Timing Analysis of Event-Driven Programs with Directed Testing

Mahdi Eslamimehr
Hesam Samimi
{eslamimehr, samimi}@ucla.edu
Communications Design Group, SAP Labs
Talk Outline

• Introduction
  – Toyota UA Case
  – Problem Definition and Previous Work
  – Review of Classic Directed Testing
  – Motivating Experiments

• Our Approach
  – Example
  – VICE: Algorithms and tools

• Experiment Results

• Conclusion
  – *Event-Based Directed Testing* improves the state-of-art
Aug. 28, 2009, San Diego CA, USA

- Toyota Lexus ES 350 sedan
  - UA Reached 100 mph+

- 911 Emergency Phone Call from passenger during event
  - All 4 occupants killed in crash

- Driver:
  Mark Saylor, 45 year old male.
  Off-duty California Highway Patrol Officer; vehicle inspector.
  - Crash was blamed on wrong floor mats causing pedal entrapment
  - Brake rotor damage indicated “endured braking”

- This event triggered escalation of investigations dating back to 2002 MY
Recalls & Public Discussion

(Brakes might not mitigate open throttle – more later)

- Floor mat recalls
  - Sept. 2007 recall to fasten floor mats
  - Wider recall Oct./Nov. 2009 after Saylor mishap

- Sticky gas pedal recall
  - Jan. 2010 and onward

- Congressional investigation
  - Toyota President testifies to US Congress, Feb. 2010
  - April 2010: Economic loss class action venue selected
NASA Investigation

- NASA team investigates UA (2010-2011)
  - Including **Electronic Throttle Control System (ETCS)**
  - Controls air + fuel + spark \(\rightarrow\) engine power

- **Timing analysis too difficult for NASA**
  - **Worst Case Execution Time** difficult due to busy-wait loops, indirect recursion, etc. [NASA App. A pp. 120-133]
  - But, no deadline misses seen [NASA App. A. pp. 120-125]
Toyota 2008 ETCS – Two CPUs

Main CPU (Contains Software)

Monitor Chip (Contains Software)

http://m.eet.com/media/1201063/Toyota_ECM.jpg

© Copyright 2014, Philip Koopman. CC Attribution 4.0 International license.
$1.6B Economic Loss Class Action

- “Lawsuit pursues claims for breach of warranties, unjust enrichment, and violations of various state consumer protection statutes, among other claims.”
- https://www.toyotaelsettlement.com/
- 2002 through 2010 models of Toyota vehicles
- Toyota denies claims; settled for $1.6 Billion in Dec. 2012
- Brake override firmware update for in some recent models

Please be advised that the Brake Override System installation is now available for the following make and model vehicles:
- 2008 - 2010 Toyota LandCruiser
- 2009 - 2010 Toyota Corolla
- 2009 - 2010 Toyota Corolla Matrix
- 2008 - 2010 Toyota Highlander
- 2006 - 2010 Toyota RAV4
- 2003 - 2009 Toyota 4Runner
- 2007 - 2010 Toyota Tundra
- 2008 - 2010 Lexus LX
- 2010                        Lexus RX

https://www.toyotaelsettlement.com/
3 August 2014

© Copyright 2014, Philip Koopman. CC Attribution 4.0 International license.
Bookout/Schwarz Trial

- October 2013, Oklahoma
  - Fatal 2007 crash of a 2005 Toyota Camry
  - Neither floor mat nor sticky pedal recalls cover this MY; no “fixes” announced

- Toyota blamed driver error for crash
  - Mr. Arora (Exponent) testified as Toyota software expert
  - “[Toyota’s counsel] theorized that Bookout mistakenly pumped the gas pedal instead of the brake, and by the time she realized her mistake and pressed the brake, it was too late to avoid the crash” [http://bigstory.ap.org/article/oklahoma-jury-considers-toyota-acceleration-case]

- Plaintiffs blamed ETCS
  - Dr. Koopman & Mr. Barr testified as software experts
  - Testified about defective safety architecture & software defects
  - 150 feet of skid marks implied open throttle while braking

© Copyright 2014, Philip Koopman. CC Attribution 4.0 International license.
Controllers in safety time-critical embedded systems are expected to finish their tasks within reliable time bounds.

- **Underestimation** causes missing deadlines and leads to bugs
- **Overestimation** wastes process availability.

**Question:** what is the exact WCET across all inputs?

1. Program $P$, $K$ is the WCET of all executions of $P$, if $P$’s WCET never grows beyond $K$.
2. There is a possible schedule of events and an execution of the program $P$ such that the WCET becomes $K$.  

