Automated Software Testing with Inferred Program Properties

Tao Xie
Dept. of Computer Science & Engineering
University of Washington

Joint work with David Notkin

July 2004

Motivation

- Existing automated test generation tools are powerful
  - produce a **large** number of test inputs

- This **large** number of test inputs
  - contain high percentage of redundant tests
  - are impractical to eyeball-inspect for correctness (when lack of specifications)

Solutions – Inferred Properties

- High percentage of redundant tests
  - Equivalence properties [ASE 04] [1]
    - redundancy detection and avoidance
- Impractical to eyeball-inspect for correctness
  - Operational abstractions [ASE 03]
    - test selection
  - Statistical algebraic abstractions
    - test selection
  - Observer abstractions [ICFEM 04]
    - test abstraction

[1]. Joint work also with Darko Marinov

Solutions – Inferred Properties

- High percentage of redundant tests
  - Equivalence properties [ASE 04] [1]
    - redundancy detection and avoidance
- Impractical to eyeball-inspect for correctness
  - Operational abstractions [ASE 03]
    - test selection
  - Statistical algebraic abstractions
    - test selection
  - Observer abstractions [ICFEM 04]
    - test abstraction

[1]. Joint work also with Darko Marinov

Testing with Equivalence Properties

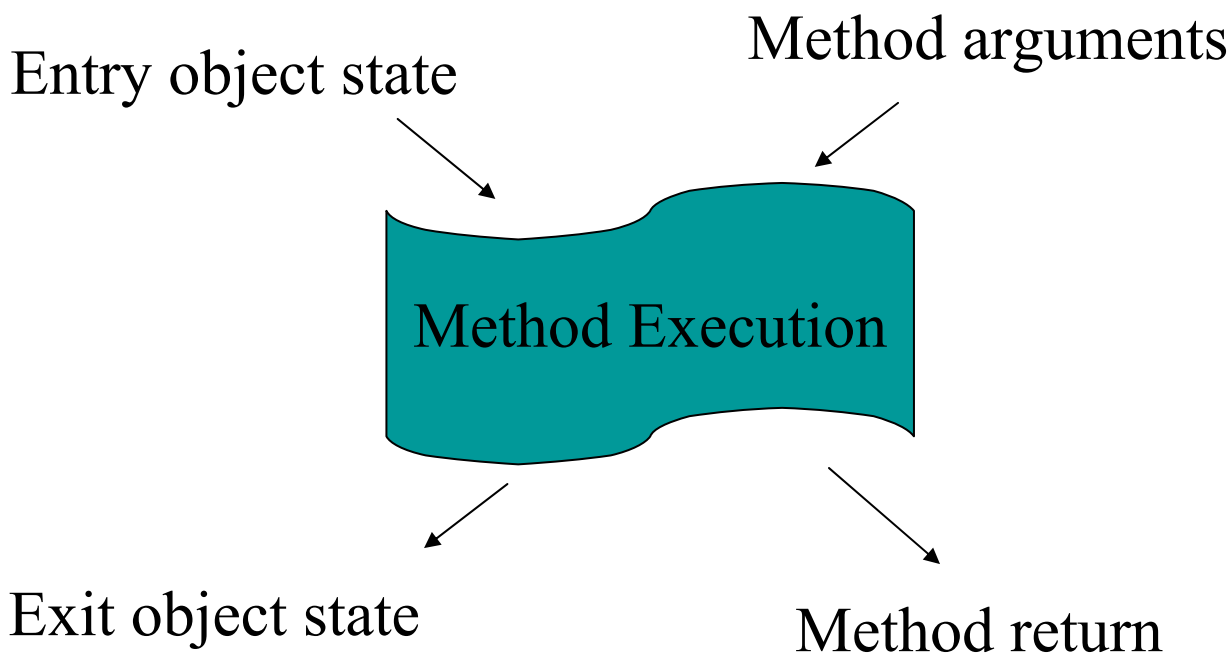
- Problem: High redundancy among automatically generated tests
  - consider two tests are nonequivalent if they have different test source code
- Solution: Detect and avoid redundant tests by inferring equivalence properties
  - among object states, method executions, tests

Class Example - IntStack

```
public class IntStack {
  private int[] store;
  private int size;
  private static final int INITIAL_CAPACITY = 10;

  public IntStack() {
    this.store = new int[INITIAL_CAPACITY];
    this.size = 0;
  }

  public void push(int value) { …  }
  public int pop() { …   }
  public boolean isEmpty() { …  }

  public boolean equals(Object other) {
    if (other == null) return false;
    if (!(other instanceof IntStack)) return false;
    IntStack s = (IntStack)other;
    if (this.size != s.size) return false;
    for (int i = 0; i < this.size; i++)
      if (this.store[i] != s.store[i]) return false;
    return true;
  }
}
```

Test Examples

```
Test 1 (T1):
  IntStack s1 = new IntStack();
  s1.isEmpty();
  s1.push(3);
  s1.push(2);
  s1.pop();
  s1.push(5);

Test 2 (T2):
  IntStack s2 = new IntStack();
  s2.push(3);
  s2.push(5);

Test 3 (T3):
  IntStack s3 = new IntStack();
  s3.push(3);
  s3.push(2);
  s3.pop();
```



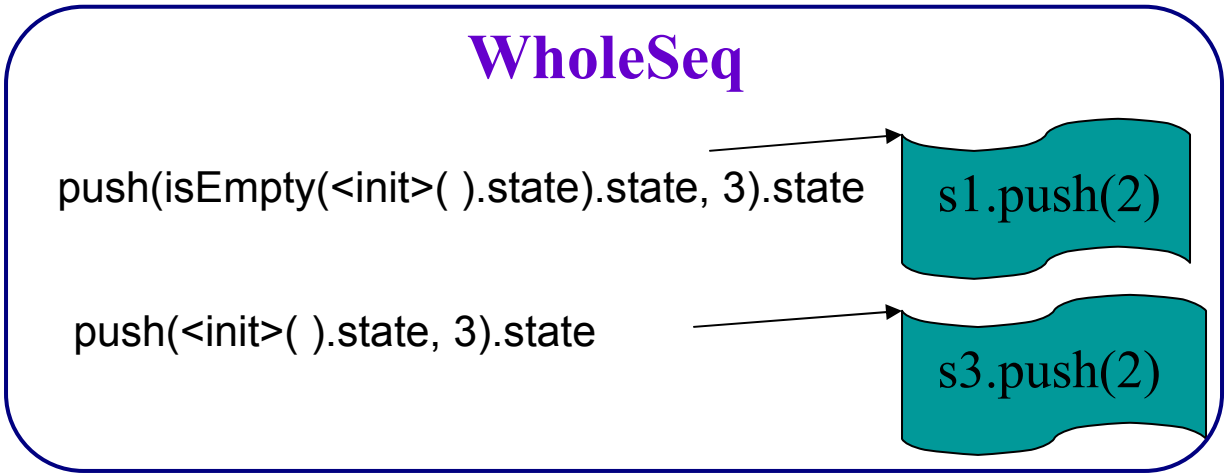
Rostra
Redundant-Test Detection Framework

