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
  - automic 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

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){
            //some error
        }
    }
}
```

Symbolic Execution - Example

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){
            //some error
        }
    }
}
```

symbolic domain:

σ	PC
$\emptyset$	true

symbolic state

path condition

Symbolic Execution - Example

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){
            //some error
        }
    }
}
```

symbolic domain:

σ	PC
$x \mapsto x_0$	true
$y \mapsto y_0$	

Symbolic Execution - Example

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){
            //some error
        }
    }
}
```

symbolic domain:

σ	PC
$x \mapsto x_0$	
$y \mapsto y_0$	true
$z \mapsto 2y_0$	

Symbolic Execution - Example

symbolic domain:

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){
            //some error
        }
    }
}
```

$$PC = PC \wedge b$$
$$PC' = PC' \wedge \neg b$$

1

σ	PC
$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto 2y_0$	$(x_0 \neq 2y_0)$

satisfiable ?

2

σ	PC
$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto 2y_0$	$(x_0 = 2y_0)$

satisfiable ?

Symbolic Execution - Example

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){    ⇨
            //some error
        }
    }
}
```

symbolic domain:

1

σ	PC
$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto 2y_0$	$(x_0 \neq 2y_0)$

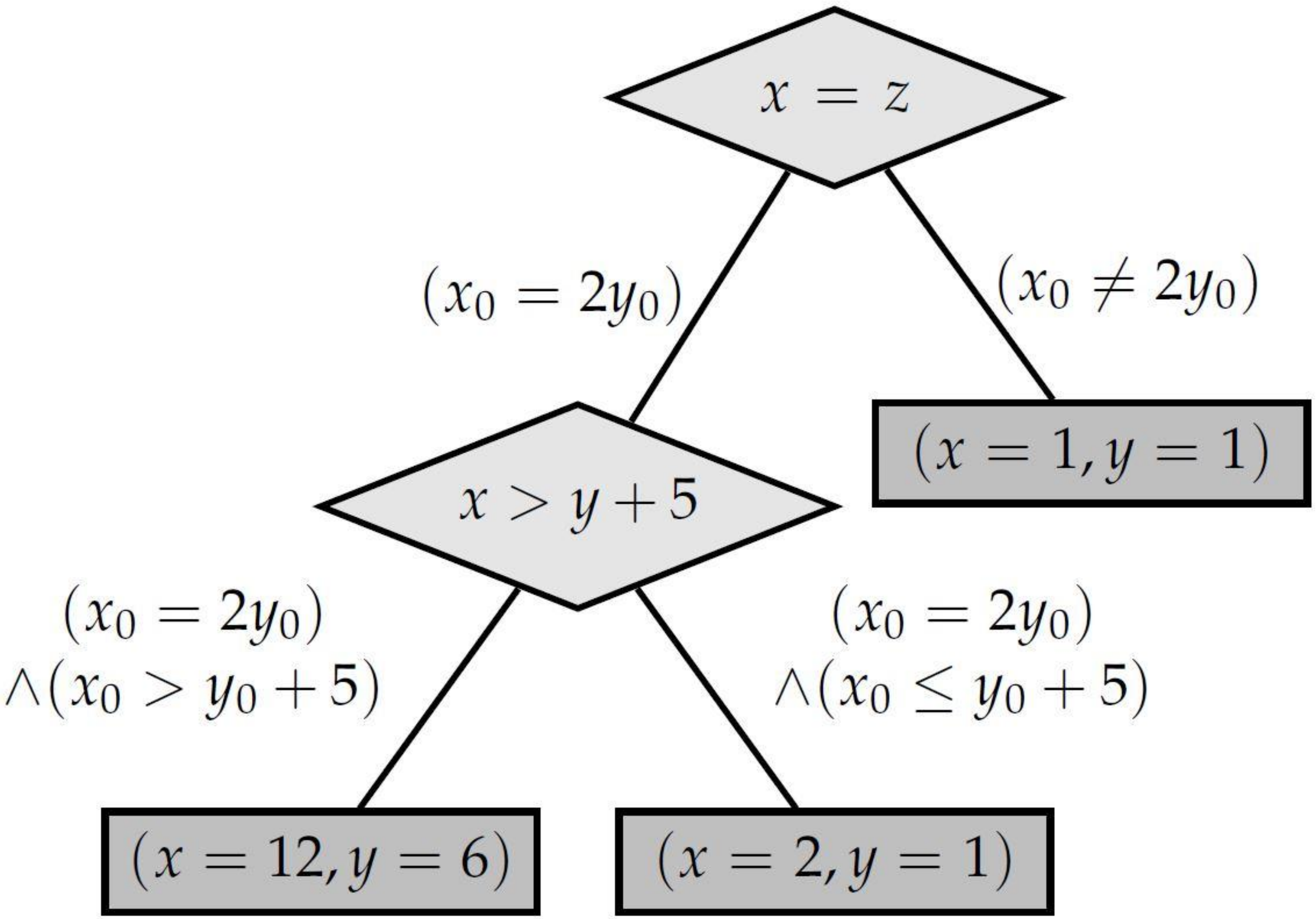
2

σ	PC