**WCET**
WCET in Literature

– **Dynamic Analysis**
  - Random Algorithms:
    - [Bernat et. Al., *RTSS’02*]
  - Genetic Algorithm:
    - [Atanassov et. Al., *EWDC’01*]
  - Classic Directed Testing:
    - [N. Williams and M. Roger, *AST’09*]

– **Static Analysis**
  - [Holsti et. Al., *ESA’2000*]
  - [C. Ferdinand, *BIS’04*]
  - [Gustafsson and Ermedahl, *RTSS’06*]
Classic Directed Testing

- Generate concrete inputs one by one
  - each input leads program along a different path

- On each input execute program both concretely and symbolically
  - concrete execution *guides* the symbolic execution
  - concrete execution *enables* symbolic execution to overcome incompleteness of theorem prover
  - symbolic execution *helps to generate* concrete input for next execution
    - increases coverage

Input = Random

While all paths covered?

Yes

End

No

Concolically execute and collect path constraints

Solving constraints and generate inputs
Example

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
Directed Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
```

Concrete Execution

Symbolic Execution

<table>
<thead>
<tr>
<th>Concrete state</th>
<th>Symbolic state</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 22, y = 7</td>
<td>x = x₀, y = y₀</td>
<td></td>
</tr>
</tbody>
</table>
Directed Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution
- concrete state
- path condition

Symbolic Execution
- symbolic state
- path condition

\[ x = 22, \ y = 7, \ z = 14 \]

\[ x = x_0, \ y = y_0, \ z = 2*y_0 \]
Directed Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

Symbolic Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 22, y = 7, z = 14</td>
<td>x = x₀, y = y₀, z = 2*y₀</td>
<td>2*y₀ != x₀</td>
</tr>
</tbody>
</table>

Concrete state:
- x = 22, y = 7
- z = 14

Symbolic state:
- x = x₀
- y = y₀
- z = 2*y₀

Path condition:
- 2*y₀ != x₀
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

x = 22, y = 7, z = 14
x = x₀, y = y₀, z = 2*y₀

Solve: 2*y₀ == x₀
Solution: x₀ = 2, y₀ = 1

2*y₀ != x₀
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}

Directed Testing Approach

Concrete Execution

Symbolic Execution

concrete state

symbolic state

path condition

x = 2, y = 1, z = 2
x = x₀, y = y₀, z = 2 * y₀

Concrete Execution

Symbolic Execution

Concrete state

Symbolic state

Path condition

x = 2, y = 1, z = 2
x = x₀, y = y₀, z = 2 * y₀
Directed Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y + 10) {
            // ERROR;
        }
    }
}
```