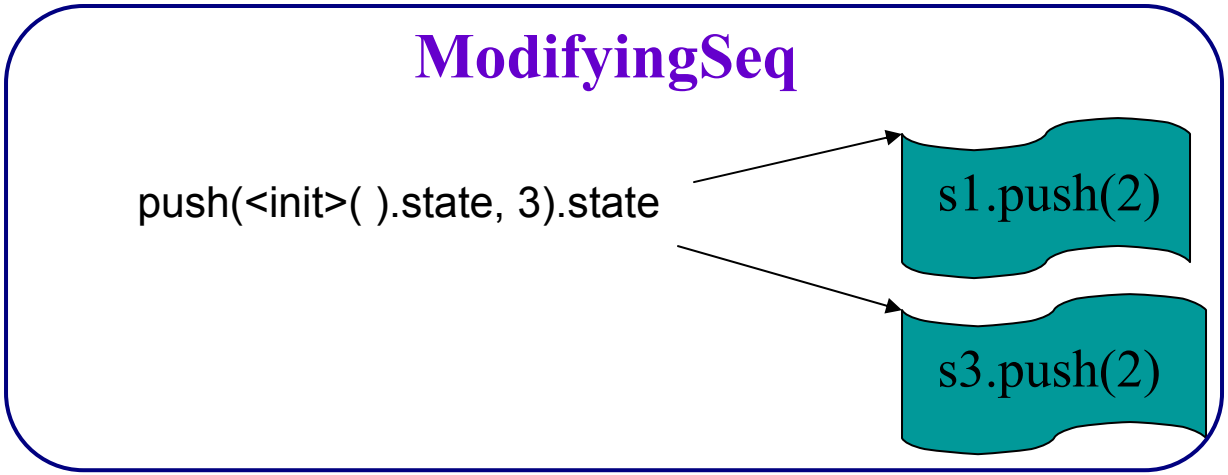
- Five techniques of object state representations
  - Two method-sequence representations
  - Three concrete-state representations

- Definitions of equivalent object states, equivalent method executions, redundant tests

Method-Sequence Representations

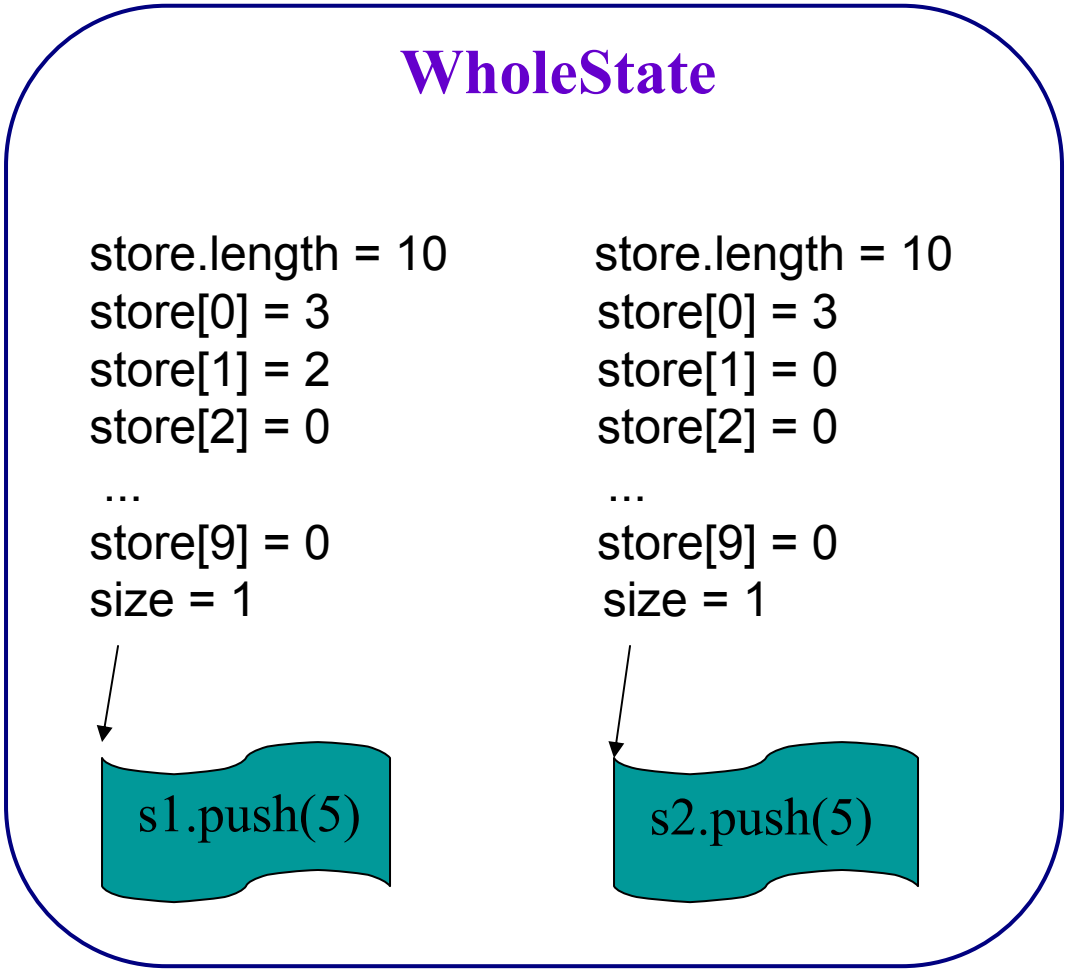
```
Test 1 (T1):
  IntStack s1 = new IntStack();
  s1.isEmpty();
  s1.push(3);
  s1.push(2);
  s1.pop();
  s1.push(5);

Test 2 (T2):
  IntStack s2 = new IntStack();
  s2.push(3);
  s2.push(5);

Test 3 (T3):
  IntStack s3 = new IntStack();
  s3.push(3);
  s3.push(2);
  s3.pop();
```





Concrete-State Representations - I

```
Test 1 (T1):
  IntStack s1 = new IntStack();
  s1.isEmpty();
  s1.push(3);
  s1.push(2);
  s1.pop();
  s1.push(5);

Test 2 (T2):
  IntStack s2 = new IntStack();
  s2.push(3);
  s2.push(5);

Test 3 (T3):
  IntStack s3 = new IntStack();
  s3.push(3);
  s3.push(2);
  s3.pop();
```

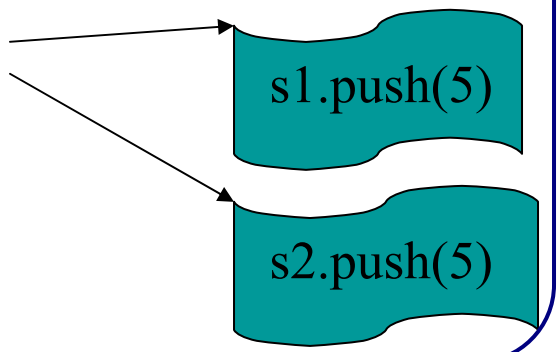


Concrete-State Representations - II

```
Test 1 (T1):
  IntStack s1 = new IntStack();
  s1.isEmpty();
  s1.push(3);
  s1.push(2);
  s1.pop();
  s1.push(5);

Test 2 (T2):
  IntStack s2 = new IntStack();
  s2.push(3);
  s2.push(5);

Test 3 (T3):
  IntStack s3 = new IntStack();
  s3.push(3);
  s3.push(2);
  s3.pop();
```

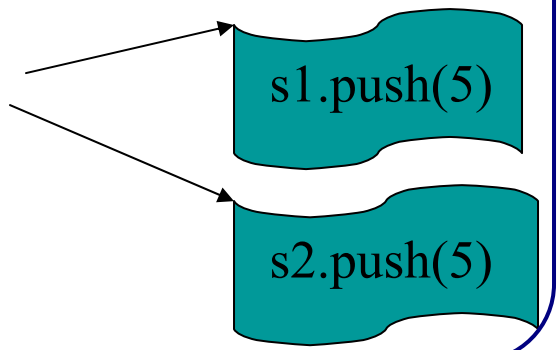
MonitorEquals obj.equals(obj)

store.length = 10
store[0] = 3
size = 1



PairwiseEquals

s1.equals(s2) == true



Redundant-Test Detection

- Equivalent object states
  - the same object state representations.
- Equivalent method executions
  - the same method name and signatures, equivalent method arguments, and equivalent entry object states.
- Redundant test:
  - A test *t* is redundant for a test suite *S* iff for each method execution of *t*, exists an equivalent method execution of some test in *S*.

