Test-suite Augmentation for Evolving Software

Raul Santelices[†]

Pavan K. Chittimalli[‡]
Alessandro Orso[†]

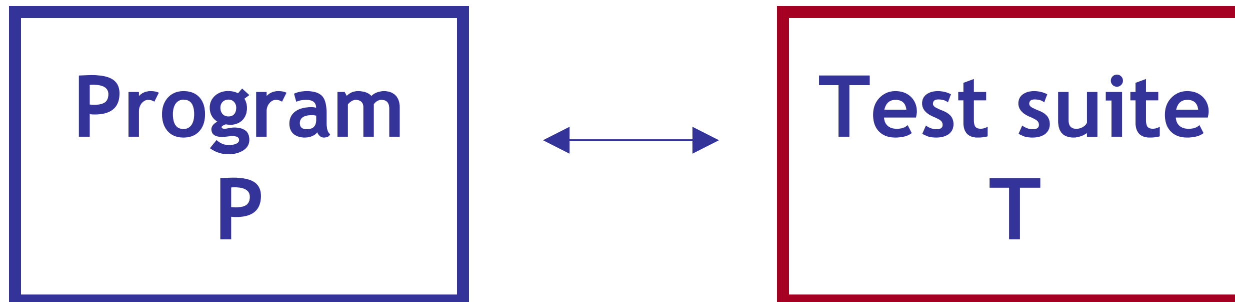
Taweesup Apiwattanapong[#]
Mary Jean Harrold[†]

[†]Georgia Tech [‡]TCS, Ltd. [#]NECTEC

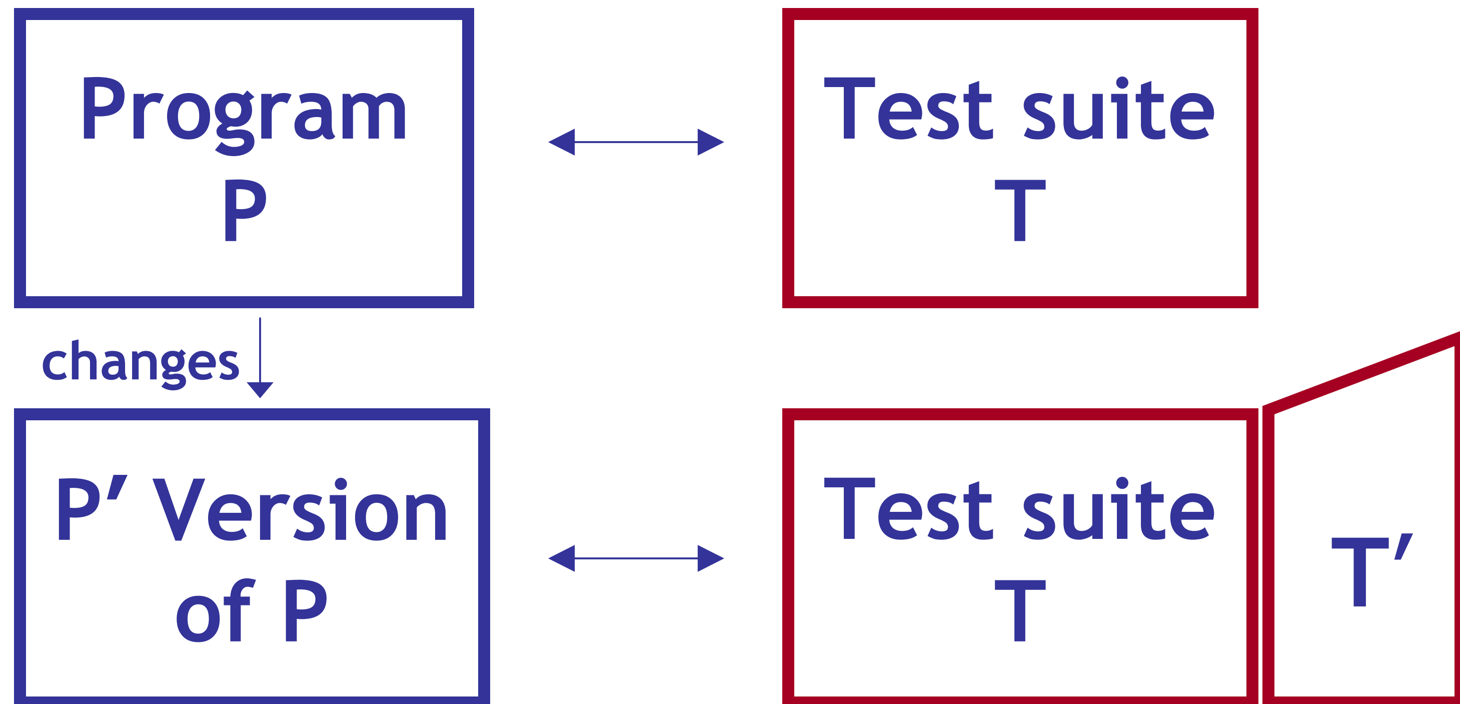
Work supported by TCS Ltd., NSF CCR-0306372, SBE-0123532, and CCF-0725202

