

# Experiments on automation of formal verification of devices at the binary level

Thomas Lacroix

INSA Lyon  
Soutenance de PFE (Option R&D)

Wednesday, June 19, 2019



# Section 1

## Motivation

# Table of Contents

- 1 Motivation
  - Security critical systems
  - Formal verification
  - Network Interface Controllers (NIC)
- 2 Automatic contract-based verification
  - Pipeline
  - How trustful is it?
  - How powerful is it?
- 3 Proof-producing verification
- 4 Conclusion

## Privacy

- Smartphones
- Smart TVs

## Integrity

- Hospital equipment
- Traffic control systems
- Power plants

## Privacy

- Smartphones
- Smart TVs

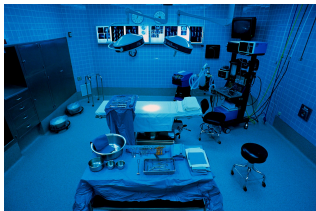
## Integrity

- Hospital equipment
- Traffic control systems
- Power plants

**Problem:** complex systems almost always contain bugs

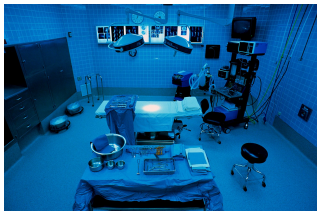
# Security critical systems - vulnerable

# Security critical systems - vulnerable



**Figure:** “It’s Insanely Easy to Hack Hospital Equipment” [7]

# Security critical systems - vulnerable



**Figure:** “It’s Insanely Easy to Hack Hospital Equipment” [7]



**Figure:** “Remote Exploitation of an Unaltered Passenger Vehicle” [3, 5]





MINIX 3







Formal proof [2]:

- The binary code correctly implements its **abstract specification**.
- The specification guarantees **integrity** and **confidentiality**.



Formal proof [2]:

- The binary code correctly implements its **abstract specification**.
- The specification guarantees **integrity** and **confidentiality**.
- **Integrity**: data cannot be *changed* without permission.
- **Confidentiality**: data cannot be *read* without permission.

Proof assumptions [1]:

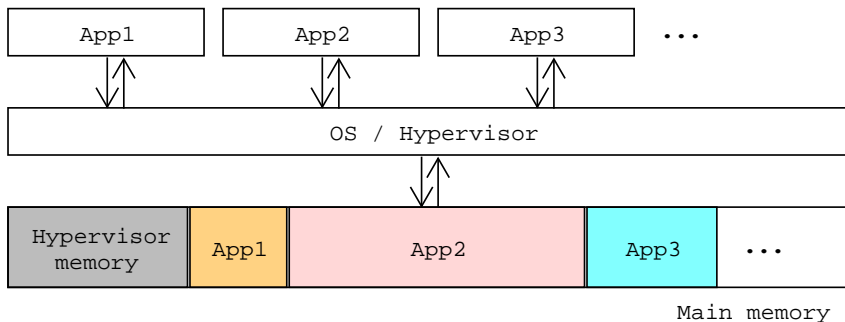




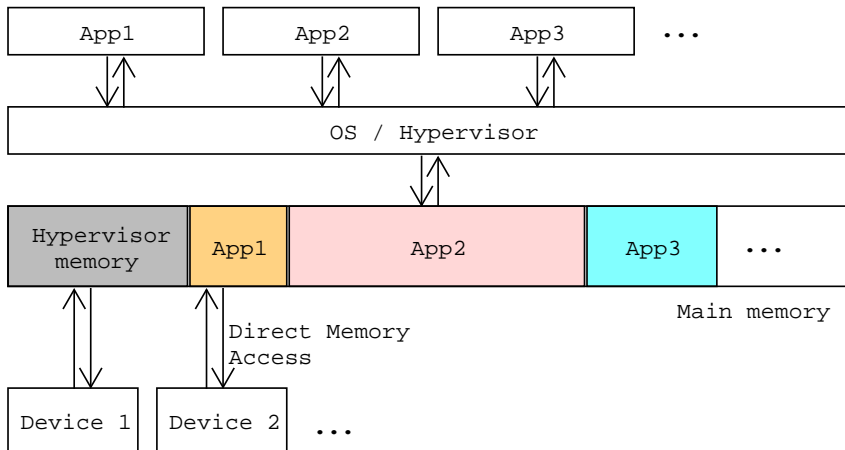
Proof assumptions [1]:

- Use of Direct Memory Access (DMA) is excluded, or only allowed for **trusted drivers that have to be formally verified by the user.**

# What is DMA?



# What is DMA?





# Table of Contents

## 1 Motivation

- Security critical systems
- **Formal verification**
- Network Interface Controllers (NIC)

## 2 Automatic contract-based verification

- Pipeline
- How trustful is it?
- How powerful is it?

## 3 Proof-producing verification

## 4 Conclusion

**Objective:** show absence of errors in modelisation of real systems

**Objective:** show absence of errors in modelisation of real systems

## Formal proof

machine checkable proofs using  
rigorous semantic

Use small reliable kernels

→ produced theorems are trustworthy

*Examples:* HOL4, Coq, Isabelle

**Objective:** show absence of errors in modelisation of real systems

## Formal proof

machine checkable proofs using  
rigorous semantic

Use small reliable kernels  
→ produced theorems are trustworthy

*Examples:* HOL4, Coq, Isabelle

## Non proof-producing verification

specialized programs or procedures  
that check a given property

Classic bug-prone software  
→ need tests, less trustworthy

SMT solvers, model checkers

# Table of Contents

- 1 Motivation
  - Security critical systems
  - Formal verification
  - Network Interface Controllers (NIC)
- 2 Automatic contract-based verification
  - Pipeline
  - How trustful is it?
  - How powerful is it?
- 3 Proof-producing verification
- 4 Conclusion

