Effective Verification of Low-Level Software with Nested Interrupts

Li Hao Liang

Joint work with Daniel Kroening, Tom Melham, Peter Schrammel
University of Oxford
&
Michael Tautschnig (Queen Mary, University of London)

Design, Automation and Test in Europe 2015
Motivation

- Interrupts are widely used in all styles of computing platforms - servers, embedded systems, mobile devices
- Efficient I/O
- Context switching/thread scheduling
- Power-efficient operations
Motivation

- Interrupt-related bugs are difficult to find by testing
- Detected bugs are hard to reproduce and diagnose
- Situation gets worse if multiple interrupts are present
Nested Execution of Interrupts

Taken from “Understanding the Linux Kernel”, 3rd Edition
Q: Are they just concurrent programs? What is the difference?  
A: Subtle yet important.

Taken from “Understanding the Linux Kernel”, 3rd Edition
Motivating Example - Interrupt-Driven

```c
irqreturn_t handler1(...) {
    ...
    y = 2
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}

irqreturn_t handler2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    return IRQ_HANDLED;
}
```
Motivating Example - Interrupt-Driven

```c
irqreturn_t handler1(...) {
    ...
    y = 2
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}
```

```c
irqreturn_t handler2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    ...
    return IRQ_HANDLED;
}
```
Motivating Example - Interrupt-Driven

```c
irqreturn_t handler1(...)
{
    ...
    y = 2;
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}
```

```c
irqreturn_t handler2(...)
{
    ...
    enable_irq(1);
    y = 1;
    ...
    return IRQ_HANDLED;
}
```
Motivating Example - Interrupt-Driven

```c
irqreturn_t handler1(...) {
    ...
    y = 2
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}
```

```c
irqreturn_t handler2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    ...
    return IRQ_HANDLED;
}
```
Motivating Example - Interrupt-Driven

```c
irqreturn_t handler1(...) {
    ...
    y = 2
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}
```

```c
irqreturn_t handler2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    return IRQ_HANDLED;
}
```
Motivating Example - Interrupt-Driven

```c
interrupt_handler1(...)
{
    ...
    y = 2
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}

interrupt_handler2(...)
{
    ...
    enable_irq(1);
    y = 1;
    ...
    ...
    return IRQ_HANDLED;
}
```
Motivating Example - Interrupt-Driven

```c
irqreturn_t handler1(...) {
    ...
    y = 2;
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}
```

```c
irqreturn_t handler2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    ...
    return IRQ_HANDLED;
}
```

Holds :)
Motivating Example - Concurrent

```c
irqreturn_t thread1(...) {
    ...
    y = 2
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}
```

```c
irqreturn_t thread2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    ...
    return IRQ_HANDLED;
}
```
Motivating Example - Concurrent

irqreturn_t thread1(...) {
    ...
    y = 2
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}

irqreturn_t thread2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    ...
    return IRQ_HANDLED;
}
Motivating Example - Concurrent

```c
irqreturn_t thread1(...) {
    ...
    y = 2;
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}
```

```c
irqreturn_t thread2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    return IRQ_HANDLED;
}
```
Motivating Example - Concurrent

```c
int thread1(...) {
    ...  
    y = 2
    x = 3*y;
    assert(x == 6);
    ...  
    return IRQ_HANDLED;
}
```

```c
int thread2(...) {
    ...  
    enable_irq(1);
    y = 1;
    ...
    ...  
    return IRQ_HANDLED;
}
```
Motivating Example - Concurrent

```c
irqreturn_t thread1(...) {
    ...
    y = 2
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}
```

```c
irqreturn_t thread2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    return IRQ_HANDLED;
}
```
Motivating Example - Concurrent

```c
thread1(...) {
  ...
  y = 2
  x = 3*y;
  assert(x == 6);
  ...
  return IRQ_HANDLED;
}

thread2(...) {
  ...
  enable_irq(1);
  y = 1;
  ...
  return IRQ_HANDLED;
}
```
Motivating Example - Concurrent

```c
irqreturn_t thread1(...) {
    ...
    y = 2;
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}

irqreturn_t thread2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    return IRQ_HANDLED;
}
```
Motivating Example - Concurrent

```c
irqreturn_t thread1(...) {
    ...
    y = 2
    x = 3*y;
    assert(x == 6);
    ...
    return IRQ_HANDLED;
}
```

```c
irqreturn_t thread2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    return IRQ_HANDLED;
}
```

