Using Unfoldings in Automated Testing of Multithreaded Programs

Kari Kähkönen, Olli Saarikivi, Keijo Heljanko
(keijo.heljanko@aalto.fi)

Published in ASE ’12
The Problem

• How to automatically test the local state reachability in multithreaded programs that read input values
  – 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 systematically explore different execution paths of the program under test

```plaintext
x = input
x = x + 5
if (x > 10) {
    ...
}
...  
```

Control flow graph
Dynamic Symbolic Execution

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

```java
x = input
x = x + 5
if (x > 10) {
    ...
}
...
```

Control flow graph
Dynamic Symbolic Execution

• Symbolic execution generates constraints at branch points that define input values leading to true and false branches

\[
x = \text{input}\n\]
\[
x = x + 5
\]
\[
\text{if } (x > 10) \{
\]
\[
\quad \ldots
\]
\[
\}
\[
\quad \ldots
\]

Control flow graph

\[
c_1 = \text{input}_1 + 5 > 10
\]
\[
c_2 = \text{input}_1 + 5 \leq 10
\]
Dynamic Symbolic Execution

• A conjunction of symbolic constraints along an execution path is called a path constraint
  – Solved using SMT-solvers to obtain concrete test inputs for unexplored execution paths
  – E.g., pc: $\text{input}_1 + 5 > 10 \land \text{input}_2 \times \text{input}_1 = 50$
  – Solution: $\text{input}_1 = 10$ and $\text{input}_2 = 5$
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
Dynamic Partial-Order Reduction (DPOR)

- DPOR algorithm by Flanagan and Godefroid (2005) calculates what additional interleavings need to be explored from the history of the current execution.
- Once DPOR has fully explored the subtree from a state, it will have explored a persistent set of operations from that state:
  - Will find all assertion violations and deadlocks.
  - Will execute at least one test run from each Mazurkiewicz trace.
Identifying Backtracking Points in DPOR

• When a race is identified during execution, a backtracking point is added to be explored later
• To do so, DPOR tracks the causal relationships of global operations in order to identify backtracking points
• In typical implementations the causal relationships are tracked by using vector clocks
• An optimized DPOR approach can be found from:
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
  • When the executed tests cover the symbolic representation completely, the testing process can be stopped
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
Encoding Shared variables and locks

- **Read global variable**
  - `pc_i` -> `X_{1,1}`
  - `pc_j` -> `X_{1,2}`

- **Write global variable**
  - `pc_i` -> `X_{1,1}`
  - `pc_j` -> `X_{1,2}`

- **Symbolic branching**
  - `pc_i` -> `true`
  - `pc_i` -> `false`

- **Release lock l**
  - `pc_j` -> `l_x`

- **Acquire lock l**
  - `pc_j` -> `l_y`
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();
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
  – Can potentially be very expensive

• Key observation: It is 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 quite a bit more efficient
Computing Potential Extensions

• In a Petri net representation of a program under test (not constructed explicitly in our algorithm) the places for shared variables are always marked.

• This results in a tree like connection of the unfolded shared variable places and allows efficient potential extension computations.

Thread 1:
local int a = x; (read)

Thread 2:
x = 5; (write)
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 generates canonical representation regardless of the execution order
Comparison with DPOR and Race Detection and Flipping

• Unfolding approach is computationally more expensive per test run but requires less test runs
  – The reduction to the number of test runs can be exponential
  – Consider a system with 2n threads and n shared variables, which consist of a thread reading ($r_i$) and writing ($w_i$) variable $i$.
  – System with exponential number of Mazurkiewics traces but a linear size unfolding:
Additional Observations

- The unfolding approach is especially useful for programs whose control depends heavily on input values
  - DPOR might have to explore large subtrees generated by DSE multiple times if it does not manage to ignore all irrelevant interleavings of threads
- One limitation of the current algorithm is that it does not cleanly support dynamic thread creation
  - Possible solution: contextual nets with read-arcs
## Experiments (old from ASE ’12)

<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>
## Experiments (new)

<table>
<thead>
<tr>
<th>program</th>
<th>Unfolding paths</th>
<th>Unfolding time</th>
<th>DPOR paths</th>
<th>DPOR time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Szymanski</td>
<td>65138</td>
<td>1m 5s</td>
<td>65138</td>
<td>0m 30s</td>
</tr>
<tr>
<td>Filesystem 1</td>
<td>3</td>
<td>0m 0s</td>
<td>135</td>
<td>0m 3s</td>
</tr>
<tr>
<td>Filesystem 2</td>
<td>3</td>
<td>0m 0s</td>
<td>992</td>
<td>0m 30s</td>
</tr>
<tr>
<td>Fib 1</td>
<td>19605</td>
<td>0m 17s</td>
<td>21034</td>
<td>0m 19s</td>
</tr>
<tr>
<td>Fib 2</td>
<td>218243</td>
<td>3m 16s</td>
<td>232631</td>
<td>2m 58s</td>
</tr>
<tr>
<td>Updater 1</td>
<td>33269</td>
<td>5m 51s</td>
<td>33464</td>
<td>6m 1s</td>
</tr>
<tr>
<td>Updater 2</td>
<td>33497</td>
<td>6m 3s</td>
<td>34033</td>
<td>6m 8s</td>
</tr>
<tr>
<td>Locking</td>
<td>2520</td>
<td>0m 8s</td>
<td>2520</td>
<td>0m 6s</td>
</tr>
<tr>
<td>Synthetic 1</td>
<td>926</td>
<td>0m 5s</td>
<td>1650</td>
<td>0m 7s</td>
</tr>
<tr>
<td>Synthetic 2</td>
<td>8205</td>
<td>0m 42s</td>
<td>22475</td>
<td>1m 36s</td>
</tr>
</tbody>
</table>
Conclusions

• A new approach to test multithreaded programs
• The restricted form of the unfoldings allows efficient implementation of the algorithm
• Unfoldings are competitive with existing approaches and can be substantially faster in some cases
• Future work:
  • Encoding the unfolding as SMT formulas in order to check global properties of the program under test
  • Even more compact representations with read-arcs