Effective Verification of Low-Level Software with Nested Interrupts

Lihao Liang

Joint work with Daniel Kroening, Tom Melham, Peter Schrammel

University of Oxford

&

Michael Tautschnig (Queen Mary, University of London)

Université Paris 13, March 19th, 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.
Motivating Example - Interrupt-Driven

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;
}

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

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;
}
```

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

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

\begin{verbatim}
irqreturn_t thread2(...) {
    ...
    enable_irq(1);
    y = 1;
    ...
    ...
    return IRQ_HANDLED;
}
\end{verbatim}
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;
}
```
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;
}

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
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
A High Level Overview

Source Code
A High Level Overview

Source Code

Control-Flow Graph
A High Level Overview

- Source Code
- Control-Flow Graph
- Static Single Assignment (SSA) Form
A High Level Overview

Source Code

Control-Flow Graph

Static Single Assignment (SSA) Form

Symbolic Event Structure
A High Level Overview

Source Code

Control-Flow Graph

Static Single Assignment (SSA) Form → Symbolic Event Structure → Symbolic Partial Orders
A High Level Overview

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

Conjunction
A High Level Overview

Source Code → Control-Flow Graph

Static Single Assignment (SSA) Form → Symbolic Event Structure

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 shared 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

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

A:
\[ x_1 = 1 \]
\[ i_0 = y_1 + 1 \]

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

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

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

program order
Symbolic Partial Orders - Program Order

clock(A1) < clock(A2) \land clock(A2) < clock(A3) → (A1)Wx,1 (A2)Ry,y_1 (A3)Wi,i_0

clock(B1) < clock(B2) \land clock(B2) < clock(B3) → (B1)Wy,1 (B2)Rx,x_2 (B3)Wj,j_0

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

program order
Symbolic Partial Orders - Write Serialisation

(A1) $w_x, 1$
(A2) $r_{y, y_1}$
(A3) $w_{i, i_0}$

(B1) $w_{y, 1}$
(B2) $r_{x, x_2}$
(B3) $w_{j, j_0}$

(m1) $w_x, 0$
(m2) $w_y, 0$
Symbolic Partial Orders - Write Serialisation

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

\[ (m1)Wx_0,0 \]
\[ (m2)Wy_0,0 \]
\[ (A1)Wx_1,1 \]
\[ (A2)Wy_1,1 \]
\[ (A3)Wi,i_0 \]
\[ (B1)Wy_1,1 \]
\[ (B2)Rx_2,x_2 \]
\[ (B3)Wj,j_0 \]

write serialisation
Symbolic Partial Orders - Write Serialisation

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

\[\text{clock}(m2) < \text{clock}(B1) \lor \text{clock}(B1) < \text{clock}(m2)\]
Symbolic Partial Orders - Read-From

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

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

(m1) \( W_{x,0} \)
(m2) \( W_{y,0} \)
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) $W_{y_1, 1}$
(B2) $R_{x, x_2}$
(B3) $W_{j, j_0}$

read-from
Symbolic Partial Orders - Read-From

(A1) \( x, 1 \)
(A2) \( y, y_1 \)
(A3) \( w, i_0 \)

(B1) \( y, 1 \)
(B2) \( x, x_2 \)
(B3) \( w, j_0 \)

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

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

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

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

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

\[\text{clock}(A1) < \text{clock}(B2) \implies x_2 = 1\]

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

read-from
Symbolic Partial Orders - Read-From

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

\( \text{clock}(m2) < \text{clock}(A2) \)
\[ \Rightarrow \]
\[ y_1 = 0 \]

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

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

\( \text{clock}(B1) < \text{clock}(A2) \)
\[ \Rightarrow \]
\[ y_1 = 1 \]

\( \text{clock}(B1) < \text{clock}(A2) \)
\[ \Rightarrow \]
\[ y_1 = 0 \]

\( \text{clock}(B2) < \text{clock}(A2) \)
\[ \Rightarrow \]
\[ x_2 = 1 \]
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) \( Wi, i_0 \)

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

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

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

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

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

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

\( (A1) \) Write \( x, 1 \)
\( (A2) \) Read \( y, y_1 \)
\( (A3) \) Write \( i, i_0 \)

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

\( \Rightarrow x_2 = 0 \)

