Ultimate Referee, Ultimate Automizer, and Incremental Verification

Matthias Heizmann

University of Freiburg

CPAchecker Workshop 2019
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**
Running Example and Floyd-Hoare Annotation

\begin{align*}
\ell_0: & \quad \text{assume } p \neq 0; \\
\ell_1: & \quad \text{while}(n \geq 0) \\
& \quad \{ \\
\ell_2: & \quad \text{assert } p \neq 0; \\
& \quad \text{if}(n == 0) \\
& \quad \quad \{ \\
\ell_3: & \quad p := 0; \\
& \quad \} \\
\ell_4: & \quad n--; \\
& \quad \}
\end{align*}

pseudocode

\begin{tikzpicture}
  \node (l0) at (0,0) {$\ell_0$};
  \node (l1) at (1,-1) {$\ell_1$};
  \node (l2) at (2,-2) {$\ell_2$};
  \node (l3) at (3,-3) {$\ell_3$};
  \node (l4) at (4,-4) {$\ell_4$};
  \node (l5) at (5,-5) {$\ell_5$};
  \node (lerr) at (5,-6) {$\ell_{\text{err}}$};

  \path
    (l0) edge [->] node {p \neq 0} (l1)
    (l1) edge [->] node {n < 0} (l5)
    (l1) edge [->] node {n \geq 0} (l2)
    (l2) edge [->] node {p == 0} (lerr)
    (l2) edge [->] node {n == 0} (l3)
    (l3) edge [->] node {n \neq 0} (l4)
    (l4) edge [->] node {p := 0} (l2)
    (l4) edge [->] node {n --} (l2);
\end{tikzpicture}

control flow graph
Running Example and Floyd-Hoare Annotation

Definition:

\[
\{ \varphi \} \text{ st } \{ \varphi' \} \text{ is valid Hoare triple}
\]

if program is in state that satisfies \( \varphi \) and program executes \( \text{ st } \)
then program is in a state that satisfies \( \varphi' \)

Example:

\[
\{ p \neq 0 \lor n = -1 \} \quad n \geq 0 \quad \{ p \neq 0 \} 
\]

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 $\varphi_i \rightarrow \ell_i \rightarrow \ell_j \rightarrow \varphi_j$ only if the Hoare triple $\{ \varphi \} \ell \rightarrow \ell' \{ \varphi' \}$ 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.
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 annotated 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 annotated 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


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

---

<table>
<thead>
<tr>
<th>examples</th>
<th>infeasible</th>
<th>feasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>error trace of $P$</td>
<td>$p \neq 0$</td>
<td>$n \geq 0$</td>
</tr>
<tr>
<td>not error trace of $P$</td>
<td>$n = 0$</td>
<td>$n--$</td>
</tr>
</tbody>
</table>
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.
Trace Abstraction: Technical Implementation

Implementation based on automata theory

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

- 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
\[
\ell_0: \text{ assume } p \neq 0; \\
\ell_1: \text{ while(} n \geq 0 \text{) } \\
\quad \{ \\
\quad \quad \ell_2: \text{ assert } p \neq 0; \\
\quad \quad \quad \text{ if(} n == 0 \text{) } \\
\quad \quad \quad \quad \{ \\
\quad \quad \quad \quad \quad \ell_3: \quad p := 0; \\
\quad \quad \quad \quad \} \\
\quad \quad \ell_4: \quad n--; \\
\quad \} \\
\text{ pseudocode}
\]
Trace Abstraction: Example

\[\ell_0: \text{assume } p \neq 0;\]
\[\ell_1: \text{while}(n \geq 0)\]
\[\{\]
\[\ell_2: \text{assert } p \neq 0;\]
\[\text{if}(n == 0)\]
\[\{\]
\[\ell_3: \quad p := 0;\]
\[\}\]
\[\ell_4: \quad n--;\]
\[\}\]

pseudocode

control flow graph
1. take trace $\pi_1$
Trace Abstraction: Example

1. take trace $\pi_1$
2. consider trace as program $A_1$

1: assume $p \neq 0$;
2: assume $n \geq 0$;
3: assert $p \neq 0$;

pseudocode of $A_1$

Diagram of $A_1$: 
- Initial state with $p \neq 0$
- Transition to state with $n \geq 0$
- Transition to state with $p = 0$
- Final state
1. take trace $\pi_1$
2. consider trace as program $A_1$
3. analyze correctness of $A_1$
1. take trace $\pi_1$
2. consider trace as program $A_1$
3. analyze correctness of $A_1$
4. generalize program $A_1$
   - add transitions

