Explaining Concurrency Bugs with Interpolants

Andreas Holzer\textsuperscript{2}, Daniel Schwartz-Narbonne\textsuperscript{3}, Mitra Tabaei\textsuperscript{1}, Georg Weissenbacher\textsuperscript{1}, Thomas Wies\textsuperscript{3}

\textsuperscript{1}Vienna University of Technology
\textsuperscript{2}University of Toronto
\textsuperscript{3}New York University

AVM, Austria
Debugging

Test input → input → Execution → output → Violating Specification
Debugging

Test input

Execution

What went wrong?

Error Explanation
  Cause of failure

Fault Localization
  Location of fault

P

input

output

Violating Specification
Automatic Debugging Techniques

• Dynamic analysis:
  o Comparison of failing and passing traces
    • Quality of test suite

• Symbolic execution analysis:
  o Max-SAT
    • Cause clue clauses [PLDI11]
  o Interpolation
    • Error Invariant [FM12]
    • Flow-sensitive Fault Localization [VMCAI13]
    • Hybrid Algorithm [VSSTE14]
Overview of our Method

• A concurrency bug explanation technique:
  o Symbolic execution analysis
  o Interpolation

• A general framework for concurrency bug explanation
  o Not relying on specific bug characteristic
  o No given pattern templates or annotations
Outline

• Notion of Interpolant
• Interpolants for debugging sequential traces
  o Encoding control-dependencies
    ▪ Flow-sensitive slices

• Interpolants for explaining concurrency bugs
  o Encoding:
    ▪ Locks
    ▪ Inter-thread data-dependencies

• Empirical Evaluation
Interpolants

Given: an unsatisfiable conjunction of formulas $A \land B$:

$A \land B \equiv \text{false}$

An Interpolant for $A \land B$ is a formula $I$ s.t.:

- $A \Rightarrow I$
- $I \land B \equiv \text{false}$
- $I$ is only on common variables of $A$ and $B$
Trace Formula
Trace Formula

Failing trace

Trace in SSA form

In SSA form:
- Every variable is assigned once:
  Example:
  \[ x_0 = 1 \]
  \[ y = x_0 + 5 \]
  \[ x_1 = 2 \]
Trace Formula

Failing trace

Trace in SSA form

program statement

assertion

input

input

$T_i$: SSA form of program statements

 Unsatisfiable trace formula:

$$input \land T_1 \land T_2 \ldots \land T_n \land \text{assertion}$$
Interpolants for Debugging

Trace formula

Interpolant at Position P:

\[ X \implies I_P \implies \neg Y \]
Interpolants for Debugging

Interpolant Sequence

true

$I_0 \rightarrow I_1 \rightarrow \ldots \rightarrow I_p \rightarrow \ldots \rightarrow I_{n-1} \rightarrow$ false

Trace formula
Interpolants for Debugging

Over-approximation of reachable states at p

Trace formula

\[ \text{true} \rightarrow I_0 \rightarrow I_1 \rightarrow \ldots \rightarrow I_p \rightarrow \ldots \rightarrow I_{n-1} \rightarrow \text{false} \]
Interpolants for Debugging

\[ I_0 \rightarrow \cdots \rightarrow I_{p-1} \equiv I_p \rightarrow I_p \rightarrow \cdots \rightarrow I_{n-1} \rightarrow \text{false} \]

\[ S_{p-1} = S_p \]
Interpolants for Debugging

$\text{true} \xrightarrow{} I_0 \xrightarrow{} I_1 \xrightarrow{} \cdots \xrightarrow{} I_{p-1} \times I_p \xrightarrow{} \cdots \xrightarrow{} I_{n-1} \xrightarrow{} \text{false}$

