Traces, Interpolants, and Automata: Ultimate Automizer’s Approach to Software Verification

Matthias Heizmann

University of Freiburg

EPIT 2018, Aussois, 2018-05-07
Ultimate program analysis framework

source code at GitHub (mainly LGPL)

received contributions from more than 50 people

Automata Library

Petri

alternating

tree

visibly pushdown

Büchi

tools available via web interface

software verification tools

Kojak

Taipan

Automizer
ULTIMATE

Automizer

automata-based software verification

for non-reachability, memory safety, termination, overflows
ULTIMATE

automata-based software verification
for non-reachability, memory safety, termination, overflows

Achievements at SV-COMP (https://sv-comp.sosy-lab.org)

- 2015: Silver Overall
- 2016: Gold Overall, Gold Falsification Overall
- 2017: Gold Overall, Gold Termination
- 2018: Silver Overall, Gold Termination, Gold NoOverflow
Reason for success? Standing on the sholders of giants!

SMT

automata theory
Outline

- **Software verification:**
  Example, Hoare triples, Floyd-Hoare annotation

- Trace abstraction: new paradigm

- Trace abstraction: example
  - Excursus: Correctness proofs for straightline code

- Trace abstraction: algorithm

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

*pseudocode*

**control flow graph**
\(p \neq 0 \lor n = -1\)

\(n \geq 0\)

\(n < 0\)

\(n = 0\)

\(n \neq 0\)

\(p = 0\)

\(p := 0\)

\(n = 0\)

control flow graph
Definition:

\{ \phi \} \text{s.t.} \{ \phi' \} \text{ is valid Hoare triple iff}

if program is in state that satisfies \phi
and program executes \text{s.t.}
then program is in a state that satisfies \phi'

Example:

\{ p \neq 0 \lor n = -1 \} \text{n>0} \{ p \neq 0 \}

is a valid Hoare triple

Example:

\{ n \neq 0 \} \text{n--} \{ n \neq 0 \}

is not a valid Hoare triple
Example:

\[
\{ n \neq 0 \} \text{n--} \{ n \neq 0 \}
\]
is not a valid Hoare triple

SMT script

(declare-fun n () Int)
(declare-fun n' () Int)
(assert (not (= n 0)))
(assert (= n' (- n 1)))
(assert (not (not (= n' 0))))
(check-sat)
Definition:

A Floyd-Hoare annotation is a mapping that assigns each location $\ell_i$ a formula $\varphi_i$ such that there is an edge $\varphi_i \xrightarrow{\text{stmt}} \varphi_j$ only if the Hoare triple $\{ \varphi \} \text{stmt} \{ \varphi' \}$ is valid.

Theorem:

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

- Software verification:
  Example, Hoare triples, Floyd-Hoare annotation

- Trace abstraction: new paradigm

- Trace abstraction: example
  - Excursus: Correctness proofs for straightline code

- Trace abstraction: algorithm

- Termination analysis
“A program defines a language over the alphabet of statements.”
New View on Programs

“A program defines a language over the alphabet of statements.”

- Set of statements: alphabet of formal language
  e.g., \( \Sigma = \{ p != 0, n >= 0, n == 0, p := 0, n != 0, p == 0, n-- , n < 0 \} \)
New View on Programs

“A program defines a language over the alphabet of statements.”

- Set of statements: alphabet of formal language
e.g., $\Sigma = \{ \text{p \neq 0}, \text{n \geq 0}, \text{n == 0}, \text{p := 0}, \text{n \neq 0},$

  - $\text{p == 0}, \text{n--}, \text{n < 0} \}$

- Control flow graph: automaton over the alphabet of statements
- Error location: accepting state of this automaton
“A program defines a language over the alphabet of statements.”

- **Set of statements**: alphabet of formal language
  
  \[ \Sigma = \{ p \neq 0, \quad n \geq 0, \quad n = 0, \quad p := 0, \quad n \neq 0, \quad p = 0, \quad n-- , \quad n < 0 \} \]

- **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
\( \ell_0: \) assume p \(!=\) 0;

\( \ell_1: \) while(\( n \geq 0 \))

\( \ell_2: \) assert p \(!=\) 0;

\( \ell_3: \) if(\( n == 0 \))

\( \ell_4: \) p := 0;

\( \ell_5: \) n--;

pseudocode
Some program together with a specification in our formalism
Trace Abstraction

Our paradigm:

Computer programs are collections of statements, the definition of a program consists of two parts:

