MaTRIX
Maintenance-Oriented Test Requirements Identifier and Examiner

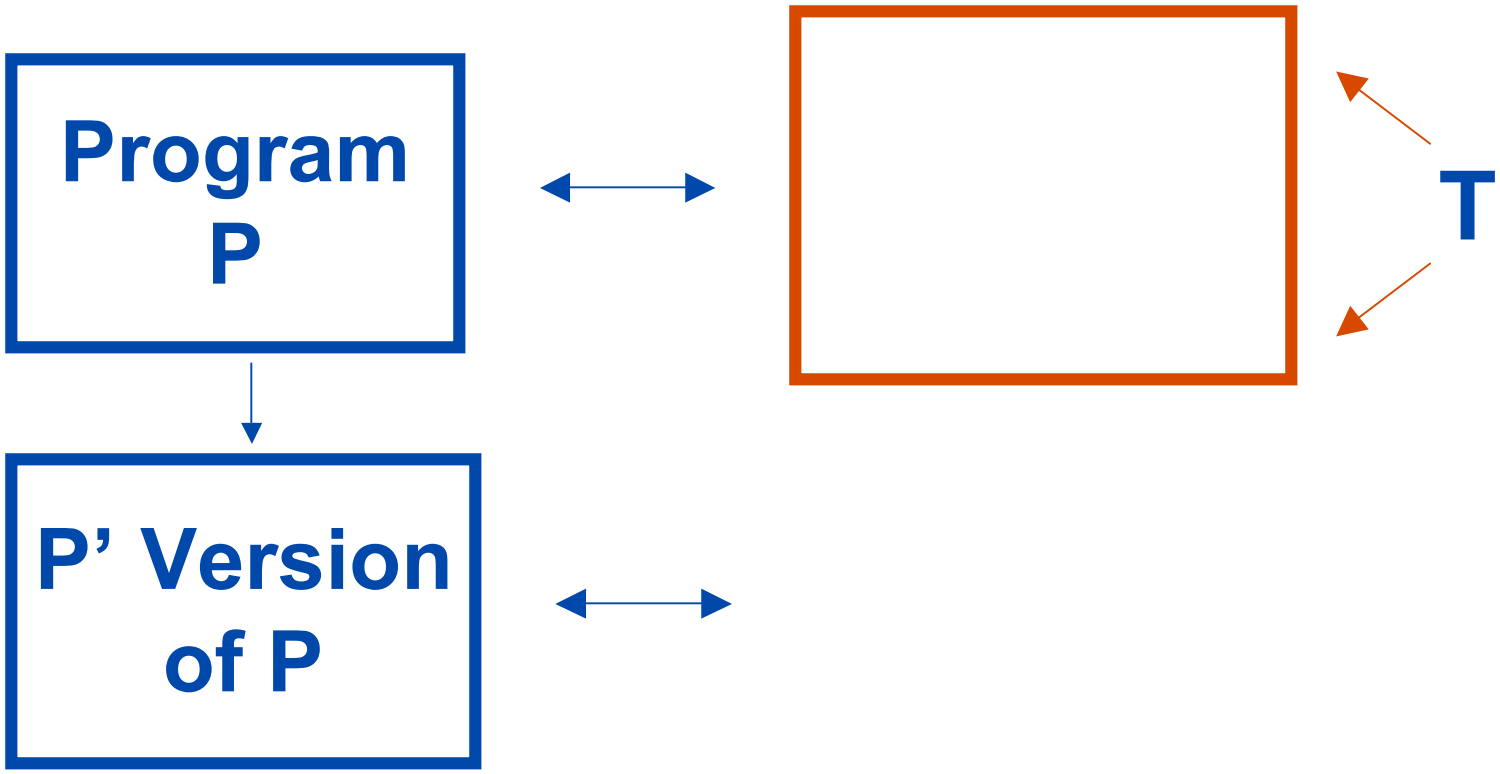
Mary Jean Harrold†

Taweesup (Term) Apiwattanapong,† Raúl Santelices,†
Pavan Kumar Chittimalli,‡ Alessandro Orso†