$I_{p-1} \equiv I_p$

$S_{p-1} = S_p$
## Interpolants for Debugging

### Sample Trace

<table>
<thead>
<tr>
<th>Step</th>
<th>Code</th>
<th>Trace SSA encoding</th>
<th>Interpolants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>x=3;</td>
<td>$x_0=3$</td>
<td>true</td>
</tr>
<tr>
<td>2.</td>
<td>y=5;</td>
<td>$y_0=5$</td>
<td>$x_0=3$</td>
</tr>
<tr>
<td>3.</td>
<td>m=y+x;</td>
<td>$m_0=y_0+x_0$</td>
<td>$m_0=y_0+3$, $x_0=3$</td>
</tr>
<tr>
<td>4.</td>
<td>n=y-x;</td>
<td>$n_0=y_0-x_0$</td>
<td>$m_0=n_0+6$</td>
</tr>
<tr>
<td>5.</td>
<td>y=y+1;</td>
<td>$y_1=y_0+1$</td>
<td>$m_0=n_0+6$</td>
</tr>
<tr>
<td>6.</td>
<td>assert(n&gt;m);</td>
<td>$n_0&gt;m_0$</td>
<td>false</td>
</tr>
</tbody>
</table>

### Interpolants

- Interpolants
- contain enough information to understand the failure
### Interpolants for Debugging

#### Sample Trace

1. \( x=3; \)
2. \( y=5; \)
3. \( m=y+x; \)
4. \( n=y-x; \)
5. \( y=y+1; \)
6. \( \text{assert}(n>m); \)

#### Trace SSA encoding

<table>
<thead>
<tr>
<th>Sample Trace</th>
<th>Trace SSA encoding</th>
<th>Interpolants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( x=3; )</td>
<td>1. ( x_0=3 )</td>
<td>( x_0=3 ) <strong>true</strong></td>
</tr>
<tr>
<td>2. ( y=5; )</td>
<td>2. ( y_0=5 )</td>
<td>Irrelevant</td>
</tr>
<tr>
<td>3. ( m=y+x; )</td>
<td>3. ( m_0=y_0+x_0 )</td>
<td>( m_0=y_0+3, x_0=3 ) <strong>true</strong></td>
</tr>
<tr>
<td>4. ( n=y-x; )</td>
<td>4. ( n_0=y_0-x_0 )</td>
<td>( m_0=n_0+6 ) <strong>true</strong></td>
</tr>
<tr>
<td>5. ( y=y+1; )</td>
<td>5. ( y_1=y_0+1 )</td>
<td>Irrelevant</td>
</tr>
<tr>
<td>6. ( \text{assert}(n&gt;m); )</td>
<td>6. ( n_0&gt;m_0 )</td>
<td>Irrelevant</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>true</strong></td>
</tr>
</tbody>
</table>

**Unchanged**

AVM, May 4-6 2015

17
Error Explanation:

- **Slice**: Isolating relevant *statements* for assertion violation

- **Error Invariants**: Revealing the relevant *variables*

Sample Trace

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>Error Explanation (Slice)</th>
<th>Error Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>x=3;</td>
<td>1. x=3</td>
<td>true</td>
</tr>
<tr>
<td>2.</td>
<td>y=5;</td>
<td></td>
<td>x=3</td>
</tr>
<tr>
<td>3.</td>
<td>m=y+x;</td>
<td>3. m=y+x</td>
<td>m=y+3, x=3</td>
</tr>
<tr>
<td>4.</td>
<td>n=y-x;</td>
<td>4. n=y-x</td>
<td>m=n+6</td>
</tr>
<tr>
<td>5.</td>
<td>y=y+1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>assert(n&gt;m);</td>
<td>6. assert(n&gt;m)</td>
<td>false</td>
</tr>
</tbody>
</table>
Error Explanation

Sample Trace

1. x=3;
2. y=5;
3. m=y+x;
4. n=y-x;
5. y=y+1;
6. assert(n>m);

Error Explanation (Slice)

1. x=3
2. x=3
3. m=y+x
3. m=y+3,x=3
4. n=y-x
4. m=n+6
5. m=n+6
6. assert(n>m)
6. false

Error Invariants

Error Explanation:

Interpolants are not unique: any formula between WP and SP

• Error Invariants: Revealing the relevant variables
Sound Error Explanation Slices

