From Symbolic Execution to Concolic Testing

Daniel Paqué
Structure

- Symbolic Execution
- Concolic Testing
- Execution Generated Testing
- Concurrency in Concolic Testing
Motivation

- Software Testing “usually accounts for 50% of software development cost”
  [Source: “The economic impacts of inadequate infrastructure for software testing”, NIST]
- complex and large Software Systems complicate finding small test suites with high coverage

- Symbolic Execution
  - automatic test case generation
  - high code coverage
Symbolic Execution - **Idea**

- execute the program in symbolic domain
  - explore all possible execution paths
  - for each path the constraints of the branching points are collected
  - generate test input based on the constraints
Symbolic Execution - Example

```c
1   foo(int x, int y){
2       z = 2*y;
3       if (x == z){
4           if (x > y + 5){
5               //some error
6               }
7           }
8       }
```
Symbolic Execution - Example

```java
1   foo(int x, int y) {
2       z = 2*y;
3       if (x == z) {
4           if (x > y + 5) {
5               //some error
6           }
7       }
8   }
```

symbolic domain:

<table>
<thead>
<tr>
<th>σ</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊘</td>
<td>true</td>
</tr>
</tbody>
</table>

symbolic state

path condition
Symbolic Execution - Example

```
1   foo(int x, int y) {
2       z = 2*y;
3       if (x == z) {
4           if (x > y + 5) {
5               //some error
6           }
7       }
8   }
```

symbolic domain:

<table>
<thead>
<tr>
<th>( \sigma )</th>
<th>( PC )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \mapsto x_0 )</td>
<td>true</td>
</tr>
<tr>
<td>( y \mapsto y_0 )</td>
<td></td>
</tr>
</tbody>
</table>
Symbolic Execution - Example

```c
1    foo(int x, int y) {
2        z = 2*y;
3        if (x == z) {
4            if (x > y + 5) {
5                // some error
6            }
7        }
8    }
```

symbolic domain:

<table>
<thead>
<tr>
<th>(\sigma)</th>
<th>(PC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x \mapsto x_0)</td>
<td>true</td>
</tr>
<tr>
<td>(y \mapsto y_0)</td>
<td></td>
</tr>
<tr>
<td>(z \mapsto 2y_0)</td>
<td></td>
</tr>
</tbody>
</table>
Symbolic Execution - Example

```c
1  foo(int x, int y) {
2      z = 2*y;
3      if (x == z) {
4          if (x > y + 5) {
5              // some error
6          }
7      }
8  }
```

**symbolic domain:**

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \mapsto x_0$</td>
<td>$(x_0 \neq 2y_0)$</td>
</tr>
<tr>
<td>$y \mapsto y_0$</td>
<td></td>
</tr>
<tr>
<td>$z \mapsto 2y_0$</td>
<td></td>
</tr>
</tbody>
</table>

$PC = PC \land b$

$PC' = PC' \land \neg b$

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \mapsto x_0$</td>
<td>$(x_0 = 2y_0)$</td>
</tr>
<tr>
<td>$y \mapsto y_0$</td>
<td></td>
</tr>
<tr>
<td>$z \mapsto 2y_0$</td>
<td></td>
</tr>
</tbody>
</table>

satisfiable?
Symbolic Execution - Example

```
1  foo(int x, int y) {
2      z = 2*y;
3      if (x == z) {
4          if (x > y + 5) {
3                // some error
4          }
5      }
6  }
7  }
8 }
```

symbolic domain:

<table>
<thead>
<tr>
<th>σ</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ← x₀</td>
<td>(x₀ ≠ 2y₀)</td>
</tr>
<tr>
<td>y ← y₀</td>
<td></td>
</tr>
<tr>
<td>z ← 2y₀</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>σ</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ← x₀</td>
<td>(x₀ = 2y₀)</td>
</tr>
<tr>
<td>y ← y₀</td>
<td>∧(x₀ ≤ y₀ + 5)</td>
</tr>
<tr>
<td>z ← 2y₀</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>σ</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ← x₀</td>
<td></td>
</tr>
<tr>
<td>y ← y₀</td>
<td>(x₀ = 2y₀)</td>
</tr>
<tr>
<td>z ← 2y₀</td>
<td>∧(x₀ &gt; y₀ + 5)</td>
</tr>
</tbody>
</table>

satisfiable?