$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto 2y_0$	satisfiable ? $(x_0 = 2y_0)$ $\wedge(x_0 \leq y_0 + 5)$

3

σ	PC
$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto 2y_0$	satisfiable ? $(x_0 = 2y_0)$ $\wedge(x_0 > y_0 + 5)$

Symbolic Execution - Example



Limits of Symbolic Execution

symbolic domain:

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

satisfying assignments…

1

σ	PC
$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto bar(y_0)$	$(x_0 \neq bar(y_0))$

satisfiable ?

2

σ	PC
$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto bar(y_0)$	$(x_0 = bar(y_0))$

satisfiable ?

Solution

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

Symbolic Execution

- 1979
- J.C. King

mix concrete with symbolic execution

\+ improvements in constraint solving

Concolic Testing

- 2005,
- Godefroid, Sen

Execution Generated Testing (EGT)

- 2006
- Cadar et. al

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

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){
            //some error
        }
    }
}
```

$$\{x = 29, y = 4\}$$

symbolic state:

σ	PC
$x \mapsto x_0$	true
$y \mapsto y_0$	

Concolic Testing - Example

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){
            //some error
        }
    }
}
```

$\rightarrow$ false

$\{x = 29, y = 4\}$

symbolic state:

σ	PC
$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto 2y_0$	$(x_0 \neq 2y_0)$

Concolic Testing - Example

```
foo(int x,int y){
    z = 2*y;
    if (x == z){
        if (x > y + 5){
            //some error
        }
    }
}
```