# Network Interface Controller (NIC)

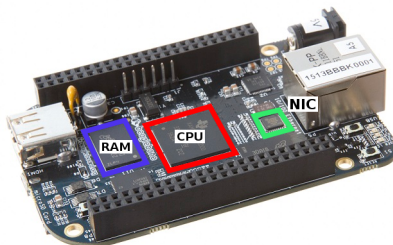
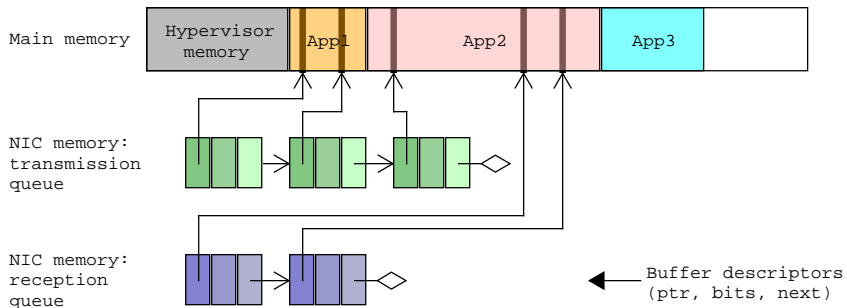
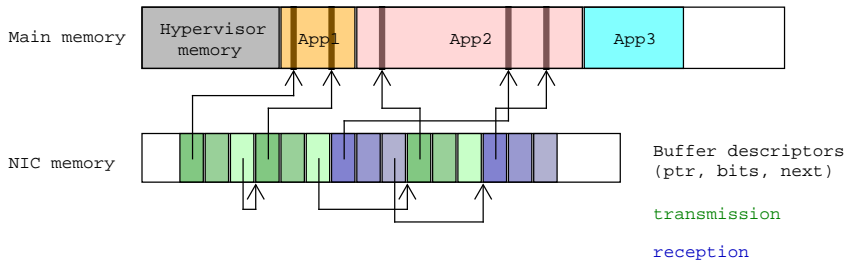


Figure: BeagleBone Black.

# NIC: How it works

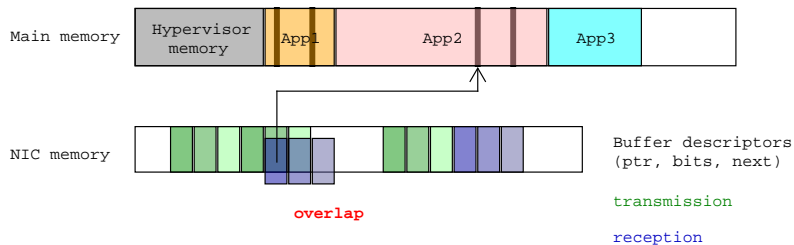


# NIC: How it works

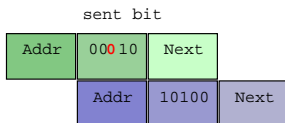
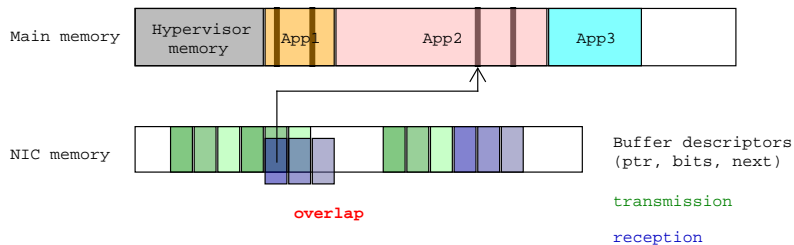




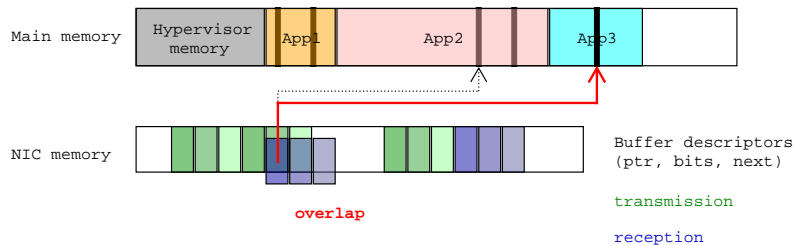
# NIC: How it can fail



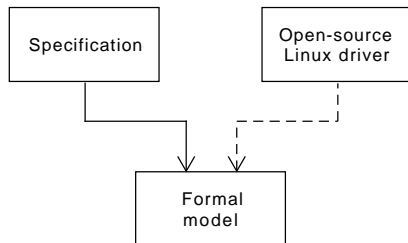
# NIC: How it can fail



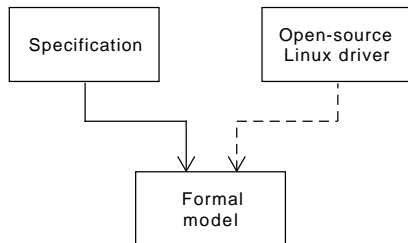
# NIC: How it can fail



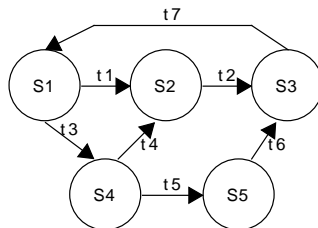
# NIC: How it has been modeled [4]



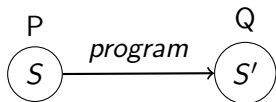
# NIC: How it has been modeled [4]



Transition system:



$$\forall S. P(S) \wedge S' = \text{program}(S) \implies Q(S')$$



$$\{P\} \text{ program } \{Q\}$$

Weakest precondition  $WP$  such that:

$$\{WP\} \text{ program } \{Q\}$$

$$\left( \forall S. P(S) \implies WP(S) \right) \implies \{P\} \text{ program } \{Q\}$$

Weakest precondition  $WP$  such that:

$$\{WP\} \text{ program } \{Q\}$$

$$\left( \forall S. P(S) \implies WP(S) \right) \implies \{P\} \text{ program } \{Q\}$$

$$WP = f(\text{program}, Q)$$



# NIC: What the verification looks like [4]

# NIC: What the verification looks like [4]

Low-level lemmas:

# NIC: What the verification looks like [4]

Low-level lemmas:

- $\{\neg \text{dead} \wedge \text{well\_configured}\} \text{ transition } \{\neg \text{dead}\}$

# NIC: What the verification looks like [4]

Low-level lemmas:

- $\{\neg \text{dead} \wedge \text{well\_configured}\} \text{ transition } \{\neg \text{dead}\}$
- $\{\neg \text{overlapping} \wedge \neg \text{cyclic}\} \text{ transition } \{\neg \text{overlapping}\}$

# NIC: What the verification looks like [4]

Low-level lemmas:

- $\{\neg \text{dead} \wedge \text{well\_configured}\} \text{ transition } \{\neg \text{dead}\}$
- $\{\neg \text{overlapping} \wedge \neg \text{cyclic}\} \text{ transition } \{\neg \text{overlapping}\}$
- $\{\neg \text{overlapping} \wedge \neg \text{cyclic}\} \text{ transition } \{\neg \text{cyclic}\}$