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

\( \Rightarrow x_2 = 0 \)
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) $W_{j,j_0}$

clock(m1) < clock(B2) ∧
clock(m1) < clock(A1)  \[\Rightarrow\]
clock(B2) < clock(A1)

clock(m1) < clock(A1)

\[\Rightarrow\]

$x_2 = 0$

clock(m1) < clock(B2)

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

\[(A1) W(x, 1) \]
\[(A2) R(y, y_1) \]
\[(A3) W_i, i_0 \]

\[(B1) W(y, 1) \]
\[(B2) R(x, x_2) \]
\[(B3) W_j, j_0 \]

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

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

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

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

\[
\begin{align*}
(A1) & \text{ } W(x, 1) \\
(A2) & \text{ } R(y, y_1) \\
(A3) & \text{ } W(i, i_0)
\end{align*}
\]

\[
\begin{align*}
(m1) & \text{ } W(x, 0) \\
(m2) & \text{ } W(y, 0)
\end{align*}
\]

clock(m1) < clock(A1) \Rightarrow clock(B2) < clock(A1)

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

clock(m1) < clock(B2) \land clock(m1) < clock(A1) \Rightarrow clock(B2) < clock(A1)

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

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

\[\text{clock}(m1) < \text{clock}(A1) \]
\[\text{clock}(m2) < \text{clock}(A2) \land \text{clock}(m2) < \text{clock}(B1) \implies \text{clock}(A2) < \text{clock}(B1) \]
\[\text{clock}(m1) < \text{clock}(B2) \land \text{clock}(m1) < \text{clock}(A1) \implies \text{clock}(B2) < \text{clock}(A1) \]
\[\text{clock}(m1) < \text{clock}(A1) \]
\[\text{clock}(m1) < \text{clock}(B2) \]
\[\implies x_2 = 0 \]

\[\text{read-from} \to \text{write serialisation} \to \text{from-read} \]
Symbolic Partial Orders - From-Read

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

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

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

\[ x_2 = 0 \]
Symbolic Partial Orders - From-Read

A cycle means this is not a valid execution

clock(m1) < clock(A1) 
\(\Rightarrow\) 
clock(m2) < clock(A2) \(\land\) clock(m2) < clock(B1) 
\(\Rightarrow\) 
clock(A2) < clock(B1)

clock(m1) < clock(B2) \(\land\) clock(m1) < clock(A1) 
\(\Rightarrow\) 
clock(B2) < clock(A1)

\((\text{A}1)\) \(W_x,1\) 
\((\text{A}2)\) \(R_y, y_1\) 
\((\text{A}3)\) \(W_i, i_0\)

\((\text{m}1)\) \(W_x,0\) 
\((\text{m}2)\) \(W_y,0\)

clock(m1) < clock(A1) 
\(\Rightarrow\) 
clock(m1) < clock(B2) 
\(\Rightarrow\) 
x_2 = 0

\((\text{B}1)\) \(W_y,1\) 
\((\text{B}2)\) \(R_x, x_2\) 
\((\text{B}3)\) \(W_j, j_0\)
Symbolic Partial Orders - Interrupts

- Define $\text{preempts}(e_1, e_2)$ to be true if the ISR call that contains memory event $e_1$ preempts the ISR call containing $e_2$

- For all memory events $e_1, e_2$ such that $\text{preempts}(e_1, e_2)$, if $(e_1, e_2) \in \text{write-serialisation} \cup \text{read-from} \cup \text{from-read}$, then for all events $e$ such that $(e_1, e) \in \text{program-order}$, we have $(e, e_2) \in \text{interrupts}$
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
Symbolic Partial Orders - Interrupts

Handler A

A_1
A_2
A_3
\ldots
A_n

Program Order

Handler B

B_1
B_2
B_3
\ldots
B_m

program order
Symbolic Partial Orders - Interrupts

Program Order

Handler A

A_1

A_2

A_3

\ldots

A_n

Handler B

B_1

B_2

B_3

\ldots

B_m

Preemption

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

Handler A

A₁
A₂
A₃
...
Aₙ

Program Order

Handler B

B₁
B₂
B₃
...
Bₙ

Preemption

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

Handler A

A₁
A₂
A₃
...
Aₙ

Handler B

B₁
B₂
B₃
...
Bₘ

Program Order

Preemption

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

Program Order

Handler A

\[ A_1 \]

\[ A_2 \]

\[ A_3 \]

\[ \ldots \]

\[ A_n \]

