Microsoft® Research
Faculty Summit 2010

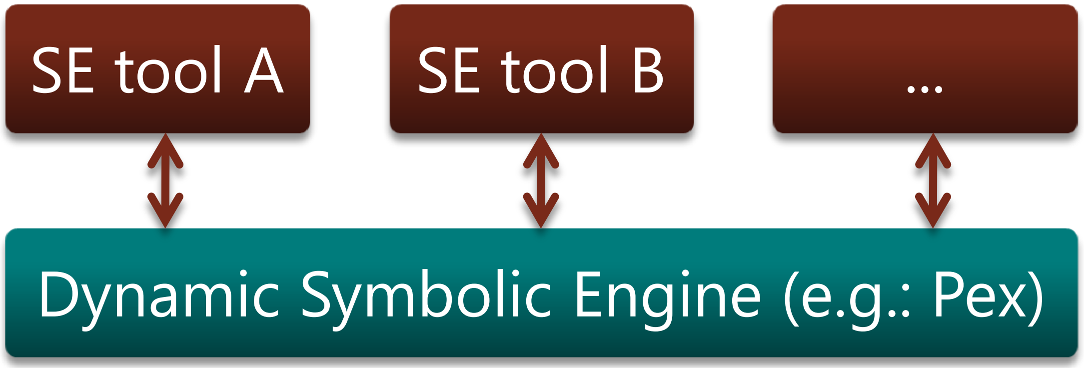
Testing, reverse engineering, data structure repair, etc., via Dynamic Symbolic Execution

Christoph Csallner, University of Texas at Arlington (UTA)

Joint work with:
Nikolai Tillmann (MSR), Yannis Smaragdakis (UMass),
Ishtiaque Hussain (UTA), Chengkai Li (UTA)

Overview

- Dynamic symbolic execution
  - Pioneered by Godefroid et al: Dart [PLDI'05], Cadar et al [SPIN'05]
  - Why it is great
  - What kind of software engineering problems it may be useful for
  - How it works
- Example problems and solutions (tools)
  - Each tool implemented on top of a dynamic symbolic execution engine



Dynamic Symbolic Execution (DSE) is great because it is a:

100% sound program analysis

If DSE says:	program P does X for input I
then:	program P does X for input I

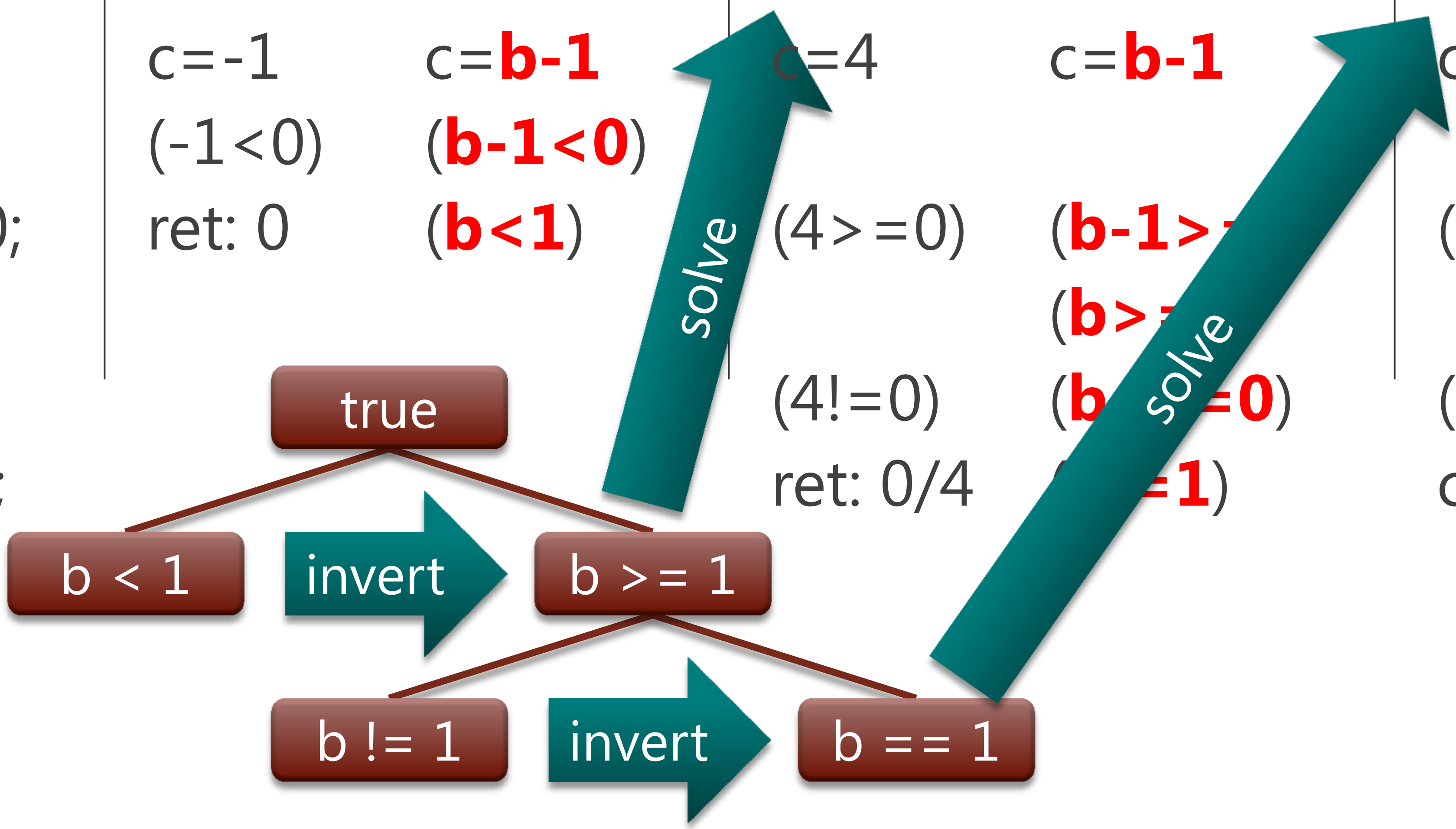
Useful For Reasoning About Individual Execution Paths