# NIC: What the verification looks like [4]

Low-level lemmas:

- $\{\neg \text{dead} \wedge \text{well\_configured}\} \text{ transition } \{\neg \text{dead}\}$
- $\{\neg \text{overlapping} \wedge \neg \text{cyclic}\} \text{ transition } \{\neg \text{overlapping}\}$
- $\{\neg \text{overlapping} \wedge \neg \text{cyclic}\} \text{ transition } \{\neg \text{cyclic}\}$

Intermediate lemmas:

- *Invariant: rx\_invariant\_well\_defined*
- *Invariant: tx\_invariant\_well\_defined*

# NIC: What the verification looks like [4]

## Low-level lemmas:

- $\{\neg \text{dead} \wedge \text{well\_configured}\} \text{ transition } \{\neg \text{dead}\}$
- $\{\neg \text{overlapping} \wedge \neg \text{cyclic}\} \text{ transition } \{\neg \text{overlapping}\}$
- $\{\neg \text{overlapping} \wedge \neg \text{cyclic}\} \text{ transition } \{\neg \text{cyclic}\}$

## Intermediate lemmas:

- *Invariant: rx\_invariant\_well\_defined*
- *Invariant: tx\_invariant\_well\_defined*

## Security theorems:

- $\forall tx\_bd. \text{readable}(tx\_bd)$
  - $\forall rx\_bd. \text{writable}(rx\_bd)$
- BD = Buffer Descriptor*

**Can we apply traditional software verification techniques and tools to show security properties of hardware devices?**



- Verification platform at binary level
- Centered around its Intermediate Language, BIR
- Features proof-producing tools
  - ▶ Weakest precondition generation

## Section 2

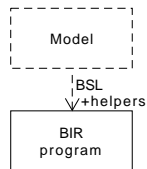
# Automatic contract-based verification

# Table of Contents

- 1 Motivation
  - Security critical systems
  - Formal verification
  - Network Interface Controllers (NIC)
- 2 Automatic contract-based verification
  - Pipeline
    - How trustful is it?
    - How powerful is it?
- 3 Proof-producing verification
- 4 Conclusion

# Contract-based verification pipeline

## 0. Translate the model in BIR

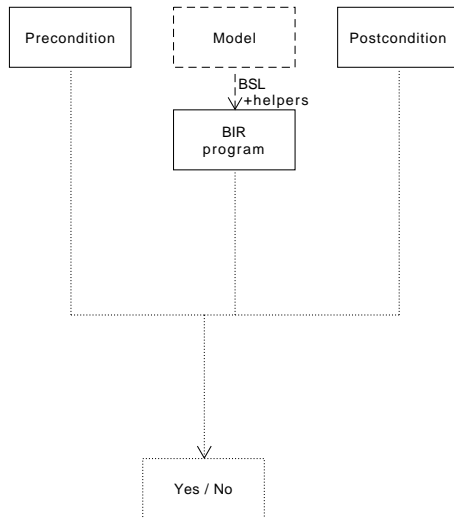


*transition*<sub>BIR</sub>

# Contract-based verification pipeline

0. Translate the model in BIR
1. Formulate a Hoare Triple

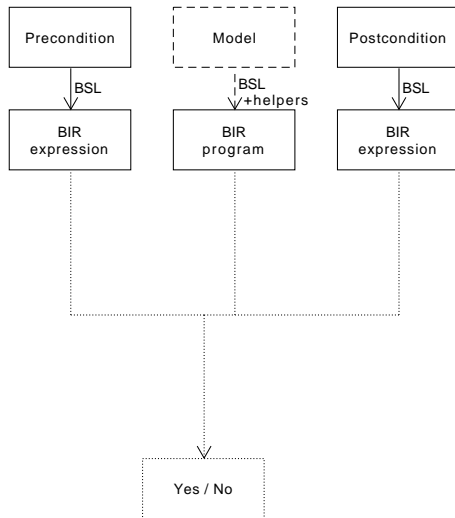
$\{P\} \text{ transition}_{BIR} \{Q\}$



# Contract-based verification pipeline

0. Translate the model in BIR
1. Formulate a Hoare Triple
2. Translate P and Q to BIR

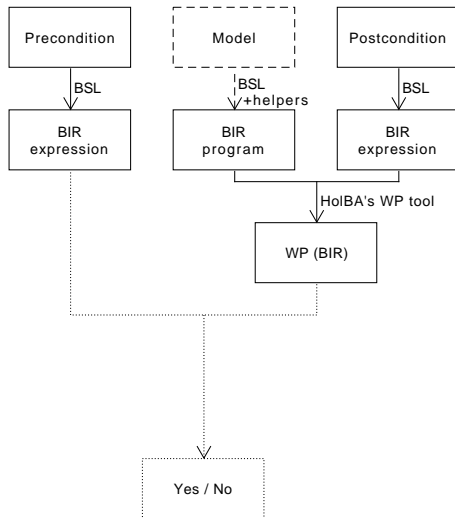
$\{P_{BIR}\} \text{ transition}_{BIR} \{Q_{BIR}\}$



# Contract-based verification pipeline

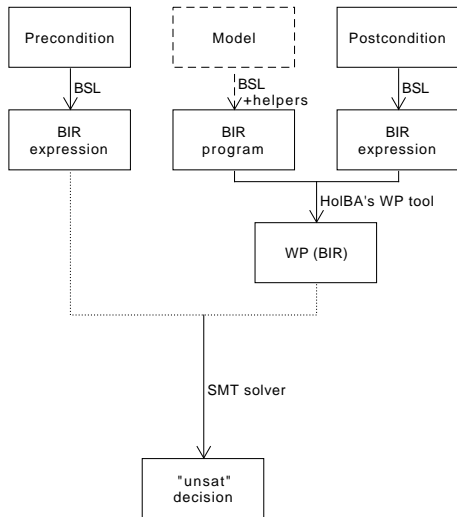
0. Translate the model in BIR
1. Formulate a Hoare Triple
2. Translate P and Q to BIR
3. Generate the WP