Presenter supported by SIGSOFT CAPS

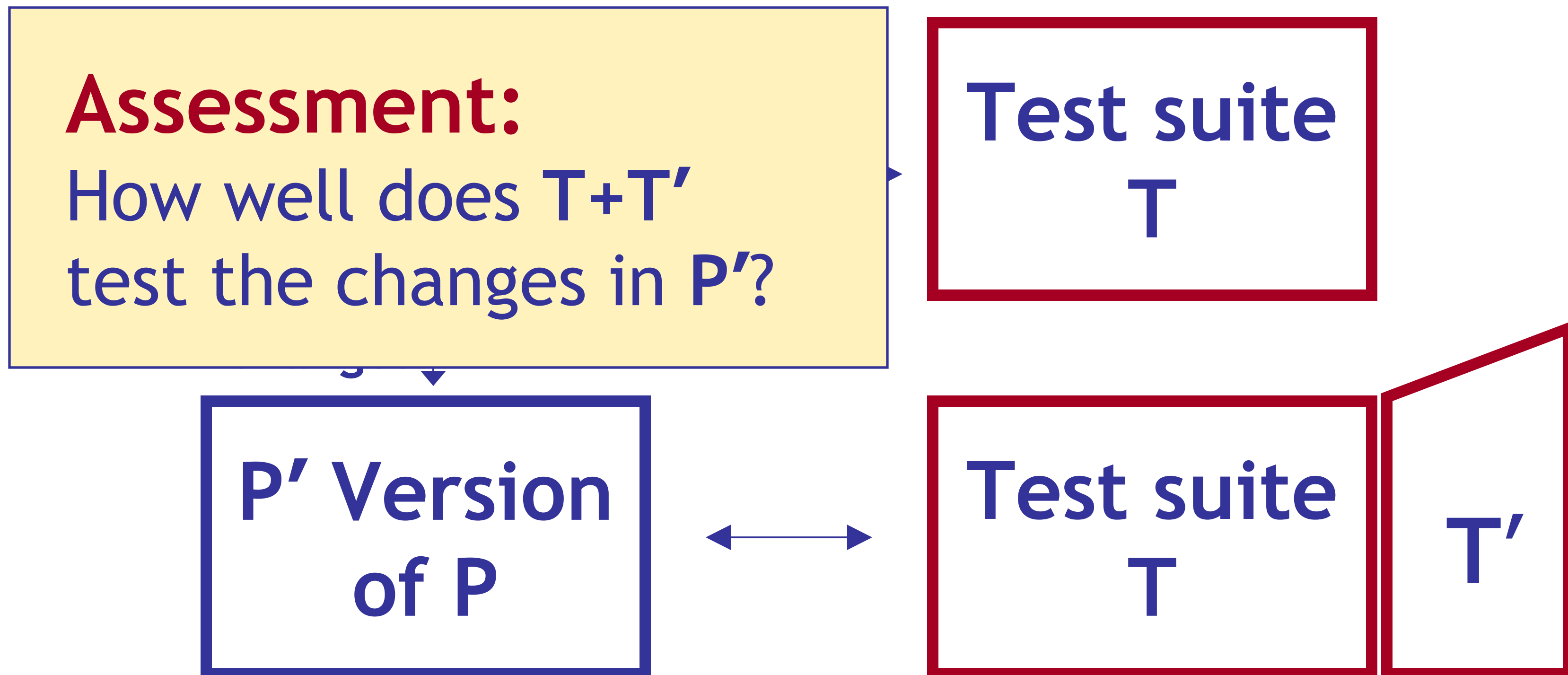
Regression Testing



Regression Testing



Regression Testing



Regression Testing

Assessment:
How well does **T+T′** test the changes in **P′**?

Addition:
What else needs to be tested for those changes?

Test suite T

Test suite T

T′

T″

Regression Testing

Assessment:
How well does **T+T'** test the changes in **P'**?

Addition:
What else needs to be tested for those changes?

Test-suite Augmentation

need to compute **requirements**

Outline

- Background
- Our Technique
- Study
- Future Work and Contributions

Example

100 inputs

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)
2.   y = y + 1
   else
3.   y = y - 1
4. z = abs(x)
5. if (y > 2)
6.   print 1
   else
7.   print 0
8. ... = z
```

Test suite T

test	input		out
	x	y	A
t1	1	10	1
t2	10	1	0

Example

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)    // A': if (x > 2)
2.     y = y + 1
   else
3.     y = y - 1
4. z = abs(x)
5. if (y > 2)
6.     print 1
   else
7.     print 0
8. ... = z
```

Test suite **T**

test	input x	input y	out A	out A′
t1	1	10	1	1
t2	10	1	0	0

no difference

different output A → A′	x ≤ 2	x > 2
y = 2	1 → 0	0 → 1
y = 3	1 → 0	0 → 1

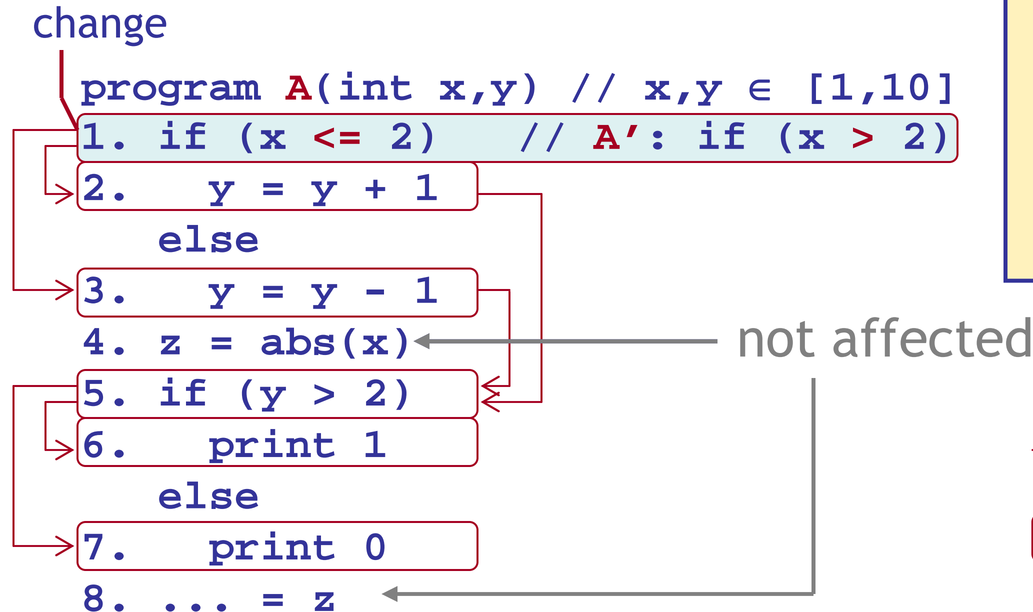
20/100 inputs have ≠ output!