- Soundness of explanation
  - Slice forms an \textit{unsatisfiable} formula

\begin{itemize}
  \item \textbf{Sound Slice}
    \begin{itemize}
      \item 1. \(x=3\) \hspace{1cm} \text{true}
      \item 3. \(m=y+x\) \hspace{1cm} \text{x=3}
      \item 4. \(n=y-x\) \hspace{1cm} \text{m=y+3,x=3}
      \item 6. \text{assert}(n>m) \hspace{1cm} \text{false}
    \end{itemize}
\end{itemize}
Sound Error Explanation Slices

- Soundness of explanation
  - Achieved by *Inductive Interpolant Sequence* [VSSTE 2014]

\[ I_{p-1} \land T_p \Rightarrow I_p \]   Inductive property
Sound Error Explanation Slices

• Soundness of explanation
  ○ Achieved by *Inductive Interpolant Sequence* [VSSTE 2014]

\[ I_{p-1} \land T_p \Rightarrow I_p \quad \text{Inductive property} \]

\[
\begin{align*}
I_{j-1} & \equiv I_j \\
\end{align*}
\]

\[
\begin{align*}
I_{k-1} & \equiv I_k \\
\end{align*}
\]
Interpolants for Debugging

• Generating unsatisfiable trace formula
  o by SSA encoding

• Computing inductive interpolants
  o for each position in the trace

• Excluding statements
  o with stationary surrounding interpolants
## Encoding Conditions

<table>
<thead>
<tr>
<th>Sample Code</th>
<th>Sample Trace</th>
<th>Slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. x=1;</td>
<td>x_0=1</td>
<td></td>
</tr>
<tr>
<td>2. y=*;</td>
<td>y_0=-10</td>
<td></td>
</tr>
<tr>
<td>3. if (y &lt; 0)</td>
<td>(y_0&lt;0)\land x_1=0</td>
<td></td>
</tr>
<tr>
<td>4. x=0;</td>
<td>x_1\neq 0</td>
<td>x=0</td>
</tr>
<tr>
<td>5. assert(x!=0);</td>
<td></td>
<td>assert(x != 0)</td>
</tr>
</tbody>
</table>
Encoding Conditions

SSA Trace

1. $x_0 = 1$;  
2. $y_0 = -10$;  
3. $(y_0 < 0) \land x_1 = 0$;  
4. $x_1 \neq 0$  

Flow-sensitive SSA Trace

1. $x_0 = 1$;  
2. $y_0 = -10$;  
3. $(y_0 < 0) \Rightarrow x_1 = 0$;  
4. $x_1 \neq 0$

1. $x = 1$;  
2. $y = *$;  
3. if $(y < 0)$  
4. $x = 0$;  
5. assert($x \neq 0$);

1. $x = 1$;  
2. $y = *$;  
3. if $(y < 0)$  
4. $x = 0$;  
5. assert($x \neq 0$);
### Flow-sensitive Slices

#### SSA Trace

<table>
<thead>
<tr>
<th>Step</th>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$x_0 = 1$</td>
<td>true</td>
</tr>
<tr>
<td>2.</td>
<td>$y_0 = -10$</td>
<td>true</td>
</tr>
<tr>
<td>3.</td>
<td>$(y_0 &lt; 0) \land x_1 = 0$</td>
<td>false</td>
</tr>
<tr>
<td>4.</td>
<td>$x_1 \neq 0$</td>
<td>false</td>
</tr>
</tbody>
</table>

#### Flow-sensitive SSA Trace

<table>
<thead>
<tr>
<th>Step</th>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$x_0 = 1$</td>
<td>true</td>
</tr>
<tr>
<td>2.</td>
<td>$y_0 = -10$</td>
<td>true</td>
</tr>
<tr>
<td>3.</td>
<td>$(y_0 &lt; 0) \implies x_1 = 0$</td>
<td>true</td>
</tr>
<tr>
<td>4.</td>
<td>$x_1 \neq 0$</td>
<td>false</td>
</tr>
</tbody>
</table>

- **Encoding of control-dependency:**
  - Conditions are encoded as implications in SSA traces:
    \[
    \left( \bigwedge_{c \in \text{conds}} c \right) \implies x = e
    \]
Model of Concurrent Traces