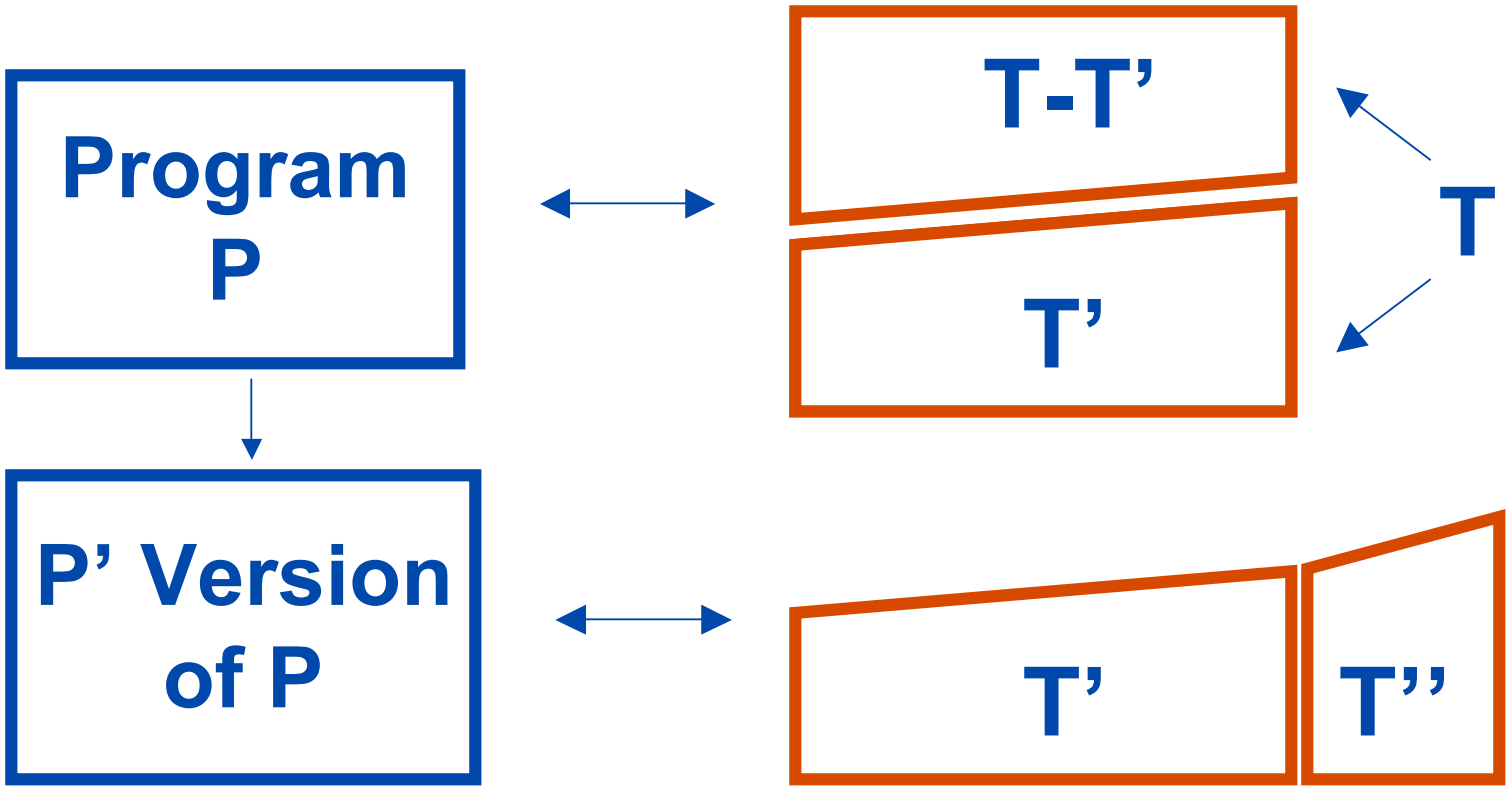
†College of Computing, Georgia Institute of Technology
‡Tata Research Development & Design Centre, TCS Limited