Violated!
Contributions

- Symbolic encoding that faithfully models the interleaving semantics of programs with nested interrupts
- Implementation in i-CBMC
- Significantly outperforms conventional approaches in terms of precision
A High Level Overview

- Source Code
  - Control-Flow Graph
    - Static Single Assignment (SSA) Form
    - Symbolic Event Structure
    - Symbolic Partial Orders
      - Conjunction
        - Decision Procedure
Static Single Assignment Form

• Turn program assignments into equalities
• A fresh index is used to annotate each occurrence of a share variable
• Each equation is associated with its branch condition
• Unwind loops to fixed depths
volatile unsigned \( x = 0, y = 0; \)
volatile unsigned \( i, j; \)

void* A(void* arg) {
    \( x = 1; \)
    \( i = y + 1; \)
}

void* B(void* arg) {
    \( y = 1; \)
    \( j = x + 1; \)
}
Symbolic Event Structure

• Four-tuple to represent symbolic read and write events to shared memory locations

• Event ID, direction, memory location, symbolic value

• Each occurrence of a shared variable in the right/left-hand side of an assignment is a symbolic read/write
Symbolic Event Structure

\[ x_0 = 0, \quad y_0 = 0 \]

\[ \text{A:} \]
\[ x_1 = 1 \]
\[ i_0 = y_1 + 1 \]

\[ \text{B:} \]
\[ y_2 = 1 \]
\[ j_0 = x_2 + 1 \]

\[ (m1) Wx,0 \]
\[ (m2) Wy,0 \]

\[ (A1) Wx,1 \]
\[ (A2) Ry,y_1 \]
\[ (A3) Wi,i_0 \]

\[ (B1) Wy,1 \]
\[ (B2) Rx,x_2 \]
\[ (B3) Wj,j_0 \]
Symbolic Partial Orders

- Each symbolic event \( e \) is associated with a clock variable \( \text{clock}(e) \)
- Values of clock variables are integers
- Order between events is the strictly less than \((<)\) relation over integers
Symbolic Partial Orders

- (program order): per-thread program order
- (write serialisation): per-address total order on writes
- (read-from): links the value of a read to a write
- (from-read): orders reads and writes
- (interrupts): orders events of nested interrupts
Symbolic Partial Orders - Program Order

(A1) Wx, 1
(A2) Ry, y_1
(A3) Wi, i_0

(B1) Wy, 1
(B2) Rx, x_2
(B3) Wj, j_0

(m1) Wx, 0
(m2) Wy, 0
Symbolic Partial Orders - Program Order

(A1) Wx,1  
(A2) Ry,y_1  
(A3) Wi,i_0

(B1) Wy,1  
(B2) Rx,x_2  
(B3) Wj,j_0

(m1) Wx,0  
(m2) Wy,0

program order
Symbolic Partial Orders - Program Order

\( \text{clock}(A1) < \text{clock}(A2) \)
\( \land \)
\( \text{clock}(A2) < \text{clock}(A3) \)

\( (A1)Wx,1 \)
\( (A2)Ry,y_1 \)
\( (A3)Wi,i_0 \)

\( \text{clock}(B1) < \text{clock}(B2) \)
\( \land \)
\( \text{clock}(B2) < \text{clock}(B3) \)

\( (B1)Wy,1 \)
\( (B2)Rx,x_2 \)
\( (B3)Wj,j_0 \)

\( \text{program order} \)
Symbolic Partial Orders - Write Serialisation

(A1) $W_{\textbf{x},1}$
(A2) $R_{\textbf{y},y_1}$
(A3) $W_{i,i_0}$

(m1) $W_{\textbf{x},0}$
(m2) $W_{\textbf{y},0}$

(B1) $W_{\textbf{y},1}$
(B2) $R_{\textbf{x},x_2}$
(B3) $W_{j,j_0}$
Symbolic Partial Orders - Write Serialisation

clock(m1) < clock(A1) ∨ clock(A1) < clock(m1)