\[
\begin{align*}
\{ p \neq 0 \} & \text{ n-- } \{ p \neq 0 \} \quad \text{is valid Hoare triple}
\end{align*}
\]
1. take trace $\pi_1$
2. consider trace as program $A_1$
3. analyze correctness of $A_1$
4. generalize program $A_1$
   ▶ add transitions

\[ \{ p \neq 0 \} \text{ n--} \{ p \neq 0 \} \]
\[ \{ p \neq 0 \} \text{ n != 0} \{ p \neq 0 \} \]

is valid Hoare triple

\[ \{ p \neq 0 \} \text{ n >= 0} \{ p \neq 0 \} \]

is valid Hoare triple

\[ \{ p \neq 0 \} \text{ n >= 0} \{ p \neq 0 \} \]

\[ \{ p \neq 0 \} \text{ n != 0} \{ n-- \} \]

\[ \{ p \neq 0 \} \text{ n >= 0} \{ p \neq 0 \} \]

\[ \{ p \neq 0 \} \text{ n != 0} \{ n-- \} \]

\[ \{ p \neq 0 \} \text{ n >= 0} \{ p \neq 0 \} \]

\[ \{ p \neq 0 \} \text{ n != 0} \{ n-- \} \]

\[ \{ p \neq 0 \} \text{ n >= 0} \{ p \neq 0 \} \]

\[ \{ p \neq 0 \} \text{ n != 0} \{ n-- \} \]

\[ \{ p \neq 0 \} \text{ n >= 0} \{ p \neq 0 \} \]

\[ \{ p \neq 0 \} \text{ n != 0} \{ n-- \} \]

\[ \{ p \neq 0 \} \text{ n >= 0} \{ p \neq 0 \} \]

\[ \{ p \neq 0 \} \text{ n != 0} \{ n-- \} \]
Trace Abstraction: Example

1. take trace $\pi_1$
2. consider trace as program $A_1$
3. analyze correctness of $A_1$
4. generalize program $A_1$
   - add transitions

\[
\begin{align*}
\{p \neq 0\} & \quad n-- \quad \{p \neq 0\} & \text{is valid Hoare triple} \\
\{p \neq 0\} & \quad n \neq 0 \quad \{p \neq 0\} & \text{is valid Hoare triple} \\
\{p \neq 0\} & \quad n \geq 0 \quad \{p \neq 0\} & \text{is valid Hoare triple}
\end{align*}
\]
1. take trace $\pi_1$
2. consider trace as program $A_1$
3. analyze correctness of $A_1$
4. generalize program $A_1$
   ▶ 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

Diagram:

- **true**
  - $p \neq 0$
  - $p = 0$
- **false**
  - $n \geq 0$
  - $p \neq 0$
  - $p = 0$

Arrows:
- From **true** to **true**
- From **false** to **false**
- From **false** to **true**
- From **true** to **false**

Equations:

- $p \neq 0$
- $p = 0$
- $n \geq 0$
- $p \neq 0$
- $p = 0$
1. take trace \( \pi_1 \)
2. consider trace as program \( A_1 \)
3. analyze correctness of \( A_1 \)
4. generalize program \( A_1 \)
   ▶ add transitions
   ▶ merge locations
Trace Abstraction: Example

Program $P$

- $\ell_0$
  - $p \neq 0$
- $\ell_1$
  - $n < 0$
  - $n \geq 0$
- $\ell_2$
  - $p = 0$
  - $n = 0$
  - $n \neq 0$
- $\ell_3$
- $\ell_4$
- $\ell_{err}$

Program $A_1$

- $q_0$
  - $\Sigma$
  - $p \neq 0$
- $q_1$
  - $\Sigma \setminus \{p := 0\}$
  - $p = 0$
- $q_2$
  - $\Sigma$
  - $false$

Consider $P$ and $A_1$ as automata and consider the set theoretic difference $L(P) \setminus L(A_1)$. $P \setminus A_1$
Trace Abstraction: Example

Consider program $P$ and program $A_1$ as automata and consider set theoretic difference $L(P) \setminus L(A_1)$. $P A_1$
Consider \( \mathcal{P} \) and \( \mathcal{A}_1 \) as automata and consider construct set theoretic difference \( L(\mathcal{P}) \setminus L(\mathcal{A}_1) \).
Consider $\mathcal{P}$ and $\mathcal{A}_1$ as automata and consider construct set theoretic difference $L(\mathcal{P}) \setminus L(\mathcal{A}_1)$. 
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 $\pi_2$
1. take trace $\pi_2$
2. consider trace as program $A_2$
1. take trace $\pi_2$
2. consider trace as program $A_2$
3. analyze correctness of $A_2$
Trace Abstraction: Example