Supported by Tata Consultancy Services (TCS) Limited and by NSF

Regression Testing



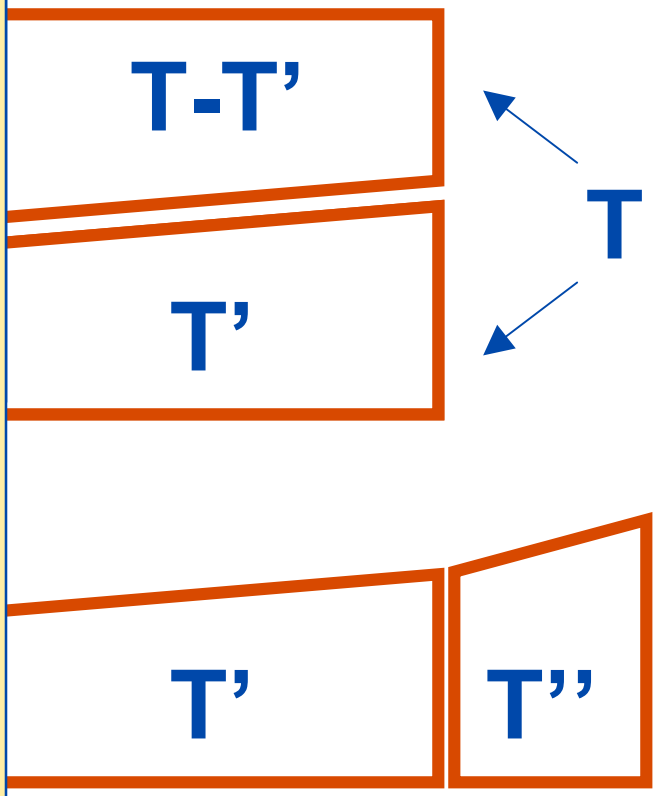
Regression Testing



Regression Testing

How well do T, T',T,'' or any test suites exercise P' with respect to changes?

Is there suitable guidance for creating new test cases that target the modified behavior?

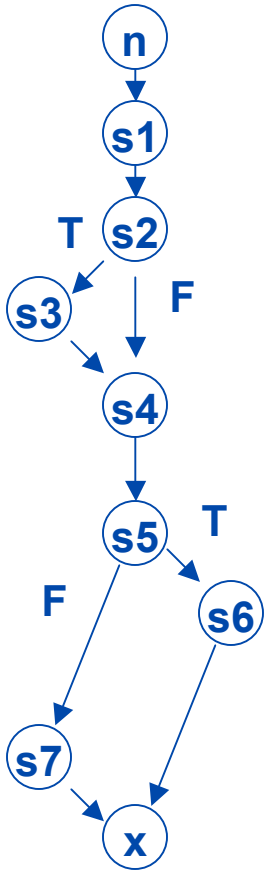


Motivating Example

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x > 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

c: `if (x >= 5){`

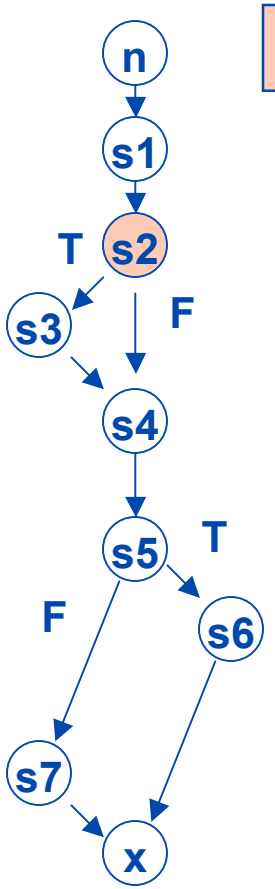


Motivating Example

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x > 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