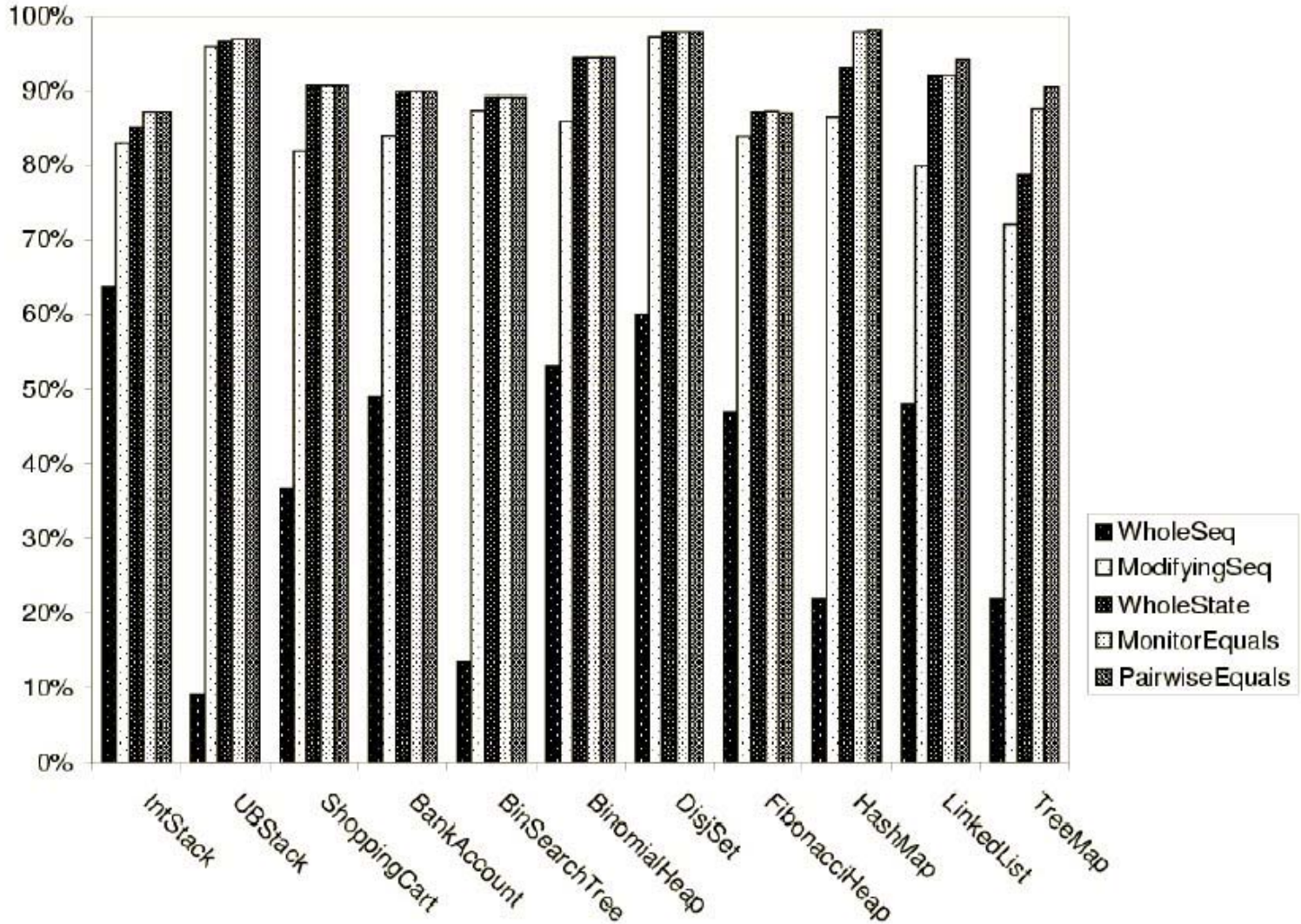
Detected Redundant Tests

```
Test 1 (T1):
  IntStack s1 = new IntStack();
  s1.isEmpty();
  s1.push(3);
  s1.push(2);
  s1.pop();
  s1.push(5);

Test 2 (T2):
  IntStack s2 = new IntStack();
  s2.push(3);
  s2.push(5);

Test 3 (T3):
  IntStack s3 = new IntStack();
  s3.push(3);
  s3.push(2);
  s3.pop();
```

technique	detected redundant tests
WholeSeq	
ModifyingSeq	T3
WholeState	T3
MontiorEquals	T3, T2
PairwiseEquals	T3, T2

Experimental Results – I

How much do we benefit?



Percentage of redundant tests among Jtest-generated tests

Experimental Results – II
Does redundant test removal decrease test suite quality?

- Measurements of original test suite quality
  - Avg # uncaught exceptions: 4
  - Avg Branch cov %: 77%
  - Avg Ferastra mutant killing %: 52%
  - Avg Jmutation mutant killing %: 54%
- Minimized test suites using the first three techniques preserve all measurements.
- Minimized test suites using the two *equals* techniques doesn't preserve
  - branch cov % (2 programs)
  - Ferastra mutant killing % (2 programs).

Testing with Equivalence Properties - Summary

- Generating, executing, and inspecting redundant tests are expensive but without gaining any benefit
- Rostra framework helps
  - compare quality of different test suites
  - select tests to augment an existing test suite
  - minimize a test suite for correctness inspection/regression execution
  - guide test generation tools to avoid generating redundant tests
- We have built test minimization and test generation tools upon Rostra

Solutions – Inferred Properties

- High percentage of redundant tests
  - Equivalence properties [ASE 04] [1]
    - redundancy detection and avoidance
- Impractical to eyeball-inspect for correctness
  - Operational abstractions [ASE 03]
    - test selection
  - Statistical algebraic abstractions
    - test selection
  - Observer abstractions [ICFEM 04]
    - test abstraction