1. meaning of the statements
2. the way how the statements are arranged
Our paradigm:

Computer programs are collections of statements, the definition of a program consists of two parts:

<table>
<thead>
<tr>
<th></th>
<th>defined by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>meaning of the statements</td>
</tr>
<tr>
<td>2.</td>
<td>the way how the statements are arranged</td>
</tr>
</tbody>
</table>
Our paradigm:

Computer programs are collections of statements, the definition of a program consists of two parts:

1. **meaning of the statements** defined by **semantics of programming language** our formalism **SMT**
2. **the way how the statements are arranged** defined by **control flow graph (resp. program code)** our formalism **automata theory**
Outline

- **Software verification:**
  Example, Hoare triples, Floyd-Hoare annotation

- **Trace abstraction: new paradigm**

- **Trace abstraction: example**
  - Excursus: Correctness proofs for straightline code

- **Trace abstraction: algorithm**

- **Termination analysis**
\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 & \quad \text{if}(n == 0) \\
\quad & \quad \quad \{ \\
\ell_3: & \quad \quad p := 0; \\
\quad & \quad \} \\
\ell_4: & \quad n--; \\
\}
\end{align*}

pseudocode

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

**pseudocode**

**control flow graph**
1. take trace $\pi_1$
1. take trace $\pi_1$
2. consider trace as program $P_1$

```
1: assume p != 0;
2: assume n >= 0;
3: assert p != 0;
```

pseudocode of $P_1$
Outline

- Software verification:
  Example, Hoare triples, Floyd-Hoare annotation

- Trace abstraction: new paradigm

- Trace abstraction: example
  - Excursus: Correctness proofs for straightline code

- Trace abstraction: algorithm

- Termination analysis
Excursus: Correctness proofs for straightline code

- Step 1: Analyze correctness

- Naive approach: symbolic execution
- Alternatives:
  - symbolic execution + unsat cores
  - Craig interpolation

Excursus: Correctness proofs for straightline code

- Step 1: Analyze correctness

SMT script

```
(declare-fun x () Int)
(declare-fun n () Int)
(assert (not (= p 0)))
(assert (>= n 0))
(assert (= p 0))
(check-sat)
```
Excursus: Correctness proofs for straightline code

- Step 1: Analyze correctness
- Step 2: Construct proof
  - Naive approach: symbolic execution

```
p \neq 0
n \geq 0
p = 0
true
p \neq 0
p \neq 0 \land n \geq 0
false
```
Excursus: Correctness proofs for straightline code

- Step 1: Analyze correctness
- Step 2: Construct proof
  - Naive approach: symbolic execution
  - Alternatives:
    - symbolic execution + unsat cores
    - Craig interpolation
Excursus: Correctness proofs for straightline code

- Step 1: Analyze correctness
- Step 2: Construct proof
  - Naive approach: symbolic execution
  - Alternatives:
    - symbolic execution + unsat cores
    - Craig interpolation

SMT script

```
(declare-fun x () Int)
(declare-fun n () Int)
(assert (! (not (= p 0)) :named stmt1))
(assert (! (>= n 0) :named stmt2))
(assert (! (= p 0) :named stmt3))
(check-sat)
(get-interpolants stmt1 stmt2 stmt3)
```
1. take trace $\pi_1$
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness of $\mathcal{P}_1$
1. take trace $\pi_1$
2. consider trace as program $P_1$
3. analyze correctness or $P_1$
4. generalize program $P_1$
   - add transitions

\[
\begin{align*}
\{ p \neq 0 \} & \quad n-- \quad \{ p \neq 0 \} \quad \text{is valid Hoare triple}
\end{align*}
\]
Trace Abstraction: Example

1. take trace $\pi_1$
2. consider trace as program $P_1$
3. analyze correctness or $P_1$
4. generalize program $P_1$
   ▶ add transitions

\[
\{ p \neq 0 \} \quad n-- \quad \{ p \neq 0 \}
\]
is valid Hoare triple

\[
\{ p \neq 0 \} \quad n ! = 0 \quad \{ p \neq 0 \}
\]
is valid Hoare triple

\[
\{ p \neq 0 \} \quad n >= 0
\]
is valid Hoare triple
Trace Abstraction: Example

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

\[ \{p \neq 0\} \text{ n-- } \{p \neq 0\} \text{ is valid Hoare triple} \]
\[ \{p \neq 0\} \text{ n != 0 } \{p \neq 0\} \text{ is valid Hoare triple} \]
\[ \{p \neq 0\} \text{ n >= 0 } \{p \neq 0\} \text{ is valid Hoare triple} \]
Trace Abstraction: Example