$$P_{BIR}(S) \implies WP_{BIR}(S)$$



# Contract-based verification pipeline

0. Translate the model in BIR
1. Formulate a Hoare Triple
2. Translate P and Q to BIR
3. Generate the WP



## Satisfiability Modulo Theories

- external tools
- SMT-LIB 2.0

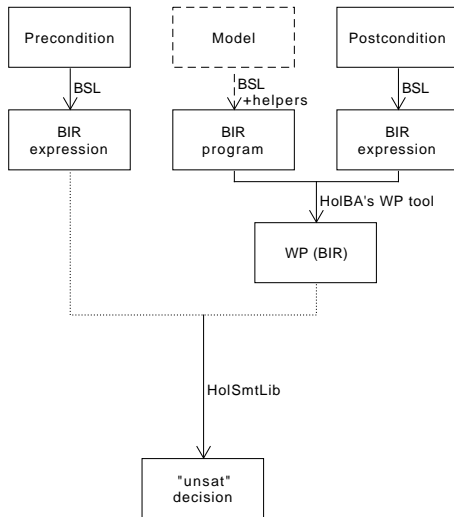


# Contract-based verification pipeline

0. Translate the model in BIR
1. Formulate a Hoare Triple
2. Translate P and Q to BIR
3. Generate the WP

$$\neg(P_{BIR}(S) \implies WP_{BIR}(S))$$

“unsat”?

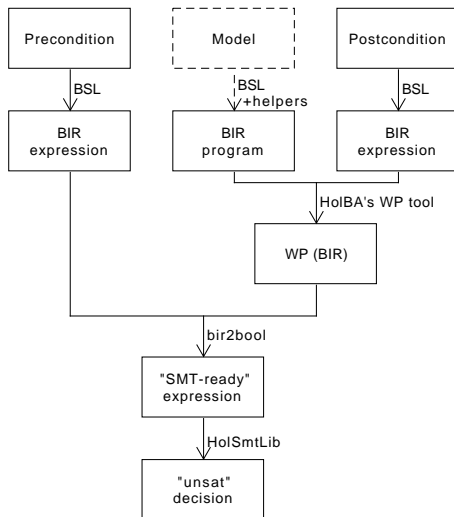


# Contract-based verification pipeline

0. Translate the model in BIR
1. Formulate a Hoare Triple
2. Translate P and Q to BIR
3. Generate the WP
4. Translate the goal into a SMT-compatible expression

$$\neg \left( P(S) \implies WP(S) \right)_{SMT}$$

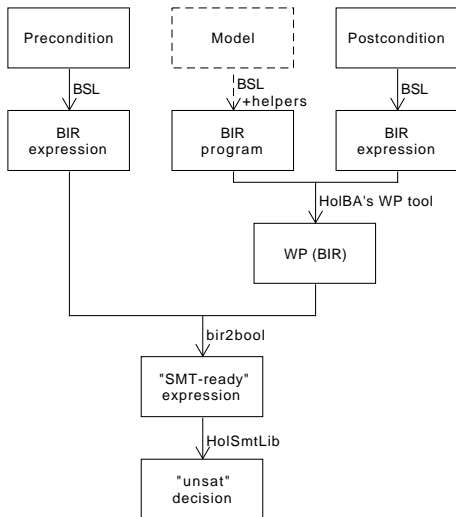
“unsat”?



# Table of Contents

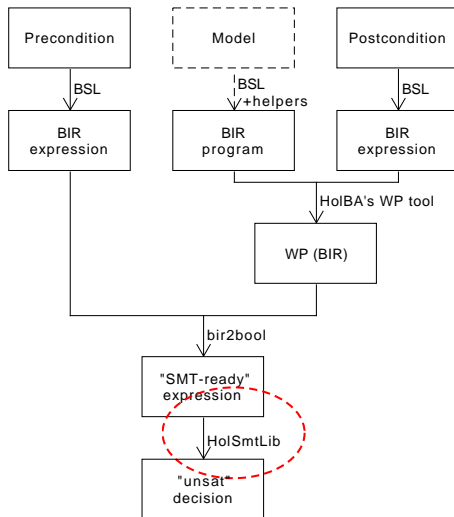
- 1 Motivation
  - Security critical systems
  - Formal verification
  - Network Interface Controllers (NIC)
- 2 Automatic contract-based verification
  - Pipeline
  - How trustful is it?
  - How powerful is it?
- 3 Proof-producing verification
- 4 Conclusion

# How trustful is it?



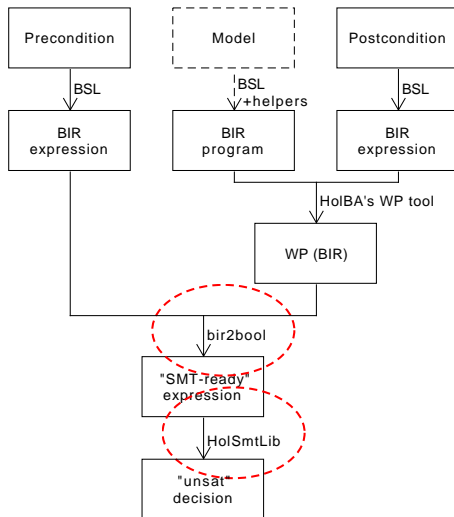
# How trustful is it?

- SMT solvers don't produce proofs



# How trustful is it?

- SMT solvers don't produce proofs
- bir2bool isn't proof-producing



# Table of Contents

- 1 Motivation
  - Security critical systems
  - Formal verification
  - Network Interface Controllers (NIC)
- 2 Automatic contract-based verification
  - Pipeline
  - How trustful is it?
  - How powerful is it?
- 3 Proof-producing verification
- 4 Conclusion

# How powerful is it?



# How powerful is it?

Not proof-producing

Easier non-proof producing platforms exist

# How powerful is it?

Not proof-producing

Easier non-proof producing platforms exist

Limited by SMT solvers' logics

# How powerful is it?

## Not proof-producing

Easier non-proof producing platforms exist

## Limited by SMT solvers' logics

- $\{\neg overlapping \wedge \neg cyclic\} \text{ transition } \{\neg overlapping\}$

# How powerful is it?

## Not proof-producing

Easier non-proof producing platforms exist

## Limited by SMT solvers' logics

- $\{\neg overlapping \wedge \neg cyclic\}$  *transition*  $\{\neg overlapping\}$
- SMT logic: **QF**\_AUFBV  $\rightarrow$  **Q**uantifier-**F**ree

# How powerful is it?

## Not proof-producing

Easier non-proof producing platforms exist

## Limited by SMT solvers' logics

- $\{\neg overlapping \wedge \neg cyclic\} \text{ transition } \{\neg overlapping\}$
- SMT logic: **QF\_AUFBV**  $\rightarrow$  **Quantifier-Free**