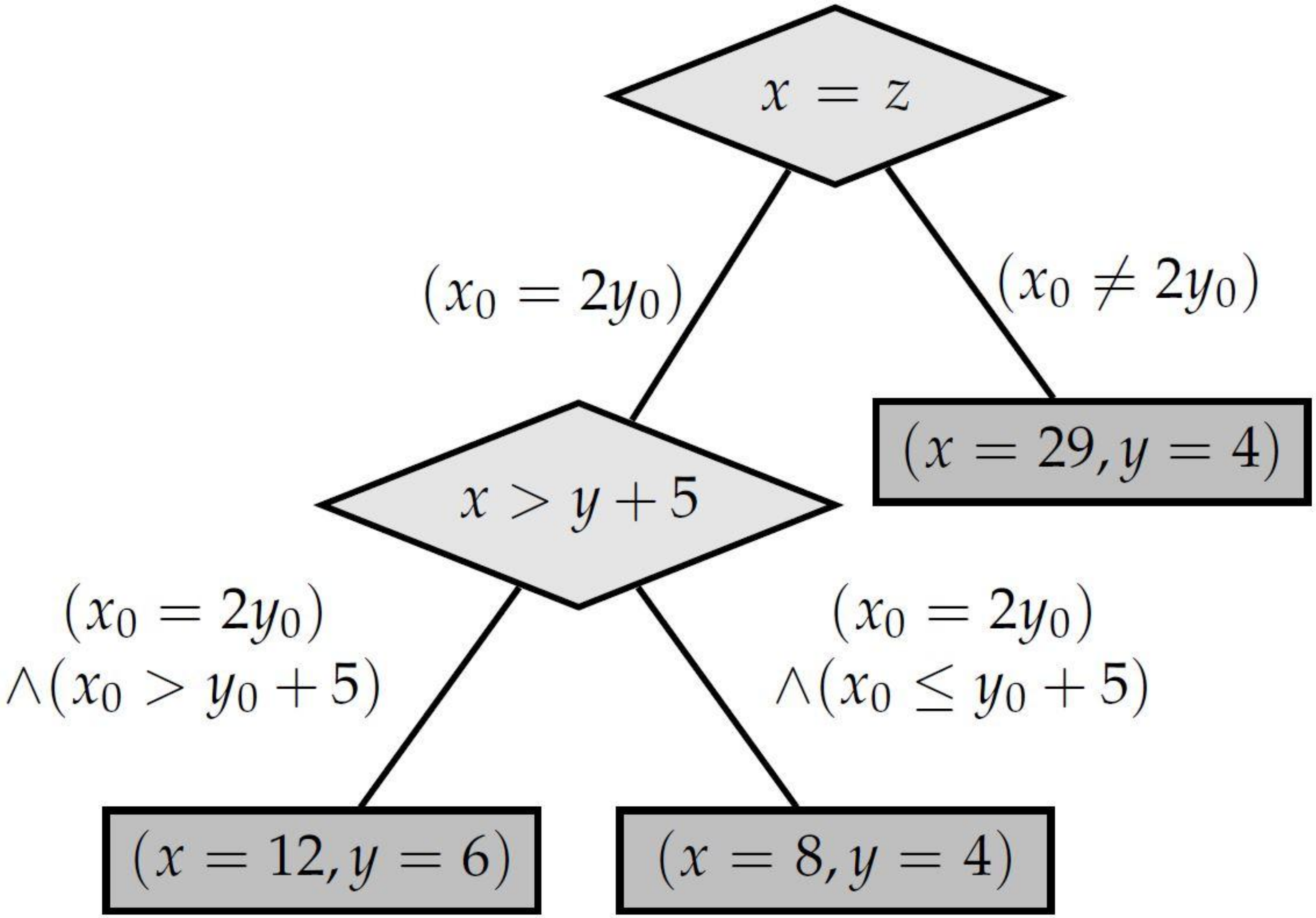
symbolic domain:

σ	PC
$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto 2y_0$	$(x_0 \neq 2y_0)$

$$\neg(x_0 = 2y_0)$$

new input: $\{x = 8, y = 4\}$

Concolic Testing - Example



Concolic Testing - Example

symbolic domain:

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

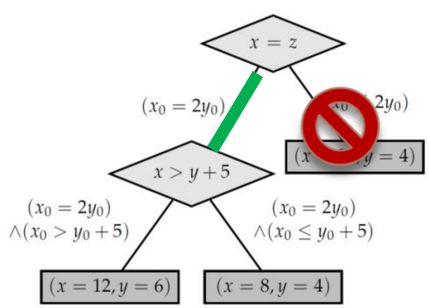
$$(x_0 = bar(y_0))$$

evaluate condition in concrete

$\{x = 8, y = 4\}$

true

σ	PC
$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto bar(y_0)$	$(x_0 = bar(y_0))$



Concolic Testing - Example

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

symbolic state:

$$(x_0 = bar(y_0))$$

evaluate bar() in concrete

$\{x = 8, y = 4\}$

σ	PC
$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto 8$	$(x_0 = 8)$

Symbolic Execution

- 1979
- J.C. King

mix concrete with symbolic execution

Concolic Testing

- sequential path exploration
- guided by concrete input

Execution Generated Testing (EGT)

- fork execution for each path
- guided by 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
    - elsewise the operation is executed symbolical

```
foo3(int x){
   y = 2;
   z = 3*y;
   if (x == z){
      //...
```