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

\[
\begin{align*}
\text{true} \\
p \neq 0 \\
p \neq 0 \\
\text{n} \geq 0 \\
p \neq 0 & \quad \text{all } \{p := 0\} \\
p = 0 \\
\text{false}
\end{align*}
\]
Trace Abstraction: Example

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

\[
\begin{align*}
true & \quad \text{all} \\
p \neq 0 & \quad \text{all} \{ p := 0 \} \\
p = 0 & \quad \text{all} \{ p := 0 \} \\
false & \quad \text{all}
\end{align*}
\]
1. take trace $\pi_1$
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions
   - merge locations
Trace Abstraction: Example

Consider only traces in set theoretic difference $\mathcal{P} \setminus \mathcal{P}_1$. 

Program $\mathcal{P}$

Program $\mathcal{P}_1$
Trace Abstraction: Example

Consider only traces in set theoretic difference $P \setminus P_1$.
Trace Abstraction: Example

Consider only traces in set theoretic difference $\mathcal{P} \setminus \mathcal{P}_1$. 
Consider only traces in set theoretic difference $\mathcal{P} \setminus \mathcal{P}_1$. 
Trace Abstraction: Example

1. take trace $\pi_2$

\[\begin{align*}
p &\neq 0 \\
n &\geq 0 \\
n &\equiv 0 \\
p &:= 0 \\
n &:= 0 \\
n &\geq 0 \\
p &\equiv 0
\end{align*}\]
Trace Abstraction: Example

1. take trace $\pi_2$
2. consider trace as program $\mathcal{P}_2$
1. take trace $\pi_2$
2. consider trace as program $\mathcal{P}_2$
3. analyze correctness or $\mathcal{P}_2$
Trace Abstraction: Example

1. take trace $\pi_2$
2. consider trace as program $\mathcal{P}_2$
3. analyze correctness or $\mathcal{P}_2$
4. generalize program $\mathcal{P}_2$
   - add transitions
   - merge locations
Trace Abstraction: Example

program $\mathcal{P}$

program $\mathcal{P}_1$

program $\mathcal{P}_2$
Trace Abstraction: Example

Program $\mathcal{P}$

- $\ell_0$
  - $p \neq 0$
- $\ell_1$
  - $n < 0 \rightarrow \ell_5$
  - $n \geq 0$
  - $n = 0$
- $\ell_2$
  - $p = 0 \rightarrow \ell_{\text{err}}$
  - $n = 0$
  - $n \neq 0$
- $\ell_4$
  - $p := 0$

Program $\mathcal{P}_1$

- $q_0$
  - $p \neq 0$
- $q_1$
  - $p = 0$
    - $\{ p := 0 \}$
- $q_2$
  - $n --$
  - $\{ p := 0 \}$

Program $\mathcal{P}_2$

- $q_0$
  - $n = 0$
- $q_1$
  - $n --$
  - $\{ n -- \}$
- $q_2$
  - $n = 0$
- $q_3$
  - $n --$
  - $\{ n -- \}$

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

- Software verification: Example, Hoare triples, Floyd-Hoare annotation

- Trace abstraction: new paradigm

- Trace abstraction: example
  - Excursus: Correctness proofs for straightline code

- Trace abstraction: algorithm

- Termination analysis
**Trace Abstraction: Verification Algorithm**

- **Program** $\mathcal{P}$
- $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P}_1) \cup \cdots \cup \mathcal{L}(\mathcal{P}_n)$
- **Construct infeasibility proof for $\pi$**
- **Construct generalized program $\mathcal{P}_i$**
- **Is $\pi$ feasible?**
  - **No**
    - Pick new error trace $\pi$
    - **Yes**
  - **Yes**

**“$\mathcal{P}$ is correct”**

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

$L(\mathcal{P}) \subseteq L(\mathcal{P}_1) \cup \cdots \cup L(\mathcal{P}_n)$

- yes
  - "\(\mathcal{P}\) is correct"
  - no
    - pick new error trace \(\pi\)
    - no
      - construct infeasibility proof for \(\pi\)
      - yes
        - construct generalized program \(\mathcal{P}_i\)
  - no
    - is \(\pi\) feasible?
      - no
        - construct infeasibility proof for \(\pi\)
      - yes
        - construct generalized program \(\mathcal{P}_i\)
Trace Abstraction: Verification Algorithm

Program $\mathcal{P}$

$\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P}_1) \cup \cdots \cup \mathcal{L}(\mathcal{P}_n)$

Is $\pi$ feasible?