## Cannot compose theorems

# How powerful is it?

## Not proof-producing

Easier non-proof producing platforms exist

## Limited by SMT solvers' logics

- $\{\neg overlapping \wedge \neg cyclic\} \text{ transition } \{\neg overlapping\}$
- SMT logic: **QF\_AUFBV**  $\rightarrow$  **Quantifier-Free**

## Cannot compose theorems

- Work in progress in HolBA

## Section 3

# Proof-producing verification

# Goal



→ Some theorems cannot be proved with previous pipeline

# Goal

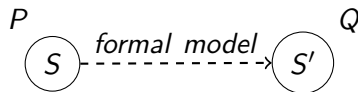
- Some theorems cannot be proved with previous pipeline
- We would like to prove them anyway

# Goal

- Some theorems cannot be proved with previous pipeline
- We would like to prove them anyway
- We want to use them with the current proof

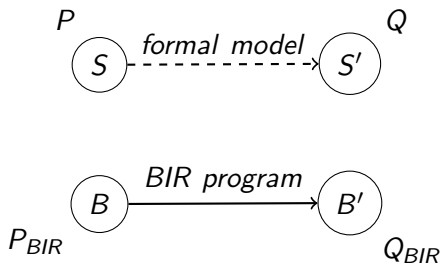
# Goal

- Some theorems cannot be proved with previous pipeline
- We would like to prove them anyway
- We want to use them with the current proof



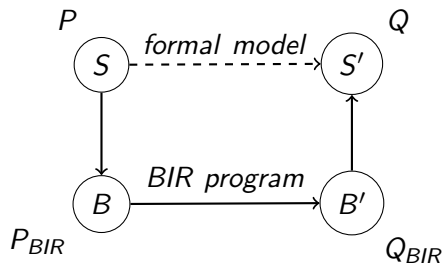
# Goal

- Some theorems cannot be proved with previous pipeline
- We would like to prove them anyway
- We want to use them with the current proof

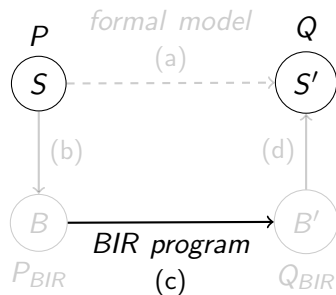


# Goal

- Some theorems cannot be proved with previous pipeline
- We would like to prove them anyway
- We want to use them with the current proof



$$\begin{aligned}
 &\{P\} \text{ BIR\_prog } \{Q\} \\
 &\quad \equiv \\
 &\forall S \ S'. \text{ exec } S \text{ BIR\_prog } S' \\
 &\quad \implies P \ S \implies Q \ S'
 \end{aligned}$$

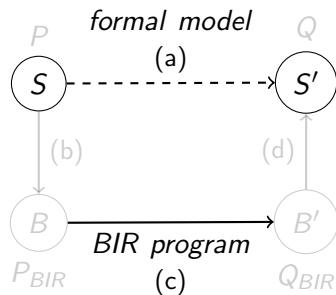


$$\{P\} \text{ BIR\_prog } \{Q\}$$

$$\equiv$$

$$\forall S \ S'. \text{ exec } S \text{ BIR\_prog } S' \\ \implies P \ S \implies Q \ S'$$

$$\forall S \ S'. \text{ exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \\ \forall B \ B'. (B' = \text{BIR\_exec BIR\_prog } B \\ \wedge R \ S \ B) \implies R \ S' \ B'$$



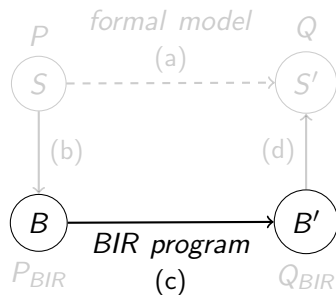


$$\{P\} \text{ BIR\_prog } \{Q\}$$

$$\equiv$$

$$\forall S \ S'. \text{ exec } S \text{ BIR\_prog } S' \\ \implies P \ S \implies Q \ S'$$

$$\forall S \ S'. \text{ exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \\ \forall B \ B'. (B' = \mathbf{BIR\_exec} \text{ BIR\_prog } B \\ \wedge R \ S \ B) \implies R \ S' \ B'$$

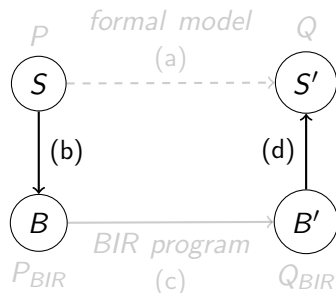


$$\{P\} \text{ BIR\_prog } \{Q\}$$

$$\equiv$$

$$\forall S \ S'. \text{ exec } S \text{ BIR\_prog } S' \\ \implies P \ S \implies Q \ S'$$

$$\forall S \ S'. \text{ exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \\ \forall B \ B'. (B' = \text{BIR\_exec BIR\_prog } B \\ \wedge R \ S \ B) \implies R \ S' \ B'$$

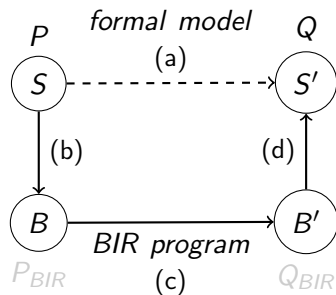


$$\{P\} \text{ BIR\_prog } \{Q\}$$

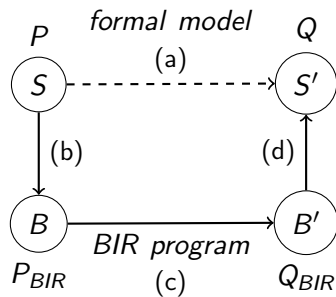
$$\equiv$$

$$\forall S \ S'. \text{ exec } S \text{ BIR\_prog } S' \\ \implies P \ S \implies Q \ S'$$

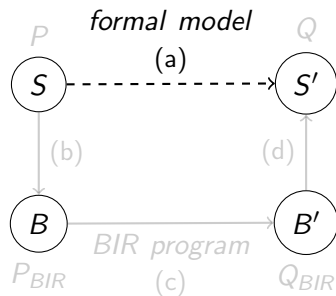
$$\forall S \ S'. \text{ exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \\ \forall B \ B'. (B' = \mathbf{BIR\_exec} \text{ BIR\_prog } B \\ \wedge R \ S \ B) \implies R \ S' \ B'$$