Multi-threaded Programs

\[
\begin{array}{ccc}
  t_1 & t_2 & \cdots & t_n \\
  \text{Shared Variables} \\
\end{array}
\]
Model of Concurrent Traces

Multi-threaded Programs modeled as

Thread CFG

Atomic Statements

\[ [\neg c] \]

\[ [c] \]

\[ y = -1 \]

\[ x = 0 \]
Model of Concurrent Traces

Multi-threaded Programs

\[ t_1 | t_2 | \ldots | t_n \]

modeled as

Thread CFG

\[ [-c] \]

\[ [c] \]

\[ y=-1 \]

\[ x=0 \]

Atomic Statements

executing

Concurrent Trace

\[ e_1 \]

\[ e_2 \]

\[ e_3 \]

\[ \ldots \]

\[ e_n \]

Total order (interleaving semantics)
Concurrent Trace Formula

- SSA encoding of variables
- Encoding control-dependency as implication

- Modeling Locks as:
  - Atomic guarded assignments:
    acquire $\ell$: ($\ell = 0$) $\triangleright$ $\ell := tid$
    release $\ell$: ($\ell = tid$) $\triangleright$ $\ell := 0$
  - Encoding locks as implications (similar to control-dependency)
Interpolants for Debugging Concurrent Bugs

Main Thread

balance := 40
withdrawal := 20
deposit := 10
...

Thread 1

acquire ℓ
bal = balance
release ℓ

bal = bal + deposit

acquire ℓ
balance = bal
release ℓ

Thread 2

acquire ℓ
bal = balance
release ℓ

bal = bal - withdrawal

acquire ℓ
balance = bal
release ℓ

assert(balance = 40 – 20 + 10)
Interpolants for Debugging Concurrent Bugs

Main Thread

- balance := 40
- withdrawal := 20
- deposit := 10
...

Thread 1

- acquire ℓ
- bal = balance
- release ℓ

- bal = bal + deposit

- acquire ℓ
- balance = bal
- release ℓ

Thread 2

- acquire ℓ
- bal = balance
- release ℓ

- bal = bal - withdrawal

- acquire ℓ
- balance = bal
- release ℓ

assert(balance = 40 - 20 + 10)
Interpolants for Debugging Concurrent Bugs

**Main Thread**

- `balance := 40`
- `withdrawal := 20`
- `deposit := 10`
- ...

**Thread 1**

- `acquire ℓ`
- `bal = balance`
- `release ℓ`
- `bal = bal + deposit`
- `acquire ℓ`
- `balance = bal`
- `release ℓ`

**Thread 2**

- `acquire ℓ`
- `bal = balance`
- `release ℓ`
- `bal = bal - withdrawal`
- `acquire ℓ`
- `balance = bal`
- `release ℓ`

**Main Thread**

- `balance := 40`
- `withdrawal := 20`
- `deposit := 10`
- ...

**Main Thread**

- `assert(balance = 40 - 20 + 10)`
Interpolants for Debugging Concurrent Bugs

Main Thread
balance := 40
withdrawal := 20
deposit := 10
...

Thread 1
acquire ℓ
bal = balance
release ℓ
bal = bal + deposit
acquire ℓ
balance = bal
release ℓ

Thread 2
acquire ℓ
bal = balance
release ℓ
bal = bal - withdrawal
acquire ℓ
balance = bal
release ℓ

Main Thread
assert(balance = 40 – 20 + 10)
Interpolants for Debugging Concurrent Bugs

Thread 1

acquire \( \ell \)
bal = balance
release \( \ell \)

bal = bal + deposit

acquire \( \ell \)
balance = bal
release \( \ell \)

Thread 2

acquire \( \ell \)
bal = balance
release \( \ell \)

bal = bal - withdrawal

acquire \( \ell \)
balance = bal
release \( \ell \)

Main Thread

balance := 40
withdrawal := 20
deposit := 10

assert(balance = 30)
Interpolants for Debugging Concurrent Bugs

Main Thread
balance := 40
withdrawal := 20
deposit := 10
...

