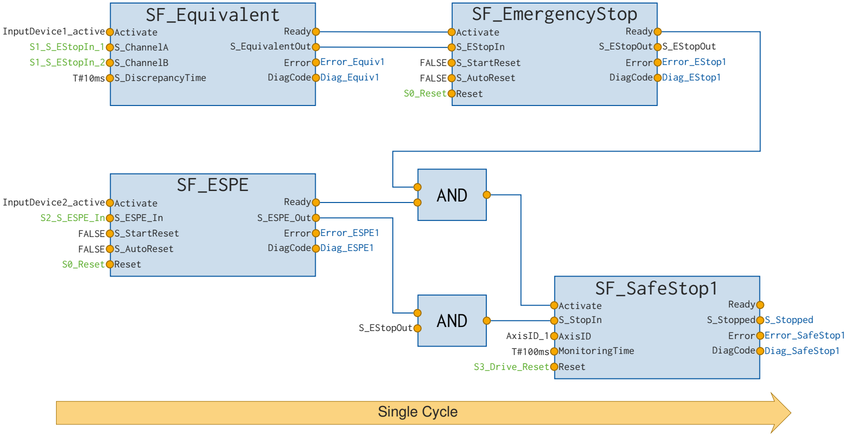
Recent years:

- Decoupling Language Details from Verifier Implementations
- Modular structure & off-the-shelf components

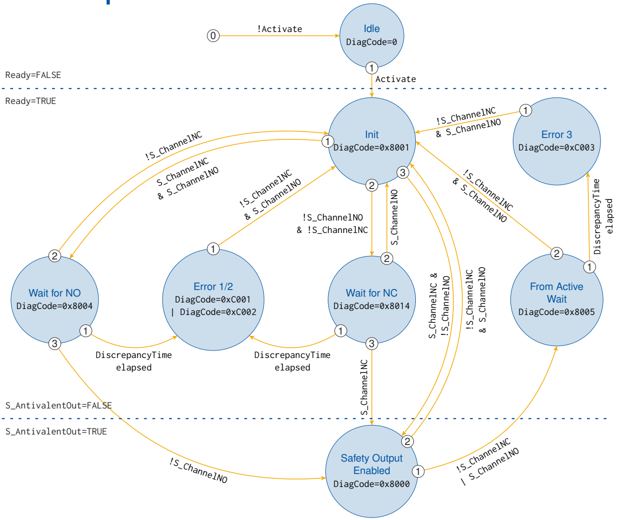
Intermediate Verification Language:

- BOOGIE [Lei08]
  - used by SMACK [RE14]
  - checked by CORRAL [LQL12]
- Constrained Horn Clauses (CHCs) [Bjø+15]
  - used by SEAHORN [Gur+15]
  - checked by SPACER [KGC14] or Z3 [MB08]

PLCopen Safety Application



Typical Block Specification



Intuition for Logical Characterisation

- Reason about program semantics by leveraging Satisfiability Modulo Theories (SMT) solving

⇒ Characterise semantics via first order logic formulae

Condition Formula over the program's variables

$$[\![x > y + 1]\!] = x > y + 1$$

Statement Formula over pre- & post variables instances

$$[\![x := y + 1]\!] = \left(x' = y + 1\right) \land \left(y' = y\right)$$

Procedure Predicate over pre- & post variables instances

$$[\![\mathrm{abs}(x, x')]\!] = \left(x \geq 0 \to x' = x\right) \lor \left(x < 0 \to x' = -x\right)$$

Unclear Characterisation of procedures with loops

Intuition for Logical Characterisation

- Reason about program semantics by leveraging Satisfiability Modulo Theories (SMT) solving

⇒ Characterise semantics via first order logic formulae

Condition Formula over the program's variables

$$[\![x > y + 1]\!] = x > y + 1$$

Statement Formula over pre- & post variables instances

$$[\![x := y + 1]\!] = \left(x' = y + 1\right) \wedge \left(y' = y\right)$$

Procedure Predicate over pre- & post variables instances

$$[\![\mathrm{abs}(x, x')]\!] = \left(x \geq 0 \rightarrow x' = x\right) \vee \left(x < 0 \rightarrow x' = -x\right)$$