# Proof overview

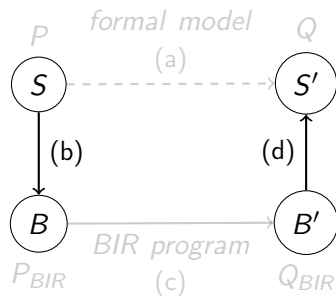


$$\begin{aligned} \text{(a)} \quad & \forall S \ S'. \text{exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \\ & \forall B \ B'. (B' = \text{BIR\_exec BIR\_prog } B \\ & \wedge \mathbf{R} \ S \ B) \implies \mathbf{R} \ S' \ B' \end{aligned}$$



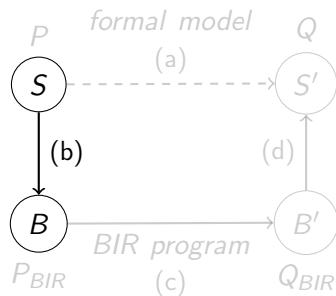
# Proof overview

- (a)  $\forall S S'. \text{exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \forall B B'. (B' = \text{BIR\_exec BIR\_prog } B \wedge \mathbf{R} S B) \implies \mathbf{R} S' B'$
- Relation between  $S$  and  $B$ :  $\mathbf{R} S B$



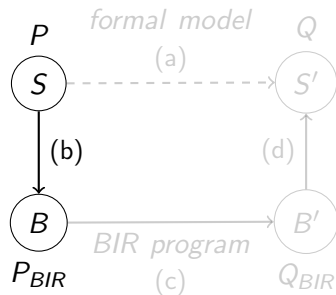
# Proof overview

- (a)  $\forall S S'. \text{exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \forall B B'. (B' = \text{BIR\_exec BIR\_prog } B \wedge \mathbf{R} S B) \implies \mathbf{R} S' B'$
- Relation between  $S$  and  $B$ :  $\mathbf{R} S B$
- (b) Injectivity:  $\forall S. \exists B. \mathbf{R} S B$



# Proof overview

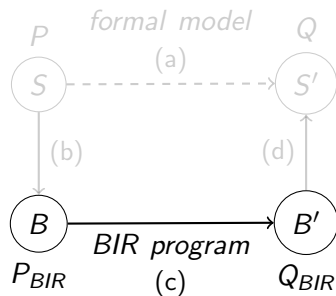
- (a)  $\forall S S'. \text{exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \forall B B'. (B' = \text{BIR\_exec BIR\_prog } B \wedge R S B) \implies R S' B'$
- Relation between  $S$  and  $B$ :  $R S B$
- (b) Injectivity:  $\forall S. \exists B. R S B$
- (b)  $\forall B. (\exists S. R S B \wedge P S) \implies P_{\text{BIR}} B$





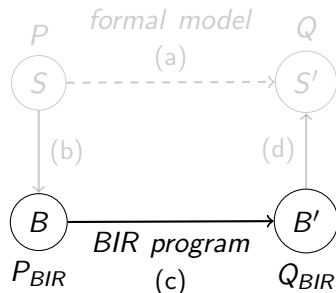
# Proof overview

- (a)  $\forall S S'. \text{exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \forall B B'. (B' = \text{BIR\_exec BIR\_prog } B \wedge R S B) \implies R S' B'$
- Relation between  $S$  and  $B$ :  $R S B$
- (b) Injectivity:  $\forall S. \exists B. R S B$
- (b)  $\forall B. (\exists S. R S B \wedge P S) \implies P_{\text{BIR}} B$
- (c)  $\forall B B'. (P_{\text{BIR}} B \wedge B' = \text{BIR\_exec BIR\_prog } B) \implies Q_{\text{BIR}} B'$



# Proof overview

- (a)  $\forall S S'. \text{exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \forall B B'. (B' = \text{BIR\_exec BIR\_prog } B \wedge \mathbf{R} S B) \implies \mathbf{R} S' B'$ 
  - Relation between  $S$  and  $B$ :  $\mathbf{R} S B$
- (b) Injectivity:  $\forall S. \exists B. \mathbf{R} S B$
- (b)  $\forall B. (\exists S. \mathbf{R} S B \wedge \mathbf{P} S) \implies \mathbf{P}_{\text{BIR}} B$
- (c)  $\{P_{\text{BIR}}\} \text{ BIR program } \{Q_{\text{BIR}}\}$



# Proof overview

$$(a) \quad \forall S \ S'. \text{exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \\ \forall B \ B'. (B' = \mathbf{BIR\_exec} \text{ BIR\_prog } B \\ \wedge \mathbf{R} \ S \ B) \implies \mathbf{R} \ S' \ B'$$

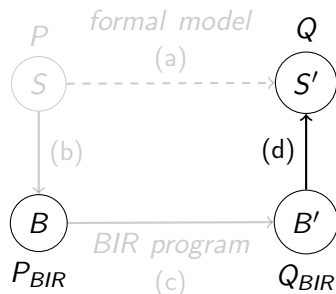
– Relation between  $S$  and  $B$ :  $\mathbf{R} \ S \ B$

$$(b) \text{ Injectivity: } \forall S. \exists B. \mathbf{R} \ S \ B$$

$$(b) \quad \forall B. (\exists S. \mathbf{R} \ S \ B \wedge \mathbf{P} \ S) \implies \mathbf{P}_{\mathbf{BIR}} \ B$$

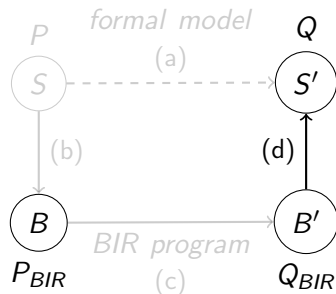
$$(c) \quad \{P_{\mathbf{BIR}}\} \text{ BIR program } \{Q_{\mathbf{BIR}}\}$$

$$(d) \quad \forall B'. \mathbf{Q}_{\mathbf{BIR}} \ B' \implies \\ (\forall S \ S'. B. \mathbf{P}_{\mathbf{BIR}} \ B \wedge \mathbf{R} \ S \ B \wedge \mathbf{R} \ S' \ B' \\ \implies \mathbf{Q} \ S \ S')$$