Thread 1
acquire \ell
bal = balance
release \ell
bal = bal + deposit
acquire \ell
balance = bal
release \ell
balance ← 50

Thread 2
acquire \ell
bal = balance
release \ell
bal = bal - withdrawal
acquire \ell
balance = bal
release \ell
balance ← 20

assert(balance = 30)
Interpolants for Debugging Concurrent Bugs

Main Thread (T₀)
- balance := 40
- withdrawal := 20
- deposit := 10

Thread 1 (T₁)
- acquire ℓ
- bal = balance
- release ℓ
- bal = bal + deposit

Thread 2 (T₂)
- acquire ℓ
- bal = balance
- release ℓ
- bal = bal - withdrawal
- acquire ℓ
- balance = bal
- release ℓ

Main Thread (T₀)
- assert(balance = 30)

Flow-insensitive Slice
- T₀ : balance := 40
- T₀ : withdrawal := 20
- T₂ : bal = balance
- T₂ : bal = bal - withdrawal
- T₂ : balance = bal
- T₀ : assert(balance = 30)

Ignoring Thread 1 altogether
Interpolants for Debugging Concurrent Bugs

Main Thread (T₀)
- balance := 40
- withdrawal := 20
- deposit := 10

Thread 1 (T₁)
- acquire ℓ
- bal = balance
- release ℓ
- bal = bal + deposit
- acquire ℓ
- balance = bal
- release ℓ

Thread 2 (T₂)
- acquire ℓ
- bal = balance
- release ℓ
- bal = bal - withdrawal
- acquire ℓ
- balance = bal
- release ℓ

Flow-sensitive Slice
- T₀: balance := 40
- T₀: withdrawal := 20
- T₂: acquire ℓ
- T₂: bal = balance
- T₂: release ℓ
- T₁: acquire ℓ
- T₁: release ℓ
- T₁: acquire ℓ
- T₁: release ℓ
- T₂: bal = bal - withdrawal
- T₂: acquire ℓ
- T₂: balance = bal
- T₂: release ℓ
- T₀: assert(balance = 30)
- T₁: balance = bal is missing

assert(balance = 30)
Data dependencies

- Data dependency:
  - Flow of data between statements

- Types of data dependency (in general)
  - Read-after-write
    ```
    a = x;
    y = a + 10;
    ```
  - Write-after-read
    ```
    x = a;
    a = y + 10;
    ```
  - Write-after-write
    ```
    a = x + 10;
    a = y + 10;
    ```

- Inter-thread data dependencies (in multi threaded programs)
  - Being able to indicate
    - conflicting accesses or hazards
Interpolants for Debugging Concurrent Bugs

Main Thread (T₀)
- balance := 40
- withdrawal := 20
- deposit := 10

Thread 1 (T₁)
- acquire ℓ
- bal = balance
- release ℓ
- bal = bal + deposit
- acquire ℓ
- balance = bal
- release ℓ

Thread 2 (T₂)
- acquire ℓ
- bal = balance
- release ℓ
- bal = bal - withdrawal
- acquire ℓ
- balance = bal
- release ℓ

Flow-sensitive Slice
- T₀: balance := 40
- T₀: withdrawal := 20
- T₂: acquire ℓ
- T₂: bal = balance
- T₂: release ℓ
- T₁: acquire ℓ
- T₁: release ℓ
- T₁: acquire ℓ
- T₁: release ℓ
- T₂: bal = bal - withdrawal
- T₂: acquire ℓ
- T₂: balance = bal
- T₂: release ℓ
- T₀: assert(balance = 30)
Hazard-sensitive Slices

• Encoding inter-thread data dependencies:
  o as implication (using auxiliary variables)

• The resulting slice:
  o Hazard-sensitive slice
Hazard-sensitive Slice

Main Thread (T₀)

- balance := 40
- withdrawal := 20
- deposit := 10

Thread 1 (T₁)

- acquire ℓ
- bal := balance
- release ℓ

- bal := bal + deposit

Thread 2 (T₂)

- acquire ℓ
- bal := balance
- release ℓ

- bal := bal - withdrawal

- acquire ℓ
- balance := bal
- release ℓ

assert(balance = 30)