Unclear Characterisation of procedures with loops

Intuition for Logical Characterisation

- ▸ Reason about program semantics by leveraging Satisfiability Modulo Theories (SMT) solving

⇒ Characterise semantics via first order logic formulae

Condition Formula over the program's variables

$$[\![x > y + 1]\!] = x > y + 1$$

Statement Formula over pre- & post variables instances

$$[\![x := y + 1]\!] = (x' = y + 1) \land (y' = y)$$

Procedure Predicate over pre- & post variables instances

$$[\![\mathrm{abs}(x, x')]\!] = (x \geq 0 \rightarrow x' = x) \lor (x < 0 \rightarrow x' = -x)$$

Unclear Characterisation of procedures with loops

Intuition for Logical Characterisation

- Reason about program semantics by leveraging Satisfiability Modulo Theories (SMT) solving

⇒ Characterise semantics via first order logic formulae

Condition Formula over the program's variables

$$[\![x > y + 1]\!] = x > y + 1$$

Statement Formula over pre- & post variables instances

$$[\![x := y + 1]\!] = \big(x' = y + 1\big) \wedge \big(y' = y\big)$$

Procedure Predicate over pre- & post variables instances

$$[\![\mathrm{abs}(x, x')]\!] = \big(x \geq 0 \rightarrow x' = x\big) \vee \big(x < 0 \rightarrow x' = -x\big)$$

Unclear Characterisation of procedures with loops

Intuition for Logical Characterisation

- Reason about program semantics by leveraging Satisfiability Modulo Theories (SMT) solving

⇒ Characterise semantics via first order logic formulae

Condition Formula over the program's variables

$$[\![x > y + 1]\!] = x > y + 1$$

Statement Formula over pre- & post variables instances

$$[\![x := y + 1]\!] = \big(x' = y + 1\big) \wedge \big(y' = y\big)$$

Procedure Predicate over pre- & post variables instances

$$[\![\mathrm{abs}(x, x')]\!] = \big(x \geq 0 \rightarrow x' = x\big) \vee \big(x < 0 \rightarrow x' = -x\big)$$

Unclear Characterisation of procedures with loops

Intuition for Logical Characterisation

- Reason about program semantics by leveraging Satisfiability Modulo Theories (SMT) solving

⇒ Characterise semantics via first order logic formulae

Condition Formula over the program's variables

$$[\![x > y + 1]\!] = x > y + 1$$

Statement Formula over pre- & post variables instances

$$[\![x := y + 1]\!] = \big(x' = y + 1\big) \land \big(y' = y\big)$$

Procedure Predicate over pre- & post variables instances

$$[\![\mathrm{abs}(x, x')]\!] = \big(x \geq 0 \rightarrow x' = x\big) \lor \big(x < 0 \rightarrow x' = -x\big)$$

Unclear Characterisation of procedures with loops

Encoding the Running Example

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = (req', in', m', h.data', h.DiagCode', h.res', out')$

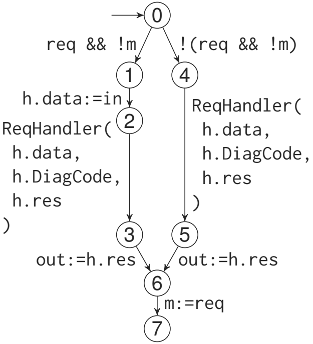
Reachability facts & rules:

- Initial configuration reachable

$$init(\vec{X}) \to p_{Main}(0, \vec{X}, 0, \vec{X})$$

- Transitive reachability

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\, h.data'' = in' \wedge id(X'' \setminus \{h.data''\})$$
$$\to p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



Encoding the Running Example

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = (req', in', m', h.data', h.DiagCode', h.res', out')$

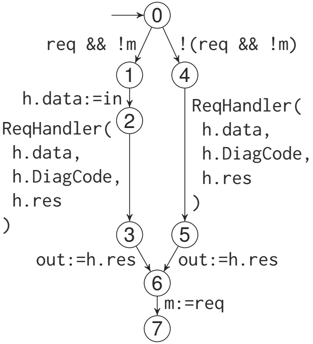
Reachability facts & rules:

- Initial configuration reachable

$$init(\vec{X}) \rightarrow p_{Main}(0, \vec{X}, 0, \vec{X})$$

- Transitive reachability

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\, h.data'' = in' \wedge id(\vec{X}'' \setminus \{h.data''\})$$
$$\rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



Encoding the Running Example

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right)$

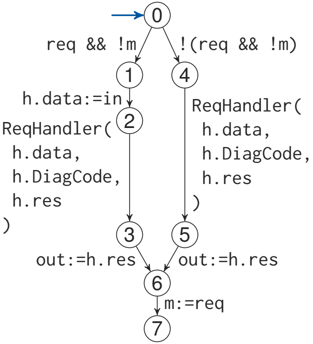
Reachability facts & rules:

- Initial configuration reachable

$$init(\vec{X}) \rightarrow p_{Main}(0, \vec{X}, 0, \vec{X})$$

- Transitive reachability

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\ h.data'' = in' \wedge id(X' \setminus \{h.data'\})$$
$$\rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



Encoding the Running Example

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right)$

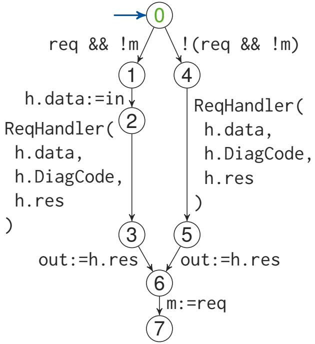
Reachability facts & rules:

- Initial configuration reachable

$$init(\vec{X}) \to p_{Main}(0, \vec{X}, 0, \vec{X})$$

- Transitive reachability

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\ h.data'' = in' \wedge id(X' \setminus \{h.data'\})$$
$$\to p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



Encoding the Running Example

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = (req', in', m', h.data', h.DiagCode', h.res', out')$

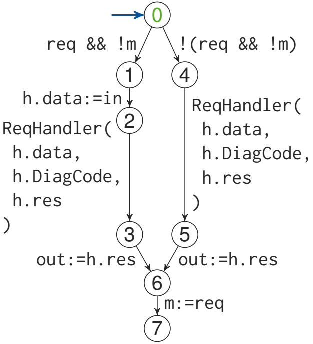
Reachability facts & rules:

- Initial configuration reachable

$$init(\vec{X}) \rightarrow p_{Main}(0, \vec{X}, 0, \vec{X})$$

- Transitive reachability

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\ h.data'' = in' \wedge id(X' \setminus \{h.data'\})$$
$$\rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



Encoding the Running Example

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right)$

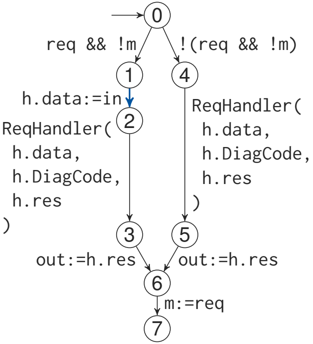
Reachability facts & rules:

- Initial configuration reachable

$$init(\vec{X}) \rightarrow p_{Main}(0, \vec{X}, 0, \vec{X})$$

- Transitive reachability

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\ h.data'' = in' \wedge id(X' \setminus \{h.data'\})$$
$$\rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



Encoding the Running Example

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right)$

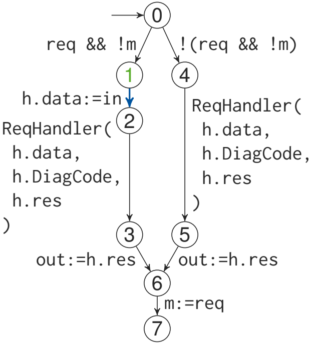
Reachability facts & rules:

- Initial configuration reachable

$$init(\vec{X}) \rightarrow p_{Main}(0, \vec{X}, 0, \vec{X})$$

- Transitive reachability

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\ h.data'' = in' \wedge id(X' \setminus \{h.data'\})$$
$$\rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