Preemption

Clock \( \text{clock}(A_n) < \text{clock}(B_3) \)

Handler B

\[ B_1 \]

\[ B_2 \]

\[ B_3 \]

\[ \ldots \]

\[ B_m \]

- program order
- 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
From Interrupt-Driven to Sequential/Concurrent Code

- Interrupt-Driven Code in C
- Sequential/Concurrent C Code
- Sequential/Concurrent Model Checkers
Alternative Technique
Sequentialisation

- Use a global array irq_enabled[] 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
void schedule(void) {
    int j = count;
    int irq;

    for (int i = 0; i < j; i++) {
        irq = nondet_int();
        assume( irq >= 0 && irq < num_irqs );

        if (count!=0 && irq_enabled[irq] &&
            nondet_bool()) {
            count--;
            switch(irq) {
            case 1: handler_1(); break;
            case 2: handler_2(); break;
            ...
            default:
            }
        }
    }
}
void schedule(void) {
    int j = count;
    int irq;

    for (int i = 0; i < j; i++) {
        irq = nondet_int();
        assume( irq >= 0 && irq < num_irqs );

        if (count!=0 && irq_enabled[irq] &&
            nondet_bool()) {
            count--;
            switch(irq) {
                case 1: handler_1(); break;
                case 2: handler_2(); break;
                ...
                default:
            }
        }
    }
}
void schedule(void) {
    int j = count;
    int irq;

    for (int i = 0; i < j; i++) {
        irq = nondet_int();
        assume( irq >= 0 && irq < num_irqs );

        if (count!=0 && irq_enabled[irq] &&
            nondet_bool()) {
            count--;
            switch(irq) {
                case 1: handler_1(); break;
                case 2: handler_2(); break;
                ...
            default:
            }
        }
    }
}
Alternative Technique
Instrumentation with Threads

- Use a global array \texttt{thread\_running[]} to indicate which interrupts are executing/preempted
- Thread \(i\) has higher priority than thread \(j\) if \(i < j\)
- Instrument atomic statements
  
  \begin{verbatim}
  assume(thread\_running[0] == 0 && ... && thread\_running[n-1] == 0)
  \end{verbatim}

  between statements of thread \(n\)

Regehr and Cooprider ENTCS07
irqreturn_t handler_n(param_list) {

    /* original code */
    ...

    return IRQ_HANDLED;
}

irqreturn_t handler_n(param_list) {
    atomic { thread_running[n] = 1; }

    /* original code */
    ...

    atomic { thread_running[n] = 0; }  
    return IRQ_HANDLED;
}
irqreturn_t handler_n(param_list) {
    atomic { thread_running[n] = 1; }

    /* original code */
    atomic_assume(thread_running[0]==0 && ... &&
                  thread_running[n-1] == 0);
    statement;
    atomic_assume(thread_running[0]==0 && ... &&
                  thread_running[n-1] == 0);
    statement;
    ...