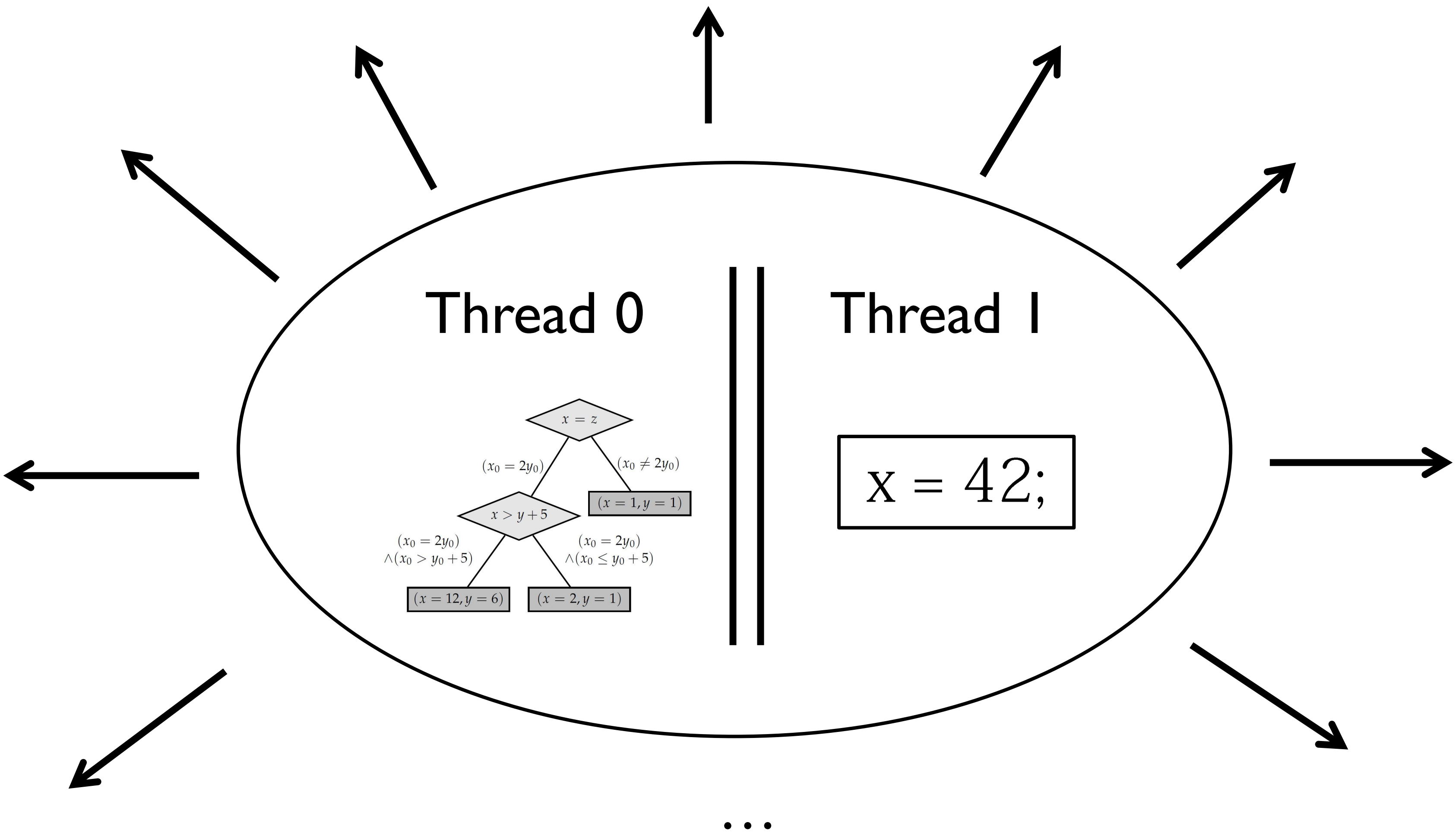
concrete

symbolic

How to deal with concurrent programs?