Encoding the Running Example

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right)$

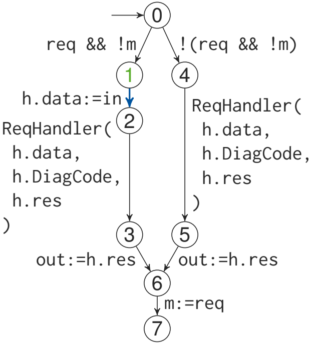
Reachability facts & rules:

- Initial configuration reachable

$$init(\vec{X}) \rightarrow p_{Main}(0, \vec{X}, 0, \vec{X})$$

- Transitive reachability

$$p_{Main}(l, \vec{X}, 1, \vec{X}')$$
$$\wedge\ h.data'' = in' \wedge id(X' \setminus \{h.data'\})$$
$$\rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



Encoding the Running Example

Variables instances:

- $\vec{X} = (req, in, m, h.data, h.DiagCode, h.res, out)$
- $\vec{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right)$

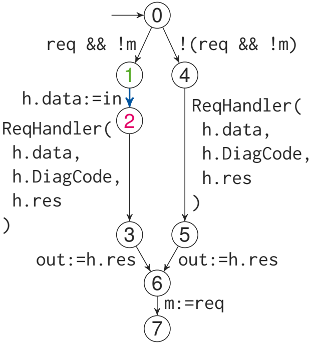
Reachability facts & rules:

- Initial configuration reachable

$$init(\vec{X}) \rightarrow p_{Main}(0, \vec{X}, 0, \vec{X})$$

- Transitive reachability

$$p_{Main}(l, \vec{X}, 1, \vec{X}') \wedge h.data'' = in' \wedge id(X' \setminus \{h.data'\}) \rightarrow p_{Main}(l, \vec{X}, 2, \vec{X}'')$$



Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

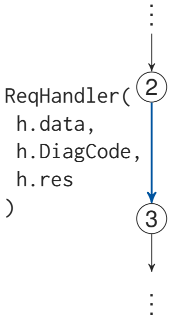
Reachability rules:

- Entry configuration reachable

$$p_{Main}(l, \vec{X}, 2, \vec{X}') \rightarrow p_{ReqHandler}(0, \vec{X}'_h, 0, \vec{X}'_h)$$

- Transitive reachability ([illegible])

$$p_{Main}(l, \vec{X}, 2, \vec{X}')$$
$$\wedge\ p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \wedge \text{[illegible]}(\vec{X}' \setminus \vec{X}''_h)$$
$$\wedge\ \text{[illegible]}\ \vec{X}'_h, \vec{X}''_h$$
$$\rightarrow p_{Main}(l, \vec{X}, 3, \vec{X}'')$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

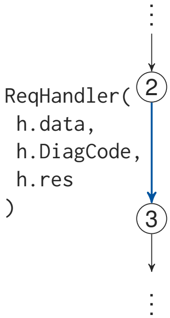
Reachability rules:

- Entry configuration reachable

$$p_{Main}(l, \vec{X}, 2, \vec{X}') \rightarrow p_{ReqHandler}(0, \vec{X}'_h, 0, \vec{X}'_h)$$

- Transitive reachability (& call summary)

$$\begin{aligned} & p_{Main}(l, \vec{X}, 2, \vec{X}') \\ \wedge\ & p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \wedge id(\vec{X}' \setminus \vec{X}'_h) \\ \wedge\ & S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h) \\ \rightarrow\ & p_{Main}(l, \vec{X}, 3, \vec{X}'') \end{aligned}$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

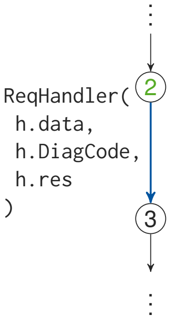
Reachability rules:

- Entry configuration reachable

$$p_{Main}(l, \vec{X}, 2, \vec{X}') \rightarrow p_{ReqHandler}(0, \vec{X}'_h, 0, \vec{X}'_h)$$

- Transitive reachability (& call summary)

$$p_{Main}(l, \vec{X}, 2, \vec{X}')$$
$$\wedge\, p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \wedge id(\vec{X}' \setminus \vec{X}'_h)$$
$$\wedge\, S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h)$$
$$\rightarrow p_{Main}(l, \vec{X}, 3, \vec{X}'')$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

