# Ultimate Referee, Ultimate Automizer, and Incremental Verification

Matthias Heizmann

University of Freiburg

CPAchecker Workshop 2019

- Running Example and Floyd-Hoare Annotations
- Ultimate Referee

  A strict proof checker.
- Trace Abstraction

  The verification approach of Ultimate Automizer
- Incremental Verification Using Trace Abstraction

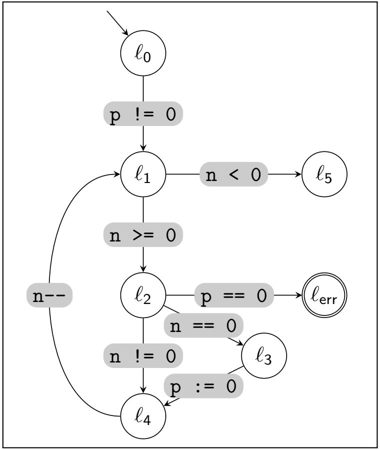
# Running Example and Floyd-Hoare Annotation

```
ℓ0:  assume p != 0;

ℓ1:  while(n >= 0)
     {
ℓ2:      assert p != 0;

         if(n == 0)
         {
ℓ3:          p := 0;
         }
ℓ4:      n--;
     }
```

pseudocode




control flow graph

# Running Example and Floyd-Hoare Annotation

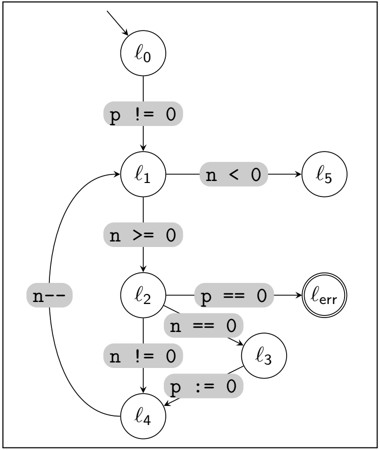
**Definition:**

$\{\boxed{\varphi}\}$ `st` $\{\boxed{\varphi'}\}$ is valid Hoare triple
iff
if program is in state that satisfies $\varphi$
and program executes *st*
then program is in a state that satisfies $\varphi'$

**Example:**

$\{\boxed{p \neq 0 \vee n = -1}\}$ `n>=0` $\{\boxed{p \neq 0}\}$

is a valid Hoare triple




control flow graph

# Running Example and Floyd-Hoare Annotation

**Definition:**

A Floyd-Hoare annotation is a mapping that assigns each location $\ell_i$ a formula $\varphi_i$ such that there is an edge

$$\boxed{\varphi_i}\ \ell_i \xrightarrow{\texttt{st}} \ell_j\ \boxed{\varphi_j}$$

only if the Hoare triple

$$\{\boxed{\varphi}\}\ \texttt{st}\ \{\boxed{\varphi'}\}$$

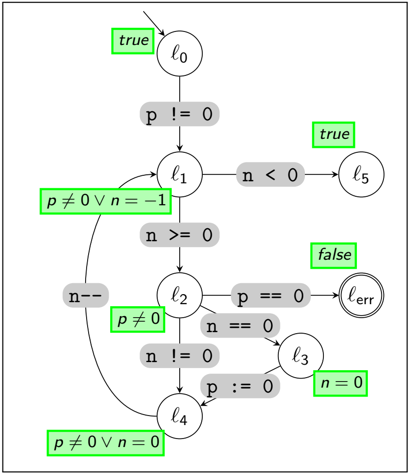
is valid.

**Proposition:**

Given a program $\mathcal{P}$, if there is a Floyd-Hoare annotation such that

- every initial location is labeled with *true* and
- every error location is labeled with *false*

then $\mathcal{P}$ is correct.




control flow graph

# Outline

- Running Example and Floyd-Hoare Annotations
- Ultimate Referee

  A strict proof checker.
- Trace Abstraction

  The verification approach of Ultimate Automizer
- Incremental Verification Using Trace Abstraction

Correctness Witnesses: Control-flow graph annotated by invariants

- not required to annoted every location
- invariants to not have to be inductive
- invariants do not have to be sufficient

Correctness Witnesses: Control-flow graph annotated by invariants

- not required to annoted every location
- invariants to not have to be inductive
- invariants do not have to be sufficient

Shortcomings of Ultimate Automizer as Witness validator

- Different tools have different notions of a control-flow graph we cannot always match invariants to the intended location.

Obstacles

- procedure entry values

Obstacles

- procedure entry values
- valid memory

Obstacles

- procedure entry values
- valid memory
- programs with gotos

# Outline

- Running Example and
  Floyd-Hoare Annotations
- Ultimate Referee

  A strict proof checker.
- Trace Abstraction

  The verification approach of Ultimate Automizer
- Incremental Verification
  Using Trace Abstraction

# Trace Abstraction

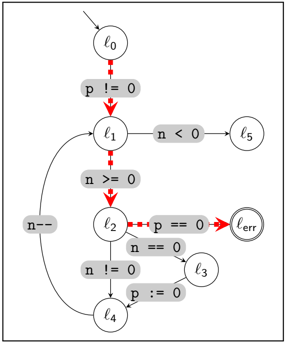
Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Refinement of Trace Abstraction". In: *SAS*. vol. 5673. Lecture Notes in Computer Science. Springer, 2009, pp. 69–85

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Nested interpolants". In: *POPL*. ACM, 2010, pp. 471–482

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Software Model Checking for People Who Love Automata". In: *CAV*. vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 36–52

# Trace Abstraction: Basic Notions

- trace
  sequence of statements
- error trace
  labeling along path from initial location to error location
- infeasible trace
  trace $\pi$ such that Hoare triple




program $\mathcal{P}$

| examples | infeasible | feasible |
|---|---|---|
| error trace of $\mathcal{P}$ | `p != 0` `n >= 0` `p == 0` | ? |
| not error trace of $\mathcal{P}$ | `n == 0` `n--` `n == 0` | `n >= 0` `n--` `n == 0` |

# Trace Abstraction: Approach

Show that every error trace is infeasible.

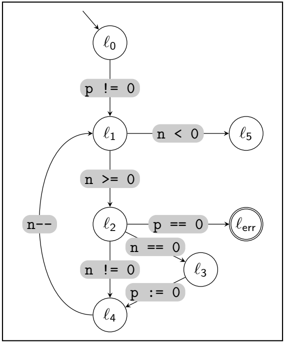
Decompose infeasible error traces into sets
such that there is a "simple" infeasibility proof for each set.