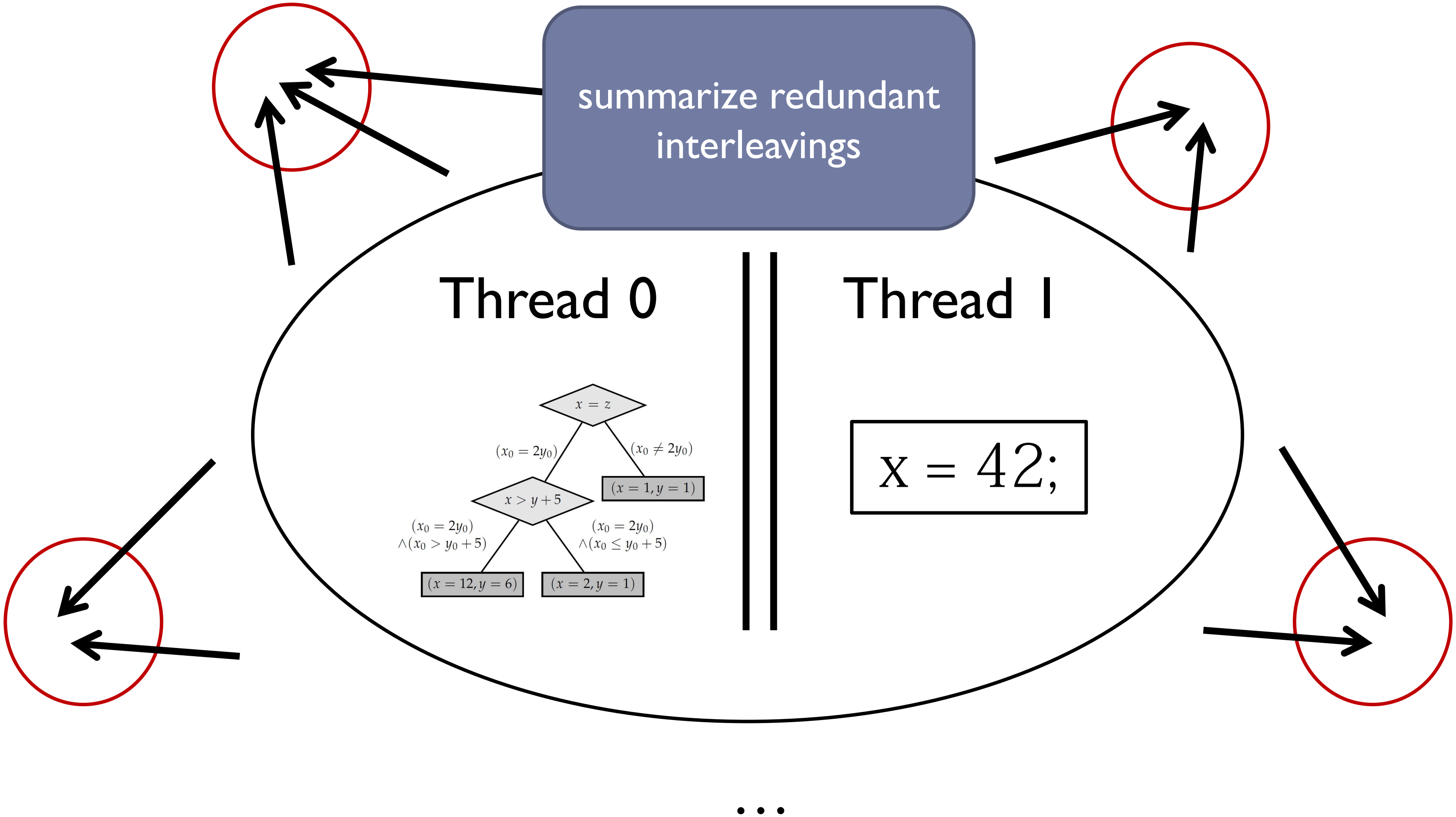
Main Challenge

Problem



Main Challenge

Idea



jCUTE

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 t_0:

```
1:  x = 3;
```

Thread t_1:

```
1:  y = 0;
2:  x = 4;
3:  z = x + 12;
```

Execution 1:

```
x = 3;
y = 0;
x = 4;
z = x + 12;
```

result: {4, 0, 16}

Execution 2:

```
y = 0;
x = 3;
x = 4;
z = x + 12;
```

result: {4, 0, 16}

Execution 3:

```
y = 0;
x = 4;
x = 3;
z = x + 12;
```

result: {3, 0, 15}

Execution 4:

```
y = 0;
x = 4;
z = x + 12;
x = 3;
```

result: {3, 0, 16}

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

	Execution 1:	***Execution 2:***	***Execution 3:***	***Execution 4:***
	$x = 3;$ $y = 0;$ $x = 4;$ $z = x + 12;$	$y = 0;$ $x = 3;$ $x = 4;$ $z = x + 12;$	$y = 0;$ $x = 4;$ $x = 3;$ $z = x + 12;$	$y = 0;$ $x = 4;$ $z = x + 12;$ $x = 3;$
result:	{4, 0, 16}	{4, 0, 16}	{3, 0, 15}	{3, 0, 16}
races:	$(t_0, 1.1) - (t_1, 1.2)$	$(t_0, 1.1) - (t_1, 1.2)$	$(t_1, 1.2) - (t_0, 1.1)$ $(t_0, 1.1) - (t_1, 1.3)$	$(t_1, 1.3) - (t_0, 1.1)$

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 possbile

schedule1:

```
x = 3;
y = 0;
x = 4;
z = x + 12;
```

schedule 2:

```
y = 0;
x = 4;
z = x + 12;
x = 3;
```

How to identify races?

How to identify races?

vector clocks

- $V : \{Threads\} \mapsto \mathbb{N}$
- can be compared *(≤)*
- *max* is componentwise
- $V \neq V'$ if neither „≤“ nor „≥“

- each thread *t* gets it‘s own vector clock V_t
- each memory location gets another two

Vector Clocks - Example

- two threads t_0, t_1
- one memory location x

	V_{t_0}		V_{t_1}		V_x^a		V_x^w	