Reachability rules:

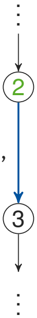
- Entry configuration reachable

$$p_{Main}(l, \vec{X}, 2, \vec{X}') \rightarrow p_{ReqHandler}(0, \vec{X}'_h, 0, \vec{X}'_h)$$

- Transitive reachability (& call summary)

$$\begin{aligned} & p_{Main}(l, \vec{X}, 2, \vec{X}') \\ \wedge\ & p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \wedge id(\vec{X}' \setminus \vec{X}'_h) \\ \wedge\ & S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h) \\ \rightarrow\ & p_{Main}(l, \vec{X}, 3, \vec{X}'') \end{aligned}$$

```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```



Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

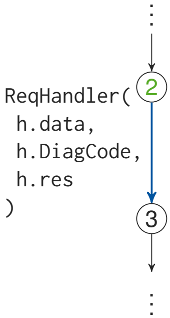
Reachability rules:

- Entry configuration reachable

$$p_{Main}(l, \vec{X}, 2, \vec{X}') \to p_{ReqHandler}(0, \vec{X}'_h, 0, \vec{X}'_h)$$

- Transitive reachability (& call summary)

$$\begin{aligned} & p_{Main}(l, \vec{X}, 2, \vec{X}') \\ \wedge\ & p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \wedge id(\vec{X}' \setminus \vec{X}'_h) \\ \wedge\ & S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h) \\ \to\ & p_{Main}(l, \vec{X}, 3, \vec{X}'') \end{aligned}$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

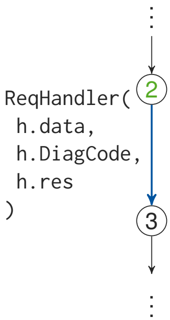
Reachability rules:

- Entry configuration reachable

$$p_{Main}(l, \vec{X}, 2, \vec{X}') \rightarrow p_{ReqHandler}(0, \vec{X}'_h, 0, \vec{X}'_h)$$

- Transitive reachability (& call summary)

$$\begin{aligned} & p_{Main}(l, \vec{X}, 2, \vec{X}') \\ \land\ & p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \land id(\vec{X}' \setminus \vec{X}'_h) \\ \land\ & S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h) \\ \rightarrow\ & p_{Main}(l, \vec{X}, 3, \vec{X}'') \end{aligned}$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

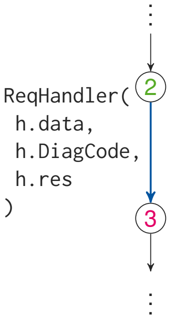
Reachability rules:

- Entry configuration reachable

$$p_{Main}(l, \vec{X}, 2, \vec{X}') \rightarrow p_{ReqHandler}(0, \vec{X}_h', 0, \vec{X}_h')$$

- Transitive reachability (& call summary)

$$\begin{aligned} & p_{Main}(l, \vec{X}, 2, \vec{X}') \\ \wedge\ & p_{ReqHandler}(0, \vec{X}_h', 42, \vec{X}_h'') \wedge id(\vec{X}' \setminus \vec{X}_h') \\ \wedge\ & S_{ReqHandler}(\vec{X}_h', \vec{X}_h'') \\ \rightarrow\ & p_{Main}(l, \vec{X}, 3, \vec{X}'') \end{aligned}$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

Encoding a Call

Variables instances:

$$\vec{X} = (req, in, m, \underbrace{h.data, h.DiagCode, h.res}_{\vec{X}_h}, out)$$

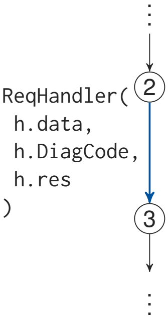
Reachability rules:

- Entry configuration reachable

$$p_{Main}(l, \vec{X}, 2, \vec{X}') \rightarrow p_{ReqHandler}(0, \vec{X}'_h, 0, \vec{X}'_h)$$

- Transitive reachability (& call summary)

$$\begin{aligned} & p_{Main}(l, \vec{X}, 2, \vec{X}') \\ \wedge\ & p_{ReqHandler}(0, \vec{X}'_h, 42, \vec{X}''_h) \wedge id(\vec{X}' \setminus \vec{X}'_h) \\ \wedge\ & S_{ReqHandler}(\vec{X}'_h, \vec{X}''_h) \\ \rightarrow\ & p_{Main}(l, \vec{X}, 3, \vec{X}'') \end{aligned}$$