# Trace Abstraction: Approach

Show that every error trace is infeasible.

Decompose infeasible error traces into sets
such that there is a "simple" infeasibility proof for each set.

- Reason 1: If we assume that `p` is not 0 and do not modify `p` then `p` cannot be 0.
- Reason 2: If we assume that `n` is 0 and we decrement `n` then `n` cannot be non-negative.



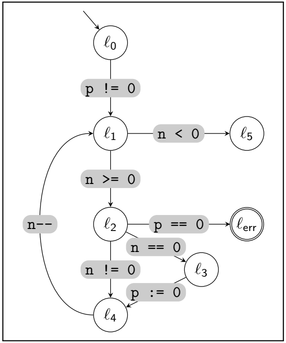

program $\mathcal{P}$

# Trace Abstraction: Technical Implementation

Implementation based on automata theory

Set of statements:
alphabet of formal language
here: $\Sigma = \{$ `p != 0`, `n >= 0`, `n == 0`, `p := 0`, `n != 0`, `p == 0`, `n--`, `n < 0` $\}$

- Set of traces:
  automaton over the alphabet of statements
- Control flow graph:
  automaton over the alphabet of statements
- Error location:
  accepting state of this automaton
- Error trace of program: word accepted by this automaton




program $\mathcal{P}$

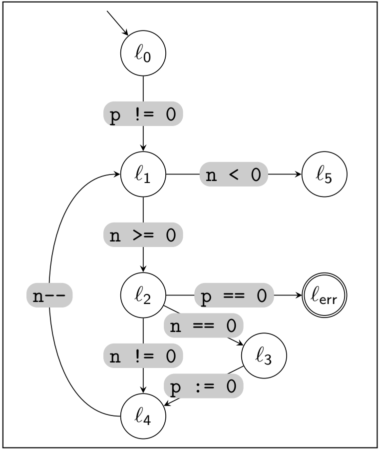
# Trace Abstraction: Example

```
ℓ0:  assume p != 0;

ℓ1:  while(n >= 0)
     {
ℓ2:      assert p != 0;

         if(n == 0)
         {
ℓ3:          p := 0;
         }
ℓ4:      n--;
     }
```

pseudocode




control flow graph

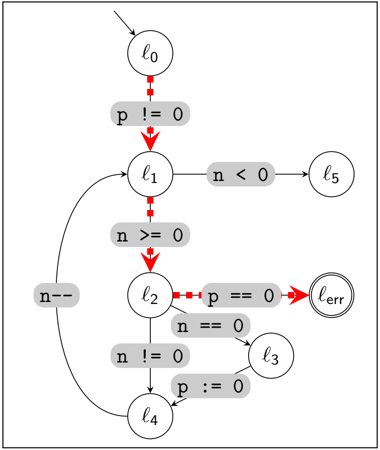
# Trace Abstraction: Example

```
ℓ0:  assume p != 0;

ℓ1:  while(n >= 0)
     {
ℓ2:      assert p != 0;

         if(n == 0)
         {
ℓ3:          p := 0;
         }
ℓ4:      n--;
     }
```

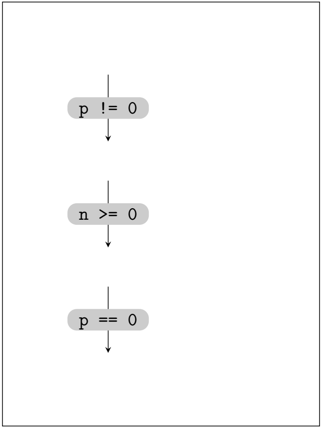
pseudocode




control flow graph

# Trace Abstraction: Example

1. take trace $\pi_1$

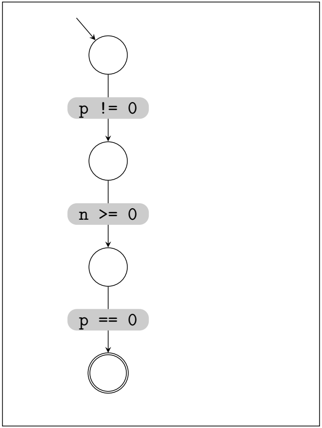


# Trace Abstraction: Example

1. take trace $\pi_1$
2. consider trace as program $\mathcal{A}_1$

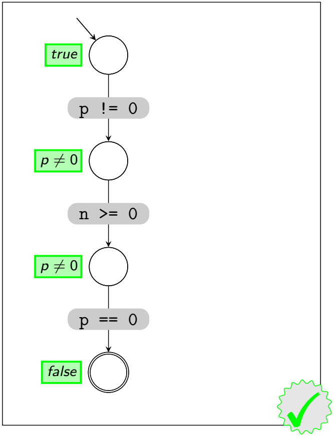
```
assume p != 0;
assume n >= 0;
assert p != 0;
```

pseudocode of $\mathcal{A}_1$



# Trace Abstraction: Example

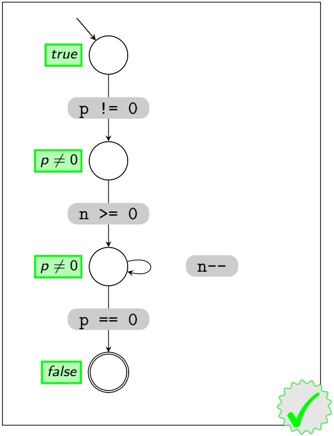
1. take trace $\pi_1$
2. consider trace as program $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$



# Trace Abstraction: Example

1. take trace $\pi_1$
2. consider trace as program $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$
4. generalize program $\mathcal{A}_1$
   - add transitions

$\{\boxed{p \neq 0}\}$ `n--` $\{\boxed{p \neq 0}\}$ is valid Hoare triple

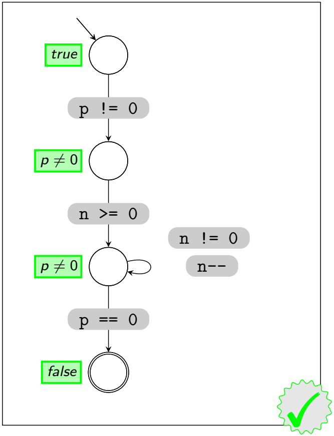


# Trace Abstraction: Example

1. take trace $\pi_1$
2. consider trace as program $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$
4. generalize program $\mathcal{A}_1$
   - add transitions