- No false warnings
  - Unlike many static analyses
  - False warning: Program P does not do X for I, even though analysis said so
  - Even if program contains “hairy” constructs: reflection, native code, …
- Drawback: 100% sound → <100% complete
  - Cannot analyze program P for all inputs I
  - … But works great for some I
- Useful for reasoning about a subset of all possible execution paths
  - Testing
  - Reverse engineering
  - Repair of data structures at runtime
  - ?

Systematic Exploration of Program Paths

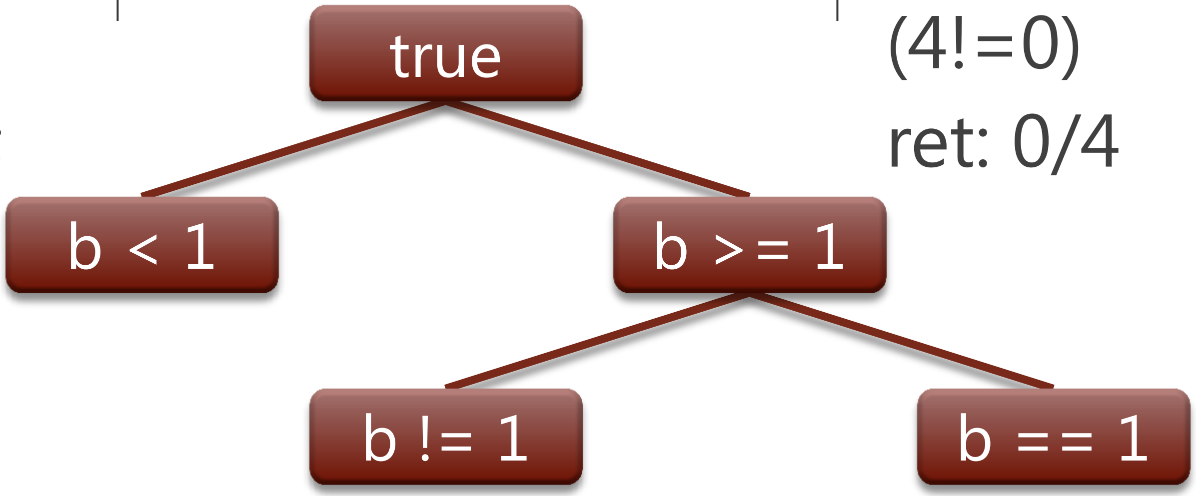
```
int p(int a, int b)
{
    int c = b-1;
    if (c<0)
        return 0;
    if (c==0)
        crash();
    return a / c;
}
```

a=0	a=**a**	a=0	a=**a**	a=0	...
b=0	b=**b**	b=5	b=**b**	b=1	
c=-1	c=**b-1**	c=4	c=**b-1**	c=0	
(-1<0)	(**b-1<0**)				
ret: 0	(**b<1**)	(4>=0)	(**b-1>=**	(0>=0)	
			(**b>=**		
		(4!=0)	(**b** **=0**)	(0==0)	
		ret: 0/4	**=1**)	crash	