[1]. Joint work also with Darko Marinov

Testing with Operational Abstractions

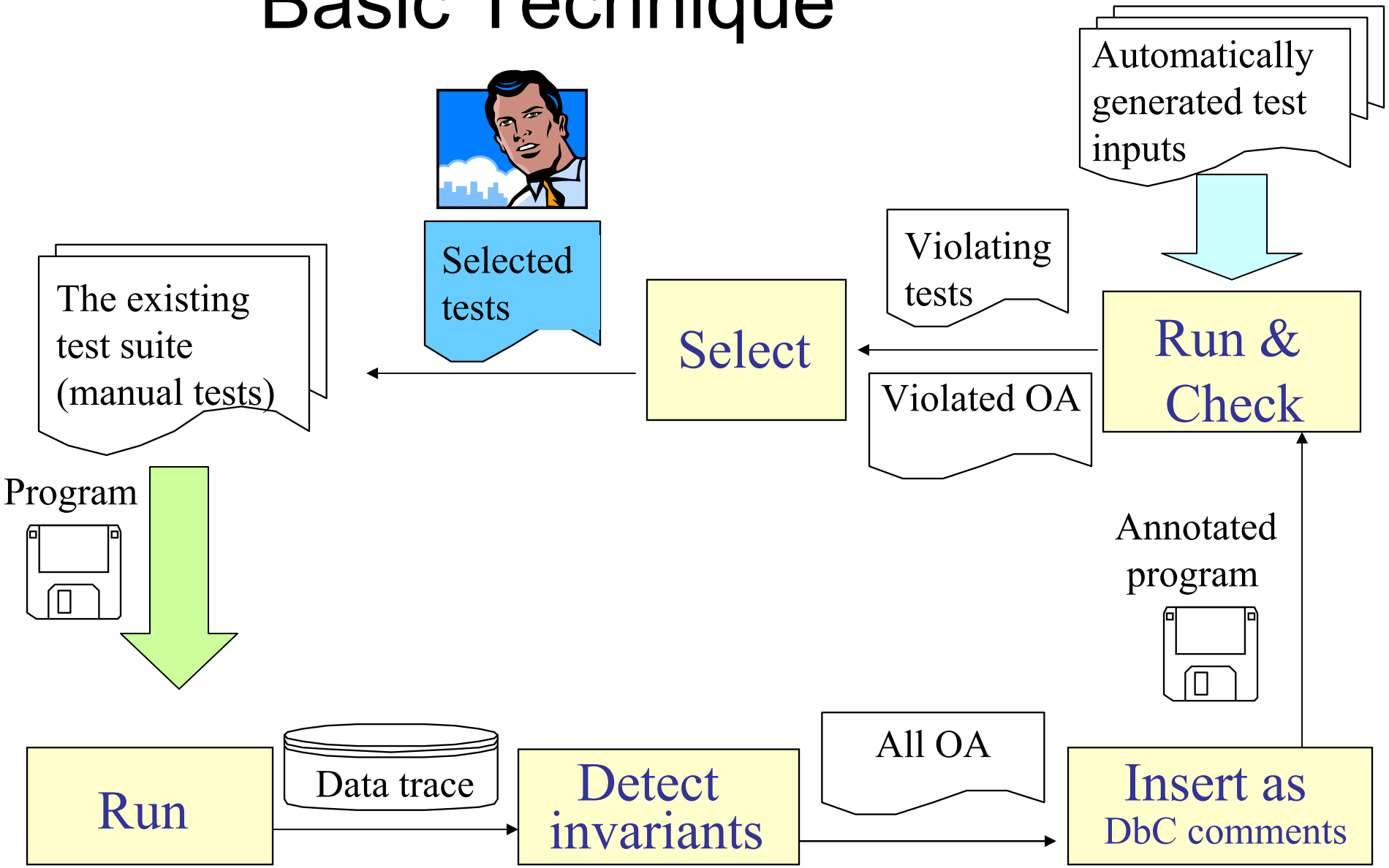
- Problem:
  - Given an existing (manual) test suite, select automatically generated tests for inspection
- Solution:
  - Use dynamic invariant detector to generate Operational Abstractions (OA) observed from existing test executions
  - Use OA to guide better test generation for violating them
  - Use OA to select violating tests for inspection
- Rationale:
  - A violating test exercises a new feature of program behavior that is not covered by the existing test suite.

Operational Abstractions [Ernst et al. 01]

- Goal: determine properties true at runtime
  (e.g. in the form of Design by Contract)
- Tool: Daikon (dynamic invariant detector)
- Approach
  1. Run test suites on a program
  2. Observe computed values
  3. Generalize

http://pag.lcs.mit.edu/daikon

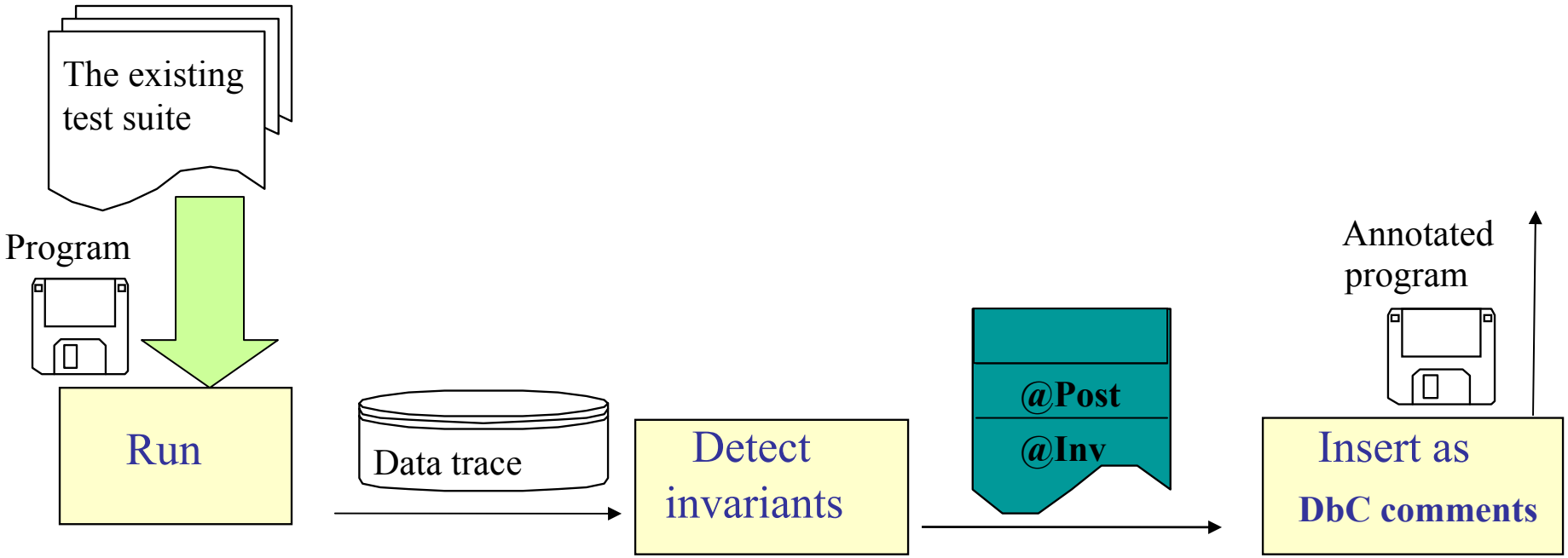
Basic Technique



OA: Operational Abstractions

Precondition Removal Technique

- Overconstrained preconditions may leave (important) legal inputs unexercised
- Solution: precondition removal technique



Motivating Example [Stotts et al. 02]

```
public class uniqueBoundedStack {
  private int[] elems;
  private int numberOfElements;
  private int max;

  public uniqueBoundedStack() {
    numberOfElements = 0;
    max = 2;
    elems = new int[max];
  }

public int getNumberOfElements() {
    return numberOfElements;
  }
        ......
};
```