**Concrete Execution**

<table>
<thead>
<tr>
<th>Concrete State</th>
<th>Symbolic State</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 2, y = 1, z = 2</td>
<td>x = x₀, y = y₀, z = 2*sy₀</td>
<td>2*sy₀ == x₀</td>
</tr>
</tbody>
</table>
Directed Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution
- **Concrete state**: $x = 2$, $y = 1$, $z = 2$
- **Symbolic state**: $x = x_0$, $y = y_0$, $z = 2y_0$
- **Path condition**: $2y_0 == x_0$ and $x_0 < y_0 + 10$
Directed Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
```

Concrete Execution
- x = 2, y = 1, z = 2
- Solve: \((2y = x) \land (x > y + 10)\)
- Solution: \(x_0 = 30, y_0 = 15\)

Symbolic Execution
- Concrete state
- Symbolic state
- path condition
- \(x = x_0, y = y_0, z = 2y_0\)
**Directed Testing Approach**

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

### Concrete Execution
- **State**: $x = 30, y = 15$

### Symbolic Execution
- **State**: $x = x_0, y = y_0$

### Path Condition
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
Explicit Path Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
Explicit Path Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
Explicit Path Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
Explicit Path Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
Explicit Path Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
Explicit Path Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
Motivating Experiments
branch coverage across testing techniques

- BinaryTree
- LinkedList
- BubbleSort
- Decoder
- Oscilloscope
- Fannkuch
- MsgKernel
- TestRadio
- TestUSART
- TestSPI
- TestADC

- Random Testing
- GA
- Classical Directed Testing
Motivating Experiments

Testing VS Static Analysis of WCET

Logarithmic Scale

WCET

Random Testing
Static Analysis
GA
Classical Directed Testing
Testing Event Driven Software

• Classical software:
  – tester only devices a suite of single inputs.

• Event-Driven software (with real-time behavior):
  – tester must device a suite of event sequences.
    – In each sequence: # of events, types of events, values associated with the events e.g. registers’ value, and timing of events.

• Challenge: Quickly generate a small number of challenging event sequences to improve branch coverage.
**Round 1**

$$[<\text{main},(723452)>,<\text{alt1},(-10038)>,<\text{main},\_>,<\text{alt1},\_>]$$

Constraints: $data_1 = msg \land data_2 = msg \land -2048 < msg \land msg < 1024$

Branch Coverage: $50\% \ (3/6)$
program Sample { entrypoint main = antenna.main, alt = antenna.alt; }
component test_antenna {
    field sending:bool = false;
    method main(data_1:int):void { dispatch_data(data_1); // code... } 
    method alt(data_2:int):void { dispatch_data(data_2); } 
    method dispatch_data(msg:int):void {
        local res:int, tmp:int = random(100)
        if (sending) return; else sending = true;
        if (-2048 < msg && msg < 1024) res = check_and_send(msg,tmp);
        sending = false;
        return;
    }
    method check_and_send(s:int, t:int):int {
        if (s==512)
            return 1;
        // send...
        return 0;
    }
}

Round 2

[<main,(-338)>,<alt1,(1001)>,<alt2,(6)>,<main, _>] 
Constraints: msg = s ∧ tmp = t ∧ s = 512 
Branch Coverage: 83% (5/6)
program Sample { entrypoint main = antenna.main, alt = antenna.alt; }
component test_antenna {
    field sending:bool = false;
    method main(data_1:int):void { dispatch_data(data_1); // code... } 
    method alt(data_2:int):void { dispatch_data(data_2); }
    method dispatch_data(msg:int):void {
        local res:int, tmp:int = random(100)
        if (sending) return; else sending = true;
        if (-2048 < msg && msg < 1024) res = check_and_send(msg,tmp);
        sending = false;
        return;
    }
    method check_and_send(s:int, t:int):int {
        if (s==512)
            return 1;
        // send...
        return 0;
    }
}

Round 3
[<main,(-338)>,<alt1,(1001)>,<alt2,(6)>,<main, _>]
Constraints: data1 = data2 = msg = s = 512
Branch Coverage: 83% (5/6)
program Sample { entrypoint main = antenna.main, alt = antenna.alt; }
component test_antenna {
    field sending:bool = false;
    method main(data_1:int):void { dispatch_data(data_1); // code... }    
    method alt(data_2:int):void { dispatch_data(data_2); }
    method dispatch_data(msg:int):void {
        local res:int, tmp:int = random(100)
        if (sending) return; else sending = true;
        if (-2048 < msg && msg < 1024) res = check_and_send(msg,tmp);
        sending = false;
        return;
    }
    method check_and_send(s:int, t:int):int {
        if (s==512)
            return 1;
        // send...
        return 0;
    }
}

Round 4
[<main,(512)>,<alt1,(512)>,<main,->_>,<alt1,->_>]
Constraints: -
Branch Coverage: 100% (6/6)
Event Based Directed Testing (EBDT)

- **compiler:**
  \[ \text{VirgilProgram} \rightarrow \text{machineCode} \]

- **avrora:**
  \[ \text{machineCode} \times \text{eventSequence} \rightarrow \text{wcet} \]

- **random:**
  \[ () \rightarrow \text{eventSequence} \]

- **timeoutCombos:**
  \[ \text{eventSequence} \rightarrow (\text{eventSequence list}) \]

- **concolic:**
  \[ (\text{Virgil program} \times \text{eventSequence} \rightarrow (\text{wcet} \times \text{branchCoverage} \times \text{constraints}) \]

- **solver:**
  \[ \text{constraints} \rightarrow \text{solution} \]

- **generator:**
  \[ \text{solution} \rightarrow \text{eventSequence} \]
Algorithm

Input: \( p: \) VirgilProgram
Output: \( \text{wcet} \times \text{branchCoverage} \times \text{eventSequence} \)
Local: \( a: \) machineCode = \text{compiler}(p), \( s: \) state = (0, 0.0, ( ), 0), \( \text{roundId}: \) int = 0
\( \text{seqs}: \) eventSequence = \text{timeoutCombos(random())}
Method: while (\( s.\text{noChange} < 2 \)) {
    foreach (\( seq \) in \( \text{seqs} \)) {
        let (\( \text{wcet}, bc, c \)) = \text{concolic}(p, seq) in
        \( s = \text{update}(s, \text{wcet}, bc, c, \text{roundId}) \)
    }
    \( \text{roundId}++ \)
    \( \text{seqs} = \text{timeoutCombos(generator(solver(s.constraints)))} \)
}
return (\( s.\text{wcet}, s.\text{coverage}, s.\text{eventSeq} \))
Experiment Results

Logarithmic Scale

Random Testing  GA  Classical Directed Testing  VICE

BinaryTree  LinkedList  BubbleSort  Decoder  Oscilloscope  Fannkuch  MsgKernel  TestRadio  TestUSART  TestSPI  TestADC
Future Works

- Formulating timeouts symbolically
- Using some static information
  - Locate places where wcet happens, and direct execution towards candidates
  - Replace random event generation with a IMR-certified model checker.