Three Test Cases, Each Represents an Execution Path

int p(int a, int b)	a=0	a=0	a=0
{	b=0	b=5	b=1
int c = b-1;	c=-1	c=4	c=0
if (c<0)	(-1<0)		
return 0;	ret: 0	(4>=0)	(0>=0)
if (c==0)			
crash();		(4!=0)	(0==0)
return a / c;		ret: 0/4	crash
}			



Summary: Dynamic symbolic execution (DSE) is great

100% sound program analysis

If DSE says:	program P does X for input I
then:	program P does X for input I

How? **It just executed P, observed X for input I**

Microsoft® Research
Faculty Summit 2010

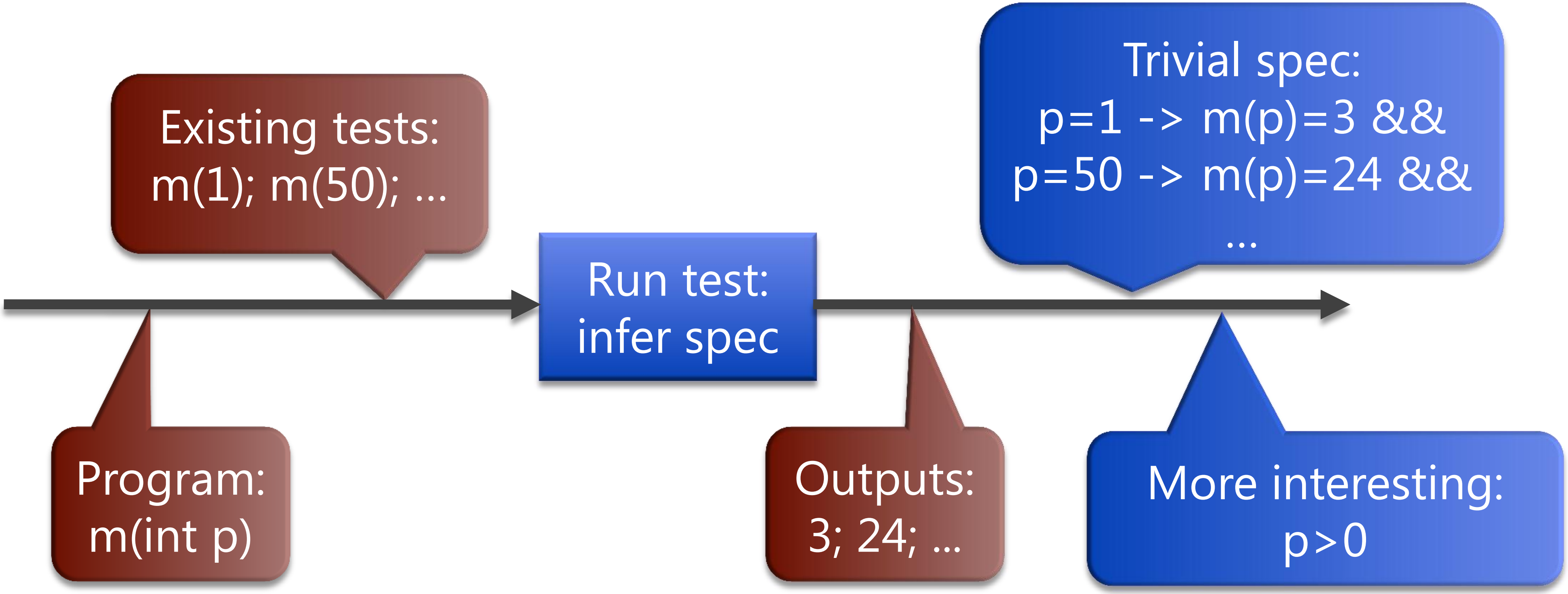
DySy: Dynamic Symbolic Execution for **Dynamic Invariant Inference**

Dynamic Invariant Inference: Reverse Engineering

- Dynamic: Execute a program with a given set of inputs
  - the inputs are assumed to be "representative"
    - e.g., a regression test suite
- Good for program comprehension, further analysis (e.g., test input generation), summaries for interprocedurality

The Key of Dynamic Invariant Inference is Generalization

- Otherwise trivial to be sound/accurate:
  - just report the (finite) observed behaviors:



Best Known Prior Work: Ernst et al: Daikon [ICSE'99]

- A predefined set of invariant templates (around 50)
  - unary, binary, ternary relations over scalars
    - compare var to const: x = a, x > 0
    - linear relationships: y = a*x + b
    - ordering: x <= y
  - relations over arrays
    - sortedness, membership: x in arr
- A gray-box approach
  - other than instantiating template for program vars,
    only observing values at method entry and exit