$\{\boxed{p \neq 0}\}$ `n--` $\{\boxed{p \neq 0}\}$ is valid Hoare triple

$\{\boxed{p \neq 0}\}$ `n != 0` $\{\boxed{p \neq 0}\}$ is valid Hoare triple



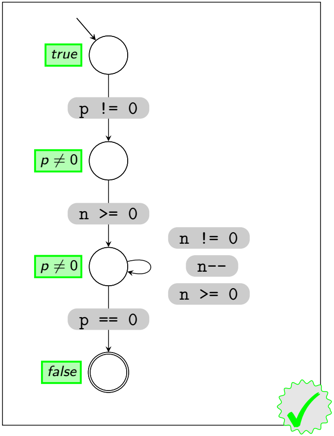
# Trace Abstraction: Example

1. take trace $\pi_1$
2. consider trace as program $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$
4. generalize program $\mathcal{A}_1$
   - add transitions

$\{p \neq 0\}$ `n--` $\{p \neq 0\}$ is valid Hoare triple

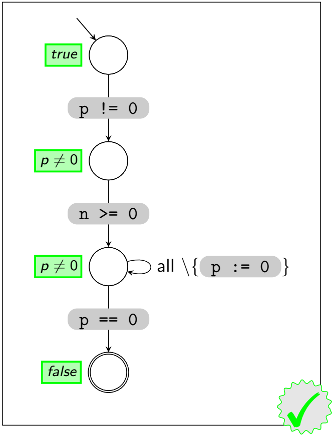
$\{p \neq 0\}$ `n != 0` $\{p \neq 0\}$ is valid Hoare triple

$\{p \neq 0\}$ `n >= 0` $\{p \neq 0\}$ is valid Hoare triple



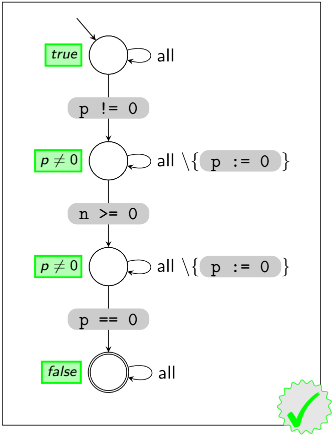
# Trace Abstraction: Example

1. take trace $\pi_1$
2. consider trace as program $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$
4. generalize program $\mathcal{A}_1$
   - add transitions



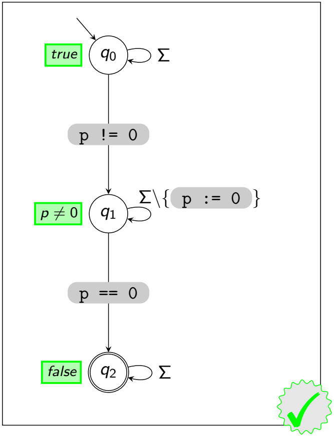
# Trace Abstraction: Example

1. take trace $\pi_1$
2. consider trace as program $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$
4. generalize program $\mathcal{A}_1$
   - add transitions

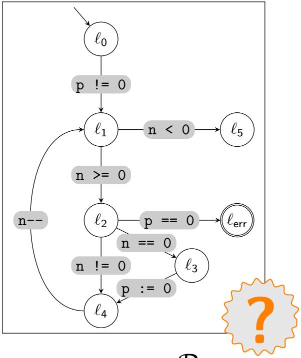


# Trace Abstraction: Example

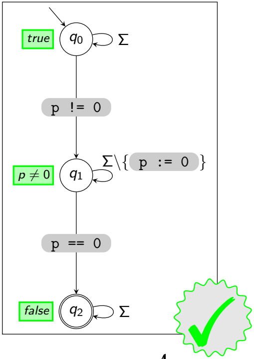
1. take trace $\pi_1$
2. consider trace as program $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$
4. generalize program $\mathcal{A}_1$
   - add transitions
   - merge locations



# Trace Abstraction: Example

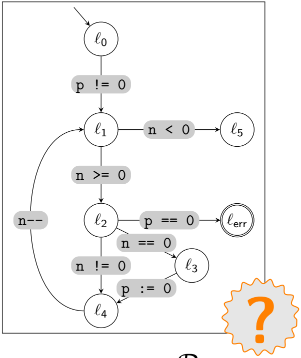



program $\mathcal{P}$

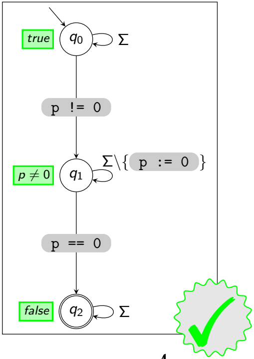



program $\mathcal{A}_1$

# Trace Abstraction: Example

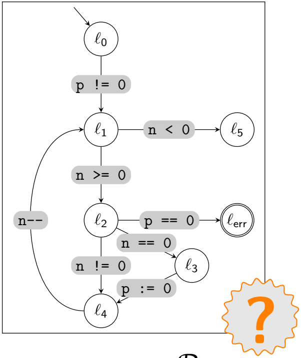



program $\mathcal{P}$

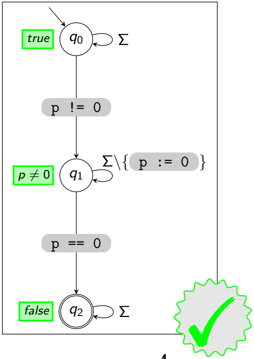



program $\mathcal{A}_1$

# Trace Abstraction: Example

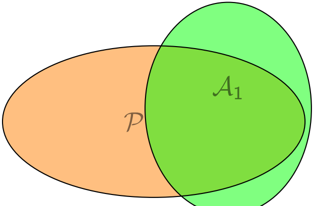



program $\mathcal{P}$

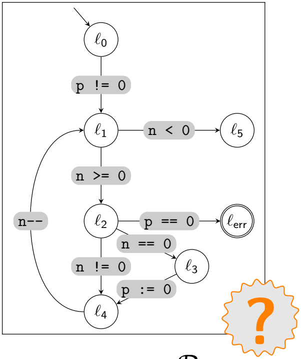



program $\mathcal{A}_1$

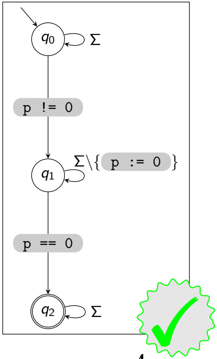
Consider $\mathcal{P}$ and $\mathcal{A}_1$ as automata and consider construct set theoretic difference $L(\mathcal{P}) \setminus L(\mathcal{A}_1)$.