From Symbolic Execution to Concolic Testing  28.11.2014
Symbolic Execution - Example

\[ x = z \]

\[ (x_0 = 2y_0) \]
\[ (x_0 \neq 2y_0) \]

\[ x > y + 5 \]

\[ (x_0 = 2y_0) \land (x_0 > y_0 + 5) \]
\[ (x_0 = 2y_0) \land (x_0 \leq y_0 + 5) \]

\[ (x = 12, y = 6) \]
\[ (x = 2, y = 1) \]
\[ (x = 1, y = 1) \]
**Limits of Symbolic Execution**

```c
1  foo(int x, int y) {
2      z = bar(y);
3      if (x == z) {
4          if (x > y + 5) {
5              // some error
6          }
7      }
8  }
```

### Symbolic Domain:

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$PC$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \mapsto x_0$</td>
<td>$(x_0 \neq bar(y_0))$</td>
</tr>
<tr>
<td>$y \mapsto y_0$</td>
<td></td>
</tr>
<tr>
<td>$z \mapsto bar(y_0)$</td>
<td></td>
</tr>
</tbody>
</table>

### Satisfying Assignments:

- $x \mapsto x_0$
- $y \mapsto y_0$
- $z \mapsto bar(y_0)$

**satisfiable?**

```
```
Solution

- Mix Symbolic Execution with Concrete Execution
  - Concolic Testing
  - Execution Generated Testing
Symbolic Execution

- 1979
- J.C. King

Concolic Testing

- 2005,
  - Godefroid, Sen

Execution Generated Testing (EGT)

- 2006
  - Cadar et. al

mix concrete with symbolic execution

+ improvements in constraint solving
Concolic Testing

- execute program with concrete values and collect symbolic constraints during execution
- explore paths sequentially instead of forking
  - infer input for next execution
- use concrete values to solve problematic constraints
Concolic Testing - Example

```c
1  foo(int x, int y){
2      z = 2*y;
3      if (x == z){
4          if (x > y + 5){
4          // some error
5      }
6  }
7  }
8 }
```

symbolic state:

<table>
<thead>
<tr>
<th>( \sigma )</th>
<th>( PC )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \mapsto x_0 )</td>
<td>true</td>
</tr>
<tr>
<td>( y \mapsto y_0 )</td>
<td></td>
</tr>
</tbody>
</table>
Concolic Testing - Example

```c
1 foo(int x, int y) {
2     z = 2*y;
3     if (x == z) { // some error
4         if (x > y + 5) {
5             false
6         }
7     }
8 }
```

symbolic state:

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$PC$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \mapsto x_0$</td>
<td>$(x_0 \neq 2y_0)$</td>
</tr>
<tr>
<td>$y \mapsto y_0$</td>
<td></td>
</tr>
<tr>
<td>$z \mapsto 2y_0$</td>
<td></td>
</tr>
</tbody>
</table>
Concolic Testing - Example

```c
1  foo(int x, int y){
2      z = 2*y;
3      if (x == z){
4          if (x > y + 5){
5              // some error
6      }
7      }
8  }
```

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$PC$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \mapsto x_0$</td>
<td>$(x_0 \neq 2y_0)$</td>
</tr>
<tr>
<td>$y \mapsto y_0$</td>
<td>$z \mapsto 2y_0$</td>
</tr>
</tbody>
</table>

new input: $\{x = 8, y = 4\}$
Concolic Testing - Example

\[ x = z \]

\[ (x_0 = 2y_0) \]

\[ (x_0 \neq 2y_0) \]

\[ x > y + 5 \]

\[ (x_0 = 2y_0) \land (x_0 > y_0 + 5) \]

\[ (x_0 = 2y_0) \land (x_0 \leq y_0 + 5) \]

\[ (x = 12, y = 6) \]

\[ (x = 8, y = 4) \]

\[ (x = 29, y = 4) \]
Concolic Testing - Example

```c
1    foo(int x, int y) {
2        z = bar(y);
3        if (x == z) {
4            if (x > y + 5) {
5                // some error
6            }
7        }
8    }
```

symbolic domain:

\[
(x_0 = \text{bar}(y_0))
\]

evaluate condition in concrete

\[
\{x = 8, y = 4\}
\]

true

\[
\begin{array}{c|c}
\sigma & PC \\
\hline
x \mapsto x_0 & (x_0 = \text{bar}(y_0)) \\
y \mapsto y_0 & \\
z \mapsto \text{bar}(y_0) & \\
\end{array}
\]
Concolic Testing - Example

```c
foo(int x, int y){
    z = bar(y);
    if (x == z){
        if (x > y + 5){
            // some error
        }
    }
}
```

