Using Unfoldings in Automated Testing of Multithreaded Programs

Kari Kähkönen, Olli Saarikivi, Keijo Heljanko
The Problem

- How to automatically test the local state reachability in multithreaded programs
  - E.g., find assertion violations, uncaught exceptions, etc.
- The main challenge: path explosion and numerous interleavings of threads
- One approach: dynamic symbolic execution (DSE) + partial order reduction
- New approach: DSE + unfoldings
Dynamic Symbolic Execution

- DSE aims to explore different execution paths of the program under test

\[ x = \text{input} \]
\[ x = x + 5 \]

\[
\text{if } (x > 10) \{
\ ...
\}
\]

\[
\text{...}
\]

Control flow graph
**Dynamic Symbolic Execution**

- DSE typically starts with a random execution
- The program is executed concretely and symbolically

```plaintext
x = input
x = x + 5

if (x > 10) {
  ...
}
...
```

Control flow graph
Dynamic Symbolic Execution

- Symbolic execution generates constraints that can be solved to obtain new test inputs for unexplored paths.

\[ x = \text{input} \]
\[ x = x + 5 \]

\[
\begin{align*}
\text{if } (x > 10) & \{ \\
& \ldots \\
& \} \\
& \ldots
\end{align*}
\]

\[ c_1 = \text{input}_1 + 5 > 10 \]
\[ c_2 = \text{input}_1 + 5 \leq 10 \]
What about Multithreaded Programs?

- Take control of the scheduler
- Execute threads one by one until a global operation (e.g., access shared variable) is reached
- Branch the execution tree for each enabled operation
What about Multithreaded Programs?

• Take control of the scheduler
• Execute threads one by one until a global operation (e.g., access shared variable) is reached
• Branch the execution tree for each enabled operation

Problem: a large number of irrelevant interleavings
One Solution: Partial-Order Reduction

• Ignore provably irrelevant parts of the symbolic execution tree

• Existing algorithms:
  – dynamic partial-order reduction
  – race detection and flipping
Another Solution?

• Can we create a symbolic representation of the executions that contain all the interleavings but in more compact form than with execution trees?
• Yes, with unfoldings
What Are Unfoldings?

- Unwinding of a control flow graph is an execution tree
- Unwinding of a Petri net is an unfolding
- Can be exponentially more compact than exec. trees
What Are Unfoldings?

- Unwinding of a control flow graph is an execution tree
- Unwinding of a Petri net is an unfolding
- Can be exponentially more compact than exec. trees
What Are Unfoldings?

- Unwinding of a control flow graph is an execution tree
- Unwinding of a Petri net is an unfolding
- Can be exponentially more compact than exec. trees

Petri net

Unfolding
What Are Unfoldings?

- Unwinding of a control flow graph is an execution tree
- Unwinding of a Petri net is an unfolding
- Can be exponentially more compact than exec. trees
What Are Unfoldings?

- Unwinding of a control flow graph is an execution tree
- Unwinding of a Petri net is an unfolding
- Can be exponentially more compact than exec. trees
Using Unfoldings with DSE

- When a test execution encounters a global operation, extend the unfolding with one of the following events:
  - read
  - write
  - lock
  - unlock

- Potential extensions for the added event are new test targets
Example

Global variables:
int x = 0;

Thread 1:
local int a = x;
if (a > 0)
  error();

Thread 2:
local int b = x;
if (b == 0)
  x = input();

Initial unfolding
Example

Global variables:
int x = 0;

Thread 1:
local int a = x;
if (a > 0)
error();

Thread 2:
local int b = x;
if (b == 0)
x = input();

First test run
Example

Global variables:
int x = 0;

Thread 1:
local int a = x;
if (a > 0)
error();

Thread 2:
local int b = x;
if (b == 0)
x = input();

Find possible extensions
Example

Global variables:
int x = 0;

Thread 1:
local int a = x;
if (a > 0)
error();

Thread 2:
local int b = x;
if (b == 0)
x = input();
Computing Potential Extensions

• Finding potential extensions is the most computationally expensive part of unfolding

• It is possible to use existing potential extension algorithms with DSE
  – Designed for arbitrary Petri nets
  – Very expensive

• Key contribution: Possible to limit the search space of potential extensions due to restricted form of unfoldings generated by the algorithm
  – Same worst case behavior, but in practice very efficient
Comparison with DPOR and Race Detection and Flipping

• The amount of reduction obtained by dynamic partial-order approaches depend on the order events are added to the symbolic execution tree.

• Unfolding approach is computationally more expensive per test run but typically requires less test runs.
  – With threads that contains high amount of independence, the reduction to the number of test runs can be even exponential.
## Experiments

<table>
<thead>
<tr>
<th>program</th>
<th>Unfolding</th>
<th></th>
<th>DPOR (ACSD ’12)</th>
<th></th>
<th>jCUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>paths</td>
<td>time</td>
<td>paths</td>
<td>time</td>
<td>paths</td>
</tr>
<tr>
<td>Indexer (12)</td>
<td>8</td>
<td>2</td>
<td>85</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>Filesystem (16)</td>
<td>3</td>
<td>0</td>
<td>16</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>Filesystem (18)</td>
<td>4</td>
<td>0</td>
<td>97</td>
<td>6</td>
<td>2026</td>
</tr>
<tr>
<td>Parallel pi (5)</td>
<td>120</td>
<td>3</td>
<td>2698</td>
<td>17</td>
<td>120</td>
</tr>
<tr>
<td>Test selector (3)</td>
<td>65</td>
<td>2</td>
<td>87</td>
<td>2</td>
<td>65</td>
</tr>
<tr>
<td>Test selector (4)</td>
<td>2576</td>
<td>70</td>
<td>8042</td>
<td>97</td>
<td>2576</td>
</tr>
<tr>
<td>Pairs (6)</td>
<td>7</td>
<td>0</td>
<td>512</td>
<td>8</td>
<td>580</td>
</tr>
<tr>
<td>Locking (4)</td>
<td>2520</td>
<td>42</td>
<td>2520</td>
<td>13</td>
<td>2520</td>
</tr>
<tr>
<td>Synthetic-1 (3)</td>
<td>984</td>
<td>15</td>
<td>3716</td>
<td>10</td>
<td>2430</td>
</tr>
<tr>
<td>Synthetic-2 (3)</td>
<td>1943</td>
<td>54</td>
<td>7768</td>
<td>56</td>
<td>4860</td>
</tr>
<tr>
<td>Synthetic-3 (4)</td>
<td>682</td>
<td>14</td>
<td>8550</td>
<td>52</td>
<td>1757</td>
</tr>
</tbody>
</table>
Conclusions

- A new approach to test multithreaded programs by combining DSE and unfoldings
- The restricted form of the unfoldings allows efficient implementation of the algorithm
- The new algorithm offers competitive performance to existing approaches
  - In some cases it can be substantially faster