# Trace Abstraction: Example

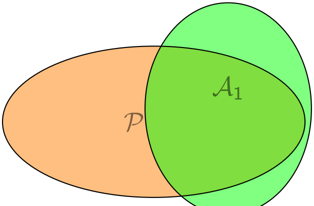



program $\mathcal{P}$

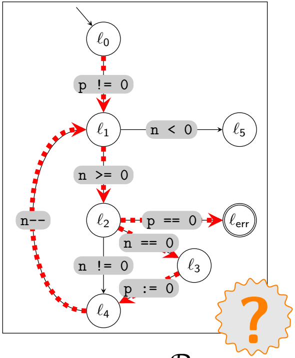



program $\mathcal{A}_1$

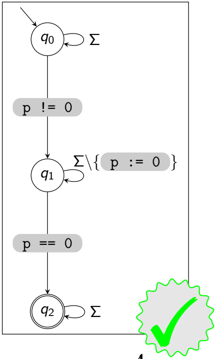
Consider $\mathcal{P}$ and $\mathcal{A}_1$ as automata and consider construct set theoretic difference $L(\mathcal{P})\setminus L(\mathcal{A}_1)$.



# Trace Abstraction: Example

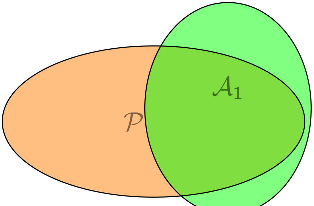



program $\mathcal{P}$



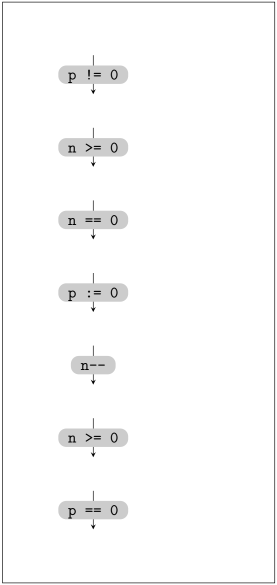

program $\mathcal{A}_1$

Consider $\mathcal{P}$ and $\mathcal{A}_1$ as automata and consider construct set theoretic difference $L(\mathcal{P}) \setminus L(\mathcal{A}_1)$.



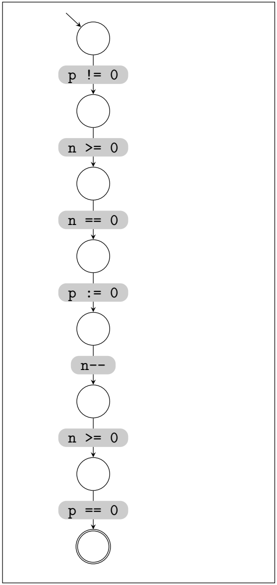
# Trace Abstraction: Example

1. take trace $\pi_2$



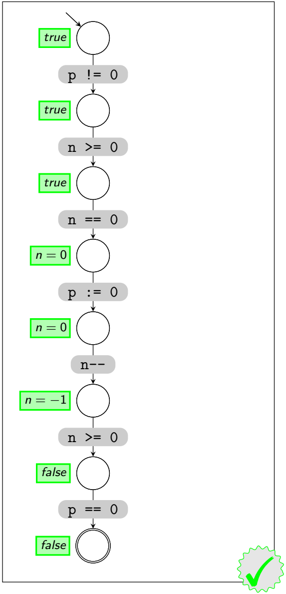
# Trace Abstraction: Example

1. take trace $\pi_2$
2. consider trace as program $\mathcal{A}_2$



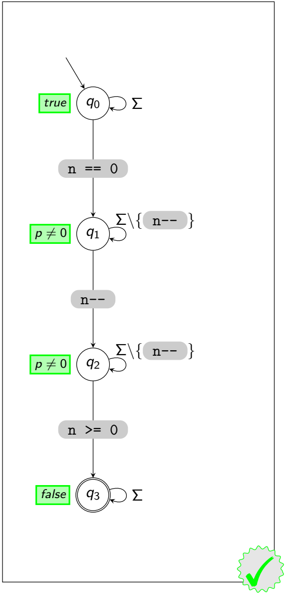
# Trace Abstraction: Example

1. take trace $\pi_2$
2. consider trace as program $\mathcal{A}_2$
3. analyze correctness of $\mathcal{A}_2$

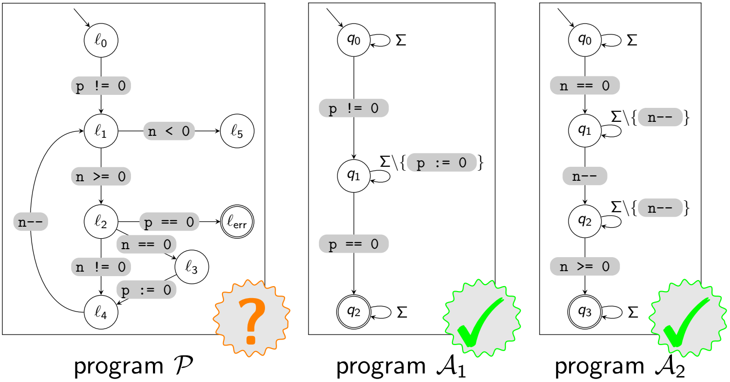


# Trace Abstraction: Example

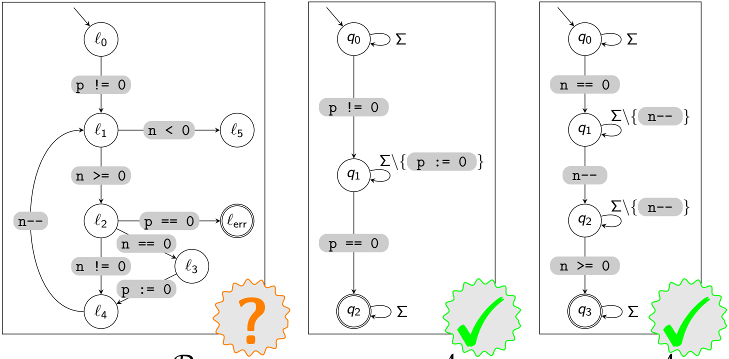
1. take trace $\pi_2$
2. consider trace as program $\mathcal{A}_2$
3. analyze correctness of $\mathcal{A}_2$
4. generalize program $\mathcal{A}_2$
   - add transitions
   - merge locations



