Concurrency Synthesis for Device-Drivers

Arjun Radhakrishna

Institute of Science and Technology, Austria

March 10, 2014

Joint work with Pavol Černý, Thomas A. Henzinger, Leonid Rhyzyk, and Thorsten Tarrach

Supported by ERC Advanced Grant QUAREM, the FWF NFN Grant S11402-N23 and S11407-N23 (RiSE), the EU NOE Grant ArtistDesign, and a gift from Intel
Concurrency bug-fixes in practice: Linux case study

- Picked **100 most recent bugs** from Linux device drivers

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>REORDER</td>
<td>Reorder program statements</td>
<td>28</td>
</tr>
<tr>
<td>LOCK</td>
<td>Protect racing code with a lock</td>
<td>17</td>
</tr>
<tr>
<td>OPTIMISTIC</td>
<td>Check if shared variable has been modified</td>
<td>10</td>
</tr>
<tr>
<td>BARRIER</td>
<td>Use system provided function to wait</td>
<td>7</td>
</tr>
<tr>
<td>ATOMIC</td>
<td>Replace sequence with equivalent atomic primitive</td>
<td>6</td>
</tr>
<tr>
<td>UPGRADE</td>
<td>Replace synchronization with stronger ones</td>
<td>5</td>
</tr>
<tr>
<td>UNSHARE</td>
<td>Create a private copy of a shared variable</td>
<td>3</td>
</tr>
<tr>
<td>CLONE</td>
<td>Replicate an idempotent statement</td>
<td>1</td>
</tr>
<tr>
<td>ADHOC</td>
<td>No previous category</td>
<td>23</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>

- Locking and atomic operations not so often used
Motivation

- Synthesis for concurrency: focus on lock- and atomic-section placement

- Aim: find programmer-like solutions/bug-fixes

- Programmer approach: write sequentially correct code and then fix concurrency bugs
IntrEnabled = 1
assume(IntrEnabled)
handled = ready
ready = 1

IntrEnabled = 0, ready = 0
IntrEnabled = 1
assume(IntrEnabled)
ready = 1
handled = ready
handled = 1

Bug: Interrupts are enabled before you are ready to handle them
Fix: Be ready to handle interrupts before enabling them

Switch!
Simplified from a real bug fix
Sequential-Semantics preserving reordering
Reordering statements

IntrEnabled = 1

IntrEnabled = 1

assume(IntrEnabled)

handled = ready

handled = 1

ready = 1

IntrEnabled = 0, ready = 0

Bug: Interrupts are enabled before you are ready to handle them

Fix: Be ready to handle interrupts before enabling them

Switch!

Simplified from a real bug fix

Sequential-Semantics preserving reordering
Reordering statements

\[
\begin{align*}
\text{IntrEnabled} &= 1 \\
\text{assume(IntrEnabled} &= 1) \\
\text{IntrEnabled} &= 1 \\
\text{assume(IntrEnabled)} \\
\text{ready} &= 1 \\
\text{handled} &= \text{ready} \\
\text{handled} &= 1
\end{align*}
\]

\[\text{IntrEnabled} = 0, \text{ready} = 0\]

\text{Bug: Interrupts are enabled before you are ready to handle them}

\text{Fix: Be ready to handle interrupts before enabling them}

\text{Switch!}

\text{Simplified from a real bug fix}

\text{Sequential-Semantics preserving reordering}
Reordering statements

\[
\begin{align*}
\text{IntrEnabled} &= 1 \\
\text{assume(IntrEnabled = 1)} \\
\text{handled} &= \text{ready}
\end{align*}
\]

\[
\begin{align*}
\text{IntrEnabled} &= 0, \text{ready} = 0 \\
\text{IntrEnabled} &= 1 \\
\text{assume(IntrEnabled)}
\end{align*}
\]

\[
\begin{align*}
\text{ready} &= 1 \\
\text{handled} &= \text{ready}
\end{align*}
\]

\[
\text{handled} = 1
\]

Bug: Interrupts are enabled before you are ready to handle them

Fix: Be ready to handle interrupts before enabling them

Switch!

Simplified from a real bug fix

Sequential-Semantics preserving reordering
Reordering statements

\[ \text{IntrEnabled} = 1 \]
\[ \text{assume(IntrEnabled} = 1) \]
\[ \text{handled} = \text{ready} \]
\[ \text{ready} = 1 \]

\[ \text{IntrEnabled} = 0, \text{ready} = 0 \]
\[ \text{assume(IntrEnabled)} \]
\[ \text{handled} = \text{ready} \]
\[ \text{handled} = 1 \]

Bug: Interrupts are enabled before you are ready to handle them

Fix: Be ready to handle interrupts before enabling them

Switch!