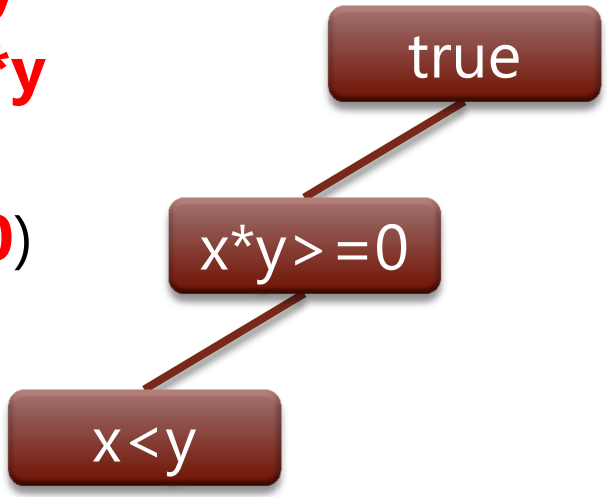
Our Work: DySy: *Dy*-namic, *Sy*-mbolic

- Why not get candidate invariants directly from the program text?
  - e.g., if-conditions, loop conditions
  - but what if these are on intermediate (local) values or after modifying input variables?
- Observation: Conditions maintained by *dynamic symbolic execution* of the program are exactly what we want!
- *Path condition*
  - predicate the inputs must satisfy for an execution to follow a particular path
  - i.e., a *precondition* for observing the current behavior!

Example: Monitor Execution Path (1)

```
int testme(int x, int y)
{
    int prod = x*y;
    if (prod < 0)
        throw new ArgumentException();
    if (x < y)    // swap them
    {
        int tmp = x;
        x = y;
        y = tmp;
    }
    int sqry = y*y;
    return prod*prod - sqry*sqry;
}
```

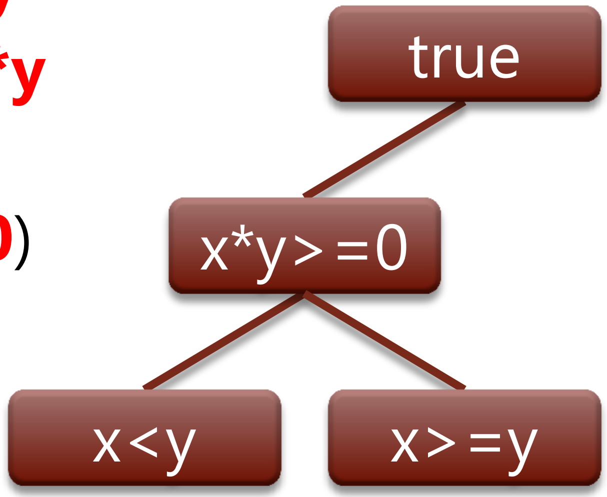
Concrete	**Symbolic**
x=2, y=5	x=**x**, y=**y**
prod=10	prod=**x*y**
(10>=0)	(**x*y>=0**)
(2<5)	(**x<y**)
tmp=2	tmp=**x**
x=5	x=**y**
y=2	y=**x**
sqry=4	sqry=**x*x**
ret: 84	ret: **x*y*x*y - x*x*x*x**



Example: Monitor Execution Path (2)

```
int testme(int x, int y)
{
    int prod = x*y;
    if (prod < 0)
        throw new ArgumentException();
    if (x < y)    // swap them
    {
        int tmp = x;
        x = y;
        y = tmp;
    }
    int sqry = y*y;
    return prod*prod - sqry*sqry;
}
```

Concrete	**Symbolic**
x=5, y=2	x=**x**, y=**y**
prod=10	prod=**x*y**
(10>=0)	(**x*y>=0**)
(5>=2)	(x>=y)
sqry=4	sqry=**y*y**
ret: 84	ret: **x*y*x*y - y*y*y*y**

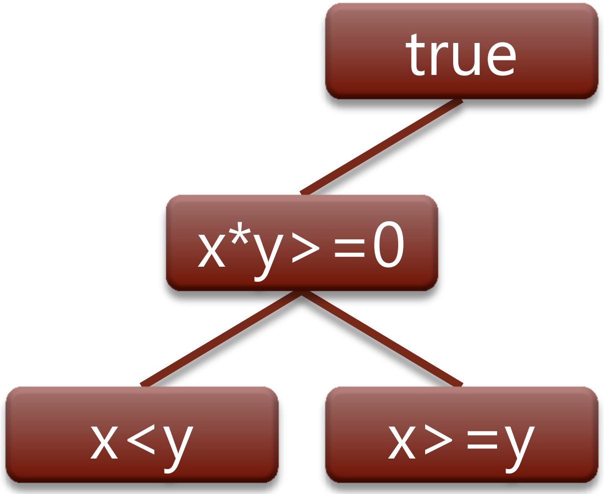


Disjunction of Path Conditions → Precondition

```
int testme(int x, int y)
{
    int prod = x*y;
    if (prod < 0)
        throw new ArgumentException();
    if (x < y)    // swap them
    {
        int tmp = x;
        x = y;
        y = tmp;
    }
    int sqry = y*y;
    return prod*prod - sqry*sqry;
}
```