# Trace Abstraction: Example



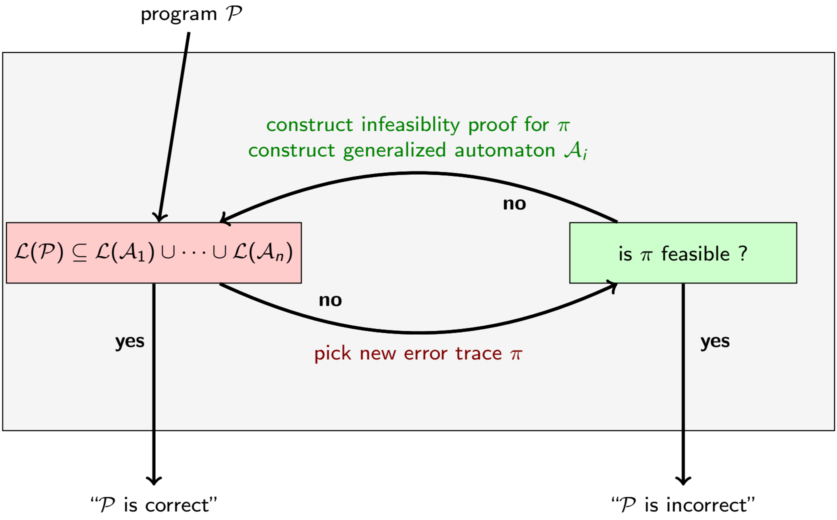
# Trace Abstraction: Example



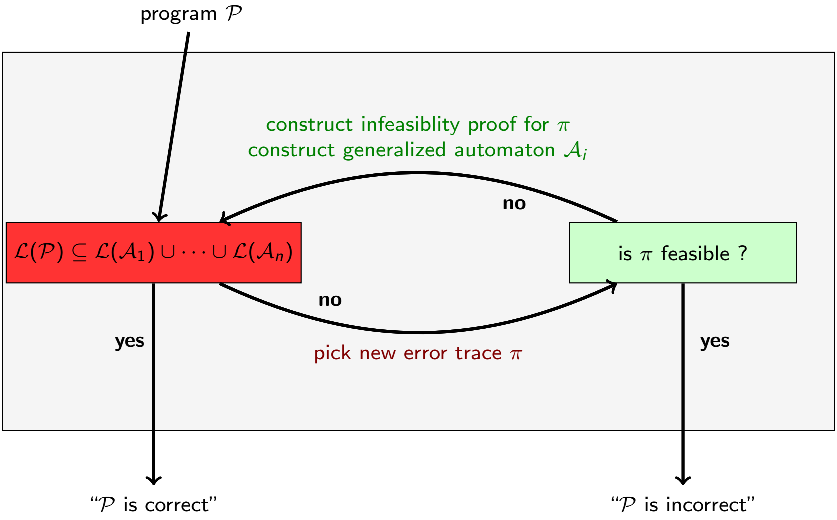

program $\mathcal{P}$ program $\mathcal{A}_1$ program $\mathcal{A}_2$

$$\mathcal{P} \subseteq \mathcal{A}_1 \cup \mathcal{A}_2$$

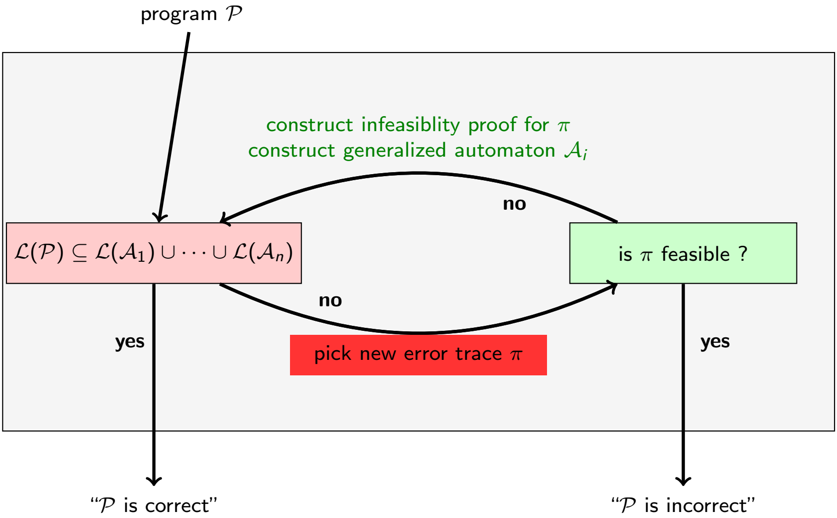
# Trace Abstraction: Verification Scheme



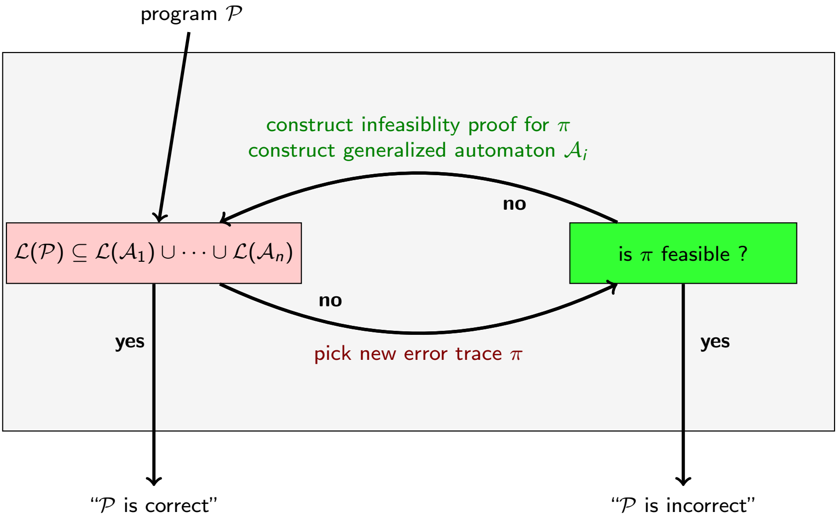
# Trace Abstraction: Verification Scheme