A manual test suite (15 tests)

Operational Violation Example

- Precondition Removal Technique

```
public int top(){
   if (numberOfElements < 1) {
     System.out.println("Empty Stack");
     return -1;
   } else {
     return elems[numberOfElements-1];
   }
}
```

```
@pre { for (int i = 0 ; i <= this.elems.length-1; i++)
          $assert ((this.elems[i] >= 0));   }
```

Daikon generates from manual test executions:

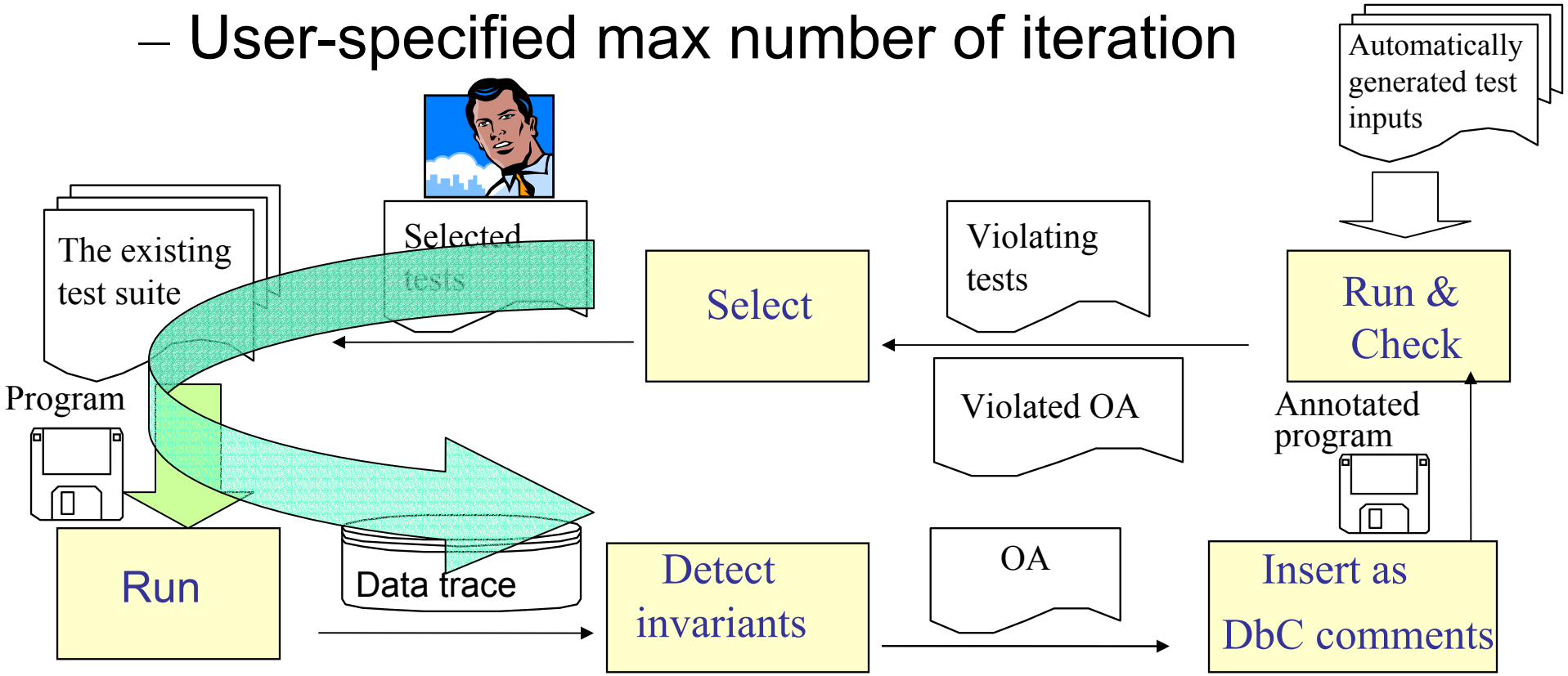
```
@post: [($result == -1) ⇔ (this.numberOfElements == 0)]
```

Jtest generates a violating test input:

```
uniqueBoundedStack THIS = new uniqueBoundedStack ();
THIS.push (-1);
int RETVAL = THIS.top ();
```

Iterations

- The existing tests augmented by selected tests are run to generate OA
- Iterates until
  - No operational violations
  - User-specified max number of iteration



Experiments

- **12** programs from assignments and texts (standard data structures)
  - Total **775** executable LOC in **127** methods
- Accompanying manual test suites
  - **~94**% branch coverage
- The number of Jtest-generated tests without operational abstraction
  - Range(**24…227**) Median(**124**)
    [test containing up to 2 method calls]
  - **Thousands** [test containing up to 3 method calls]

Experimental Results

- The number of selected tests
  - Our approach:
    - Range(**0…25**) Median(**3**)
  - Structural approach:
    - Range(**0…5**) Median(**1**)

- The percentage of anomaly-revealing tests among selected tests (median)
  - Our approach:
    - Iteration 1: **20%** (Basic) **68%** (Pre_Removal)
    - Iteration 2: **0%** (Basic) **17%** (Pre_Removal)
  - Structural approach: **0%**
    - But increase confidence on the new exercised branches
- Many anomaly-revealing tests not generated by Jtest without operational abstractions