1. take trace $\pi_2$
2. consider trace as program $A_2$
3. analyze correctness of $A_2$
4. generalize program $A_2$
   - add transitions
   - merge locations
Trace Abstraction: Example

Program $P$

Program $A_1$

Program $A_2$

$P \subseteq A_1 \cup A_2$
Trace Abstraction: Verification Scheme

program $\mathcal{P}$

$L(\mathcal{P}) \subseteq L(A_1) \cup \cdots \cup L(A_n)$

is $\pi$ feasible?

no

pick new error trace $\pi$

no

construct infeasibility proof for $\pi$
construct generalized automaton $A_i$

yes

“$\mathcal{P}$ is correct”

“$\mathcal{P}$ is incorrect”
Trace Abstraction: Verification Scheme

\[ \mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(A_1) \cup \ldots \cup \mathcal{L}(A_n) \]

- **program** \( \mathcal{P} \)
- **is \( \pi \) feasible?**
  - no
  - yes
  - pick new error trace \( \pi \)
  - construct infeasibility proof for \( \pi \)
  - construct generalized automaton \( A_i \)

- “\( \mathcal{P} \) is correct”
- “\( \mathcal{P} \) is incorrect”
Trace Abstraction: Verification Scheme

- Program $\mathcal{P}$
- $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{A}_1) \cup \cdots \cup \mathcal{L}(\mathcal{A}_n)$
- is $\pi$ feasible?

- no
  - construct infeasibility proof for $\pi$
  - construct generalized automaton $\mathcal{A}_i$

- yes
  - yes
  - no
  - pick new error trace $\pi$

- "$\mathcal{P}$ is correct"
- "$\mathcal{P}$ is incorrect"
Trace Abstraction: Verification Scheme

Let $L(P) \subseteq L(A_1) \cup \cdots \cup L(A_n)$

- If yes, then $P$ is correct.
- If no, then pick new error trace $\pi$ and continue.

- If no, then construct infeasibility proof for $\pi$ and construct generalized automaton $A_i$.

- If yes, then $P$ is incorrect.

“$P$ is correct”

“$P$ is incorrect”
Trace Abstraction: Verification Scheme

Let \( P \) be the program and \( \mathcal{L}(P) \subseteq \mathcal{L}(A_1) \cup \cdots \cup \mathcal{L}(A_n) \). If \( \pi \) is feasible?

- If yes, then \( \mathcal{P} \) is correct.
- If no, pick new error trace \( \pi \) and repeat.

If \( \pi \) is infeasible, then construct infeasibility proof for \( \pi \) and construct generalized automaton \( A_i \).
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?
Reuse automata: Example 1

program $P_{\text{new}}$ with
$\Sigma_{\text{new}} = \Sigma \cup \{n:=-2\}$

Floyd-Hoare automaton $A_1$

Floyd-Hoare automaton $A_2$

$P_{\text{new}} \cap \overline{A_1} \cap \overline{A_2} = \emptyset$ ?
Reuse automata: Example 1

Program \( P_{\text{new}} \) with
\[
\Sigma_{\text{new}} = \Sigma \cup \{ n:=-2 \}
\]

Floyd-Hoare automaton \( A_1 \)

Floyd-Hoare automaton \( A_2 \)

\[
P_{\text{new}} \cap A_1 \cap A_2 = \emptyset \ ?
\]

No! Counterexample to emptiness: \( \pi = p==0 \ \ n:=-2 \ \ n \geq 0 \ \ p==0 \)
Reuse automata: Example 1

program $P^{new}$ with

$\Sigma^{new} = \Sigma \cup \{n = -2\}$

Floyd-Hoare automaton $A_1$

Floyd-Hoare automaton $A_2$

Floyd-Hoare automaton $A_3$

$P^{new} \cap A_1 \cap A_2 \cap A_3 = \emptyset$!
Reuse automata: Example 2

program \( \mathcal{P}_{\text{new}} \) with
\[
\Sigma_{\text{new}} = \Sigma \cup \{ n := -2 \}
\]

\begin{align*}
\ell_0 & \quad p := 23 \\
\ell_1 & \quad n < 0 \quad \ell_5 \\
\ell_2 & \quad n >= 0 \\
\ell_3 & \quad p != 0 \\
\ell_4 & \quad n != 0 \\
\end{align*}