# Proof overview

Notation:  $P_{BIR} B \stackrel{def}{=} BIR\_eval P_{BIR} B$



# Proof overview

$$(a) \quad \forall S \ S'. \text{exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \\ \forall B \ B'. (B' = \mathbf{BIR\_exec} \text{ BIR\_prog } B \\ \wedge \mathbf{R} \ S \ B) \implies \mathbf{R} \ S' \ B'$$

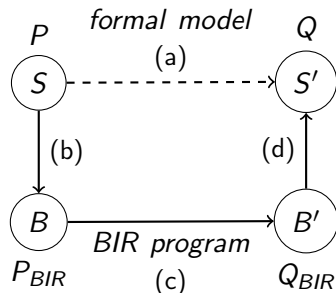
– Relation between  $S$  and  $B$ :  $\mathbf{R} \ S \ B$

$$(b) \text{ Injectivity: } \forall S. \exists B. \mathbf{R} \ S \ B$$

$$(b) \quad \forall B. (\exists S. \mathbf{R} \ S \ B \wedge \mathbf{P} \ S) \implies \mathbf{P}_{\mathbf{BIR}} \ B$$

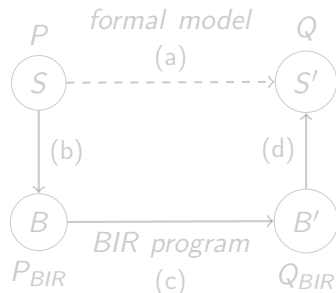
$$(c) \quad \{P_{\mathbf{BIR}}\} \text{ BIR program } \{Q_{\mathbf{BIR}}\}$$

$$(d) \quad \forall B'. \mathbf{Q}_{\mathbf{BIR}} \ B' \implies \\ (\forall S \ S'. B. \mathbf{P}_{\mathbf{BIR}} \ B \wedge \mathbf{R} \ S \ B \wedge \mathbf{R} \ S' \ B' \\ \implies \mathbf{Q} \ S \ S')$$



# Proof overview

1.  $\forall S S'. \text{exec } S \text{ BIR\_prog } S' \stackrel{\text{def}}{=} \forall B B'. (B' = \text{BIR\_exec BIR\_prog } B \wedge \mathbf{R} S B) \implies \mathbf{R} S' B'$
2. Relation between  $S$  and  $B$ :  $\mathbf{R} S B$
3. Injectivity:  $\forall S. \exists B. \mathbf{R} S B$
4.  $\forall B. (\exists S. \mathbf{R} S B \wedge \mathbf{P} S) \implies \mathbf{P}_{\text{BIR}} B$
5.  $\{P_{\text{BIR}}\} \text{BIR program } \{Q_{\text{BIR}}\}$
6.  $\forall B'. \mathbf{Q}_{\text{BIR}} B' \implies (\forall S S'. B. \mathbf{P}_{\text{BIR}} B \wedge \mathbf{R} S B \wedge \mathbf{R} S' B' \implies \mathbf{Q} S S')$



# Proof overview - automate?

Theorem	Length of proof (LoC)	Ease to automate
1. $\text{def } S \rightarrow S' (a)$	–	<i>Hard? (lifter)</i>
2. $\text{def relation } (R)$	–	Easy
3. Injectivity	10	Very easy
4. $P \rightarrow P_{BIR} (b)$	4	Very easy
5. Hoare Triple $(c)$	151	<i>Medium? *2</i>
6. $Q_{BIR} \rightarrow S (d)$	48	Should be easy *1

# Proof overview - automate?

Theorem	Length of proof (LoC)	Ease to automate
1. $\text{def } S \rightarrow S' (a)$	–	<i>Hard? (lifter)</i>
2. $\text{def relation } (R)$	–	Easy
3. Injectivity	10	Very easy
4. $P \rightarrow P_{BIR} (b)$	4	Very easy
5. Hoare Triple $(c)$	151	<i>Medium? *2</i>
6. $Q_{BIR} \rightarrow S (d)$	48	Should be easy *1

\*1 Need 2 simple tactics



# Proof overview - automate?

Theorem	Length of proof (LoC)	Ease to automate
1. $\text{def } S \rightarrow S' (a)$	–	<i>Hard? (lifter)</i>
2. $\text{def relation } (R)$	–	Easy
3. Injectivity	10	Very easy
4. $P \rightarrow P_{BIR} (b)$	4	Very easy
5. Hoare Triple $(c)$	151	<i>Medium? *2</i>
6. $Q_{BIR} \rightarrow S (d)$	48	Should be easy *1

\*1 Need 2 simple tactics

\*2 Need smart tactics (multi-pass, goal aware)

## Section 4

### Conclusion

# Conclusion

- Automation is feasible
- Can reduce proof lengths and complexity
- Trustworthy if proof-producing

# Questions

- [1] [Is seL4 proven secure? | FAQ | seL4 docs.](#)
- [2] [What is proved and what is assumed | seL4.](#)
- [3] [Andy Greenberg.](#)  
Hackers remotely kill a jeep on the highway—with me in it.
- [4] [Jonas Haglund.](#)  
Formal verification of systems software.
- [5] [Dr Charlie Miller and Chris Valasek.](#)  
Remote exploitation of an unaltered passenger vehicle.
- [6] [Thomas Tuerk.](#)  
Interactive theorem proving (ITP) course.
- [7] [Kim Zetter.](#)  
It's insanely easy to hack hospital equipment.

# HolBA overview

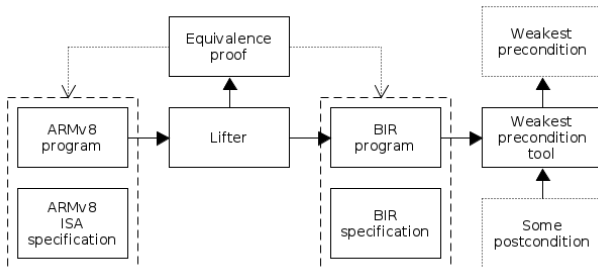


Figure: Overview of the HolBA framework (lifter and WP tool)

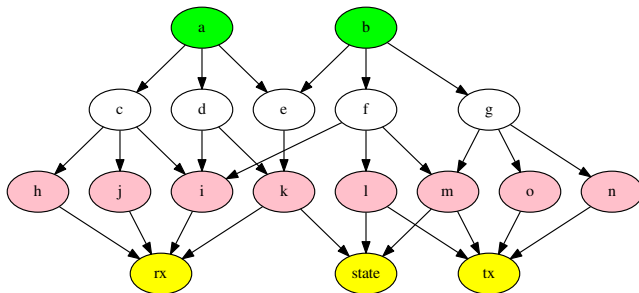


Figure: Fringe of an ideally-shaped proof

# Pipeline public interface

```
val thm = prove_contract "cjmp"  
  cjmp_prog_def  
  (* Precondition *) (blabel_str "entry", btrue)  
  (* Postcondition *) (  
    [blabel_str "end"],  
    beq ((bden o bvarimm32) "y", bconst32 100)  
  )
```