Precondition:

(x*y >= 0)



Postcondition: return:

(x<y) → (x*y*x*y - x*x*x*x)

else → (x*y*x*y - y*y*y*y)

Case Study: StackAr

- StackAr is a reference micro-benchmark for Daikon
  - Included in the Daikon distribution, discussed in papers
- We hand-inferred an “ideal” set of invariants
- Used the test inputs written by the Daikon authors
- Both DySy and Daikon found almost all reference invariants
  - 27 total, of those: DySy: 20 (25 liberally), Daikon: 19 (27 liberally)
- But Daikon inferred a lot more: many redundant or spurious
  - 89 “ideal” expressions, DySy: 133, Daikon: 316
  - Example:

```
\old(topOfStack) >= 0
==>
(\old(topOfStack) >> StackAr.DEFAULT_CAPACITY) == 0
```

Microsoft® Research
Faculty Summit 2010

Dynamic Symbolic Execution for **Automatic Data Structure Repair**

Automatic Data Structure Repair : Motivation

- Software is built on data structures
- During runtime, data structures may get corrupted by
  - Software bugs, hardware bugs,
  - Particles from space ("soft errors"): http://en.wikipedia.org/wiki/Cosmic_ray#Effect_on_electronics
- Data structure corruption may crash software
- Crash may be fatal, sometimes we do not have the time to
  - Restart system, let alone analyze, debug, fix, re-install
  - Example: Real-time systems
- Instead, we want to repair data structure automatically
  - Bring into a state that again satisfies a given correctness condition
  - Perform repair efficiently: Cannot wait forever!

Approach Hinges on Correctness Condition

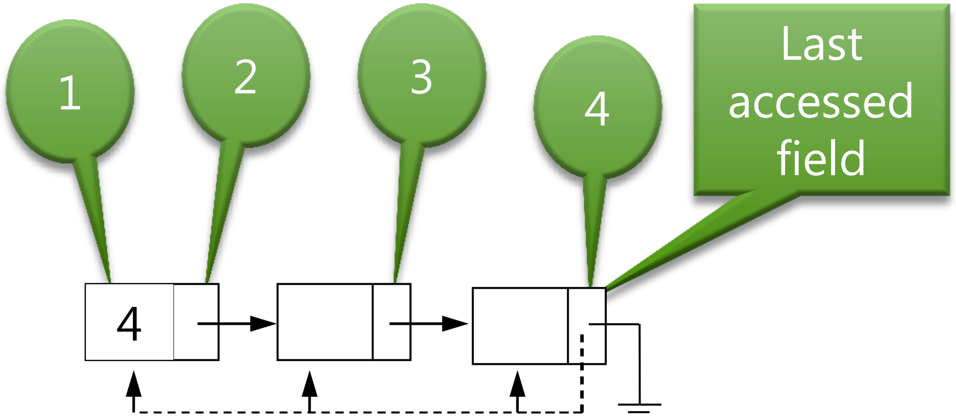
- Assume the correctness condition is correct
  - Bug in correctness condition dooms repair
  - Still better than state of the art that assumes that full program is correct
  - Correctness condition is smaller than full program → easier to understand
- Express correctness condition in same language as program
  - Easier for programmer to reason about correctness condition
  - Example: Java method that checks correctness

Prior Work: Khurshid et al: Juzi [ASE'07, ICSE'08]

```
public class LinkedList {
  Node header;
  // ..
  public boolean repOk() {
    Node n = header;
    if (n == null)
      return true;
    int length = n.value;
    int count  = 1;
    while (n.next != null) {
      count += 1;
      n = n.next;
      if (count > length)
        return false;
    }
    if (count != length)
      return false;

    return true;
  }
}
```

```
public class Node {
  int value;
  Node next;
  // ..
}
```



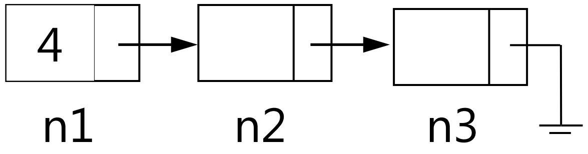
First node has a value that is equal to the number of nodes in the list.