**symbolic state:**

\[
(x_0 = \text{bar}(y_0))
\]

\{x = 8, y = 4\}

**evaluate bar() in concrete**

<table>
<thead>
<tr>
<th>(\sigma)</th>
<th>(PC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x \mapsto x_0)</td>
<td>((x_0 = 8))</td>
</tr>
<tr>
<td>(y \mapsto y_0)</td>
<td></td>
</tr>
<tr>
<td>(z \mapsto 8)</td>
<td></td>
</tr>
</tbody>
</table>
Symbolic Execution

1979
J.C. King

Concolic Testing
- sequential path exploration
- guided by concrete input

Execution Generated Testing (EGT)
- fork execution for each path
- guided by symbolic execution

mix concrete with symbolic execution
Execution Generated Testing

- further differences to Concolic Testing:
  - EGT dynamically checks if all operands are concrete
    - if so the operation can be executed in concrete
    - otherwise the operation is executed symbolical

```c
1 foo3(int x) {
2     y = 2;
3     z = 3*y;
4     if (x == z) {
5         // ...
```
How to deal with concurrent programs?
Main Challenge

From Symbolic Execution to Concolic Testing 28.11.2014
Main Challenge

summarize redundant interleavings

Thread 0

Thread 1

\[ x = 42; \]

...
Koushik Sen & Gul Agha: (2006)

- „race-detection and flipping algorithm“
  - minimize redundant executions in concurrent programs
  - uses vector clocks to identify races
Redundant Executions

**Thread t₀:**

1: \( x = 3; \)

**Thread t₁:**

1: \( y = 0; \)
2: \( x = 4; \)
3: \( z = x + 12; \)

**Execution 1:**

\[
\begin{align*}
x &= 3; \\
y &= 0; \\
x &= 4; \\
z &= x + 12; \\
\end{align*}
\]

**Execution 2:**

\[
\begin{align*}
y &= 0; \\
x &= 3; \\
x &= 4; \\
z &= x + 12; \\
\end{align*}
\]

**Execution 3:**

\[
\begin{align*}
y &= 0; \\
x &= 4; \\
x &= 3; \\
z &= x + 12; \\
\end{align*}
\]

**Execution 4:**

\[
\begin{align*}
y &= 0; \\
x &= 4; \\
x &= 3; \\
z &= x + 12; \\
\end{align*}
\]

**result:**

\[
\begin{align*}
\{4, 0, 16\} & \quad \{4, 0, 16\} & \{3, 0, 15\} & \{3, 0, 16\}
\end{align*}
\]
Redundant Executions – Race Detection

- two events are in a race if...
  - they stem from different threads
  - both access the same memory location (without locking)
  - the order both events can be permuted by changing the schedule

<table>
<thead>
<tr>
<th>Execution 1:</th>
<th>Execution 2:</th>
<th>Execution 3:</th>
<th>Execution 4:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = 3;</code></td>
<td><code>y = 0;</code></td>
<td><code>y = 0;</code></td>
<td><code>y = 0;</code></td>
</tr>
<tr>
<td><code>y = 0;</code></td>
<td><code>x = 3;</code></td>
<td><code>x = 4;</code></td>
<td><code>x = 4;</code></td>
</tr>
<tr>
<td><code>x = 4;</code></td>
<td><code>x = 4;</code></td>
<td><code>x = 3;</code></td>
<td><code>z = x + 12;</code></td>
</tr>
<tr>
<td><code>z = x + 12;</code></td>
<td><code>z = x + 12;</code></td>
<td><code>z = x + 12;</code></td>
<td><code>x = 3;</code></td>
</tr>
</tbody>
</table>

result:             {4, 0, 16}       {4, 0, 16}       {3, 0, 15}       {3, 0, 16}

races:               (t₀, 1.1)         (t₀, 1.1)         (t₁, 1.2)         (t₁, 1.3)
                      -(t₁, 1.2)        -(t₁, 1.2)        -(t₀, 1.1)        -(t₀, 1.1)
                      (t₀, 1.1)         (t₀, 1.1)         (t₁, 1.3)         (t₁, 1.3)
The Race-Detection and Flipping Algorithm

init:
- generate a random input and a schedule

loop:
- execute code with the generated input and schedule
- compute the race conditions and symbolic constraints
- generate a new schedule or a new input
- continue until all possible distinct execution paths have been explored (depth-first search strategy)
Generating new inputs/schedules