T₁: \( v \land balance = bal \)
T₂: \( v \Rightarrow balance = bal \)
Interpolants for Debugging Concurrent Bugs

Main Thread (T₀)
- balance := 40
- withdrawal := 20
- deposit := 10

Thread 1 (T₁)
- acquire ℓ
- bal = balance
- release ℓ
- bal = bal + deposit
- acquire ℓ
- balance = bal
- release ℓ

Thread 2 (T₂)
- acquire ℓ
- bal = balance
- release ℓ
- bal = bal - withdrawal
- acquire ℓ
- balance = bal
- release ℓ

Hazard-sensitive Slice
- T₀: balance := 40
- T₀: withdrawal := 20
- T₂: bal = balance
- T₁: balance = bal
- T₂: bal = bal - withdrawal
- T₂: balance = bal
- T₀: assert(balance = 30)
Interpolants for Debugging Concurrent Bugs

Main Thread (T₀)

balance := 40
withdrawal := 20
deposit := 10

Thread 1 (T₁)

acquire ℓ
bal = balance
release ℓ
bal = bal + deposit
acquire ℓ
balance = bal
release ℓ

Thread 2 (T₂)

acquire ℓ
bal = balance
release ℓ
bal = bal - withdrawal
acquire ℓ
balance = bal
release ℓ

assert(balance = 30)

Hazard-sensitive Slice

T₀: balance := 40
{balance ≤ 40}
T₀: withdrawal := 20
{balance ≤ withdrawal + 20}

T₂: bal = balance
{bal ≤ withdrawal + 20}
T₁: balance = bal
{bal ≤ withdrawal + 20}

T₂: bal = bal – withdrawal
{bal ≤ 20}

T₂: balance = bal
{balance ≤ 20}
T₀: assert(balance = 30)
Fine-Tuning Explanations

• Adding different levels of detail to the explanations
  o Encoding:
    - control- and inter-thread data-dependency (fs+hs)
    - control-dependency (fs)
    - inter-thread data-dependency (hs)
    - no dependency (∅)
  o Leading to different reductions in number of:
    - variables
    - statements
Fine-Tuning Explanations

∅

T₀ : balance := 40
T₀ : withdrawal := 20
T₂ : bal = balance
T₂ : bal = bal - withdrawal
T₂ : balance = bal
T₀ : assert(balance = 30)

fs

T₀ : balance := 40
T₀ : withdrawal := 20
T₂ : acquire ℓ
T₂ : bal = balance
T₂ : release ℓ
T₁ : acquire ℓ
T₁ : release ℓ
T₁ : acquire ℓ
T₁ : release ℓ
T₂ : bal = bal - withdrawal
T₂ : acquire ℓ
T₂ : balance = bal
T₂ : release ℓ
T₀ : assert(balance = 30)

hs

T₀ : balance := 40
T₀ : withdrawal := 20
T₂ : bal = balance
T₁ : balance = bal
T₂ : bal = bal – withdrawal
T₂ : balance = bal
T₀ : assert(balance = 30)
## Empirical Evaluation

### Quality + Quantity results:

<table>
<thead>
<tr>
<th>Program</th>
<th>Concurrency bug</th>
<th>Number of traces</th>
<th>Type of slice</th>
<th>Avg. reduction of statements(%)</th>
<th>Avg. reduction of variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock free pool</td>
<td>Linearizability problem</td>
<td>8</td>
<td>fs</td>
<td>61%</td>
<td>34%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fs+hs</td>
<td>60%</td>
<td>34%</td>
</tr>
<tr>
<td>Bank account</td>
<td>Atomicity violation</td>
<td>5</td>
<td>fs+hs</td>
<td>46%</td>
<td>23%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>hs</td>
<td>88%</td>
<td>33%</td>
</tr>
</tbody>
</table>
Conclusion

- A general framework for concurrency bug explanation
  - Interpolation
  - Symbolic execution analysis
    - Encoding of:
      - Control-dependency
      - Inter-thread data-dependency
  - Implementation
    - Interpolant computation
      - VERMEER [ICSE15]
    - Tracing failing concurrent traces
      - ConCrest [FSE13]
Thank you!