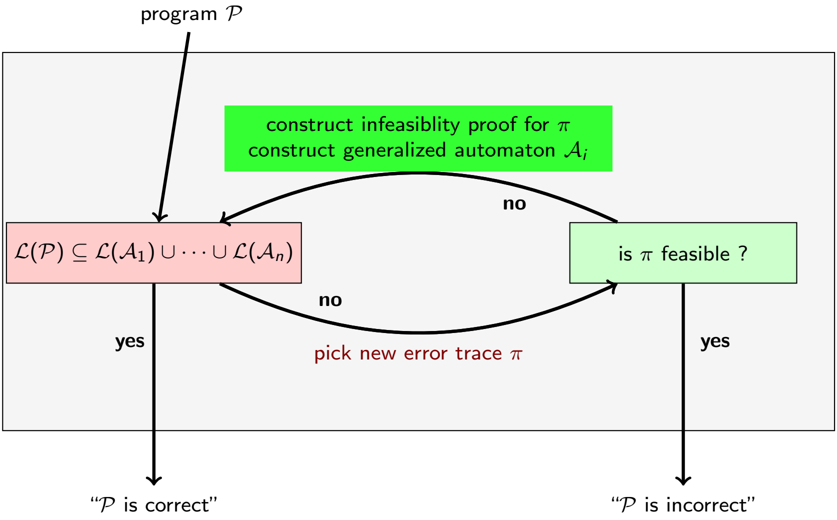
# Trace Abstraction: Verification Scheme



# Trace Abstraction: Verification Scheme



# Trace Abstraction: Verification Scheme



# Outline

- Running Example and Floyd-Hoare Annotations
- Ultimate Referee

  A strict proof checker.
- Trace Abstraction

  The verification approach of Ultimate Automizer
- Incremental Verification Using Trace Abstraction

# Motivation

verify program $\mathcal{P}$

construct modified program $\mathcal{P}'$

verify program $\mathcal{P}'$

construct modified program $\mathcal{P}''$

verify program $\mathcal{P}''$

$\vdots$

# Motivation

verify program $\mathcal{P}$

construct modified program $\mathcal{P}'$

verify program $\mathcal{P}'$

construct modified program $\mathcal{P}''$

verify program $\mathcal{P}''$

$\vdots$

Which information can we reuse

while verifying via trace abstraction?

# Incremental Verification Using Trace Abstraction

Bat-Chen Rothenberg

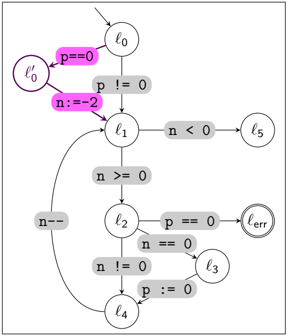
Daniel Dietsch

Matthias Heizmann

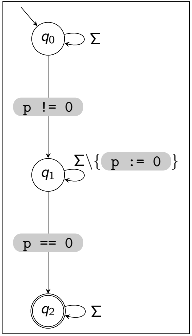
Bat-Chen Rothenberg, Daniel Dietsch, and Matthias Heizmann. "Incremental Verification Using Trace Abstraction". In: *SAS*. vol. 11002. Lecture Notes in Computer Science. Springer, 2018, pp. 364–382

# Reuse automata: Example 1

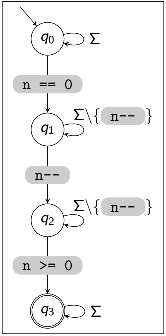
program $\mathcal{P}_{\text{new}}$ with $\Sigma_{\text{new}} = \Sigma \cup \{$ `n:=-2` $\}$




Floyd-Hoare automaton $\mathcal{A}_1$




Floyd-Hoare automaton $\mathcal{A}_2$



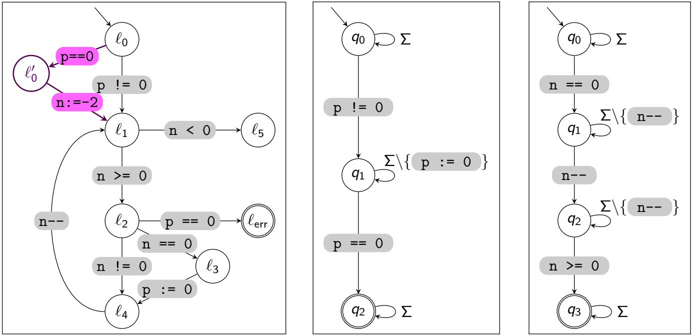

$$\mathcal{P}_{\text{new}} \cap \overline{A_1} \cap \overline{A_2} = \emptyset \ ?$$

# Reuse automata: Example 1

program $\mathcal{P}_{\text{new}}$ with $\Sigma_{\text{new}} = \Sigma \cup \{$`n:=-2`$\}$

Floyd-Hoare automaton $\mathcal{A}_1$

Floyd-Hoare automaton $\mathcal{A}_2$

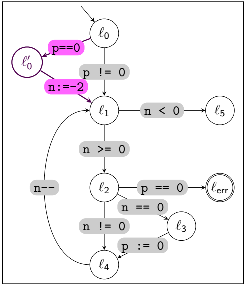



$$\mathcal{P}_{\text{new}} \cap \overline{A_1} \cap \overline{A_2} = \emptyset \ ?$$

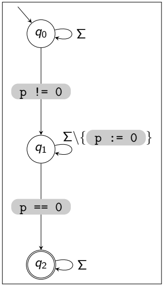
No! Counterexample to emptiness: $\pi =$ `p==0` `n:=-2` `n` $\geq$ `0` `p==0`

# Reuse automata: Example 1

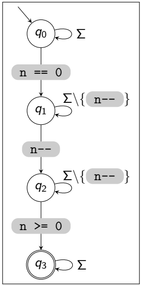
program $\mathcal{P}^{\text{new}}$ with $\Sigma^{\text{new}} = \Sigma \cup \{$ `n:=-2` $\}$



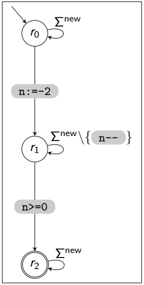

Floyd-Hoare automaton $\mathcal{A}_1$




Floyd-Hoare automaton $\mathcal{A}_2$



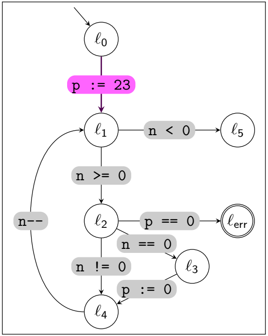

Floyd-Hoare automaton $\mathcal{A}_3$



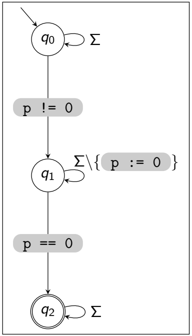

$$\mathcal{P}^{\text{new}} \cap \overline{A_1} \cap \overline{A_2} \cap \overline{A_3} = \emptyset \;!$$

# Reuse automata: Example 2

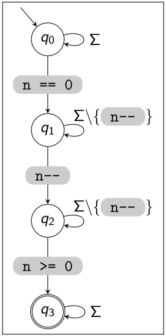
program $\mathcal{P}_{\text{new}}$ with $\Sigma_{\text{new}} = \Sigma \cup \{$ `n:=-2` $\}$




Floyd-Hoare automaton $\mathcal{A}_1$




Floyd-Hoare automaton $\mathcal{A}_2$



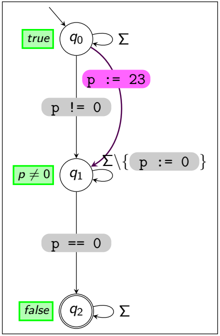

$$\mathcal{P}_{\text{new}} \cap \overline{A_1} \cap \overline{A_2} = \emptyset \text{ ?}$$

No! Automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are useless! Statement `p := 23` not in $\Sigma$.