(m1)Wx, 0
(m2)Wy, 0
(A1)Wx, 1
(A2)Wy, y_1
(A3)Wi, i_0

(B1)Wy, 1
(B2)Rx, x_2
(B3)Wj, j_0

write serialisation
Symbolic Partial Orders - Write Serialisation

\[ \text{clock}(m1) < \text{clock}(A1) \lor \text{clock}(A1) < \text{clock}(m1) \]

\[ \text{clock}(m2) < \text{clock}(B1) \lor \text{clock}(B1) < \text{clock}(m2) \]

\[ (A1) Wx, 1 \]
\[ (A2) Wy, y_1 \]
\[ (A3) Wi, i_0 \]

\[ (B1) Wy, 1 \]
\[ (B2) Rx, x_2 \]
\[ (B3) Wj, j_0 \]

write serialisation
Symbolic Partial Orders - Read-From

(A1) \( W_{x,1} \)
(A2) \( R_{y,y_1} \)
(A3) \( W_{i,i_0} \)

(m1) \( W_{x,0} \)
(m2) \( W_{y,0} \)

(B1) \( W_{y,1} \)
(B2) \( R_{x,x_2} \)
(B3) \( W_{j,j_0} \)
Symbolic Partial Orders - Read-From

(A1) $W_{\mathbf{x}, 1}$
(A2) $R_{\mathbf{y}, y_1}$
(A3) $W_{i, i_0}$

(B1) $W_{\mathbf{y}, 1}$
(B2) $R_{\mathbf{x}, x_2}$
(B3) $W_{j, j_0}$

(m1) $W_{\mathbf{x}, 0}$
(m2) $W_{\mathbf{y}, 0}$

read-from
Symbolic Partial Orders - Read-From

(A1) $W x, 1$
(A2) $R y, y_1$
(A3) $W i, i_0$

$W x, 0$
$W y, 0$

$W y, 1$
$R x, x_2$
$W j, j_0$

$\text{clock}(m1) < \text{clock}(B2)$
$x_2 = 0$

$\text{clock}(A1) < \text{clock}(B2)$
$x_2 = 1$

$\text{clock}(m1) < \text{clock}(B2)$
$x_2 = 0$

read-from
Symbolic Partial Orders - Read-From

(A1) \( Wx, 1 \)
(A2) \( Wy, y_1 \)
(A3) \( Wi, i_0 \)

(B1) \( Wy, 1 \)
(B2) \( Rx, x_2 \)
(B3) \( Wj, j_0 \)

\[ \text{clock}(m1) < \text{clock}(B2) \Rightarrow x_2 = 0 \]
\[ \text{clock}(A1) < \text{clock}(B2) \Rightarrow x_2 = 1 \]

read-from
Symbolic Partial Orders - Read-From

(A1) $Wx, 1$
(A2) $Wy, y_1$
(A3) $Wi, i_0$

clock(m2) < clock(A2) \implies y_1 = 0

(B1) $Wy, 1$
(B2) $Rx, x_2$
(B3) $Wj, j_0$

clock(B1) < clock(A2) \implies y_1 = 1

clock(m1) < clock(B2) \implies x_2 = 0

clock(A1) < clock(B2) \implies x_2 = 1

_read-from_
Symbolic Partial Orders - From-Read

(A1) $Wx, 1$
(A2) $Wy, y_1$
(A3) $Wi, i_0$

(m1) $Wx, 0$
(m2) $Wy, 0$

(B1) $Wy, 1$
(B2) $Rx, x_2$
(B3) $Wj, j_0$
Symbolic Partial Orders - From-Read

\[(A1) Wx, 1 \]
\[(A2) Ry, y_1 \]
\[(A3) Wi, i_0 \]

\[(B1) Wy, 1 \]
\[(B2) Rx, x_2 \]
\[(B3) Wj, j_0 \]

\[
\text{clock(m1)} < \text{clock(B2)} \implies x_2 = 0
\]

read-from
Symbolic Partial Orders - From-Read

\[(A1) Wx, 1 \]
\[(A2) R_{y, y_1} \]
\[(A3) Wi, i_0 \]
\[\text{clock}(m1) < \text{clock}(A1)\]

\[\text{clock}(m1) < \text{clock}(B2) \implies x_2 = 0\]

\[(m1) Wx, 0 \]
\[(m2) Wy, 0 \]

\[\text{clock}(m1) < \text{clock}(B2)\]

\[(B1) Wy, 1 \]
\[(B2) Rx, x_2 \]
\[(B3) Wj, j_0 \]