	t_0	t_1	t_0	t_1	t_0	t_1	t_0	t_1
init_0	0	0	0	0	0	0	0	0
fork_1	1	0	0	1	0	0	0	0
$(t_0, rd, x)_2$	**2**	**0**	0	1	2	0	0	0
$(t_1, rd, x)_3$	2	0	**0**	**2**	2	2	0	0
$(t_1, wr, x)_4$	2	0	**2**	**3**	2	3	2	3
$(t_0, rd, x)_5$	**3**	**3**	2	3	3	3	2	3

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
   $V_t = max\{V_t, V_m^w\}$ and $V_m^a = max\{V_m^a, V_t\}$
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
   $V_{t'} = V_t,\ V_t(t) = V_t(t) + 1$ and $V_{t'} = V_{t'} + 1$

Vector Clocks – Example

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

	V_{t_0}		V_{t_1}		V_x^a		V_x^w	
	t_0	t_1	t_0	t_1	t_0	t_1	t_0	t_1
init_0	0	0	0	0	0	0	0	0
fork_1	1	0	0	1	0	0	0	0
$(t_0, rd, x)_2$	**2**	**0**	0	1	2	0	0	0
$(t_1, rd, x)_3$	2	0	**0**	**2**	2	2	0	0
$(t_1, wr, x)_4$	2	0	**2**	**3**	2	3	2	3
$(t_0, rd, x)_5$	**3**	**3**	2	3	3	3	2	3

Vector Clocks – Example

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

	V_{t_0}		V_{t_1}		V_x^a		V_x^w	
	t_0	t_1	t_0	t_1	t_0	t_1	t_0	t_1
init_0	0	0	0	0	0	0	0	0
fork_1	1	0	0	1	0	0	0	0
$(t_0, rd, x)_2$	**2**	**0**	0	1	2	0	0	0
$(t_1, rd, x)_3$	2	0	**0**	**2**	2	2	0	0
$(t_1, wr, x)_4$	2	0	**2**	**3**	2	3	2	3
$(t_0, rd, x)_5$	**3**	**3**	2	3	3	3	2	3

Vector Clocks – Example

2. If e is a read of a shared memory location m then $V_t = max\{V_t, V_m^w\}$ and $V_m^a = max\{V_m^a, V_t\}$

	V_{t_0}		V_{t_1}		V_x^a		V_x^w	
	t_0	t_1	t_0	t_1	t_0	t_1	t_0	t_1
init_0	0	0	0	0	0	0	0	0
fork_1	1	0	0	1	0	0	0	0
$(t_0, rd, x)_2$	**2**	**0**	0	1	2	0	0	0
$(t_1, rd, x)_3$	2	0	**0**	**2**	2	2	0	0
$(t_1, wr, x)_4$	2	0	**2**	**3**	2	3	2	3
$(t_0, rd, x)_5$	**3**	**3**	2	3	3	3	2	3