# BSL: BIR Simple Language

```
bite (  
  borl [  
    ble ((bden o bvarimm64) "x", bconst64 100),  
    bnot (ble (bplus ((bden o bvarimm64) "y", bconst64 1),  
                bconst64 10)),  
    ble (bplus ((bden o bvarimm64) "x",  
                (bden o bvarimm64) "y"),  
          bconst64 20)  
  ],  
  bmult ((bden o bvarimm64) "x", bconst64 2),  
  bplus (bmult ((bden o bvarimm64) "x", bconst64 3),  
          bconst64 1)  
)
```

# BIR pretty-printer - disabled

```
BExp_IfThenElse
  (BExp_BinExp BIExp_Or
    (BExp_BinExp BIExp_Or
      (BExp_BinPred BIExp_LessOrEqual
        (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 100w)))
      (BExp_UnaryExp BIExp_Not
        (BExp_BinPred BIExp_LessOrEqual
          (BExp_BinExp BIExp_Plus (BExp_Den (BVar "y" (BType_Imm Bit64)))
            (BExp_Const (Imm64 1w))) (BExp_Const (Imm64 10w))))))
    (BExp_BinPred BIExp_LessOrEqual
      (BExp_BinExp BIExp_Plus (BExp_Den (BVar "x" (BType_Imm Bit64)))
        (BExp_Den (BVar "y" (BType_Imm Bit64)))) (BExp_Const (Imm64 20w)))
    (BExp_BinExp BIExp_Mult (BExp_Den (BVar "x" (BType_Imm Bit64)))
      (BExp_Const (Imm64 2w)))
    (BExp_BinExp BIExp_Plus
      (BExp_BinExp BIExp_Mult (BExp_Den (BVar "x" (BType_Imm Bit64)))
        (BExp_Const (Imm64 3w))) (BExp_Const (Imm64 1w)))
```

# BIR pretty-printer - enabled






```
BExp_If
  (BExp_Or
    (BExp_LessOrEqual
      (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 100w)))
    (BExp_Not
      (BExp_LessOrEqual
        (BExp_Plus
          (BExp_Den (BVar "y" (BType_Imm Bit64))) (BExp_Const (Imm64 1w)))
          (BExp_Const (Imm64 10w))))
    (BExp_LessOrEqual
      (BExp_Plus
        (BExp_Den (BVar "x" (BType_Imm Bit64)))
        (BExp_Den (BVar "y" (BType_Imm Bit64))))
      (BExp_Const (Imm64 20w))))
BExp_Then
  (BExp_Mult (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 2w)))
BExp_Else
  (BExp_Plus
    (BExp_Mult
      (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 3w)))
    (BExp_Const (Imm64 1w)))
```

# Exception pretty-printer and LogLib

```
[TRACE @ nic_helpersLib::prove_p_imp_wp] smt_ready_tm:
~(if (nic_dead = 0w) ^ (nic_init_state = 2w) then 1w else 0w) ||
~(if (nic_init_state = 1w then 1w else 0w) || if 1w = 0w then 1w else 0w) &&
((if (nic_init_state = 1w then 1w else 0w) ||
~(if (nic_init_state = 2w then 1w else 0w) || if (nic_dead = 0w then 1w else 0w) &&
((if (nic_init_state = 2w then 1w else 0w) ||
~(if (nic_init_state = 3w then 1w else 0w) || if 1w = 0w then 1w else 0w) &&
((if (nic_init_state = 3w then 1w else 0w) ||
~(if (nic_init_state = 4w then 1w else 0w) || if 1w = 0w then 1w else 0w) &&
if (nic_init_state = 4w) v (1w = 0w) then 1w else 0w))) =
1w
Handled exception: [init_automaton_doesnt_die] Z3_ORACLE_PROVE failed
HOL_ERR:
- Structure: nic_helpersLib
- Function: prove_p_imp_wp
- Message: at Z3.Z3_SMT_Oracle:
Z3 not configured: set the HOL4_Z3_EXECUTABLE environment variable to point to the Z3 executable file.

[DEBUG @ nic_helpersLib::prove_p_imp_wp] Asking Z3 for a SAT model...
[DEBUG @ nic_helpersLib::prove_p_imp_wp] Failed to compute a SAT model. Ignoring.
error in quse /NOBACKUP/tholac/holba-reborn/examples/nic/test-early-wp.sml : HOL_ERR {message = "at Z3.Z3_SMT_Oracle:\nZ3 not configured: set the HOL4_Z3_EXECUTABLE environment variable to point to the Z3 executable file.", origin_function = "prove_p_imp_wp", origin_structure = "nic_helpersLib"}
error in load test-early-wp : HOL_ERR {message = "at Z3.Z3_SMT_Oracle:\nZ3 not configured: set the HOL4_Z3_EXECUTABLE environment variable to point to the Z3 executable file.", origin_function = "prove_p_imp_wp", origin_structure = "nic_helpersLib"}
Uncaught exception: HOL_ERR {message = "at Z3.Z3_SMT_Oracle:\nZ3 not configured: set the HOL4_Z3_EXECUTABLE environment variable to point to the Z3 executable file.", origin_function = "prove_p_imp_wp", origin_structure = "nic_helpersLib"}
```

# Continuous Integration (CI) - tests

	<b>All checks have passed</b> 2 successful checks	<a href="#">Hide all checks</a>
	 <b>Travis CI - Branch</b> Successful in 33m — Build Passed	<a href="#">Details</a>
	 <b>Travis CI - Pull Request</b> Successful in 67m — Build Passed	<a href="#">Details</a>

# Continuous Integration (CI) - static analysis

holba-bot commented 5 days ago

Results of: `scripts/ci/static-analysis.sh`

---

Grep-cheat analysis:

- ▶ Found 23 occurrences of `cheat` in `src/` and 0 in `examples/`.

---

Grep-todo analysis:

- ▶ Found 52 occurrences of `TODO/FIXME` in `src/` and 0 in `examples/`.

---

**Note:** I'm a script, and I'm simple, so I may be missing something or show false positives. You can review the script [here](#).