- **Read-From**: blue arrows
- **Write Serialisation**: blue arrows
Symbolic Partial Orders - From-Read

(A1) \( \text{W}x,1 \)
(A2) \( \text{R}y, y_1 \)
(A3) \( \text{W}i, i_0 \)

\( \text{clock}(m1) < \text{clock}(A1) \)

\( \text{clock}(m1) < \text{clock}(B2) \) \( \Rightarrow \) \( x_2 = 0 \)

(B1) \( \text{W}y,1 \)
(B2) \( \text{R}x, x_2 \)
(B3) \( \text{W}j, j_0 \)

\( \text{read-from} \)
\( \text{write serialisation} \)
\( \text{from-read} \)
Symbolic Partial Orders - From-Read

\[(A1) Wx, 1 \]
\[(A2) Ry, y_1 \]
\[(A3) Wi, i_0 \]

\[(B1) Wy, 1 \]
\[(B2) Rx, x_2 \]
\[(B3) Wj, j_0 \]

\[\text{clock}(m1) < \text{clock}(B2) \wedge \text{clock}(m1) < \text{clock}(A1) \implies \text{clock}(B2) < \text{clock}(A1)\]

\[\text{clock}(m1) < \text{clock}(A1)\]

\[\text{clock}(m1) < \text{clock}(B2) \implies x_2 = 0\]

read-from
write serialisation
from-read
Symbolic Partial Orders - From-Read

- \((A1)\) \(Wx,1\)
- \((A2)\) \(Wy, y_1\)
- \((A3)\) \(Wi, i_0\)
- \((B1)\) \(Wy,1\)
- \((B2)\) \(Rx, x_2\)
- \((B3)\) \(Wj, j_0\)

\[
\text{clock}(m1) < \text{clock}(B2) \land \\
\text{clock}(m1) < \text{clock}(A1) \quad \Rightarrow \\
\text{clock}(B2) < \text{clock}(A1)
\]

\[
\text{clock}(m1) < \text{clock}(B2) \quad \Rightarrow \\
x_2 = 0
\]
Symbolic Partial Orders - From-Read

(A1) $W_x, 1$
(A2) $R_{y, y_1}$
(A3) $W_{i, i_0}$

(B1) $W_{y, 1}$
(B2) $R_{x, x_2}$
(B3) $W_{j, j_0}$

$\text{clock}(m1) < \text{clock}(B2) \land \text{clock}(m1) < \text{clock}(A1) \implies \text{clock}(B2) < \text{clock}(A1)$

$\text{clock}(m1) < \text{clock}(A1)$

$\implies x_2 = 0$

read-from
write serialisation
from-read
Symbolic Partial Orders - From-Read

(A1) \( Wx, 1 \)
(A2) \( Ry, y_1 \)
(A3) \( Wi, i_0 \)

(B1) \( Wy, 1 \)
(B2) \( Rx, x_2 \)
(B3) \( Wj, j_0 \)

\[ \text{clock}(m1) < \text{clock}(A1) \]
\[ \text{clock}(m2) < \text{clock}(A2) \land \text{clock}(m2) < \text{clock}(B1) \]
\[ \implies \text{clock}(A2) < \text{clock}(B1) \]
\[ \text{clock}(m1) < \text{clock}(B2) \land \text{clock}(m1) < \text{clock}(A1) \]
\[ \implies \text{clock}(B2) < \text{clock}(A1) \]

\[ \text{clock}(m1) < \text{clock}(B2) \]
\[ \implies x_2 = 0 \]

