Sherlock: Scalable Deadlock Detection for Concurrent Programs

Mahdi Eslamimehr, Jens Palsberg

David Wellig
ETH Zürich

11.03.2015
Clarification of terms

Types of deadlock-detection techniques:
- static: detection without running the program
- dynamic: gathers information during one or more runs
- hybrid: combination of static and dynamic

Scalable:
- The property of an algorithm of being suitably efficient and practical when applied to large situations.

Sherlock is a *dynamic* deadlock-detection algorithm for Java programs, which works well for *large schedules* and which especially determines the schedule to a deadlock.
Types of deadlocks treated by Sherlock

<table>
<thead>
<tr>
<th>$\overline{T_1}$</th>
<th>$\overline{T_2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 : synchronized($A$){</td>
<td>1 : synchronized($B$){</td>
</tr>
<tr>
<td>2 : synchronized($B$){}</td>
<td>2 : synchronized($A$){}</td>
</tr>
<tr>
<td>...}</td>
<td>...}</td>
</tr>
</tbody>
</table>

- synchronized($A$)${s}$: Thread acquires lock of A and executes statement s.
- In the state ($T_1 \triangleright 2$, $T_2 \triangleright 2$) we are deadlocked.
Basic idea of Sherlock

Basic structure of the algorithm:

\[(execute \circ permute)^i \circ execute, \quad i \in \mathbb{N}.\]
Execute function

Interface of the \textit{execute} function:

\[
\text{execute} : \text{Program} \times \text{Schedule} \times \text{Candidate} \rightarrow \\
\text{Input} \times \text{Schedule} \times \text{Bool} \oplus \{\text{none}\}
\]

Suppose we have program \(p\), schedule \(s\) and candidate \(c\):

\[
(a, \text{trace}, \text{found}) = \text{execute}(p, s, c).
\]

If \textit{found} is true, \textit{trace} is the actual schedule that leads to the candidate \(c\) which then in fact is a deadlock. \(a\) is the input to the program.

\textit{execute} uses concolic execution, which is an execution that records \textit{constraints}. 

David Wellig ETH Zürich

Sherlock: Scalable Deadlock Detection for Concurrent Programs
Let $y$ be an input, and $s$ a statement. Consider:

$$x = 6; \text{ if } (y > 4) \{ s \};$$

How to find an input that leads to $s$?

First run:

$$y = 0 \Rightarrow (y > 4) \text{ prevents executing } s \Rightarrow (y > 4) \text{ is recorded.}$$

Second run:

$$y = 8 \Rightarrow \text{ Success.}$$
Permute function (1)

Interface of the `permute` function:

\[ \text{permute} : \text{Schedule} \times \text{Candidate} \rightarrow \text{Schedule} \oplus \{ \text{none} \} \]

`permute` permutes the events of schedule \( \pi = \langle e_1, \ldots, e_n \rangle \), i.e.

\[ \text{permute}(\pi, c) = \langle e_{\sigma(1)}, \ldots, e_{\sigma(n)} \rangle, \quad \text{for a } \sigma \in S_n, \]

where \( S_n \) is the set of all permutations of \( n \) elements. Furthermore \( \sigma \) has to satisfy the following constraints

\[ a_\pi \land \beta_\pi \land \Psi(V,E) \land \delta_c. \]
Permute function (2)

Constraints:

- $a_\pi$: Happens-before relation.
- $\beta_\pi$: Write-read consistency. Read event reads value written by most recent write event.
- $\Psi_{(V,E)}$: Lock-order constraints. $(V, E)$ is the lock-order graph. Nodes $V$ are events that acquire locks.
- $\delta_c$: Representation of deadlock-candidate $c$. 
A closer look on Sherlock

```
(Deadlock set) Sherlock(Program p) {
  (Cycle set) candidates = GoodLock(p)
  Schedule s₀ = initialRun(p)
  (Deadlock set) locks = ∅

  for each Cycle c ∈ candidates do {
    boolean found = false
    boolean stalled = false
    int i = 0
    Schedule s = s₀
    while (¬ found) ∧ (¬ stalled) ∧ (i ≤ 1000) {
      case execute(p, s, c) of
        (Input × Schedule × boolean) (a, trace, true) : {
          locks = locks ∪ {(c, a, trace)}
          found = true
        }
        (Input × Schedule × boolean) (a, trace, false) : {
          case permute(trace, c) of
            Schedule s' : { s = s' }
            none : { stalled = true }
            none : { stalled = true }
          i = i + 1
        }
      }
    }
    return locks
  }
```
Example for Sherlock (1)

\[
y = \text{to be determined}
\]

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 :</td>
<td>$x := 6$</td>
<td>$x := 2$</td>
</tr>
<tr>
<td>2 :</td>
<td>synchronized($A$){</td>
<td>synchronized($B$){</td>
</tr>
<tr>
<td>3 :</td>
<td>if($y &gt; 4$){</td>
<td>if($y^2 + 5 &lt; x^2$){</td>
</tr>
<tr>
<td>4 :</td>
<td>synchronized($B$)}</td>
<td>synchronized($A$)}</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

We use abbreviations for the events:

\[
e_i = (T_1, i), \quad 1 \leq i \leq 4, \quad e_{i+4} = (T_2, i), \quad 1 \leq i \leq 4.
\]

Initial run:

\[
y = 0 \Rightarrow s = \langle e_1, e_2, e_3, e_5, e_6, e_7 \rangle.
\]
Example for Sherlock (2)

First iteration of while-loop: Record of \((y > 4)\).

\[ y = 5 \Rightarrow \text{trace} = \langle e_1, e_2, e_3, e_5, e_6, e_7, e_4 \rangle \]

Permute on trace:

\[ s = \langle e_5, e_6, e_7, e_1, e_2, e_3, e_4 \rangle \]

Second iteration of while-loop:

\[ \text{trace} = \langle e_5, e_6, e_7, e_1, e_2, e_3, e_4 \rangle \]

Permute on trace:

\[ s = \langle e_5, e_6, e_1, e_2, e_7, e_3, e_4 \rangle \]

Third iteration of while-loop:

\[ \text{trace} = \langle e_5, e_6, e_1, e_2, e_7, e_3, e_4, e_8 \rangle \]
Experimental results (1)

Figure: Numbers of deadlocks detected by technique

David Wellig ETH Zürich
Sherlock: Scalable Deadlock Detection for Concurrent Programs
Observations:

- Sherlock detects many deadlocks for more than $10^6$ execution steps.
- Sherlock only missed 15 detections of DCJJ.
Conclusions, Limitations

Conclusion:

- Sherlock finds more deadlocks than any other dynamic detection technique.
- Thanks to *permute* Sherlock scales also to long schedules.

Limitation:

- Sherlock up to now only supports synchronized methods and statements. Wait, notify and notify all are not supported.
- To find deadlock candidates Sherlock relies on GoodLock. If GoodLock misses a deadlock, so does Sherlock.
Thank you for your attention.