Previous Work on Test-suite Augmentation Criteria

- **Identify** entities **affected** by (dependent on) changes
  - statements
  - control dependencies (branches)
  - data dependencies (du-pairs)
- **Require** testing (coverage) of those entities

[Rothermel & Harrold 94, Binkley 97]

Example: Affected Dependencies



Dependencies are **affected** by a change directly or transitively

affected dependence

affected statement

Dependencies: (1,2), (1,3), (2,5), (3,5), (5,6), (5,7)

direct on 1 — (1,2), (1,3)

transitive on 1 — (2,5), (3,5), (5,6), (5,7)

Example: Augmentation with Affected Dependencies

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)     // A': if (x > 2)
2.     y = y + 1
   else
3.     y = y - 1
4. z = abs(x)
5. if (y > 2)
6.     print 1
   else
7.     print 0
8. ... = z
```

Test suite T

test	input		out A	out A'
	x	y		
t1	1	10	1	1
t2	10	1	0	0

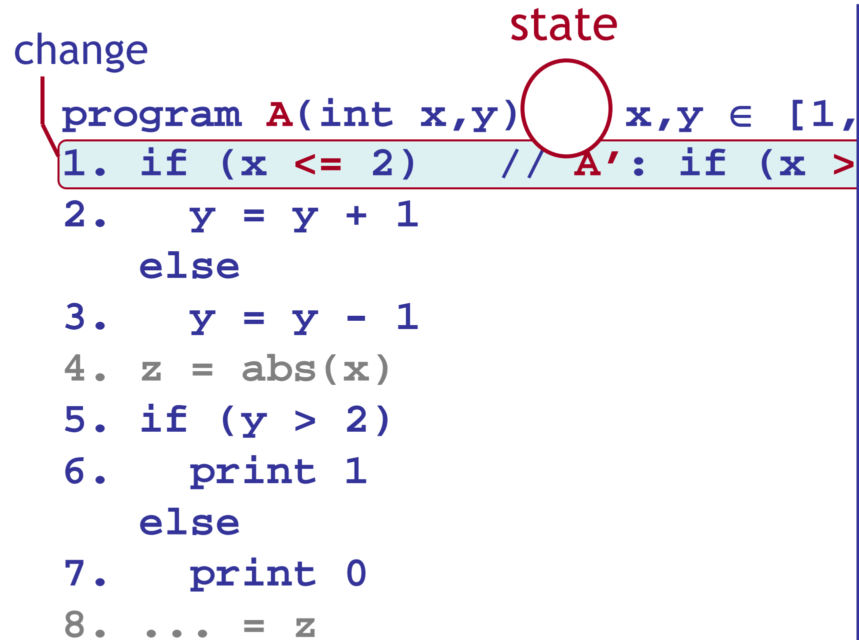
assess — no difference

add <nothing>

Coverage of dependencies on A':

test	(1,2)	(1,3)	(2,5)	(3,5)	(5,6)	(5,7)
t1		●		●		●
t2	●		●		●	

Analyzing Augmentation Criteria



PIE model [Voas 92]:

- Execute change
- Infect state
- Propagate infection to output

Analyzing Augmentation Criteria

E

```
program A(int x,y) // x,y ∈ [1,
1. if (x <= 2)    // A': if (x >
2.    y = y + 1
   else
3.    y = y - 1
4. z = abs(x)
5. if (y > 2)
6.    print 1
   else
7.    print 0
8. ... = z
```

PIE model [Voas 92]:

- Execute change
- Infect state
- Propagate infection to output

Analyzing Augmentation Criteria

E

!

```
program A(int x,y) // x,y ∈ [1,
1. if (x <= 2)    // A': if (x >
2.    y = y + 1
   else
3.    y = y - 1
4. z = abs(x)
5. if (y > 2)
6.    print 1
   else
7.    print 0
8. ... = z
```

PIE model [Voas 92]:

- Execute change
- Infect state
- Propagate infection to output

Analyzing Augmentation Criteria

```
E
   program A(int x,y) // x,y ∈ [1,
   1. if (x <= 2)     // A': if (x >
P  2.    y = y + 1
         else
   3.    y = y - 1
   4. z = abs(x)            P
   5. if (y > 2)
P  6.    print 1
         else
   7.    print 0        I
   8. ... = z
```

output infected ✓

PIE model [Voas 92]:

- Execute change
- Infect state
- Propagate infection to output

Limitations of Previous Work

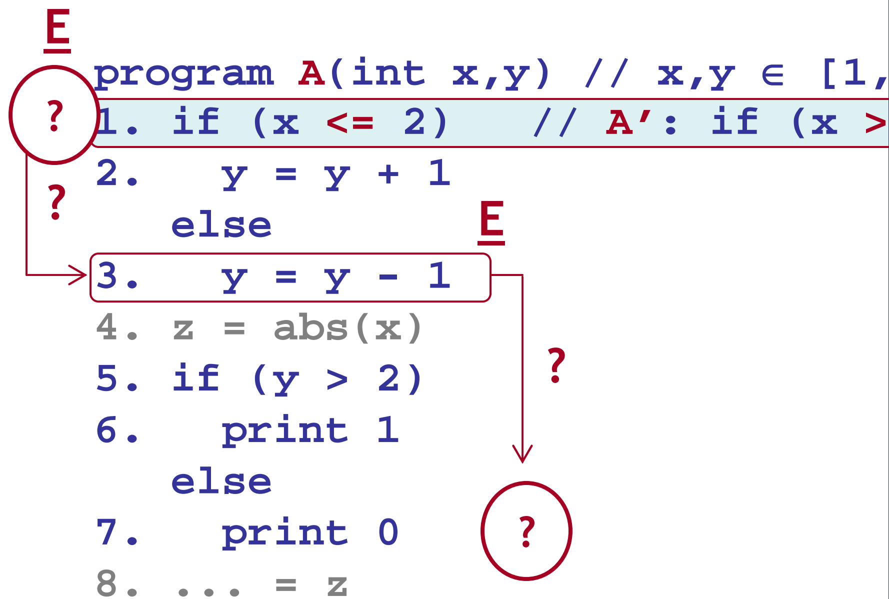
E