Solutions – Inferred Properties

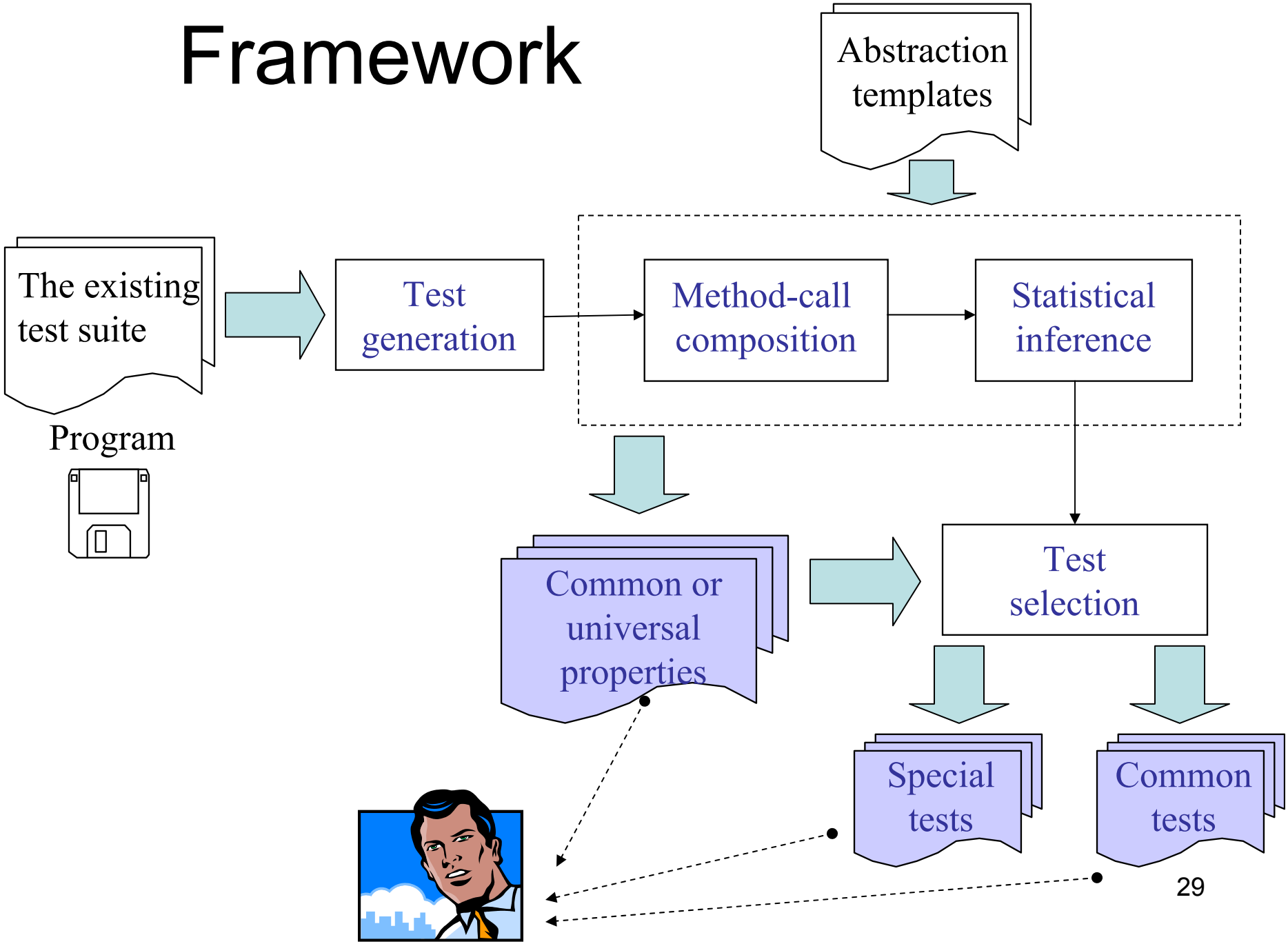
- High percentage of redundant tests
  - Equivalence properties [ASE 04] [1]
    - redundancy detection and avoidance
- Impractical to eyeball-inspect for correctness
  - Operational abstractions [ASE 03]
    - test selection
  - Statistical algebraic abstractions
    - test selection
  - Observer abstractions [ICFEM 04]
    - test abstraction

[1]. Joint work also with Darko Marinov

Testing with Statistical Algebraic Abstraction

- Problem: Special/common test selection
  - Programmers have an intuitive notion of special/common tests. How to identify special/common tests automatically?
- Solution: Use inferred statistical algebraic abstractions

Framework


Abstraction templates
The existing test suite
Program
Test generation
Method-call composition
Statistical inference
Common or universal properties
Test selection
Special tests
Common tests

Abstraction Templates - I

- foo2(foo1(S, arg1), arg2) = const
  - isEmpty (push (S, element)) == false
- foo2(foo1(S, arg1), arg2) = arg1 or arg2
  - Top (push (S, element)) == element
- foo2(foo1(S, arg1), arg2) = foo1(S, arg1)
  - equals (pop(<init>()), <init>())
- foo2(foo1(S, arg1), arg2) = S
  - equals (pop (push (S, element)), S)
- foo2(foo1(S, arg1), arg2) = foo1(foo2(S, arg2), arg1)
  - equals (push (push (S, element1), element2) ,
    push (push (S, element2), element1)
- foo1(S, arg1) = const
  - maxSize(S) == 2
- foo1(S, arg1) = S
  - equals (print(S), S)

Abstraction Templates - II

- Conditional axioms
  - foo2(foo1(S, arg1), arg2) = ((arg1 == arg2)? RHS_true : RHS_false)
  - foo2(foo1(S, arg1), arg2) = ((arg1 != arg2)? RHS_true : RHS_false)
  - foo2(foo1(S, arg1), arg2) = ((foo3(S))? RHS_true : RHS_false)

- Differencing axioms
  - foo2(foo1(S, arg1), arg2) = RHS + const

Test Selection Based on Statistical Abstractions

- Method-call composition: compose method call pair from method executions
  - Method executions of *foo1* and *foo2* are composed as *foo2(foo1(S, arg1), arg2),*
    - if *foo1.exit_state* == *foo2.entry_state*

- Statistical inference: look for equality/inequality patterns among args, return, entry state, exit state of either method in a pair
  - Based on abstraction templates

- Special/common test selection: universal axioms, common axioms

IntStack Universal Axiom Examples

- pop(<init>().state) = ArrayIndexOutOfBoundsException
  - EuqalCount:3
  - Common Test: pop(IntStack.<init>().state)

- pop(push(S, I).state).retval == I
  pop(push(S, I).state).state == S
  - EuqalCount:22
  - Common Test: pop(push(<init>().state,0).state)

- isEmpty(<init>().state).retval = true
  - EuqalCount:3
  - Common Test: isEmpty(<init>().state)

IntStack Common Axiom Examples

- isEmpty(pop(S).state).retval == isEmpty(S).retval
  - EuqalCount:16
  - Common Test:
    isEmpty(pop(push(push(<init>().state,0).state, 0).state).state).state
  - InequalCount:3
  - Special Test: isEmpty(pop(push(<init>().state, 0)).state)

- isEmpty(push(S, I).state).retval == isEmpty(S).retval
  - EuqalCount:20
  - Common Test:
    isEmpty(push(push(IntStack.<init>().state,0).state,0).state).state
  - InequalCount:4
  - Special Test: isEmpty(push(<init>().state,0).state)

Percentage threshold: 15%

Solutions – Inferred Properties

- High percentage of redundant tests
  - Equivalence properties [ASE 04] [1]
    - redundancy detection and avoidance
- Impractical to eyeball-inspect for correctness
  - Operational abstractions [ASE 03]
    - test selection
  - Statistical algebraic abstractions
    - test selection
  - Observer abstractions [ICFEM 04]
    - test abstraction