```
ReqHandler(
 h.data,
 h.DiagCode,
 h.res
)
```

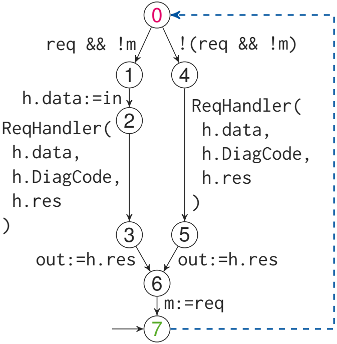
Implementation Details

- Possible cycle-edge encoding

$$p_{Main}(l, \vec{X}, 7, \vec{X}') \wedge id(\vec{X} \setminus \vec{X}_{in})$$

$$\rightarrow p_{Main}(l, \vec{X}, 0, \vec{X}'')$$

- Allow summaries for main block by modelling it like a call
- Reduce number of locations via Large Block Encoding

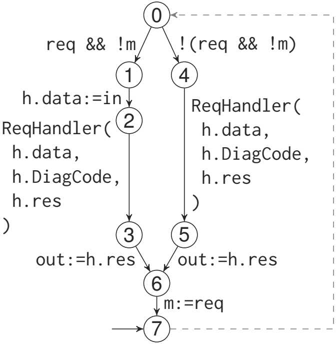


Implementation Details

- Possible cycle-edge encoding

$$p_{Main}(l, \vec{X}, 7, \vec{X}') \wedge id(\vec{X} \setminus \vec{X}_{in})$$
$$\rightarrow p_{Main}(l, \vec{X}, 0, \vec{X}'')$$

- Allow summaries for main block by modelling it like a call
- Reduce number of locations via Large Block Encoding

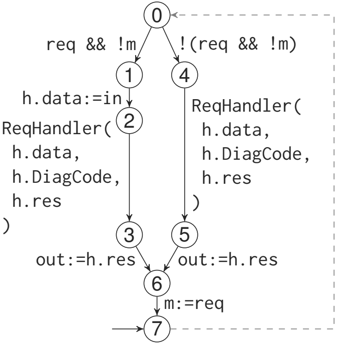


Implementation Details

- Possible cycle-edge encoding

$$p_{Main}(l, \vec{X}, 7, \vec{X}') \land id(\vec{X} \setminus \vec{X}_{in})$$
$$\rightarrow p_{Main}(l, \vec{X}, 0, \vec{X}'')$$

- Allow summaries for main block by modelling it like a call
- Reduce number of locations via Large Block Encoding



References I

[Ape+13] Sven Apel et al. “Domain Types: Abstract-Domain Selection Based on Variable Usage”. In: *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*. 2013, pp. 262–278.

[Bjø+15] Nikolaj Bjørner et al. “Horn Clause Solvers for Program Verification”. In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. 2015, pp. 24–51.

References II

[GS97] Susanne Graf and Hassen Saïdi. “Construction of Abstract State Graphs with PVS”. In: *Computer Aided Verification, 9th International Conference, CAV ’97, Haifa, Israel, June 22-25, 1997, Proceedings*. 1997, pp. 72–83.

[Gur+15] Arie Gurfinkel et al. “The SeaHorn Verification Framework”. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 2015, pp. 343–361.

References III

[KGC14] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. "SMT-Based Model Checking for Recursive Programs". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 2014, pp. 17–34.

[Lei08] Rustan Leino. "This is Boogie 2". In: Microsoft Research, 2008.

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. "Corral: A Solver for Reachability Modulo Theories". In: *Computer-Aided Verification (CAV)*. 2012.

References IV

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 2008, pp. 337–340.

References V

[RE14] Zvonimir Rakamarić and Michael Emmi. “SMACK: Decoupling Source Language Details from Verifier Implementations”. In: *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 106–113.