    atomic { thread_running[n] = 0; }
    return IRQ_HANDLED;
}
## 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>×</td>
<td>✓ 2.1 s</td>
<td>TO</td>
<td>TO</td>
<td>✓ 25.2 s</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>2,652</td>
<td>2</td>
<td>×</td>
<td>unknown</td>
<td>✓ 1425.1 s</td>
<td>TO</td>
<td>✓ 3.6 s</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>2,652</td>
<td>2</td>
<td>×</td>
<td>unknown</td>
<td>? 1420.5 s</td>
<td>TO</td>
<td>! 4.2 s</td>
</tr>
<tr>
<td><strong>RcCore</strong></td>
<td>7,035</td>
<td>3</td>
<td>×</td>
<td>unknown</td>
<td>TO</td>
<td>TO</td>
<td>✓ 75.7 s</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>7,035</td>
<td>3</td>
<td>TO</td>
<td>unknown</td>
<td>TO</td>
<td>TO</td>
<td>✓ 75.5 s</td>
</tr>
<tr>
<td><strong>Brake (1 Wheel)</strong></td>
<td>3,938</td>
<td>2</td>
<td>* 0.8 s</td>
<td>* 3.1 s</td>
<td>✓ 0.8 s</td>
<td>×</td>
<td>✓ 154.2 s</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>3,938</td>
<td>2</td>
<td>TO</td>
<td>unknown</td>
<td>? 1.1 s</td>
<td>×</td>
<td>! 3.7 s</td>
</tr>
<tr>
<td><strong>Brake (2 Wheels)</strong></td>
<td>3,938</td>
<td>3</td>
<td>TO</td>
<td>* 5.6 s</td>
<td>✓ 7.8 s</td>
<td>×</td>
<td>TO</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>3,938</td>
<td>3</td>
<td>TO</td>
<td>unknown</td>
<td>? 1.1 s</td>
<td>×</td>
<td>! 6.7 s</td>
</tr>
<tr>
<td><strong>Brake (3 Wheels)</strong></td>
<td>3,938</td>
<td>4</td>
<td>TO</td>
<td>* 10.6 s</td>
<td>✓ 854.1 s</td>
<td>×</td>
<td>TO</td>
</tr>
<tr>
<td>+ incorrect</td>
<td>3,938</td>
<td>4</td>
<td>TO</td>
<td>unknown</td>
<td>? 3.8 s</td>
<td>×</td>
<td>! 9.8 s</td>
</tr>
</tbody>
</table>

✓ = proved correct, ! = bug exposed, ? = bug missed, * = false alarm

× = tool crashes, PE = parse errors, TO = timeout 1800 s
## Experimental Results (cont.)

<table>
<thead>
<tr>
<th></th>
<th>LOC</th>
<th>Interrupts</th>
<th>Instrumentation with Threads</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>IMPARA r878</td>
<td>ESBMC 1.23</td>
</tr>
<tr>
<td><strong>Logger</strong></td>
<td>112</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>0.2 s</strong></td>
<td><strong>2.5 s</strong></td>
</tr>
<tr>
<td><strong>+ incorrect</strong></td>
<td>112</td>
<td>2</td>
<td>!</td>
<td>!</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>0.4 s</strong></td>
<td><strong>0.2 s</strong></td>
</tr>
<tr>
<td><strong>Logger (extended)</strong></td>
<td>172</td>
<td>3</td>
<td>✓</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>65.2 s</strong></td>
<td><strong>250.6 s</strong></td>
</tr>
<tr>
<td><strong>+ incorrect</strong></td>
<td>172</td>
<td>3</td>
<td>!</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>117.1 s</strong></td>
<td>!</td>
</tr>
<tr>
<td><strong>Blink</strong></td>
<td>2,652</td>
<td>2</td>
<td>✓</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>360.8 s</strong></td>
<td><strong>4.5 s</strong></td>
</tr>
<tr>
<td><strong>+ incorrect</strong></td>
<td>2,652</td>
<td>2</td>
<td>!</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>37.3 s</strong></td>
<td>?</td>
</tr>
<tr>
<td><strong>RcCore</strong></td>
<td>7,035</td>
<td>3</td>
<td>×</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>+ incorrect</strong></td>
<td>7,035</td>
<td>3</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>!</td>
</tr>
<tr>
<td><strong>Brake (1 Wheel)</strong></td>
<td>3,938</td>
<td>2</td>
<td>TO</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>+ incorrect</strong></td>
<td>3,938</td>
<td>2</td>
<td>TO</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>!</td>
</tr>
<tr>
<td><strong>Brake (2 Wheels)</strong></td>
<td>3,938</td>
<td>3</td>
<td>TO</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>+ incorrect</strong></td>
<td>3,938</td>
<td>3</td>
<td>TO</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>!</td>
</tr>
<tr>
<td><strong>Brake (3 Wheels)</strong></td>
<td>3,938</td>
<td>4</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td><strong>+ incorrect</strong></td>
<td>3,938</td>
<td>4</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>!</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 comparison between CBMC with thread instrumentation and i-CBMC (time in seconds). The x-axis represents i-CBMC (time in seconds) ranging from $10^{-1}$ to $10^3$, and the y-axis represents CBMC with thread instrumentation also ranging from $10^{-1}$ to $10^3$. There are data points indicating safe and unsafe conditions. The graph also includes a diagonal line representing equal time, and red lines indicating timeouts.]
Website

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