# Reuse automata: Example 2

Idea: extend Floyd-Hoare automata with new statements

Floyd-Hoare automaton $\mathcal{A}_1$

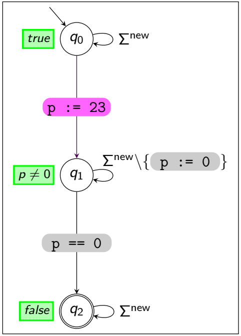



$\{$ *true* $\}$ `p := 23` $\{ p \neq 0 \}$ is valid Hoare triple

# Reuse automata: Example 2

Idea: extend Floyd-Hoare automata with new statements

Floyd-Hoare automaton $\mathcal{A}_1$




$\{\mathit{true}\}$ `p := 23` $\{p \neq 0\}$ is valid Hoare triple

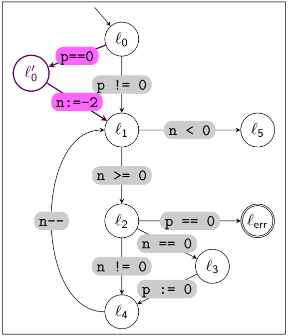
$\{\mathit{true}\}$ `p := 23` $\{\mathit{true}\}$ is valid Hoare triple

$\{\mathit{false}\}$ `p := 23` $\{\mathit{false}\}$ is valid Hoare triple

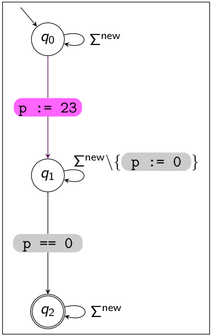
$\vdots$

# Reuse automata: Example 2

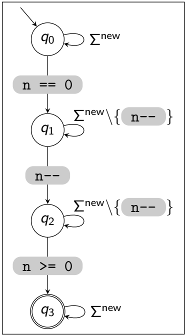
program $\mathcal{P}_{\text{new}}$ with $\Sigma_{\text{new}} = \Sigma \cup \{\texttt{n:=-2}\}$




Floyd-Hoare automaton $\mathcal{A}_1$

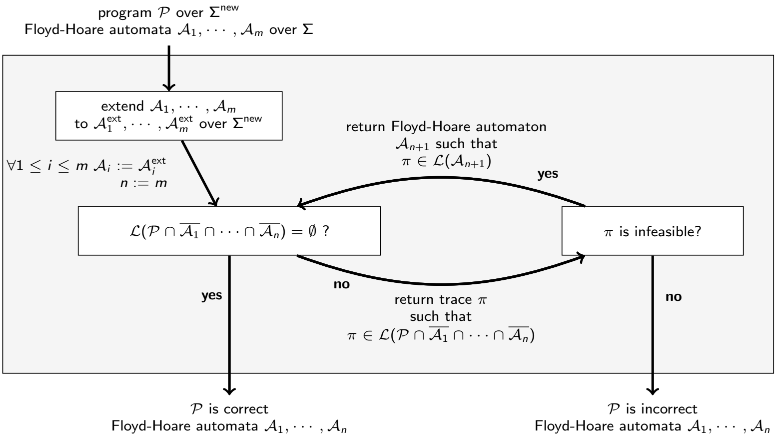



Floyd-Hoare automaton $\mathcal{A}_2$

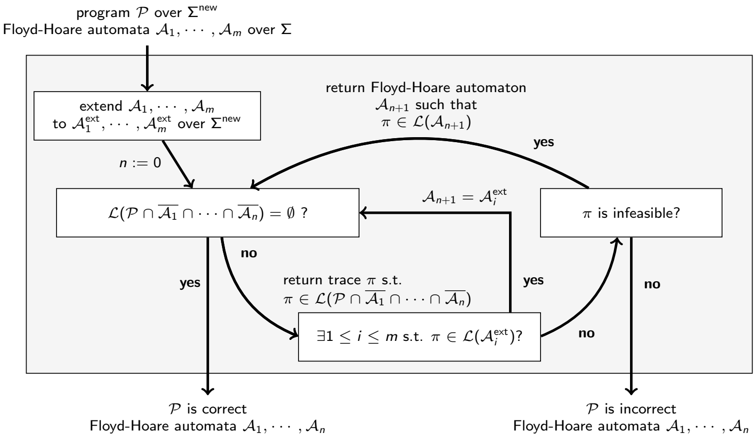



$$\mathcal{P}_{\text{new}} \cap \overline{A_1} \cap \overline{A_2} = \emptyset \ !$$

# Incremental Verification Scheme: Eager Approach



# Incremental Verification Scheme: Lazy Approach



# Will our incremental verification work in practice?

Saved costs:

- Analysis of (potentially spurious) counterexamples

  checking feasibility, computation of interpolants

- Construction of Floyd-Hoare automata

  checking Hoare triples

# Will our incremental verification work in practice?

Saved costs:

- Analysis of (potentially spurious) counterexamples

  checking feasibility, computation of interpolants

- Construction of Floyd-Hoare automata

  checking Hoare triples

Additional costs:

- Larger automata

  e.g., $\mathcal{P} \cap \overline{A_1} \cap \ldots \cap \overline{A_n}$

- Extending automata

  checking Hoare triples

- Reading and writing automata

  I/O operations on hard drive, parsing automata

# Implementation

Implemented in the Ultimate Automizer software verifier

- http://ultimate.informatik.uni-freiburg.de/
- Open source https://github.com/ultimate-pa/ultimate

Automata written in the format of the Ultimate Automata Library

# Benchmarks

- Produced by the Linux Verification Center
  `http://linuxtesting.org/`
- 4,193 verification tasks from 1,119 revisions of 62 device drivers
- Bechmark set used in related work