Construct infeasibility proof for $\pi$

Construct generalized program $\mathcal{P}_i$

Yes

Pick new error trace $\pi$

No

$\mathcal{P}$ is correct

$\mathcal{P}$ is incorrect
Trace Abstraction: Verification Algorithm

- **Program** $P$
- **$\mathcal{L}(P) \subseteq \mathcal{L}(P_1) \cup \cdots \cup \mathcal{L}(P_n)$**
- **Is $\pi$ feasible?**
  - **No**: pick new error trace $\pi$
  - **Yes**: construct infeasibility proof for $\pi$
    - **Construct generalized program $P_i$**

"$P$ is correct"  
"$P$ is incorrect"
Trace Abstraction: Verification Algorithm

\[ \mathcal{L}(P) \subseteq \mathcal{L}(P_1) \cup \cdots \cup \mathcal{L}(P_n) \]

- **Program** $P$
  - If $\mathcal{L}(P) \subseteq \mathcal{L}(P_1) \cup \cdots \cup \mathcal{L}(P_n)$, then $\pi$ is feasible.
  - If no, pick new error trace $\pi$ and go back to the beginning.
  - If yes, go to the next step.

- **Construct infeasibility proof for $\pi$**
  - If no, construct generalized program $P_i$ and go back to the beginning.
  - If yes, go to the next step.

- **Is $\pi$ feasible?**
  - If no, pick new error trace $\pi$ and go back to the beginning.
  - If yes, go to the next step.

- **“$P$ is correct”**
- **“$P$ is incorrect”**
A classical approach to software model checking:

program $P$ \overset{\text{Floyd-Hoare annotation for $P$}}{\longrightarrow} \text{correct program $P$}
A classical approach to software model checking:

[Diagram: program $\mathcal{P}$ to Floyd-Hoare annotation for $\mathcal{P}$]

Our approach to software model checking:

[Diagram: sample trace $\pi_1$ to Floyd-Hoare annotation for $\pi_1$]
A classical approach to software model checking:

- Program $\mathcal{P}$
- Floyd-Hoare annotation for $\mathcal{P}$

Our approach to software model checking:

- Sample trace $\pi_1$
- Floyd-Hoare annotation for $\pi_1$
- Correct program $\mathcal{P}_1$
A classical approach to software model checking:

- **Program** $\mathcal{P}$
- Floyd-Hoare annotation for $\mathcal{P}$

Our approach to software model checking:

- **Sample trace** $\pi_1$
- Floyd-Hoare annotation for $\pi_1$
- Correct program $\mathcal{P}_1$

$\mathcal{P} \subseteq \mathcal{P}_1$
A classical approach to software model checking:

Our approach to software model checking:

\[ P \subseteq P_1 \cup \cdots \cup P_n \]
A classical approach to software model checking:

Program $\mathcal{P}$ → Floyd-Hoare annotation for $\mathcal{P}$

Our approach to software model checking:

Sample trace $\pi_1$ → Floyd-Hoare annotation for $\pi_1$ → Correct program $\mathcal{P}_1$

Sample trace $\pi_2$ → Floyd-Hoare annotation for $\pi_2$ → Correct program $\mathcal{P}_2$

... → ... → Covering of $\mathcal{P}$ by correct programs $\mathcal{P} \subseteq \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_n$
## Extensions of Trace Abstraction Approach

### Interprocedural programs

<table>
<thead>
<tr>
<th>Authors</th>
<th>Title</th>
<th>Conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>H., Hoenicke, Podelski</td>
<td>Nested Interpolants</td>
<td>POPL 2010</td>
</tr>
<tr>
<td>Cassez, Müller, Burnett</td>
<td>Summary-Based Inter-Procedural Analysis via Modular Trace Refinement</td>
<td>FSTTCS 2014</td>
</tr>
</tbody>
</table>

### Concurrent systems

<table>
<thead>
<tr>
<th>Authors</th>
<th>Title</th>
<th>Conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Farzan, Kincaid, Podelski</td>
<td>Concurrent systems</td>
<td>POPL 2014</td>
</tr>
<tr>
<td>Cassez, Ziegler</td>
<td>Verification of Concurrent Programs Using Trace Abstraction Refinement</td>
<td>LPAR 2015</td>
</tr>
</tbody>
</table>