Simplified from a real bug fix

Sequential-Semantics preserving reordering
Reordering statements

Bug: Interrupts are enabled before you are ready to handle them

IntrEnabled = 1
assume(IntrEnabled = 1)
handled = ready
ready = 1

IntrEnabled = 0, ready = 0
assume(IntrEnabled)
handled = ready

IntrEnabled = 1
ready = 1
handled = 1
Reordering statements

IntrEnabled = 1
assume(IntrEnabled = 1)
handled = ready
ready = 1

IntrEnabled = 0, ready = 0
assume(IntrEnabled)

handled = ready

Switch!

 handled = 1
read = 1
handled = 1

Bug: Interrupts are enabled before you are ready to handle them
Fix: Be ready to handle interrupts before enabling them
Reordering statements

Bug: Interrupts are enabled before you are ready to handle them
Fix: Be ready to handle interrupts before enabling them

- Simplified from a real bug fix
- Sequential-Semantics preserving reordering
Program-Repair Problem Statement

Given: library $\mathcal{L}$ of **sequentially correct** methods: $M_1, \ldots, M_n$.

Output: library $\mathcal{L}' : M'_1, \ldots, M'_n$.
  - $\mathcal{L}'$ is correct
  - Each $M'_i$ is sequentially equivalent to $M_i$

Our approach: each $M'_i$ is **rearrangement** of $M_i$ with additional atomic sections
Programmer-like approach

- Find root cause of bug – Trace generalization
- Fix the root cause – Eliminate generalized trace
- Repeat till done – Repeat till done (termination)
Trace Generalization

\[ I_1 = x \]
\[ I_2 = x \]
\[ I_1 = I_1 + 1 \]
\[ I_2 = I_2 + 1 \]
\[ x = I_1 \]
\[ x = I_2 \]

Read into \( I_1 \) before write from \( I_2 \) and
Read into \( I_2 \) before write from \( I_1 \)
Trace Generalization

\[
\begin{align*}
    l_1 &= x \\
    l_1 &= l_1 + 1 \\
    l_2 &= x \\
    l_2 &= l_2 + 1 \\
    x &= l_1 \\
    x &= l_2 \\
    l_1 &= x \\
    l_1 &= l_1 + 1 \\
    l_2 &= x \\
    l_2 &= l_2 + 1 \\
    x &= l_1 \\
    x &= l_2
\end{align*}
\]
Trace Generalization

Read into \( l_1 \) before write from \( l_2 \)

and

Read into \( l_2 \) before write from \( l_1 \)
Trace Generalization

Read into l1 before write from l2
and
Read into l2 before write from l1
IntrEnabled = 1

assume(IntrEnabled)

handled = ready

ready = 1
Trace Generalization

IntrEnabled = 1

assume(IntrEnabled)

handled = ready

ready = 1

Interrupts Enabled
Interrupt raised
Try to handle
Before ready
Elimination of Generalization Traces

IntrEnabled = 1

\[ \text{assume(IntrEnabled)} \]

handled = ready

ready = 1
Elimination of Generalization Traces

\[ \text{IntrEnabled} = 1 \]

\[ \text{ready} = 1 \]

\[ \text{assume(IntrEnabled)} \]

\[ \text{handled} = \text{ready} \]
Elimination of Generalization Traces

\[ \text{IntrEnabled} = 1 \]

\[ \text{assume(IntrEnabled)} \]

\[ \text{handled} = \text{ready} \]

Here, required ordering constraint is \( \text{ready} = 1 \) precedes \( \text{IntrEnabled} = 1 \).

Also, \( (a \text{ precedes } b) \land (b \text{ precedes } a) \equiv \text{atomic}\{a; b\} \).
Elimination of Generalization Traces

- The order that creates a cycle, eliminates the generalized trace
- Here, required ordering constraint is \( \text{ready} = 1 \) precedes \( \text{IntrEnabled} = 1 \)
- Also, \( (a \text{ precedes } b) \land (b \text{ precedes } a) \equiv \text{atomic} \{a; b\} \).
Algorithm

\[ \phi \leftarrow \text{semantics preserving orders of } P \]

Pick \( P' \) satisfying \( \phi \)

\[ \phi = \phi \land \text{Eliminate}(gcex) \]

Verify \( P' \)

\[ gcex = \text{Generalize}(cex) \]

Return \( P \)

\[ \text{ok} \]

\[ \text{cex} \]
Algorithm

- \( \phi \leftarrow \text{semantics preserving orders of } P \)
- \( \phi = \phi \land \text{Eliminate}(gce) \)
- \( gce = \text{Generalize}(ce) \)
- Pick \( P' \) satisfying \( \phi \)
- Verify \( P' \)
- Return \( P \)
- \text{ok}
- \text{cex}

- Associate each constraint with cost and pick minimal cycle
Experiments

- Took synchronization skeleton of R8169 linux driver
  - Only logic and synchronization, no data

- 7 threads, 5 known bugs – 400 source lines of code

- Every bug fixed in 1 – 2 iterations (with trace generalization)
  - Without trace generalization, takes more iterations
  - 99% time in verifier: fewer iterations \(\implies\) fewer verifier calls
Conclusion and Future Work

- Synthesis through semantics-preserving transformations
- Find programmer-like fixes and avoid expensive locks/synchronization

- More synchronization constructs and transformations
- Learn which fixes are admissible from correct traces of the program
Questions?