(read-from)
(write serialisation)
(from-read)
Symbolic Partial Orders - From-Read

\[(A1) Wx,1 \quad (A2) Ry, y_1 \quad (A3) Wi, i_0 \]

\[\text{clock}(m1) < \text{clock}(A1) \quad \text{clock}(m2) < \text{clock}(A2) \land \text{clock}(m2) < \text{clock}(B1) \implies \text{clock}(A2) < \text{clock}(B1) \]

\[\text{clock}(m1) < \text{clock}(B2) \land \text{clock}(m1) < \text{clock}(A1) \implies \text{clock}(B2) < \text{clock}(A1) \]

\[(m1) Wx,0 \quad (m2) Wy,0 \quad (B1) Wy,1 \quad (B2) Rx, x_2 \quad (B3) Wj, j_0 \]

\[\text{clock}(m1) < \text{clock}(B2) \implies x_2 = 0 \]

- read-from
- write serialisation
- from-read
- program order
Symbolic Partial Orders - From-Read

A cycle means this is not a valid execution.

(A1) \( W_{x,1} \)
(A2) \( R_{y,y_1} \)
(A3) \( W_{i,i_0} \)

(B1) \( W_{y,1} \)
(B2) \( R_{x,x_2} \)
(B3) \( W_{j,j_0} \)

\[
\begin{align*}
\text{clock}(m1) < \text{clock}(A1) \\
\text{clock}(m2) < \text{clock}(A2) \land \\
\text{clock}(m2) < \text{clock}(B1) \\
\implies \\
\text{clock}(A2) < \text{clock}(B1) \\
\text{clock}(m1) < \text{clock}(B2) \land \\
\text{clock}(m1) < \text{clock}(A1) \\
\implies \\
\text{clock}(B2) < \text{clock}(A1) \\
x_2 = 0
\end{align*}
\]
Symbolic Partial Orders - Interrupts

- Define preempts(e₁, e₂) to be true if the ISR call that contains memory event e₁ preempts the ISR call containing e₂

- For all memory events e₁, e₂ such that preempts(e₁, e₂), if (e₁, e₂) ∈ write-serialisation ∪ read-from ∪ from-read, then for all events e such that (e₁, e) ∈ program-order, we have (e, e₂) ∈ interrupts
Symbolic Partial Orders - Interrupts

Handler A

\[ A_1 \]
\[ A_2 \]
\[ A_3 \]
\[ \ldots \]
\[ A_n \]

Handler B

\[ B_1 \]
\[ B_2 \]
\[ B_3 \]
\[ \ldots \]
\[ B_m \]

Program Order

\[ \text{program order} \]
Symbolic Partial Orders - Interrupts

Handler A

\[ A_1 \]
\[ A_2 \]
\[ A_3 \]
\[ \ldots \]
\[ A_n \]

Handler B

\[ B_1 \]
\[ B_2 \]
\[ B_3 \]
\[ \ldots \]
\[ B_m \]

Program Order

\[ \rightarrow \text{program order} \]
Symbolic Partial Orders - Interrupts

Handler A

\(A_1 \rightarrow \) Preemption

\(A_2 \rightarrow \)

\(A_3 \rightarrow \)

\(\ldots \rightarrow \)

\(A_n \rightarrow \)

Handler B

\(B_1 \rightarrow \)

\(B_2 \rightarrow \)

\(B_3 \rightarrow \)

\(\ldots \rightarrow \)

\(B_m \rightarrow \)

Program Order

\[\text{program order}\]
Symbolic Partial Orders - Interrupts

Program Order

Handler A

\[ A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow \cdots \rightarrow A_n \]

Preemption

Handler B

\[ B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow \cdots \rightarrow B_m \]

- program order
- write serialisation or read-from or from-read
Symbolic Partial Orders - Interrupts

Program Order:
- \( A_1 \)
- \( A_2 \)
- \( A_3 \)
- \( \ldots \)
- \( A_n \)

Handler A:
- \( A_1 \) preempts \( A_2 \)
- \( A_2 \) preempts \( A_3 \)
- \( \ldots \)
- \( A_n \) preempts

Handler B:
- \( B_1 \)
- \( B_2 \)
- \( B_3 \)
- \( \ldots \)
- \( B_m \)

- program order
- write serialisation or read-from or from-read
- interrupts
Symbolic Partial Orders - Interrupts