- new input: concolic testing
- new schedule:
  - pick two events which are in a race
  - delay the first event as much as possible

\[\text{schedule 1:}\]
\[
\begin{align*}
x &= 3; \\
y &= 0; \\
x &= 4; \\
z &= x + 12;
\end{align*}
\]

\[\text{schedule 2:}\]
\[
\begin{align*}
y &= 0; \\
x &= 4; \\
z &= x + 12; \\
x &= 3;
\end{align*}
\]
How to identify races?
How to identify races?

- **vector clocks**

- $V : \{\text{Threads}\} \rightarrow \mathbb{N}$
- can be compared ($\leq$)
- $max$ is componentwise
- $V \neq V'$ if neither "$\leq$" nor "$\geq$"

- each thread $t$ gets its own vector clock $V_t$
- each memory location gets another two
Vector Clocks - Example

- two threads \( t_0, t_1 \)
- one memory location \( x \)

<table>
<thead>
<tr>
<th></th>
<th>( V_{t_0} )</th>
<th>( V_{t_1} )</th>
<th>( V_{x}^{a} )</th>
<th>( V_{x}^{w} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{init}_0 )</td>
<td>( t_0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>( t_1 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( \text{fork}_1 )</td>
<td>( t_0 )</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>( t_1 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( (t_0, rd, x)_2 )</td>
<td>( t_0 )</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>( t_1 )</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( (t_1, rd, x)_3 )</td>
<td>( t_0 )</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>( t_1 )</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>( (t_1, wr, x)_4 )</td>
<td>( t_0 )</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>( t_1 )</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>( (t_0, rd, x)_5 )</td>
<td>( t_0 )</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>( t_1 )</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Vector Clocks - Algorithm

- Whenever a thread $t$ with vector clock $V_t$ generates an event $e$, the following algorithm is executed:

  1. If $e$ is not a fork event or a new thread event, then $V_t(t) = V_t(t) + 1$
  2. If $e$ is a read of a shared memory location $m$ then
     \[
     V_t = \max\{V_t, V_m^w\} \quad \text{and} \quad V_m^a = \max\{V_m^a, V_t\}
     \]
  3. If $e$ is a write, lock or unlock of a shared memory location $m$ then
     \[
     V_m^w = V_m^a = V_t = \max\{V_m^a, V_t\}
     \]
  4. If $e$ is a fork event and if $t'$ is the newly created thread then
     \[
     V_{t'} = V_t, \quad V_t(t) = V_t(t) + 1 \quad \text{and} \quad V_{t'} = V_{t'} + 1
     \]
Vector Clocks – Example

1. If $e$ is not a fork event or a new thread event, then $V_t(t) = V_t(t) + 1$

<table>
<thead>
<tr>
<th></th>
<th>$V_{t_0}$</th>
<th>$V_{t_1}$</th>
<th>$V_x^a$</th>
<th>$V_x^w$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$t_0$</td>
<td>$t_1$</td>
<td>$t_0$</td>
<td>$t_1$</td>
</tr>
<tr>
<td><strong>init</strong>$^0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>fork</strong>$^1$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$(t_0,rd,x)_2$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$(t_1,rd,x)_3$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$(t_1,wr,x)_4$</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$(t_0,rd,x)_5$</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Vector Clocks – Example