[1]. Joint work also with Darko Marinov

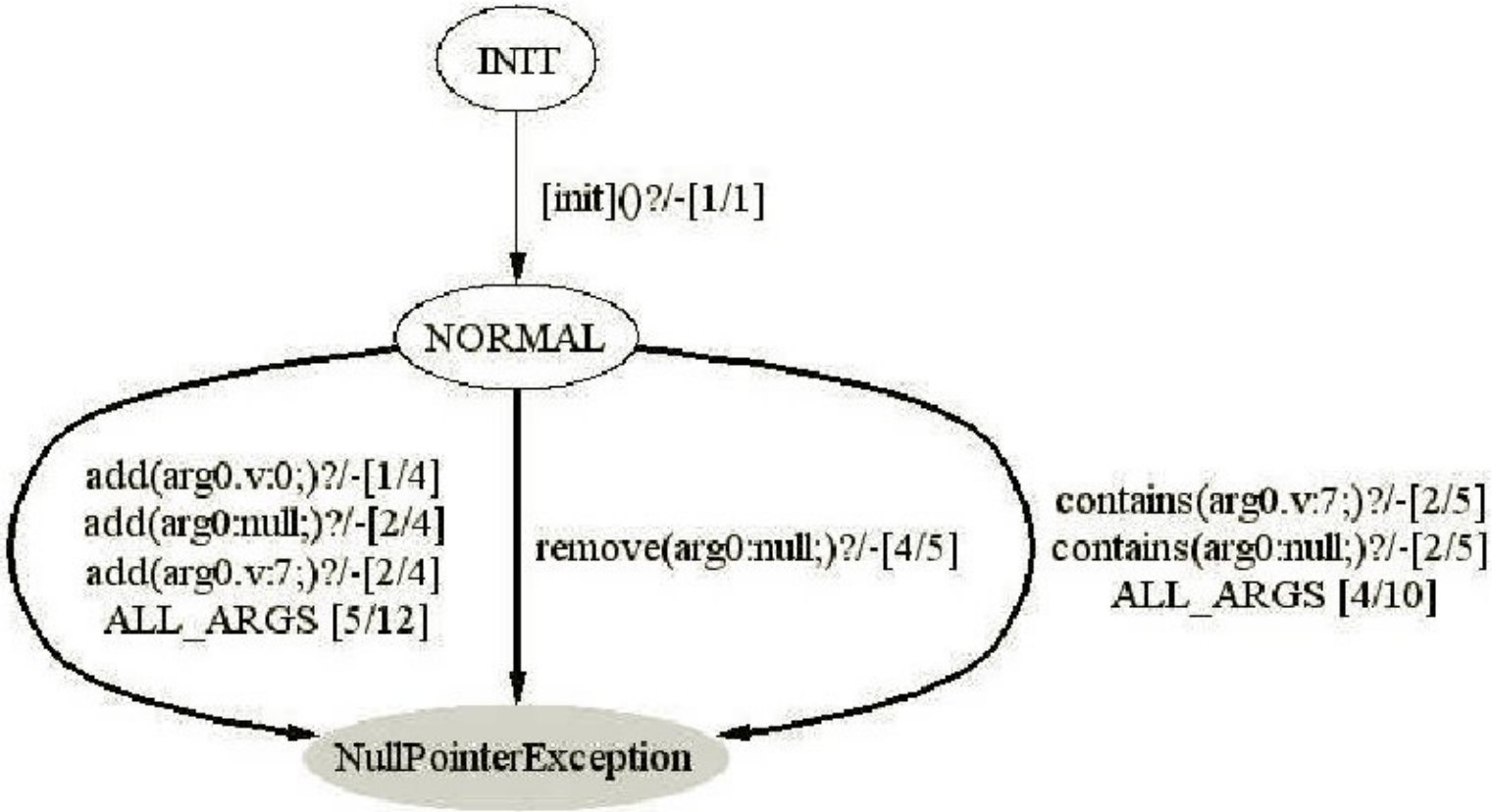
Testing with Observer Abstractions

- Object State Machines (OSM):
  - States: object states
  - Transitions: method calls
- Problem: Concrete OSM is often too large for inspection
- Solution: Use observers (methods with non-void returns) to abstract concrete states
- Summarize the behavior of all tests instead of selecting a subset

Generating Observer Abstractions

- Equivalent detection (Rostra)
  - Run existing tests and collect nonequivalent object states and method calls
- Test augmentation
  - Generate tests to exercise each combination of nonequivalent object states and method calls
- Observer abstraction construction
  - For method calls of each observer, collect their return values on each nonequivalent object state
  - Use their return values as abstract state representation
  - Construct an abstract object state machine for each observer

exception Observe Abstraction of BinarySearchTree



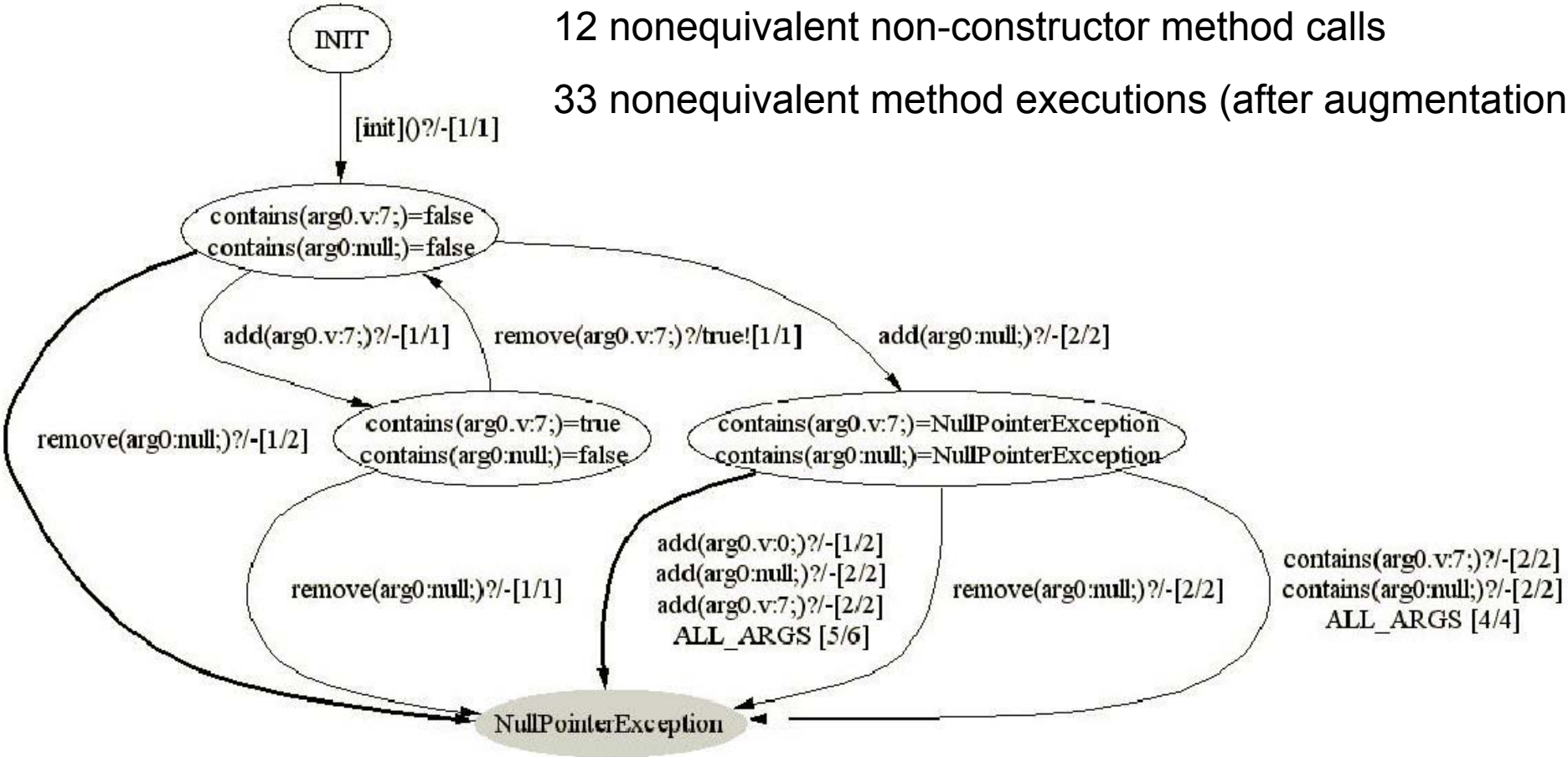
contains Observe Abstraction of BinarySearchTree