```
program A(int x,y) // x,y ∈ [1,
1. if (x <= 2)    // A': if (x >
2.    y = y + 1
   else          E
3.    y = y - 1
4. z = abs(x)
5. if (y > 2)
6.    print 1
   else
7.    print 0
8. ... = z
```

PIE model [Voas 92]:

- Execute change
- Infect state
- Propagate infection to output

Limitations of Previous Work



PIE model [Voas 92]:

- Execute change
- Infect state
- Propagate infection to output

Problems: State might not be infected

Infection might not propagate to output

Our Technique

E

```
program A(int x,y) /
1. if (x <= 2)    //
2.    y = y + 1
   else
3.    y = y - 1
4. z = abs(x)
5. if (y > 2)
6.    print 1
   else
7.    print 0
8. ... = z
```

Goal: requirements for infecting state at change and propagating infection to **output**

Our Technique

```
E
   program A(int x,y) /
   1. if (x <= 2)    //
P  2.    y = y + 1
      else
   3.    y = y - 1
   4. z = abs(x)          P
   5. if (y > 2)
P  6.    print 1
      else
   7.    print 0
   8. ... = z
```

Goal: requirements for infecting state at change and propagating infection to **output**

stage **1** identify *chains* of dependencies (propagation paths)

Example chain: **(1,3),(3,5),(5,7)**

Our Technique

```
E
program A(int x,y) /
P 1. if (x <= 2)    //
  2.    y = y + 1
     else
  3.    y = y - 1
  4. z = abs(x)            P
P 5. if (y > 2)
  6.    print 1
     else
  7.    print 0   !
  8. ... = z
```

infection propagates to end of chain!

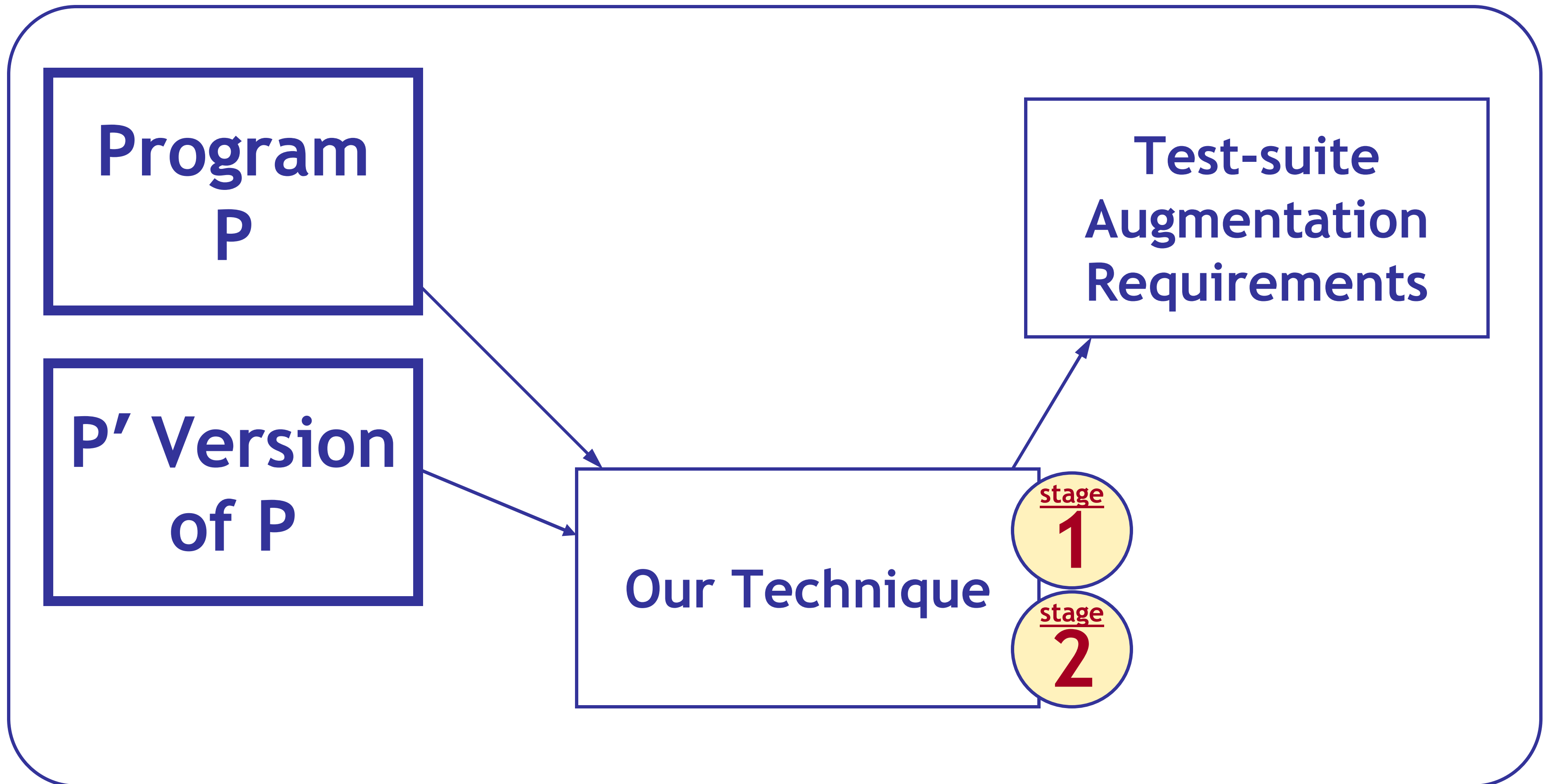
Goal: requirements for infecting state at change and propagating infection to **output**

stage **1** identify *chains* of dependencies (propagation paths)

Example chain: **(1,3),(3,5),(5,7)**

stage **2** compute *infection conditions* on chains (using *partial* symbolic execution)

Our Technique



stage 1

Dependence Chains

Problem

- Chains to output can be too long (or infinite)
- Number of chains might grow exponentially

Solution

Set limit *d* to **length** of chains

Intuition

Covering **all** dependence chains of length *d* **increases** chances of propagation to **output**

stage 1 Coverage of Chains

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)    // A': if (x > 2)
2.     y = y + 1
   else
3.     y = y - 1
4. z = abs(x)
5. if (y > 2)
6.     print 1
   else
7.     print 0
8. ... = z
```