Floyd-Hoare automaton \( \mathcal{A}_1 \)
\begin{align*}
q_0 & \quad \Sigma \\
q_1 & \quad \{ p := 0 \} \\
q_2 & \quad \{ n-- \} \\
q_3 & \quad \Sigma
\end{align*}

Floyd-Hoare automaton \( \mathcal{A}_2 \)
\begin{align*}
q_0 & \quad \Sigma \\
q_1 & \quad \{ n-- \} \\
q_2 & \quad \{ n-- \} \\
q_3 & \quad \Sigma
\end{align*}

\[
\mathcal{P}_{\text{new}} \cap \overline{\mathcal{A}_1} \cap \overline{\mathcal{A}_2} = \emptyset ?
\]

No! Automata \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \) are useless! Statement \( p := 23 \) not in \( \Sigma \).
Idea: extend Floyd-Hoare automata with new statements

Floyd-Hoare automaton $A_1$

$$\{ \text{true} \} \ p := 23 \ \{ p \neq 0 \} \text{ is valid Hoare triple}$$
Idea: extend Floyd-Hoare automata with new statements

Floyd-Hoare automaton $A_1$

- Initial state $q_0$ with $\{true\}$ and transition to $q_1$ on $\Sigma_{new}$.
- State $q_1$ with $\{p := 23\}$ and transition on $\Sigma_{new}$.
- State $q_2$ with $\{false\}$ and transition on $\Sigma_{new}$.

- $p := 23$ from $q_0$ to $q_1$ is valid Hoare triple.
- $p := 23$ from $q_1$ to $q_0$ is valid Hoare triple.
- $p := 23$ from $q_1$ to $q_2$ is valid Hoare triple.

...
Reuse automata: Example 2

**Program $P_{\text{new}}$ with**

$\Sigma_{\text{new}} = \Sigma \cup \{n:=2\}$

![Diagram](image)

**Floyd-Hoare automaton $A_1$**

**Floyd-Hoare automaton $A_2$**

$P_{\text{new}} \cap \overline{A_1} \cap \overline{A_2} = \emptyset$!
Incremental Verification Scheme: Eager Approach

Program $P$ over $\Sigma^{\text{new}}$.

Floyd-Hoare automata $A_1, \ldots, A_m$ over $\Sigma$.

Extend $A_1, \ldots, A_m$ to $A_{1}^{\text{ext}}, \ldots, A_{m}^{\text{ext}}$ over $\Sigma^{\text{new}}$.

\[ \forall 1 \leq i \leq m \; A_i := A_i^{\text{ext}} \]
\[ n := m \]

$L(P \cap A_1 \cap \cdots \cap A_n) = \emptyset$?

Yes: $\pi$ is infeasible?

Yes: $P$ is correct. Floyd-Hoare automata $A_1, \ldots, A_n$.

No: return trace $\pi$ such that $\pi \in L(A_{n+1})$.

Yes: return Floyd-Hoare automaton $A_{n+1}$ such that $\pi \in L(A_{n+1})$.

No: $P$ is incorrect. Floyd-Hoare automata $A_1, \ldots, A_n$. 
Incremental Verification Scheme: Lazy Approach

Program $P$ over $\Sigma$

Floyd-Hoare automata $A_1, \ldots, A_m$ over $\Sigma$

Extend $A_1, \ldots, A_m$ to $A_1^{\text{ext}}, \ldots, A_m^{\text{ext}}$ over $\Sigma^{\text{new}}$

$n := 0$

Return Floyd-Hoare automaton $A_{n+1}$ such that $\pi \in L(A_{n+1})$

\[ L(P \cap A_1 \cap \cdots \cap A_n) = \emptyset \] ?

\[ \exists 1 \leq i \leq m \text{ s.t. } \pi \in L(A_i^{\text{ext}}) \] ?

$\pi$ is infeasible?

Yes

$n := 0$

No

Return trace $\pi$ s.t. $\pi \in L(P \cap A_1 \cap \cdots \cap A_n)$

$A_{n+1} = A_i^{\text{ext}}$

No

Yes

$P$ is incorrect

Floyd-Hoare automata $A_1, \ldots, A_n$

$P$ is correct

Floyd-Hoare automata $A_1, \ldots, A_n$
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
  \[ \mathcal{P} \cap \overline{A_1} \cap \ldots \cap \overline{A_n} \]
- 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/](http://ultimate.informatik.uni-freiburg.de/)
- Open source [https://github.com/ultimate-pa/ultimate](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