c: `if (x >= 5){`



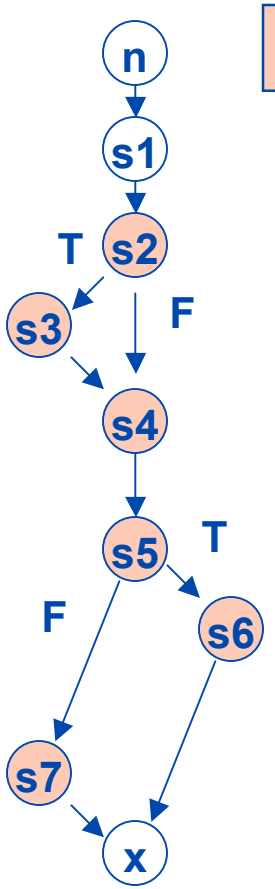
change

Motivating Example

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x > 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

c: `if (x >= 5){`



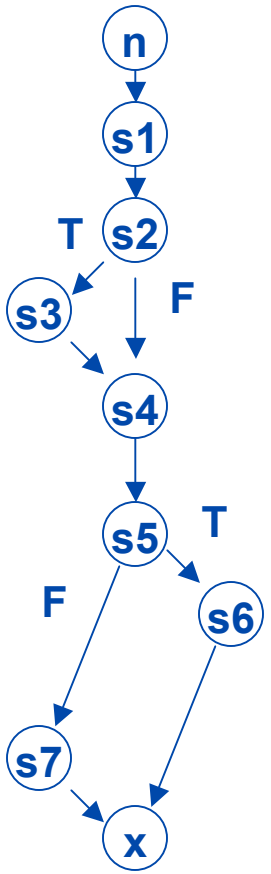
branches

Motivating Example

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x > 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

c: `if (x >= 5){`



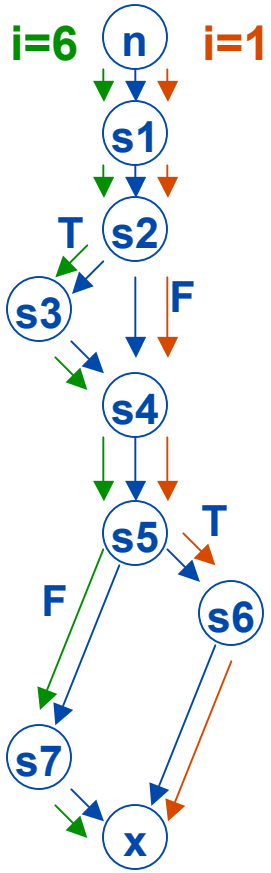
du-pairs

(s1,s2,x)
(s1,s3,x)
(s1,s4,x)
(s3,s4,x)
(s4,s5,x)
(s4,s6,x)
(s4,s7,x)

Motivating Example

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x > 5){   c: if (x >= 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