1. If $e$ is not a fork event or a new thread event, then $V_i(t) = V_i(t) + 1$

<table>
<thead>
<tr>
<th></th>
<th>$V_{t_0}$</th>
<th>$V_{t_1}$</th>
<th>$V_x^a$</th>
<th>$V_x^w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>init$_0$</td>
<td>$t_0$ 0</td>
<td>$t_0$ 0</td>
<td>$t_0$ 0</td>
<td>$t_0$ 0</td>
</tr>
<tr>
<td></td>
<td>$t_1$ 0</td>
<td>$t_1$ 0</td>
<td>$t_1$ 0</td>
<td>$t_1$ 0</td>
</tr>
<tr>
<td>fork$_1$</td>
<td>$t_0$ 1</td>
<td>$t_0$ 1</td>
<td>$t_0$ 0</td>
<td>$t_0$ 0</td>
</tr>
<tr>
<td>(t$_0$, rd, x)$_2$</td>
<td>2 0</td>
<td>0 1</td>
<td>2 0</td>
<td>0 0</td>
</tr>
<tr>
<td>(t$_1$, rd, x)$_3$</td>
<td>2 0</td>
<td>0 2</td>
<td>2 2</td>
<td>0 0</td>
</tr>
<tr>
<td>(t$_1$, wr, x)$_4$</td>
<td>2 0</td>
<td>2 3</td>
<td>2 3</td>
<td>2 3</td>
</tr>
<tr>
<td>(t$_0$, rd, x)$_5$</td>
<td>3 3</td>
<td>2 3</td>
<td>3 3</td>
<td>2 3</td>
</tr>
</tbody>
</table>
### Vector Clocks – Example

2. If $e$ is a read of a shared memory location $m$ then

$$V_t = \max\{V_t, V_m^w\} \quad \text{and} \quad V_m^a = \max\{V_m^a, V_t\}$$

<table>
<thead>
<tr>
<th></th>
<th>$V_{t_0}$</th>
<th>$V_{t_1}$</th>
<th>$V_{x}^a$</th>
<th>$V_{x}^w$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init$_0$</strong></td>
<td>$t_0$ 0</td>
<td>$t_1$ 0</td>
<td>$t_0$ 0</td>
<td>$t_1$ 0</td>
</tr>
<tr>
<td><strong>fork$_1$</strong></td>
<td>$t_0$ 1</td>
<td>$t_1$ 0</td>
<td>$t_0$ 0</td>
<td>$t_1$ 0</td>
</tr>
<tr>
<td>$(t_0, rd, x)_2$</td>
<td>$t_0$ 2</td>
<td>$t_1$ 0</td>
<td>$t_0$ 0</td>
<td>$t_1$ 2</td>
</tr>
<tr>
<td>$(t_1, rd, x)_3$</td>
<td>$t_0$ 2</td>
<td>$t_1$ 0</td>
<td>$t_0$ 2</td>
<td>$t_1$ 2</td>
</tr>
<tr>
<td>$(t_1, wr, x)_4$</td>
<td>$t_0$ 2</td>
<td>$t_1$ 2</td>
<td>$t_0$ 3</td>
<td>$t_1$ 3</td>
</tr>
<tr>
<td>$(t_0, rd, x)_5$</td>
<td>$t_0$ 3</td>
<td>$t_1$ 3</td>
<td>$t_0$ 3</td>
<td>$t_1$ 3</td>
</tr>
</tbody>
</table>
Vector Clocks – Example

3. If $e$ is a write, lock or unlock of a shared memory location $m$ then

$$V_m^w = V_m^a = V_t = \max\{V_m^a, V_t\}$$

<table>
<thead>
<tr>
<th></th>
<th>$V_{t_0}$</th>
<th>$V_{t_1}$</th>
<th>$V_x^a$</th>
<th>$V_x^w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>$t_0$</td>
<td>$t_0$</td>
<td>$t_0$</td>
<td>$t_0$</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$t_1$</td>
<td>$t_1$</td>
<td>$t_1$</td>
<td>$t_1$</td>
</tr>
<tr>
<td>$V_{t_0}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$V_{t_1}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$V_x^a$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$V_x^w$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{init}_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\text{fork}_1$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$(t_0, rd, x)_2$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$(t_1, rd, x)_3$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$(t_1, wr, x)_4$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$(t_0, rd, x)_5$</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Vector Clocks – Example

3. If $e$ is a write, lock or unlock of a shared memory location $m$ then 

$$V_m^w = V_m^a = V_t = \max\{V_m^a, V_t\}$$

<table>
<thead>
<tr>
<th></th>
<th>$V_{t_0}$</th>
<th>$V_{t_1}$</th>
<th>$V_x^a$</th>
<th>$V_x^w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>$t_1$</td>
<td>$t_0$</td>
<td>$t_1$</td>
<td>$t_0$</td>
</tr>
<tr>
<td>init</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fork_1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$(t_0, rd, x)_2$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$(t_1, rd, x)_3$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$(t_1, wr, x)_4$</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$(t_0, rd, x)_5$</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