### Timed systems

<table>
<thead>
<tr>
<th>Authors</th>
<th>Title</th>
<th>Conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wang, Sipma</td>
<td>Trace Abstraction Refinement for Timed Automata</td>
<td>ATVA 2014</td>
</tr>
<tr>
<td>Cassez, Jensen, Larsen</td>
<td>Refinement of Trace Abstraction for Real-Time Programs</td>
<td>RP 2017</td>
</tr>
</tbody>
</table>

### Solving horn clauses / Tree automata

<table>
<thead>
<tr>
<th>Authors</th>
<th>Title</th>
<th>Conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kafle, Gallagher</td>
<td>Tree Automata-Based Refinement with Application to Horn Clause Verification.</td>
<td>VMCAI 2015</td>
</tr>
<tr>
<td>Wang, Jiao</td>
<td>Trace abstraction refinement for solving Horn clauses</td>
<td>The Computer Journal 2016</td>
</tr>
</tbody>
</table>
Outline

- Software verification:
  - Example, Hoare triples, Floyd-Hoare annotation

- Trace abstraction: new paradigm

- Trace abstraction: example
  - Excursus: Correctness proofs for straightline code

- Trace abstraction: algorithm

- Termination analysis
Termination Analysis
Termination Analysis

- Challenge 1: counterexample to termination is infinite execution

```plaintext
while (x > 0) {
  x--;  
}
```
Termination Analysis

- Challenge 1: counterexample to termination is infinite execution

  Solution: consider infinite traces, use $\omega$-words and Büchi automata
Termination Analysis

- Challenge 1: counterexample to termination is infinite execution
  
  Solution: consider infinite traces, use \( \omega \)-words and Büchi automata

- Challenge 2: An infinite trace may not have any execution although each finite prefix has an execution.
  
  E.g., \((\text{x > 0} \quad \text{x--})^\omega\)

  ```plaintext
  while (x > 0) {
    x--;  
  }
  ```
Termination Analysis

- **Challenge 1:** counterexample to termination is infinite execution

  Solution: consider infinite traces, use $\omega$-words and Büchi automata

- **Challenge 2:** An infinite trace may not have any execution although each finite prefix has an execution.

  E.g., $(\underbrace{x > 0}_{x > 0} \underbrace{x--}_{x--})^\omega$

  Solution: ranking functions (here: $f(x)=x$)

**Ranking Function (for a Loop)**

Function from program states to well-founded domain such that value is decreasing while executing the loop body.
Proof by contradiction for the absence of infinite executions.
Example: Bubble Sort

program sort(int i, int a[])
  \( l_1 \) while (i>0)
  \( l_2 \)   int j:=1
  \( l_3 \)   while(j<i)
              \( l_4 \)     if (a[j]>a[i])
                           \( l_4 \)       swap(a,i,j)
              \( l_4 \)     j++
  \( l_5 \)   i--
Example: Bubble Sort

program sort(int i)
\[\ell_1\] while (i>0)
\[\ell_2\] int j:=1
\[\ell_3\] while(j<i)
\[\ell_4\] j++
\[\ell_5\] i--
Example: Bubble Sort

```
program sort(int i)
  l₁ while (i>0)
  l₂   int j:=1
  l₃ while(j<i)
  l₄   j++
  l₅   i--
```

quadratic ranking function:

\[
f(i, j) = i^2 - j
\]

lexicographic ranking function:

\[
f(i, j) = (i, i - j)
\]
program $\mathcal{P}$

$$(\text{Outer + Inner})^\omega$$

module $\mathcal{P}_1$

$$(\text{Inner}^\ast \cdot \text{Outer})^\omega$$

module $\mathcal{P}_2$

$$(\text{Inner + Outer})^\ast \cdot \text{Inner}^\omega$$

ranking function

\[
f(i, j) = i
\]

ranking function

\[
f(i, j) = i - j
\]
program $\mathcal{P}$

$$(\text{Outer} + \text{Inner})^\omega = (\text{Inner}^* \cdot \text{Outer})^\omega + (\text{Inner} + \text{Outer})^* \cdot \text{Inner}^\omega$$

module $\mathcal{P}_1$

module $\mathcal{P}_2$

```
ranking function
f(i, j) = i
```

```
ranking function
f(i, j) = i - j
```
\begin{itemize}
\item \( i-- \) \( \ell_1 \) \( i>0 \) \( l_2 \)
\item \( j>=i \) \( l_3 \) \( j:=1 \) \( l_2 \)
\item \( j++ \) \( l_4 \) \( j<i \) \( l_3 \)
\end{itemize}
input: ultimately periodic trace