Our Work: Dynamic Symbolic Repair

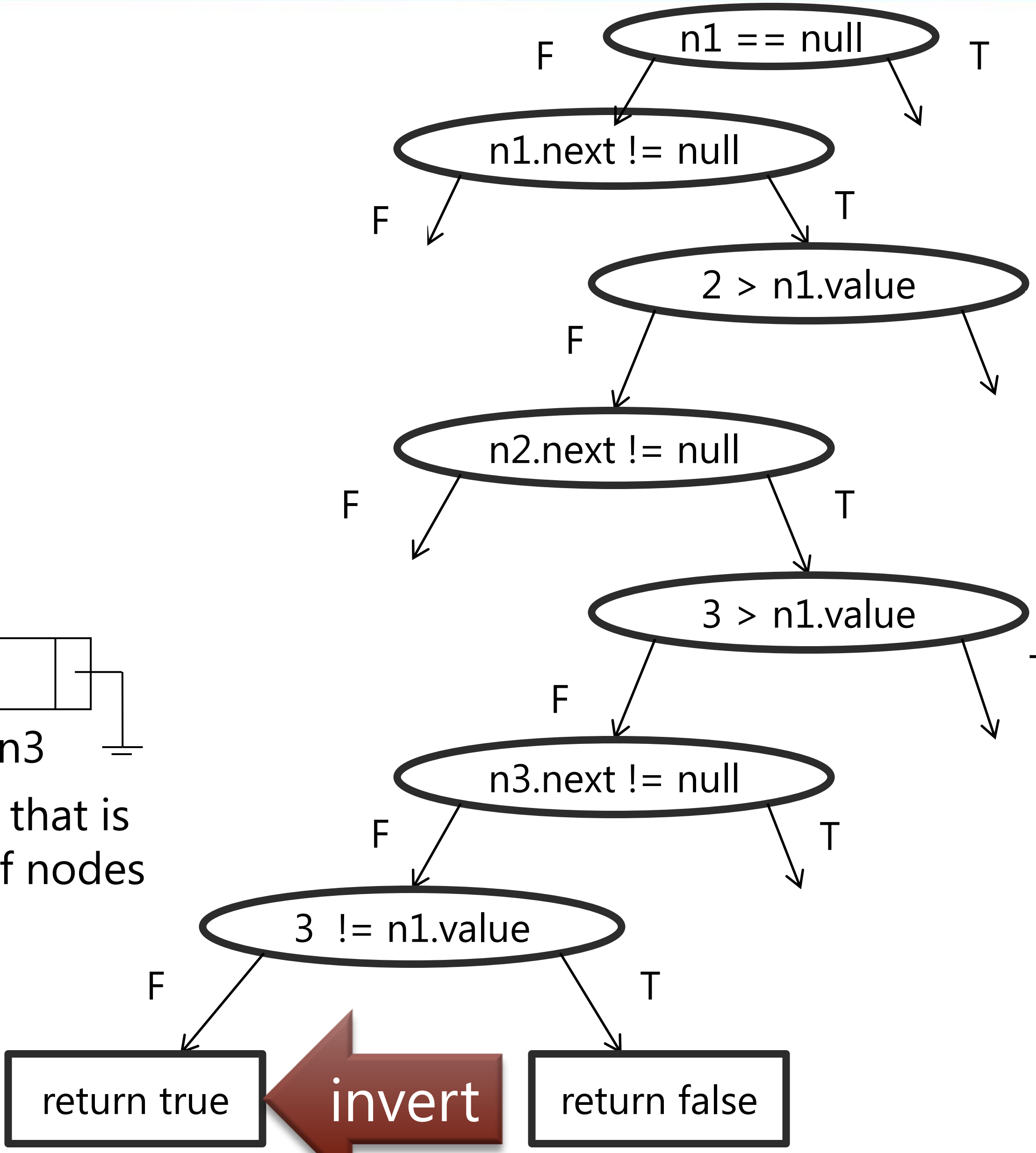
```
public class LinkedList {
  Node header;
  // ..
  public boolean repOk() {
    Node n = header;
    if (n == null)
      return true;
    int length = n.value;
    int count  = 1;
    while (n.next != null) {
      count += 1;
      n = n.next;
      if (count > length)
        return false;
    }
    if (count != length)
      return false;

    return true;
  }
}
```