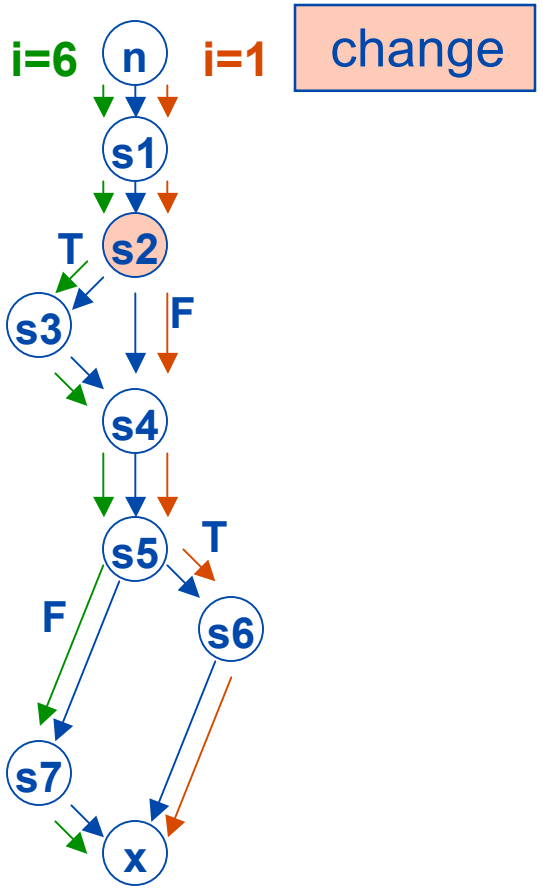


Motivating Example

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x > 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

c: `if (x >= 5){`

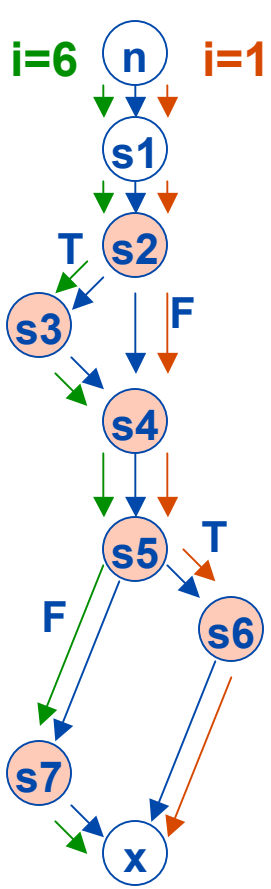


Motivating Example

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x > 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

c: `if (x >= 5){`



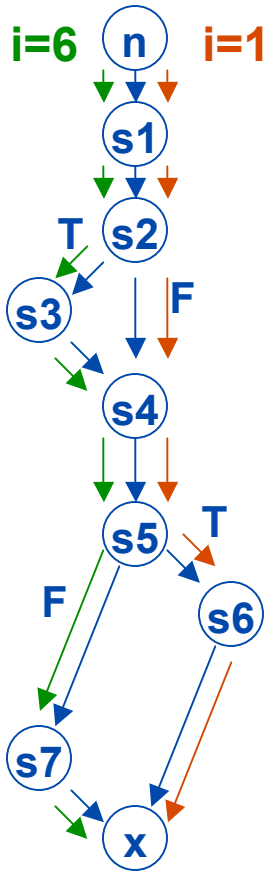
branches

Motivating Example

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x > 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

c: `if (x >= 5){`



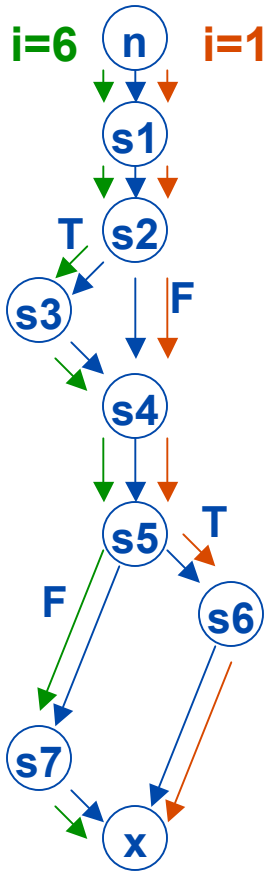
du-pairs	i=6	i=1
(s1,s2,x)	X	X
(s1,s3,x)	X	
(s1,s4,x)		X
(s3,s4,x)	X	
(s4,s5,x)	X	X
(s4,s6,x)		X
(s4,s7,x)	X	