\[ i > 0, \ j := 1 \ ( j < i, \ j++ )^\omega, \]
From $\omega$-Trace to Terminating Program – Example

input: ultimately periodic trace $\omega$

1. consider $\omega$-trace as program with single while loop

2. synthesize ranking function $f(i, j) = i - j$

3. compute rank certificate

4. add additional transitions
From \( \omega \)-Trace to Terminating Program – Example

input: ultimately periodic trace \( i > 0 \ j := 1 (j < i \ j++) \omega \),

1. consider \( \omega \)-trace as program with single while loop

\[
\begin{array}{c}
\ell_1 \quad i > 0 \rightarrow \ell_2 \ j := 1 \rightarrow \ell_3 \ j < i \ j++ \rightarrow \ell_4
\end{array}
\]

2. synthesize ranking function

\[
f(i, j) = i - j
\]

- Colón, Sipma Synthesis of Linear Ranking Functions (TACAS 2001)
- Podelski, Rybalchenko A complete method for the synthesis of linear ranking functions (VMCAI 2004)
- Bradley, Manna, Sipma Termination Analysis of Integer Linear Loops (CONCUR 2005)
- Bradley, Manna, Sipma Linear ranking with reachability (CAV 2005)
- Bradley, Manna, Sipma The polyranking principle (ICALP 2005)
- Ben-Amram, Genaim Ranking functions for linear-constraint loops (POPL 2013)
- H., Hoenicke, Leike, Podelski Linear Ranking for Linear Lasso Programs (ATVA 2013)
- Cook, Kroening, Rümmer, Wintersteiger Ranking function synthesis for bit-vector relations (FMSD 2013)
- Leike, H. Ranking Templates for Linear Loops (TACAS 2014)
From $\omega$-Trace to Terminating Program – Example

input: ultimately periodic trace $\omega$, 

1. consider $\omega$-trace as program with single while loop

2. synthesize ranking function

$$f(i, j) = i - j$$

3. compute rank certificate

$$\begin{align*}
\text{oldrk} &= \infty \\
i - j &\leq \text{oldrk} \\
i - j &\geq 0
\end{align*}$$
From $\omega$-Trace to Terminating Program – Example

input: ultimately periodic trace $\omega$:

1. consider $\omega$-trace as program with single while loop

   ![Diagram of while loop]

   $i > 0 \rightarrow j := 1 \rightarrow j < i \rightarrow j++$

2. synthesize ranking function

   $f(i, j) = i - j$

3. compute rank certificate

   ![Diagram of rank certificate]

   $\text{oldrnk} = \infty$

   $i - j \leq \text{oldrnk}$

   $i - j \geq 0$

4. add additional transitions

   ![Diagram of additional transitions]

   $\Sigma \rightarrow \Sigma$

   $\text{oldrnk} = \infty$

   $i - j < \text{oldrnk}$
Generalization of Program with Rank Certificate

▶ Case 1: $q_1$ not accepting

Hoare triple

\[
\{ \text{state assertion 1} \} \text{ stmt } \{ \text{state assertion 2} \}
\]

automaton transition

\[
q_1 \xrightarrow{\text{stmt}} q_2
\]

state assertion 1  state assertion 2
Case 1: $q_1$ not accepting

Hoare triple

\[
\{ \text{state assertion 1} \} \text{ stmt } \{ \text{state assertion 2} \}
\]

automaton transition

\[
q_1 \xrightarrow{\text{stmt}} q_2
\]

state assertion 1

state assertion 2

Case 2: $q_1$ accepting

Hoare triple

\[
\{ \text{state assertion 1} \} \text{ oldrnk:=f(x) stmt } \{ \text{state assertion 2} \}
\]

automaton transition

\[
q_1 \xrightarrow{\text{oldrnk:=f(x) stmt}} q_2
\]

state assertion 1

state assertion 2
int main() {
    int p, n;
    p = 42;
    while (n >= 0) {
        // assert p != 0;
        if (n == 0) {
            p = 0;
        }
        n--;
    }
    return 0;
}
Thank you for your attention!
Matthias Heizmann.  
*Traces, interpolants, and automata: a new approach to automatic software verification.*  

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski.  
Software model checking for people who love automata.  

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski.  
Termination analysis by learning terminating programs.  