From Symbolic Execution to Concolic Testing  28.11.2014
Vector Clock Theorem

**Theorem 1.** Two events $e$ and $e'$ are race related if following holds:

1. $V\{e\} \neq V\{\text{prev}(e')\}$ given that $\text{prev}(e')$ exists, and
2. $V\{\text{next}(e)\} \neq V\{e'\}$ given that $\text{next}(e)$ exists, and
3. $V\{e\} \leq V\{e'\}$, and
4. $VS_e \neq VS_{e'}$
Questions?
Precise Definitions

(just in case)
Race Relation – Simple Definition:

Any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ are race related (denoted by $e < e'$) iff:

1. $e \nleq e'$, and
2. $e <_m e'$ and there exists no $e_1$ such that $e_1 \neq e, e_1 \neq e', e \leq e_1$ and $e_1 \leq e'$

$(t_i, l_i, a_i)$ $\rightarrow$ (thread, label, type of access)

$e \nleq e'$ $\rightarrow$ sequentially not related

$e <_m e'$ $\rightarrow$ access on the same memory location

$e \leq e_1$ $\rightarrow$ causally related
sequentially related

In an execution path $\tau \in \text{Ex}(P)$, any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ appearing in $\tau$ are sequentially related (denoted by $e \triangleleft e'$) iff:

1. $e = e'$, or
2. $t_i = t_j$ and $e$ appears before $e'$ in $\tau$, or
3. $t_i \neq t_j$, $t_i$ created the thread $t_j$, and $e$ appears before $e''$ in $\tau$, where $e''$ is the fork event on $t_i$ creating the thread $t_j$, or
4. there exists an event $e''$ in such that $e \triangleleft e''$ and $e'' \triangleleft e'$

We say $e \triangledown e'$ iff $e \not\triangleleft e'$ and $e' \not\triangleleft e$. 

access precedence related

In an execution path \( \tau \in Ex(P) \), any two events \( e = (t_i, l_i, a_i) \) and \( e' = (t_j, l_j, a_j) \) appearing in \( \tau \) are shared-memory access precedence related (denoted by \( e \prec_m e' \)) iff:

1. \( e \) appears before \( e' \) in \( \tau \), and
2. \( e \) and \( e' \) both access the same memory location \( m \), and
3. one of them is an update (not a read) of \( m \).
causally related

In an execution path \( \tau \in \text{Ex}(P) \), any two events \( e = (t_i, l_i, a_i) \) and \( e' = (t_j, l_j, a_j) \) appearing in \( \tau \) are causally related (denoted by \( e \preceq e' \)) iff:

1. \( e \triangleleft e' \), or
2. \( e <_m e' \) for some shared-memory location \( m \), or
3. there exists an event \( e'' \) in such that \( e \preceq e'' \) and \( e'' \preceq e' \)

The causal relation is a partial-order relation. We say that \( e \parallel e' \) iff \( e \not\preceq e' \) and \( e' \not\preceq e \).
race related

Any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ are race related (denoted by $e < e'$) iff:

1. $e \uparrow e'$, and
2. if $e$ is lock event and $e''$ is the corresponding unlock event, then $e'' <_m e'$ and there exists no $e_1$ such that $e_1 \neq e'', e_1 \neq e', e'' \preceq e_1$ and $e_1 \preceq e'$, and
3. if $e$ is read or write event, then $e <_m e'$ and there exists no $e_1$ such that $e_1 \neq e, e_1 \neq e', e \preceq e_1$ and $e_1 \preceq e'$
Race-Detection and Flipping Algorithm
Detailed Example

**Thread** $t_0$

1 $x = 3$

**Thread** $t_1$ (with $z$ as input)

1 $x = 2$
2 $\text{if } (x == 2*z+1)$
3 $\text{error}$
4 $\ldots$

Ex.1: [$z = 8$, $sched_0$]
   $(t_0,l.1), (t_1,l.1), (t_1,l.2), (t_1,l.4)$
Ex.2: [$z = 8$, $sched_1$]
   $(t_1,l.1), (t_1,l.2), (t_1,l.4), (t_0,l.1)$
Ex.3: [$z = 8$, $sched_2$]
   $(t_1,l.1), (t_0,l.1), (t_1,l.2), (t_1,l.4)$
Ex.4: [$z = 1$, $sched_2$]
   $(t_1,l.1), (t_0,l.1), (t_1,l.2), (t_1,l.3), (t_1,l.4)$