Chain length: 1

chain on A′	x	y	# inputs	≠ output
$ch_1 = (1,2)$	x > 2	[1,10]	80	
$ch_2 = (1,3)$	x ≤ 2	[1,10]	20	

different output A → A′	x ≤ 2	x > 2
y = 2	1 → 0	0 → 1
y = 3	1 → 0	0 → 1

stage 1

Coverage of Chains

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)     // A': if (x > 2)
2.     y = y + 1
   else
3.     y = y - 1
4. z = abs(x)
5. if (y > 2)
6.     print 1
   else
7.     print 0
8. ... = z
```

Chain length: 1

chain on A′	x	y	# inputs	≠ output
ch_1 = (1,2)	x > 2	[1,10]	80	16
ch_2 = (1,3)	x ≤ 2	[1,10]	20	

8×2 = 16

different output A → A′	x ≤ 2	x > 2
y = 2	1 → 0	0 → 1
y = 3	1 → 0	0 → 1

stage 1 Coverage of Chains

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)    // A': if (x > 2)
2.    y = y + 1
   else
3.    y = y - 1
4. z = abs(x)
5. if (y > 2)
6.    print 1
   else
7.    print 0
8. ... = z
```

Chain length: 1

chain on A′	x	y	# inputs	≠ output
$ch_1 = (1,2)$	$x > 2$	[1,10]	80	16
$ch_2 = (1,3)$	$x \leq 2$	[1,10]	20	4

different output A → A′	$x \leq 2$	$x > 2$
$y = 2$	1 → 0	0 → 1
$y = 3$	1 → 0	0 → 1

$2 \times 2 = 4$

stage 1 Coverage of Chains

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)    // A': if (x > 2)
2.    y = y + 1
   else
3.    y = y - 1
4. z = abs(x)
5. if (y > 2)
6.    print 1
   else
7.    print 0
8. ... = z
```

Chain length: 1

chain on A′	x	y	# inputs	≠ output
$ch_1 = (1,2)$	x > 2	[1,10]	80	16
$ch_2 = (1,3)$	x ≤ 2	[1,10]	20	4

Probability of test suite finding a difference =

$p_{diff}(ch_1) + (1 - p_{diff}(ch_1)) \times p_{diff}(ch_2) =$

$16/80 + (1-16/80) \times 4/20 =$ **36%**

stage 1 Coverage of Chains

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)        // A': if (x > 2)
2.     y = y + 1
   else
3.     y = y - 1
4. z = abs(x)
5. if (y > 2)
6.     print 1
   else
7.     print 0
8. ... = z
```

Chain length: 3

chain on A′	x	y	# inputs	≠ output
(1,2),(2,5),(5,6)	x > 2	y > 1	72	16
(1,2),(2,5),(5,7)	x > 2	y ≤ 1	8	0
(1,3),(3,5),(5,6)	x ≤ 2	y > 3	14	0
(1,3),(3,5),(5,7)	x ≤ 2	y ≤ 3	6	4

Probability of test suite finding a difference = **83%** ← much better

different output A → A′	x ≤ 2	x > 2
y = 2	1 → 0	0 → 1
y = 3	1 → 0	0 → 1

stage 2

Infection Conditions

change

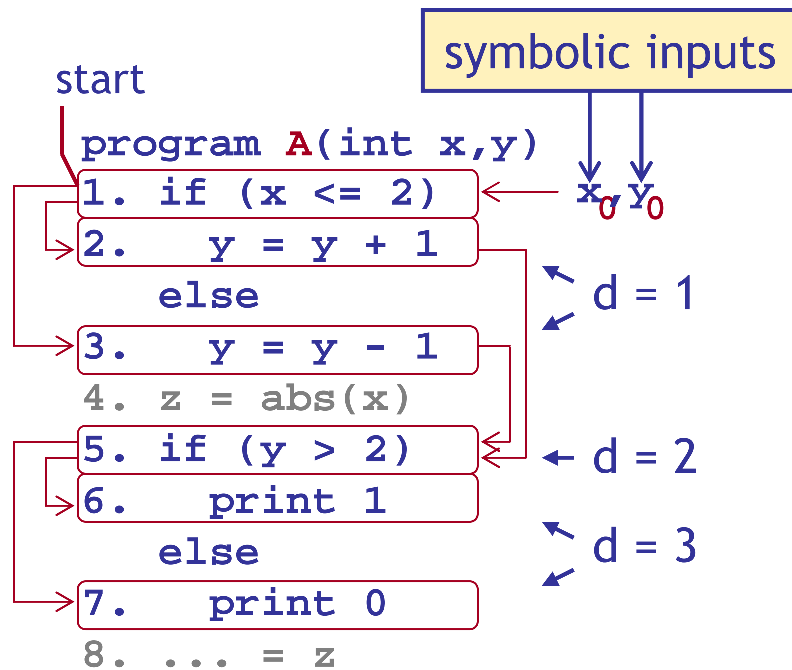
```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)    // A': if (x > 2)
2.     y = y + 1
   else
3.     y = y - 1
4. z = abs(x)
5. if (y > 2)
6.     print 1
   else
7.     print 0
8.
```

Our technique adds conditions to **guarantee** infection of state **and** propagation of infection to **end** of each chain.