- Benchmark 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

<table>
<thead>
<tr>
<th>Driver</th>
<th>Spec Tasks</th>
<th>Overall</th>
<th>Overall</th>
<th>Analysis</th>
<th>Speedup Overall</th>
<th>Speedup Analysis</th>
<th>[5] Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>dvb-usb-rtl28xxu</td>
<td>08.1a 10</td>
<td>20.509</td>
<td>0.352</td>
<td>0.187</td>
<td>0.416</td>
<td>0.257</td>
<td>49.30</td>
</tr>
<tr>
<td>dvb-usb-rtl28xxu</td>
<td>08.1a 10</td>
<td>110.893</td>
<td>4.081</td>
<td>1.992</td>
<td>4.059</td>
<td>2.546</td>
<td>27.32</td>
</tr>
<tr>
<td>dvb-usb-rtl28xxu</td>
<td>08.1a 10</td>
<td>35.551</td>
<td>1.306</td>
<td>0.725</td>
<td>1.550</td>
<td>0.844</td>
<td>22.93</td>
</tr>
<tr>
<td>dvb-usb-az6007</td>
<td>08.1a 5</td>
<td>4.620</td>
<td>0.173</td>
<td>0.118</td>
<td>0.187</td>
<td>0.132</td>
<td>24.70</td>
</tr>
<tr>
<td>dvb-usb-az6007</td>
<td>39.7a 5</td>
<td>17.952</td>
<td>1.378</td>
<td>0.862</td>
<td>1.425</td>
<td>0.989</td>
<td>12.59</td>
</tr>
<tr>
<td>cx231xx-dvb</td>
<td>08.1a 13</td>
<td>3.330</td>
<td>0.303</td>
<td>0.206</td>
<td>0.323</td>
<td>0.228</td>
<td>10.30</td>
</tr>
<tr>
<td>panasonic-laptop</td>
<td>08.1a 16</td>
<td>3.466</td>
<td>0.337</td>
<td>0.222</td>
<td>0.384</td>
<td>0.257</td>
<td>9.02</td>
</tr>
<tr>
<td>sscp8x5</td>
<td>43.1a 13</td>
<td>5.531</td>
<td>0.632</td>
<td>0.437</td>
<td>0.618</td>
<td>0.432</td>
<td>8.94</td>
</tr>
<tr>
<td>panasonic-laptop</td>
<td>08.1a 4</td>
<td>0.623</td>
<td>0.100</td>
<td>0.061</td>
<td>0.072</td>
<td>0.051</td>
<td>8.65</td>
</tr>
<tr>
<td>panasonic-laptop</td>
<td>39.7a 16</td>
<td>18.961</td>
<td>2.377</td>
<td>1.654</td>
<td>2.617</td>
<td>1.906</td>
<td>7.24</td>
</tr>
<tr>
<td>leds-bd2802</td>
<td>68.1a 4</td>
<td>1.039</td>
<td>0.180</td>
<td>0.112</td>
<td>0.191</td>
<td>0.123</td>
<td>5.43</td>
</tr>
<tr>
<td>leds-bd2802</td>
<td>32.1 4</td>
<td>0.484</td>
<td>0.089</td>
<td>0.057</td>
<td>0.097</td>
<td>0.064</td>
<td>4.98</td>
</tr>
<tr>
<td>wm831x-dcdc</td>
<td>32.1a 3</td>
<td>0.330</td>
<td>0.063</td>
<td>0.044</td>
<td>0.066</td>
<td>0.047</td>
<td>5.00</td>
</tr>
<tr>
<td>cx231xx-dvb</td>
<td>39.7a 13</td>
<td>17.536</td>
<td>3.389</td>
<td>2.425</td>
<td>3.464</td>
<td>2.517</td>
<td>5.06</td>
</tr>
<tr>
<td>ems_usb</td>
<td>08.1a 21</td>
<td>2.334</td>
<td>0.502</td>
<td>0.327</td>
<td>0.543</td>
<td>0.362</td>
<td>4.29</td>
</tr>
</tbody>
</table>

... (for full results cf. http://batg.cs.wp.cs.technion.ac.il/publications/)

<table>
<thead>
<tr>
<th>Sum (All)</th>
<th>2,660 3</th>
<th>048.373</th>
<th>434.853</th>
<th>334.603</th>
<th>448.424</th>
<th>349.69</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean (All)</td>
<td>15</td>
<td>16.749</td>
<td>2.389</td>
<td>1.838</td>
<td>2.464</td>
<td>1.921</td>
</tr>
</tbody>
</table>

[5] Speedup
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!