277 Jtest-generated tests

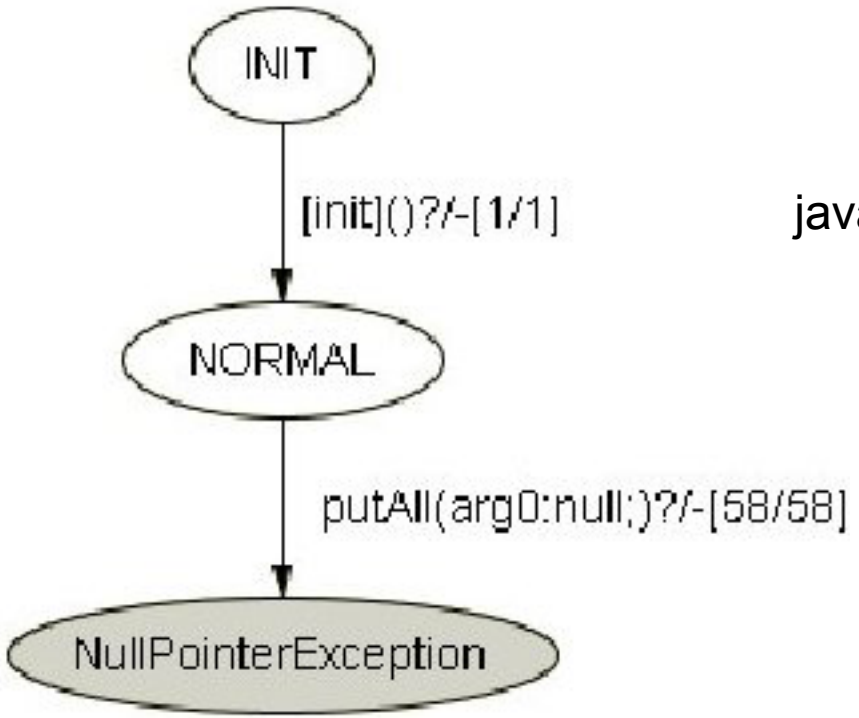
5 concrete states

12 nonequivalent non-constructor method calls

33 nonequivalent method executions (after augmentation)



exception Observe Abstraction of java.util.HashMap



java.util.HashMap putAll API documentation

putAll
public void **putAll**(Map m)

….

Specified by:
putAll in interface Map

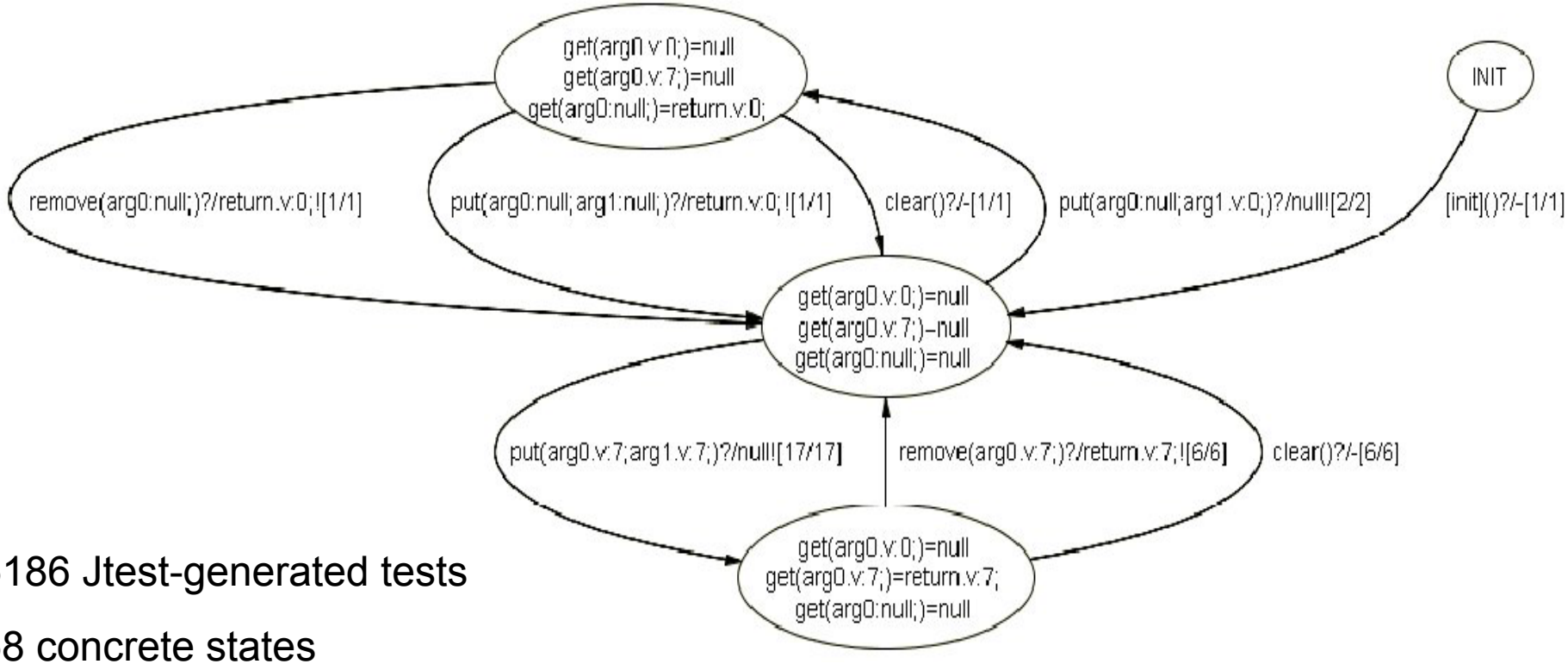
Overrides:
putAll in class AbstractMap

Parameters:
m - mappings to be stored in this map.

Throws:
NullPointerException - if the specified map is null.

http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html

get Observe Abstraction of java.util.HashMap



5186 Jtest-generated tests

58 concrete states

29 nonequivalent non-constructor method calls

1683 nonequivalent method executions (after augmentation)

Flaw in java.util.HashMap *get* API documentation

http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html

get

```
public Object get(Object key)
```

Returns the value to which the specified key is mapped in this identity hash map, or `null` if the map contains no mapping for this key. A return value of `null` does not *necessarily* indicate that the map contains no mapping for the key; it is also possible that the map explicitly maps the key to `null`. The `containsKey` method may be used to distinguish these two cases.

Specified by:
`get` in interface `Map`
Overrides:
`get` in class `AbstractMap`
Parameters:
`key` - the key whose associated value is to be returned.
Returns:
the value to which this map maps the specified key, or `null` if the map contains no mapping for this key.
See Also:
`put(Object, Object)`

A return value of null does not *necessarily* indicate that the map contains no mapping for the key; it is also possible that the map explicitly maps the key to null.

but

the value to which this map maps the specified key, or null if the map contains no mapping for this key.

Conclusion

- High percentage of redundant tests
  - Equivalence properties [ASE 04]
    - redundancy detection and avoidance
    - support multithreading programs
    - test generation (symbolic executions)
- Impractical to eyeball-inspect for correctness
  - Operational abstractions [ASE 03]
    - Test selection
  - Statistical algebraic abstractions
    - Test selection
  - Observer abstractions [ICFEM 04]
    - Test summarization
  - Symmetry abstractions
  - Protocol abstractions
- Regression testing using value spectra [ICSM 04]