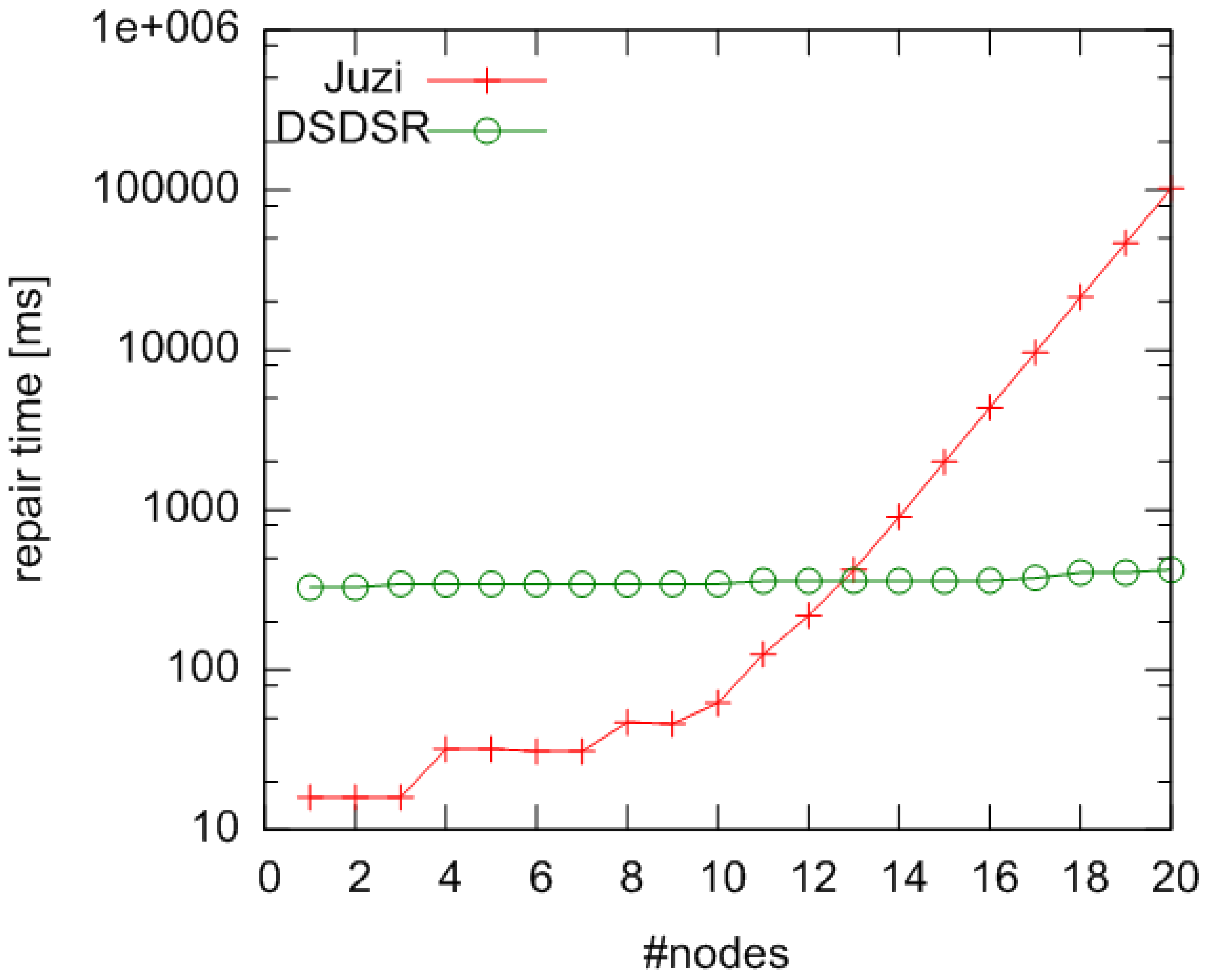
```
public class Node {
  int value;
  Node next;
  // ..
}
```



First node has a value that is equal to the number of nodes in the list.



Preliminary Results: Singly-Linked List



- Lower is better
- Backtracking search in list of field accesses in Juzi leads to exponential behavior
- No such backtracking in Dynamic Symbolic Repair (DSDSR)
- More evaluation needed
  - Larger structures
  - Different subjects

Microsoft® Research
Faculty Summit 2010

Dynamic Symbolic Execution for **Database Application Testing**

Focused Testing of Database-Centric Applications

- Many business applications are coded against existing databases
  - Databases contain valuable business data
  - Databases are large, fairly static, almost append-only
  - Example: Insurance company claims database
- Application expected to work well with the data stored in such an existing database
- Application has huge number of potential execution paths
- **But not all paths are equally interesting**
- Goal: Focus on paths that can be triggered with the existing data
  - Need to make sure application works with the existing data

Goal: Cover Paths That Are Reachable With Existing Data

```
public void dbfoo(String q)
{
   String query = "Select * From r Where "+q;
   Tuple[] tuples = db.execute(query);
   for (Tuple t: tuples) {
      int x = t.getValue(1);
      bar(x);
   }
}
```


```
public void bar(int x)
{
   int z = -x;
   if (z > 0) {        // c1
      if (z < 100)   // c2
   // ..
}
```