Motivating Example

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x > 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
}
```

C: `if (x >= 5){`



Tests satisfy test requirements for criteria but don't reveal fault in s3

Motivating Example

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x > 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
}
```

c: `if (x >= 5){`

Tests satisfy test requirements for criteria but don't reveal fault in s3

Criteria require

- **Execution of the change and entities affected by change**

But don't require

- **Infection of the state after change**
- **Propagation of state to output where it can be observed**

Computation of Testing Requirements

Our technique adds these requirements to the criteria

Criteria require

- **Execution of the change and entities affected by change**

But don't require

- **Infection of the state after change**
- **Propagation of state to output where it can be observed**

Computation of Testing Requirements

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x >= 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

PC	SS(x)	PC'	SS'(x)
true	i_0	true	i_0
"	"	"	"
$(i_0>5)$	$5/(i_0-5)$	$(i_0>=5)$	$5/(i_0-5)$
$(i_0<=5)$	i_0-1	$(i_0<5)$	i_0-1
or $(i_0>5)$	$5/(i_0-5)-1$	or $(i_0>=5)$	$5/(i_0-5)-1$
"	"	"	"
$(i_0==0)$	0	$(i_0==0)$	0
$(i_0<=5) \wedge (i_0!=0)$	i_0-1	$(i_0<5) \wedge (i_0!=0)$	i_0-1
or $(i_0>5) \wedge (i_0!=0)$	$5/(i_0-5)-1$	or $(i_0>=5) \wedge (i_0!=0)$	$5/(i_0-5)-1$

PC—path condition SS—symbolic state

Computation of Testing Requirements

```
publ
  vo
s1
s2
s3
s4
s5
s6
s7      print(10/x);
    }
  }
  ...
}
```

Conditions for propagation of infected states:

1. **The execution in P' reaches s_i' and the execution in P does not reach s_i; or**
2. **The execution in P' reaches s_i' and the execution in P reaches s_i; however, s_i' and s_i have different symbolic states.**

$(i_0<=5)\wedge(i_0!=0)$	i_0-1	$(i_0<5)\wedge(i_0!=0)$	i_0-1
or		or	
$(i_0>5)\wedge(i_0!=0)$	$5/(i_0-5)-1$	$(i_0>=5)\wedge(i_0!=0)$	$5/(i_0-5)-1$

PC—path condition SS—symbolic state

Computation of Testing Requirements

```
publ
  vo
s1
s2
s3
s4
s5
s6
s7
  }
  ...
}
```

	SS'(x)
	i_0
	"
	$5/(i_0-5)$
	i_0-1
	$5/(i_0-5)-1$
	"
	0
∧	i_0-1
)	
∧	$5/(i_0-5)-1$
$(i_0!=0)$	$(i_0!=0)$

But (as we discussed yesterday)

- **symbolic execution is expensive**
- **won't scale to large programs**
- **can't be applied for entire paths**
- **etc.**

Our technique has two ways to improve efficiency

PC—path condition SS—symbolic state

Computation of Testing Requirements

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x >= 5){
s3    x = (5/(x-5));
    }
}
```

PC	SS(x)	PC'	SS'(x)

1. Perform partial symbolic execution (PSE) beginning immediately before the change
   - computes conditions in terms of variables immediately before change
   - avoids symbolic execution from beginning of program to change

Computation of Testing Requirements

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x >= 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

PC	SS(x)
--	--
true	x_0
$(x_0>5)$	$5/(x_0-5)$

PC'	SS'(x)
--	--
true	x_0
$(x_0>=5)$	$5/(x_0-5)$

Computation of Testing Requirements

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x >= 5){
s3    x = (5/(x-5));
    }
```

PC	SS(x)
--	--
true	x_0
$(x_0>5)$	$5/(x_0-5)$

PC'	SS'(x)
--	--
true	x_0
$(x_0>=5)$	$5/(x_0-5)$

1. **Perform partial symbolic execution (PSE) beginning immediately before the change**
   - **computes conditions in terms of variables immediately before change**
   - **avoids symbolic execution from beginning of program to change**