Intuition

Propagating infection to end of chain **increases** chances of propagating to the **output**.

stage 2 Partial Symbolic Execution



For program **A**:

always $x = x_0$

statement	pc	state
1	true	$y = y_0$
2	$x_0 \le 2$	$y = y_0$
3	$x_0 > 2$	$y = y_0$
5	$x_0 \le 2$	$y = y_0 + 1$
	$x_0 > 2$	$y = y_0 - 1$
6	$x_0 \le 2 \wedge y_0+1 > 2$	$y = y_0 + 1$
	$x_0 > 2 \wedge y_0-1 > 2$	$y = y_0 - 1$
7	$x_0 \le 2 \wedge y_0+1 \le 2$	$y = y_0 + 1$
	$x_0 > 2 \wedge y_0-1 \le 2$	$y = y_0 - 1$

[Apiwattanapong et al., 2006]

stage 2 Partial Symbolic Execution

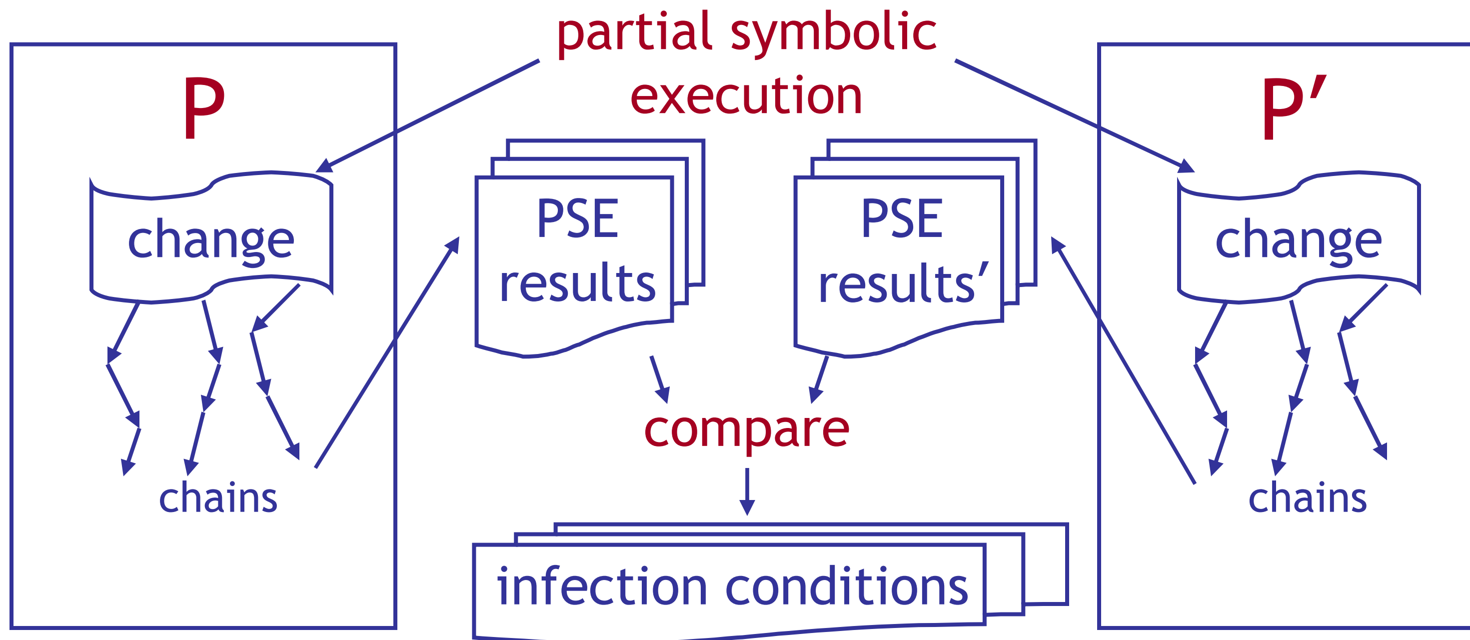
Summary

1. Starts at arbitrary point (change) in program (symbolic inputs are variables at that point)
2. Stops at given distance *d*

Our technique uses partial symbolic execution to compute **infection conditions.**

[Apiwattanapong et al., 2006]

stage 2 Infection Conditions



For each chain in P', compare results on P and P':

1. ***Live* states of P' and P differ at *end* of the chain**
2. **P' covers the chain, but P does not reach its *end***

stage 2 Concrete Example

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)      // A': if (x > 2)
2.     y = y + 1
   else
3.     y = y - 1
4. z = abs(x)
5. if (y > 2)
6.     print 1
   else
7.     print 0
8. ... = z
```

chain *h* = (1,2),(2,5),(5,6)

program	pc	live state
A	$x_0<2 \wedge y_0+1>2$	-
	$x_0 \geq 2 \wedge y_0-1>2$	-
A′	$x_0 \geq 2 \wedge y_0+1>2$	-

conditions for chain (1,2),(2,5),(5,6)

Types of infection for a chain *h* in A′:

1. *Live* states of A′ and A differ at end of *h* → state(A′,*h*) ≠ state(A,*h*)
2. A′ covers *h*, but A does not reach *end(h)* → pc(A′,*h*) ∧ ¬pc(A,end(*h*))

stage 2

Concrete Example

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)      // A′: if (x > 2)
2.    y = y + 1
   else
3.    y = y - 1
4. z = abs(x)
5. if (y > 2)
6.    print 1
   else
7.    print 0
8. ... = z
```

chain *h* = (1,2),(2,5),(5,6)

program	pc	live state
A	$x_0<2 \wedge y_0+1>2$ $x_0 \geq 2 \wedge y_0-1>2$	- -
A′	$x_0 \geq 2 \wedge y_0+1>2$	-

conditions for chain (1,2),(2,5),(5,6)

Types of infection for a chain *h* in A′:

1. *Live* states of A′ and A differ at end of *h* → *false*
2. A′ covers *h*, but A does not reach *end(h)* → pc(A′,*h*) ∧ ¬pc(A,end(*h*))

stage 2 Concrete Example

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)     // A': if (x > 2)
2.     y = y + 1
   else
3.     y = y - 1
4. z = abs(x)
5. if (y > 2)
6.     print 1
   else
7.     print 0
8. ... = z
```

chain h = (1,2),(2,5),(5,6)

program	pc	live state
A	$x_0<2 \wedge y_0+1>2$ $x_0\geq2 \wedge y_0-1>2$	- -
A′	$x_0\geq2 \wedge y_0+1>2$	-

Types of infection for a chain *h* in A′:

~~1. *Live* states of A′ and A differ at end of *h* → *false*~~

2. A′ covers *h*, but A does not reach *end(h)* → $(x_0\geq2 \wedge y_0+1>2) \wedge \neg ((x_0<2 \wedge y_0+1>2) \vee (x_0\geq2 \wedge y_0-1>2))$

conditions for chain (1,2),(2,5),(5,6)

stage 2 Concrete Example

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)     // A': if (x > 2)
2.    y = y + 1
   else
3.    y = y - 1
4. z = abs(x)
5. if (y > 2)
6.    print 1
   else
7.    print 0
8. ... = z
```

chain *h* = (1,2),(2,5),(5,6)

program	pc	live state
A	$x_0<2 \wedge y_0+1>2$ $x_0\geq 2 \wedge y_0-1>2$	- -
A′	$x_0\geq 2 \wedge y_0+1>2$	-

conditions for chain (1,2),(2,5),(5,6)

Types of infection for a chain *h* in A′:

~~1. *Live* states of A′ and A differ at end of *h* → *false*~~

2. A′ covers *h*, but A does not reach *end(h)* → $x_0 \geq 2 \wedge y_0 \in \{2,3\}$

8×2=**16** inputs

stage 1 Chains + stage 2 Infection Conditions

change

```
program A(int x,y) // x,y ∈ [1,10]
1. if (x <= 2)    // A’: if (x > 2)
2.    y = y + 1
   else
3.    y = y - 1
4. z = abs(x)
5. if (y > 2)
6.    print 1
   else
7.    print 0
8. ... = z
```

stage 1 + stage 2

chain	# inputs covering chain	# inputs infecting chain	p_{diff} before → after
(1,2),(2,5),(5,6)	72	16	**16/72 → 16/16**
(1,2),(2,5),(5,7)	8	0	**0 → 0**
(1,3),(3,5),(5,6)	14	0	**0 → 0**
(1,3),(3,5),(5,7)	6	4	**4/6 → 4/4**

Technique **guarantees** that test suite finds a difference in this example!

Work Related to Our Technique

Theory of fault-based testing [Morell 84]

↑ refines

RELAY framework [Richardson & Thompson 88]

↑ complements

PIE dynamic estimation technique [Voas 92]

Study: Goal and Setup

Goal: evaluate effectiveness

Toolset: MATRIX-RELOADED for Java bytecode*

- Successor of MATRIX [Apiwattanapong et al., 2006]

Subjects:

subject	SLOC	tests	changes
Tcas	131	1608	6
NanoXML	~ 4000	216	9

* Based on the Soot Analysis Framework

Experiment Design

Augmentation criteria

Statement: changed statements

Data: chains of data-only dependencies

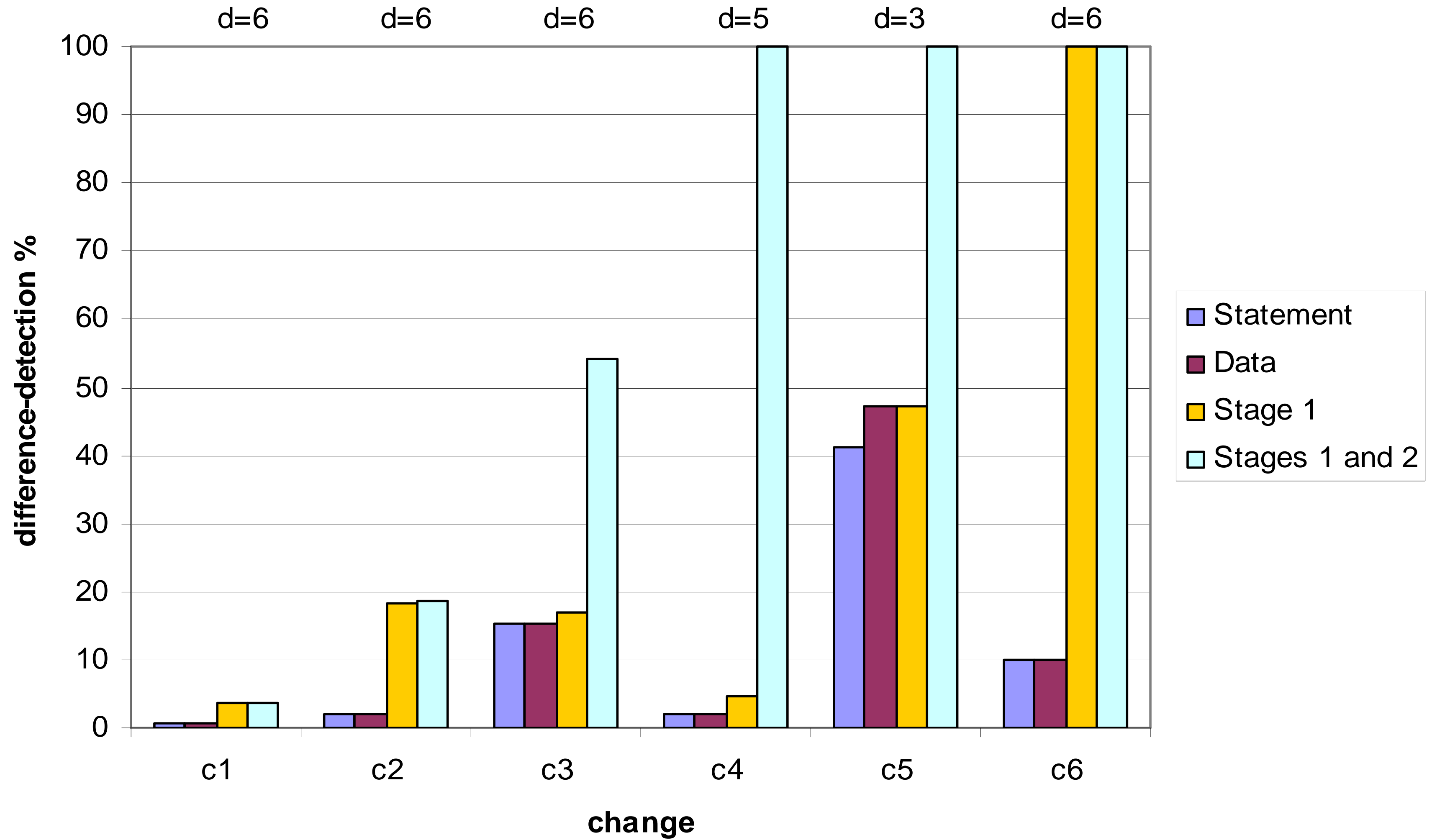
stage 1 **Stage 1**: chains of data/control dependencies