Handler A

Program Order

\( A_1 \)
\( A_2 \)
\( A_3 \)
\( \ldots \)
\( A_n \)

Handler B

\( B_1 \)
\( B_2 \)
\( B_3 \)
\( \ldots \)
\( B_m \)

clock(A_\text{n}) < clock(B_3)

Preemption

Write serialisation or read-from or from-read

Interrupts
Experimental Studies

- Benchmarks derived from embedded software and Linux device drivers (0.1K - 7K lines of code)
- Compare with two conventional techniques based on program instrumentation
  - Sequentialisation
  - Instrumentation with threads
Alternative Technique
Sequentialisation

- Use a global array to indicate which interrupts are enabled
- Encode a scheduling function that non-deterministically invokes interrupt handlers for enabled interrupts
- Apply partial-order reduction to reduce the number of calls to the scheduling function

Kidd, Jagannathan, and Vitek SPIN10
Alternative Technique
Instrumentation with Threads

- Use a global array `thread_running` to indicate which interrupts are executing/preempted

- Thread $i$ has higher priority than thread $j$ if $i < j$

- Instrument atomic statements

  ```
  assume(thread_running[0] == 0 && ... &&
  thread_running[n-1] == 0)
  ```

  between statements of thread $n$

Regehr and Cooprider ENTCS07
## Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>LOC</th>
<th>Interrupts</th>
<th>BLAST 2.7.2</th>
<th>CPAChecker 1.3.4</th>
<th>UFO SV-COMP14</th>
<th>CBMC r4781</th>
<th>i-CBMC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logger</strong></td>
<td>112</td>
<td>2</td>
<td>✓ 17.0 s</td>
<td>✓ 1.8 s</td>
<td>✓ 1.4 s</td>
<td>TO</td>
<td>✓ 0.2 s</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>112</td>
<td>2</td>
<td>! 26.7 s</td>
<td>? 2.0 s</td>
<td>! 36.2 s</td>
<td>TO</td>
<td>! 0.2 s</td>
</tr>
<tr>
<td><strong>Logger (extended)</strong></td>
<td>172</td>
<td>3</td>
<td>✓ 21.1 s</td>
<td>TO</td>
<td>TO</td>
<td>✓ 25.2 s</td>
<td></td>
</tr>
<tr>
<td>+ incorrect</td>
<td>172</td>
<td>3</td>
<td>TO</td>
<td>? 2.4 s</td>
<td>TO</td>
<td>TO</td>
<td>! 22.5 s</td>
</tr>
<tr>
<td><strong>Blink</strong></td>
<td>2652</td>
<td>2</td>
<td>✓ unknown</td>
<td>✓ 1425.1 s</td>
<td>TO</td>
<td>✓ 3.6 s</td>
<td></td>
</tr>
<tr>
<td>+ incorrect</td>
<td>2652</td>
<td>2</td>
<td>✓ unknown</td>
<td>✓ 1420.5 s</td>
<td>TO</td>
<td>✓ 4.2 s</td>
<td></td>
</tr>
<tr>
<td><strong>RcCore</strong></td>
<td>7035</td>
<td>3</td>
<td>✓ 41.1 s</td>
<td>TO</td>
<td>TO</td>
<td>✓ 75.7 s</td>
<td></td>
</tr>
<tr>
<td>+ incorrect</td>
<td>7035</td>
<td>3</td>
<td>✓ 37.1 s</td>
<td>TO</td>
<td>TO</td>
<td>✓ 75.5 s</td>
<td></td>
</tr>
<tr>
<td><strong>Brake (1 Wheel)</strong></td>
<td>3938</td>
<td>2</td>
<td>✓ 0.8 s</td>
<td>✓ 3.1 s</td>
<td>✓ 0.8 s</td>
<td>X</td>
<td>✓ 154.2 s</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>3938</td>
<td>2</td>
<td>TO</td>
<td>unknown</td>
<td>? 1.1 s</td>
<td>X</td>
<td>! 3.7 s</td>
</tr>
<tr>
<td><strong>Brake (2 Wheels)</strong></td>
<td>3938</td>
<td>3</td>
<td>TO</td>
<td>* 5.6 s</td>
<td>✓ 7.8 s</td>
<td>X</td>
<td>TO</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>3938</td>
<td>3</td>
<td>TO</td>
<td>unknown</td>
<td>? 1.1 s</td>
<td>X</td>
<td>! 6.7 s</td>
</tr>
<tr>
<td><strong>Brake (3 Wheels)</strong></td>
<td>3938</td>
<td>4</td>
<td>TO</td>
<td>* 10.6 s</td>
<td>✓ 854.1 s</td>
<td>X</td>
<td>TO</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>3938</td>
<td>4</td>
<td>TO</td>
<td>unknown</td>
<td>? 3.8 s</td>
<td>X</td>
<td>! 9.8 s</td>
</tr>
</tbody>
</table>