Vector Clocks – Example

3. If e is a write, lock or unlock of a shared memory location m then $V_m^w = V_m^a = V_t = max\{V_m^a, V_t\}$

	V_{t_0}		V_{t_1}		V_x^a		V_x^w	
	t_0	t_1	t_0	t_1	t_0	t_1	t_0	t_1
init_0	0	0	0	0	0	0	0	0
fork_1	1	0	0	1	0	0	0	0
$(t_0, rd, x)_2$	**2**	**0**	0	1	2	0	0	0
$(t_1, rd, x)_3$	2	0	**0**	**2**	2	2	0	0
$(t_1, wr, x)_4$	2	0	**2**	**3**	2	3	2	3
$(t_0, rd, x)_5$	**3**	**3**	2	3	3	3	2	3

Vector Clocks – Example

3. If e is a write, lock or unlock of a shared memory location m then
$V_m^w = V_m^a = V_t = max\{V_m^a, V_t\}$

	V_{t_0}		V_{t_1}		V_x^a		V_x^w	
	t_0	t_1	t_0	t_1	t_0	t_1	t_0	t_1
init_0	0	0	0	0	0	0	0	0
fork_1	1	0	0	1	0	0	0	0
$(t_0, rd, x)_2$	**2**	**0**	0	1	2	0	0	0
$(t_1, rd, x)_3$	2	0	**0**	**2**	2	2	0	0
$(t_1, wr, x)_4$	2	0	**2**	**3**	2	3	2	3
$(t_0, rd, x)_5$	**3**	**3**	2	3	3	3	2	3

Vector Clock Theorem

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

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

Questions?

Precise Definitions (just in case)

Race Relation – Simple Defintion:

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

1. $e \not\Updownarrow e'$, and
2. $e <_m e'$ and there exists no e_1 such that $e_1 \neq e, e_1 \neq e', e \preccurlyeq e_1$ and $e_1 \preccurlyeq e'$

- (t_i, l_i, a_i) — (thread, label, type of access)
- $e \not\Updownarrow e'$ — sequentially not related
- $e <_m e'$ — access on the same memory location
- $e \preccurlyeq e_1$ — causally related

sequentially 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 τ are *sequentially related* (denoted by $e \lhd e'$) iff:

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

We say $e \Updownarrow e'$ iff $e \ntriangleleft e'$ and $e' \ntriangleleft e$.

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 τ are *shared-memory access precedence related* (denoted by $e <_m e'$) iff:

1. e appears before e' in τ, 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 $\tau \in Ex(P)$, any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ appearing in τ are *causally related* (denoted by $e \preccurlyeq e'$) iff:

1. $e \lhd e'$, or
2. $e <_m e'$ for some shared-memory location m, or
3. there exists an event e'' in such that $e \preccurlyeq e''$ and $e'' \preccurlyeq e'$

The causal relation is a partial-order relation. We say that $e \parallel e'$ iff $e \not\preccurlyeq e'$ and $e' \not\preccurlyeq e$.

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 $e \lessdot e'$) iff:

1. $e \Updownarrow e'$, 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 $e_1 \neq e'', e_1 \neq e'$, $e'' \preccurlyeq e_1$ and $e_1 \preccurlyeq e'$, and
3. if e is read or write event, then $e <_m e'$ and there exists no e_1 such that $e_1 \neq e$, $e_1 \neq e', e \preccurlyeq e_1$ and $e_1 \preccurlyeq e'$

Race-Detection and Flipping Algorithm Detailled Example

Thread t_0

```
1   x = 3;
```

Thread t_1 (with z as input)

```
1   x = 2;
2   if (x == 2*z+1)
3       error;
4   ...
```

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: $[z = 1,\ sched_2]$
$(t_1, l.1), (t_0, l.1), (t_1, l.2), (t_1, l.3), (t_1, l.4)$