stage 1+2 **Stages 1 and 2**: chains **with** infection conditions

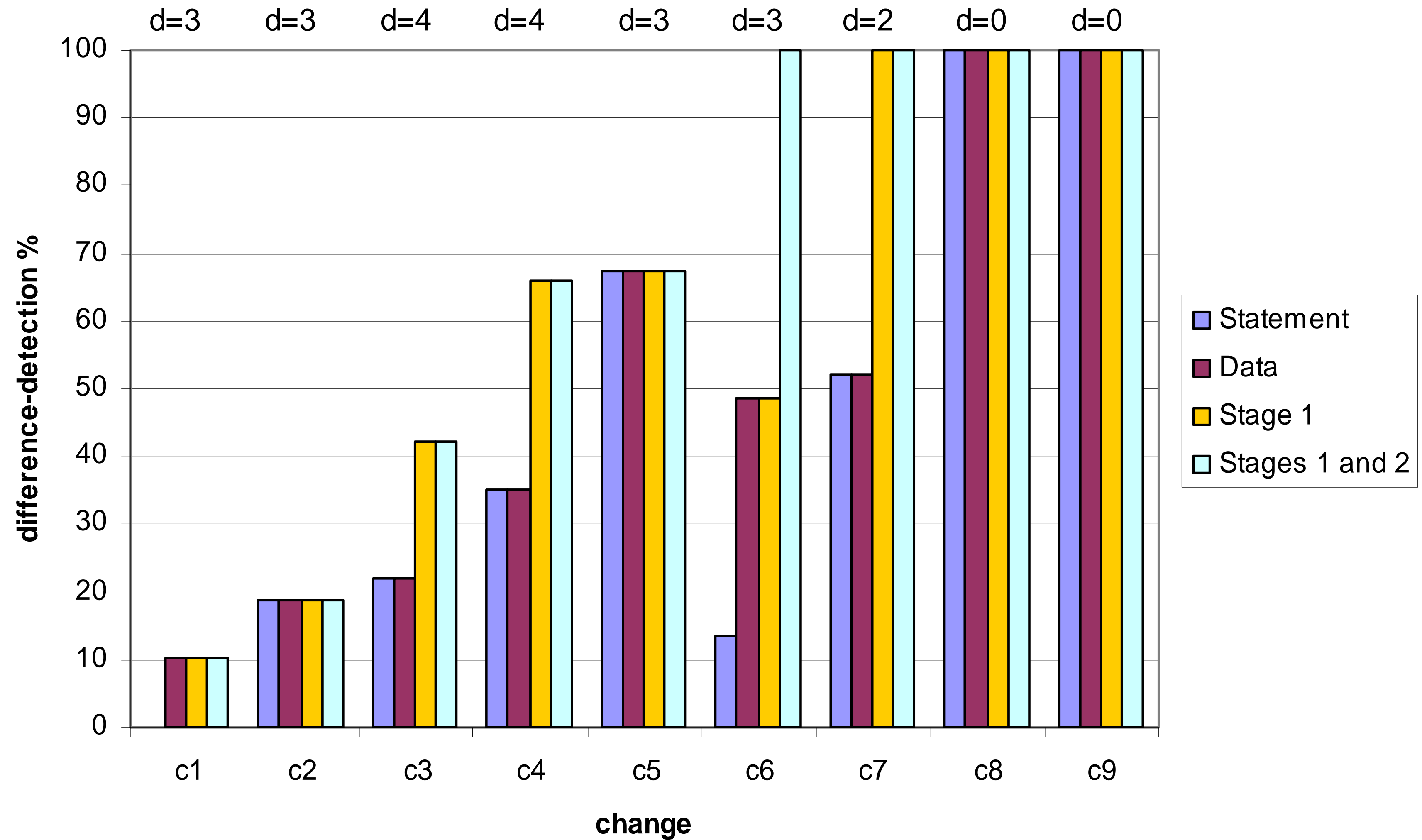
For each criterion

- Generated 100 test suites satisfying criterion
- Computed % of test suites revealing output difference
- Used maximum distance our toolset reached

Results for Tcas



Results for NanoXML



Future Work

1. Improve toolset and perform more experiments
2. Improve chances of propagating infection by finding “infection-killing” points beyond limit
3. Address the problem of large changes
4. Automate generation of test cases for our requirements

Contributions

1. Approach is propagation-based, rather than just coverage-based
2. Technique increases probability of difference detection in two stages

   Stage 1: "propagation paths" (chains)

   Stage 2: infection conditions (partial symbolic execution)
3. Technique is practical

   distance limit reduces **length** and **number** of chains

   partial symbolic execution controls infection conditions