   Don't need to solve conditions—can still monitor for their satisfaction

Computation of Testing Requirements

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x >= 5){
s3    x = (5/(x-5));
    }
}
```

PC	SS(x)
--	--
true	x_0
$(x_0>5)$	$5/(x_0-5)$

PC'	SS'(x)
--	--
true	x_0
$(x_0>=5)$	$5/(x_0-5)$

2. Perform PSE for some specified distance (user selected) instead of to output statements

- **computes conditions on states at intermediate points (i.e., distances)**
- **bounds depth, avoids symbolic execution to outputs**

Computation of Testing Requirements

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x >= 5){
s3    x = (5/(x-5)
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

PC	SS(x)	PC'	SS'(x)
--	--	--	--
		true	x_0
		>=5)	$5/(x_0-5)$

Distance 0—after change

Distance 1—after 1 dependence

Distance 2—after 2 dependences

Distance 3—after 3 dependences

Distance 3—after 3 dependences

Distance 3—after 3 dependences

And so on until output

Computation of Testing Requirements

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x >= 5){
s3    x = (5/(x-5));
    }
```

PC	SS(x)
--	--
true	x_0
(x_0>5)	5/(x_0-5)

PC'	SS'(x)
--	--
true	x_0
(x_0>=5)	5/(x_0-5)

2. Perform PSE for some specified distance (user selected) instead of to output statements

- **computes conditions on states at intermediate points (i.e., distances)**
- **bounds depth, avoids symbolic execution to outputs**

Greater distances improve confidence in propagation to output

Computation of Testing Requirements

```
public class E {
  void simple (int i) {
s1  int x = i;
s2  if (x >= 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x
s6    prin
    } else
s7    prin
    }
  }
  …
}
```

PC	SS(x)
--	--
true	x_0
$(x_0>5)$	$5/(x_0-5)$

PC'	SS'(x)
--	--
true	x_0
$(x_0>=5)$	$5/(x_0-5)$

Distance 1

PC'(s3) and (not PC(s3))

→ (x_0 >= 5) and (not (x_0 > 5))

→ (x_0 >= 5) and (x_0 <= 5)

→ (x_0 == 5)

Use of Testing Requirements

```
public class E {
  void simple (int i)
s1  int x = i;
s2  if (x >= 5){
s3    x = (5/(x-5));
    }
s4  x = x - 1;

s5  if (x == 0){
s6    print(x);
    } else {
s7    print(10/x);
    }
  }
  …
}
```

1. Instrument program so that probe checks for condition before change (e.g., after s1)
2. Assist developer in satisfying criterion and improving confidence in testing
3. Generate test if condition can be satisfied (future work)

Empirical Study: Setup

Goal:

To compare the effectiveness of our changed-based criteria with statement and all-uses coverage criteria (based on changes)

Implementation: uses differencing, Java Pathfinder, instrumenter, data-/control-dependence analysis, etc.