✓ = proved correct, ! = bug exposed, ? = bug missed, * = false alarm
X = tool crashes, PE = parse errors, TO = timeout 1800 s
### Experimental Results (cont.)

<table>
<thead>
<tr>
<th></th>
<th>LOC</th>
<th>Interrupts</th>
<th>IMPARA r878</th>
<th>ESBMC 1.23</th>
<th>CBMC r4781</th>
<th>i-CBMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logger</td>
<td>112</td>
<td>2</td>
<td>✓ 0.2 s</td>
<td>✓ 2.5 s</td>
<td>✓ 0.4 s</td>
<td>✓ 0.2 s</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>112</td>
<td>2</td>
<td>! 0.4 s</td>
<td>! 0.2 s</td>
<td>! 1.1 s</td>
<td>! 0.2 s</td>
</tr>
<tr>
<td>Logger (extended)</td>
<td>172</td>
<td>3</td>
<td>✓ 65.2 s</td>
<td>TO</td>
<td>✓ 250.6 s</td>
<td>✓ 25.2 s</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>172</td>
<td>3</td>
<td>! 117.1 s</td>
<td>TO</td>
<td>! 318.5 s</td>
<td>! 22.5 s</td>
</tr>
<tr>
<td>Blink</td>
<td>2,652</td>
<td>2</td>
<td>✓ 360.8 s</td>
<td>TO</td>
<td>✓ 4.5 s</td>
<td>✓ 3.6 s</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>2,652</td>
<td>2</td>
<td>! 37.3 s</td>
<td>TO</td>
<td>? 14.6 s</td>
<td>! 4.2 s</td>
</tr>
<tr>
<td>RcCore</td>
<td>7,035</td>
<td>3</td>
<td>X</td>
<td>TO</td>
<td>✓ 87.3 s</td>
<td>✓ 75.7 s</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>7,035</td>
<td>3</td>
<td>X</td>
<td>X</td>
<td>! 94.4 s</td>
<td>! 75.5 s</td>
</tr>
<tr>
<td>Brake (1 Wheel)</td>
<td>3,938</td>
<td>2</td>
<td>TO</td>
<td>X</td>
<td>✓ 42.2 s</td>
<td>✓ 154.2 s</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>3,938</td>
<td>2</td>
<td>TO</td>
<td>X</td>
<td>! 7.2 s</td>
<td>! 3.7 s</td>
</tr>
<tr>
<td>Brake (2 Wheels)</td>
<td>3,938</td>
<td>3</td>
<td>TO</td>
<td>X</td>
<td>✓ 797.3 s</td>
<td>TO</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>3,938</td>
<td>3</td>
<td>TO</td>
<td>X</td>
<td>! 29.3 s</td>
<td>! 6.7 s</td>
</tr>
<tr>
<td>Brake (3 Wheels)</td>
<td>3,938</td>
<td>4</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
</tbody>
</table>

✓ = proved correct, ! = bug exposed, ? = bug missed, * = false alarm
X = tool crashes, PE = parse errors, TO = timeout 1800 s
Experimental Results (cont.)

![Graph showing experimental results between CBMC with thread instrumentation and i-CBMC (time in seconds). The graph includes data points for safe and unsafe scenarios, with a diagonal line indicating perfect correlation. The vertical line indicates timeout for both CBMC and i-CBMC.]
Website

- http://www.cprover.org/interrupts
- Tools, benchmarks, and experimental data