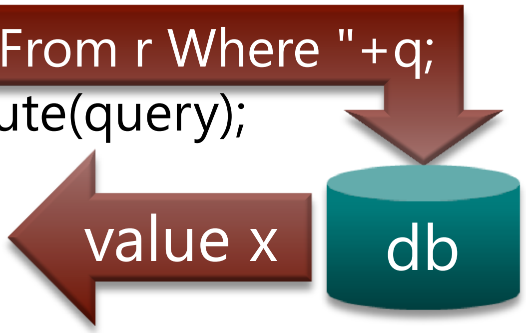
- Application issues database queries
  - Constrained by user input
  - Example: Select a particular customer
  - Input: User-supplied query
- Query results may be used by program logic (= branch conditions)
- Different values from database may trigger different paths
- Different queries may result in different execution paths

Prior Work: Generate Mock Databases

- Generating mock databases
  - Generate database contents to trigger additional execution paths
- But are the generated mock databases representative of real database?
  - Real database may contain subtle data patterns
- Hard problem

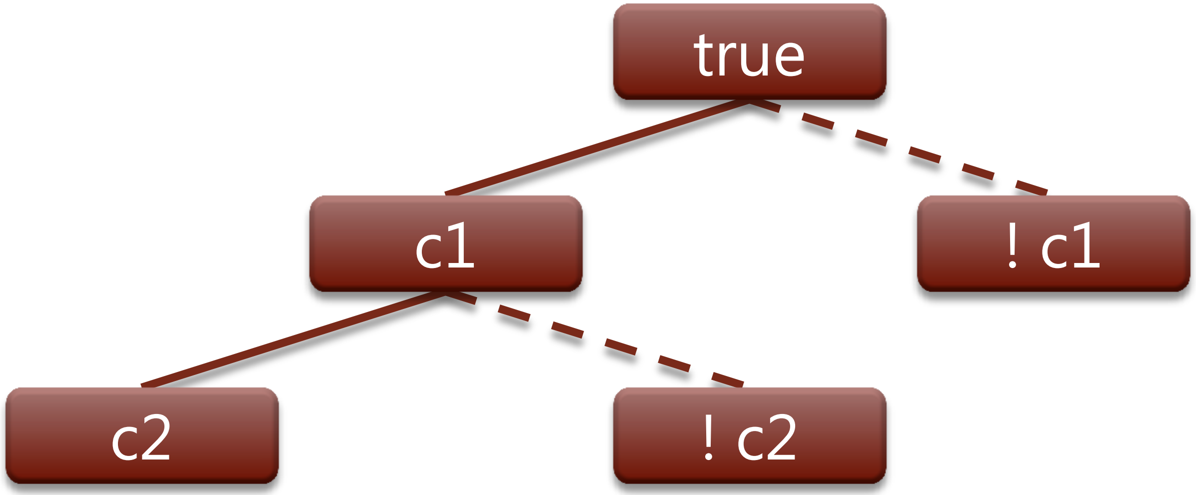
Our Work: Collect Path-Conditions + Use as DB-Query

```
public void dbfoo(String q)
{
  String query = "Select * From r Where "+q;
  Tuple[] tuples = db.execute(query);
  for (Tuple t: tuples) {
    int x = t.getValue(1);
    bar(x);
  }
}
```




```
public void bar(int x)
{
  int z = -x;
  if (z > 0) {        // c1
    if (z < 100)  // c2
  // ..
}
```

- Map each candidate execution path to a database query
- Get multiple candidate queries:
- Query 1 = c1 && !c2
- Query 2 = !c1

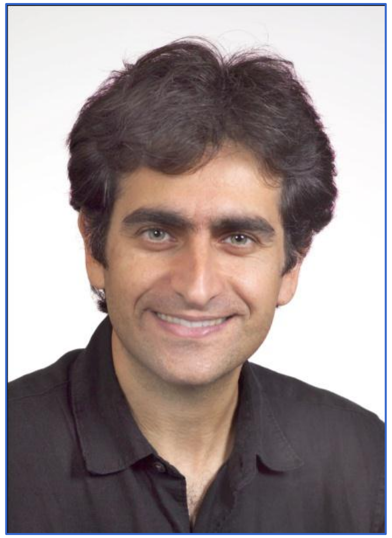


Microsoft® Research
Faculty Summit 2010

Credits and References

Credits and References

"DySy: Dynamic symbolic execution for invariant inference" by **Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis**. In Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE), May 2008, pp. 281-290.



Credits and References

"Dynamic symbolic data structure repair" by
Ishtiaque Hussain and Christoph Csallner. In Proc. 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Volume 2, Emerging Results Track, May 2010, pp. 215-218.

Credits and References

“Dynamic symbolic database application testing” by **Chengkai Li and Christoph Csallner**.
In 3rd International Workshop on Testing Database Systems (DBTest), June 2010.

Microsoft® Research
Faculty Summit 2010

Microsoft® Research

Faculty Summit 2010