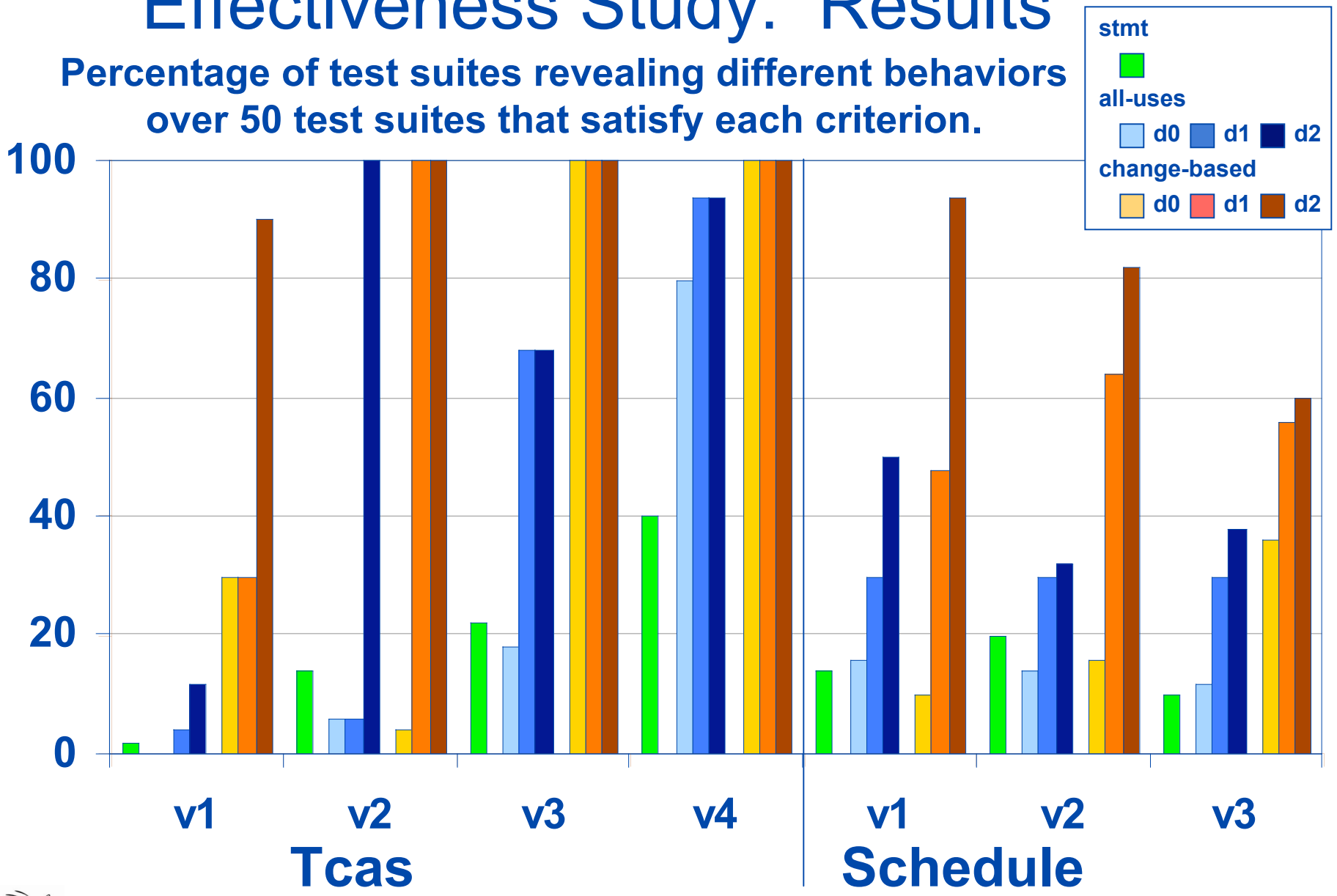
Subjects: Tcas (4 versions) and Schedule (3 versions) (each version has one fault)

Method:

- Randomly generate 50 test suites per criterion.
- Record the number of test suites that produce different outputs.

Effectiveness Study: Results

Percentage of test suites revealing different behaviors over 50 test suites that satisfy each criterion.



Conclusions

New technique

- Identifies (creates), examines (monitors) test requirements related to change(s)
- Uses symbolic execution but gains efficiency
  - partial symbolic execution so avoids performing symbolic execution from beginning of program
  - partial symbolic execution to specified distances from change so bounds depth of symbolic execution
- Size of symbolic execution tree related to change instead of size of program
- Empirical evaluation show promise of approach

Current and Future Work

Current

- Completing infrastructure
- Performing experiments—additional subjects, more complex changes, scalability, limitations

Future

- Expand technique to handle multiple changes, changes involving multiple statements
- Use conditions for automatic test-case generation

Questions?