  Dirk Beyer et al. “Precision reuse for efficient regression verification”. In: *ESEC/SIGSOFT FSE*. ACM, 2013, pp. 389–399

  `https://www.sosy-lab.org/research/cpa-reuse/regression-benchmarks`

# Results

| Driver | Spec | Tasks | Default Overall | Eager Overall | Eager Analysis | Lazy Overall | Lazy Analysis | Speedup Overall | Speedup Analysis | [5] Speedup |
|---|---|---|---|---|---|---|---|---|---|---|
| dvb-usb-rtl28xxu | 08_1a | 10 | 20.509 | 0.352 | 0.187 | 0.416 | 0.257 | 49.30 | 79.80 | 3.6 |
| dvb-usb-rtl28xxu | 39_7a | 10 | 110.893 | 4.081 | 1.992 | 4.059 | 2.546 | 27.32 | 43.55 | 6.3 |
| dvb-usb-rtl28xxu | 32_7a | 10 | 35.551 | 1.306 | 0.725 | 1.550 | 0.844 | 22.93 | 42.12 | 4.9 |
| dvb-usb-az6007 | 08_1a | 5 | 4.620 | 0.173 | 0.118 | 0.187 | 0.132 | 24.70 | 35.00 | 3.5 |
| dvb-usb-az6007 | 39_7a | 5 | 17.952 | 1.378 | 0.862 | 1.425 | 0.989 | 12.59 | 18.15 | 4.9 |
| cx231xx-dvb | 08_1a | 13 | 3.330 | 0.303 | 0.206 | 0.323 | 0.228 | 10.30 | 14.60 | 1.8 |
| panasonic-laptop | 08_1a | 16 | 3.466 | 0.337 | 0.222 | 0.384 | 0.257 | 9.02 | 13.48 | 2.4 |
| spcp8x5 | 43_1a | 13 | 5.531 | 0.632 | 0.437 | 0.618 | 0.432 | 8.94 | 12.80 | 1.6 |
| panasonic-laptop | 32_1 | 4 | 0.623 | 0.100 | 0.061 | 0.072 | 0.051 | 8.65 | 12.21 | 3.4 |
| panasonic-laptop | 39_7a | 16 | 18.961 | 2.377 | 1.654 | 2.617 | 1.906 | 7.24 | 9.94 | 3.6 |
| leds-bd2802 | 68_1 | 4 | 1.039 | 0.180 | 0.112 | 0.191 | 0.123 | 5.43 | 8.44 | 4.4 |
| leds-bd2802 | 32_1 | 4 | 0.484 | 0.089 | 0.057 | 0.097 | 0.064 | 4.98 | 7.56 | 3.9 |
| wm831x-dcdc | 32_1 | 3 | 0.330 | 0.063 | 0.044 | 0.066 | 0.047 | 5.00 | 7.02 | 2.1 |
| cx231xx-dvb | 39_7a | 13 | 17.536 | 3.389 | 2.425 | 3.464 | 2.517 | 5.06 | 6.96 | 3.2 |
| ems_usb | 08_1a | 21 | 2.334 | 0.502 | 0.327 | 0.543 | 0.362 | 4.29 | 6.44 | 2.9 |
| ...(for full results cf. http://batg.cswp.cs.technion.ac.il/publications/) | | | | | | | | | | |
| ar7part | 32_7a | 6 | 0.071 | 0.067 | 0.056 | 0.074 | 0.063 | 0.95 | 1.12 | 1.3 |
| metro-usb | 08_1a | 25 | 0.394 | 0.497 | 0.330 | 0.518 | 0.356 | 0.76 | 1.10 | 2.1 |
| rtc-max6902 | 32_7a | 9 | 0.133 | 0.124 | 0.106 | 0.147 | 0.126 | 0.90 | 1.05 | 1.1 |
| i2c-algo-pca | 43_1a | 7 | 0.012 | 0.018 | 0.018 | 0.019 | 0.019 | 1.00 | 1.00 | 1.0 |
| dvb-usb-vp7045 | 43_1a | 2 | 0.001 | 0.002 | 0.002 | 0.027 | 0.027 | 1.00 | 1.00 | 2.6 |
| cfag12864b | 43_1a | 2 | 0.036 | 0.039 | 0.036 | 0.040 | 0.037 | 0.90 | 0.97 | 1.0 |
| rtc-max6902 | 43_1a | 5 | 0.278 | 0.273 | 0.262 | 0.303 | 0.291 | 0.91 | 0.95 | 1.1 |
| magellan | 32_7a | 2 | 0.015 | 0.018 | 0.016 | 0.018 | 0.016 | 0.83 | 0.93 | 0.93 |
| vsxxxaa | 43_1a | 2 | 0.030 | 0.037 | 0.033 | 0.036 | 0.032 | 0.83 | 0.93 | 6.8 |
| ar7part | 43_1a | 2 | 0.036 | 0.043 | 0.038 | 0.044 | 0.039 | 0.81 | 0.92 | 1.2 |
| **Sum (All)** | | 2,660 | 3, 048.373 | 434.853 | 334.603 | 448.424 | 349.69 | | | |
| **Mean (All)** | | 15 | 16.749 | 2.389 | 1.838 | 2.464 | 1.921 | 6.798 | 8.717 | 4.3 |

# Future work

- Measure semantical similarity of programs
- Floyd-Hoare automata with alpha renaming
- Database of Floyd-Hoare automata in the cloud
- Machine learning to determine most promising Floyd-Hoare automata from database

Thank you for your attention!

# References I


- Dirk Beyer et al. "Correctness witnesses: exchanging verification results between verifiers". In: *SIGSOFT FSE*. ACM, 2016, pp. 326–337.
- Dirk Beyer et al. "Precision reuse for efficient regression verification". In: *ESEC/SIGSOFT FSE*. ACM, 2013, pp. 389–399.
- Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Nested interpolants". In: *POPL*. ACM, 2010, pp. 471–482.
- Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Refinement of Trace Abstraction". In: *SAS*. Vol. 5673. Lecture Notes in Computer Science. Springer, 2009, pp. 69–85.
- Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Software Model Checking for People Who Love Automata". In: *CAV*. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 36–52.
- Bat-Chen Rothenberg, Daniel Dietsch, and Matthias Heizmann. "Incremental Verification Using Trace Abstraction". In: *SAS*. Vol. 11002. Lecture Notes in Computer Science. Springer, 2018, pp. 364–382.