

Strategic Automated Software Testing in the Absence of Specifications

Tao Xie

Dept. of Computer Science & Engineering
University of Washington

Parasoft Co. Nov. 2004

Motivation

- How do we generate “useful” tests automatically?
 - With specifications, we can partition input space into subdomains and generate samples from these subdomains [Myers 79]
 - Korat [Boyapati et al. 02]: repOk (partitioning input space into valid and invalid subdomains)
 - AsmLT/SpecExplorer [MSR FSE]: abstract state machine
- How do we know the generated tests run incorrectly in the absence of uncaught exceptions?
 - With specifications, we know a fault is exposed when a postcondition is violated by a precondition-satisfying input.
- We know that specifications are often not written in practice

Our Strategic Approaches

- How do we generate “useful” tests automatically?
 - Detect and avoid redundant tests during/after test generation **[Xie, Marinov, and Notkin ASE 04]**
 - Based on inferred equivalence properties among object states
 - Detected redundant tests do not improve reliability
 - no changes in fault detection, structural coverage, confidence
- How do we know the program runs incorrectly in the absence of uncaught exceptions?
 - It is infeasible to inspect the execution of each single test
 - Select the most “valuable” subset of generated tests for inspection **[Xie and Notkin ASE 03]**
 - Based on inferred properties from existing (manual) tests
 - Select any test that violates one of these properties (deviation from “normal”)

Overview

- Motivation
- Redundant-test detection based on object equivalence
- Test selection based on operational violations
- Conclusions

Example Code

[Henkel&Diwan 03]

```
public class IntStack {
    private int[] store;
    private int size;
    public IntStack() { ... }
    public void push(int value) { ... }
    public int pop() { ... }
    public boolean isEmpty() { ... }
    public boolean equals(Object o) { ... }
}
```

Example Generated 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);
```

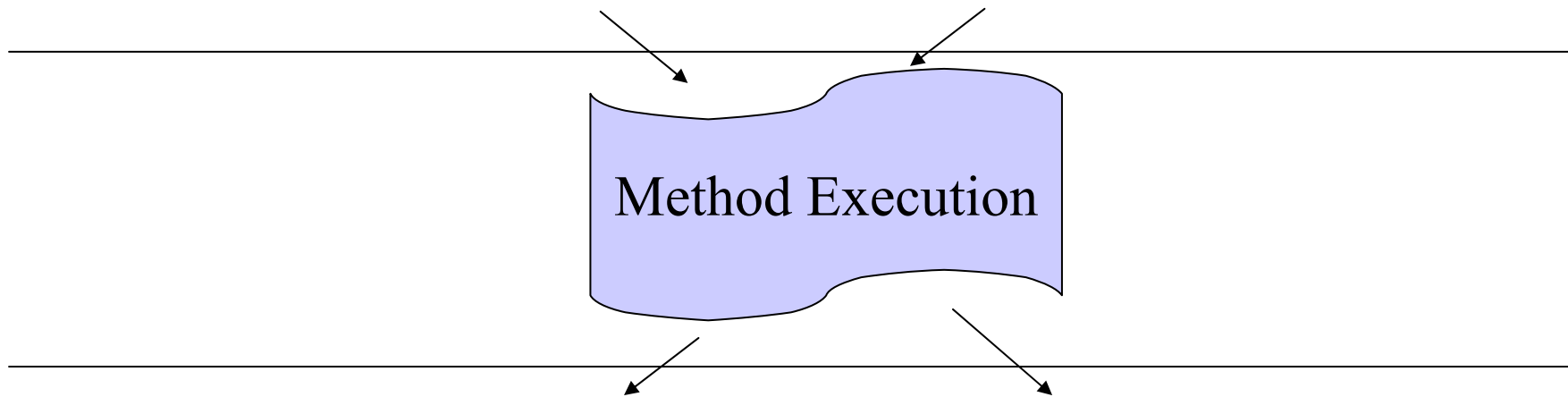
Test 3 (T3):

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

Same inputs \Rightarrow Same behavior

Assumption: deterministic method

Input = object state @entry + Method arguments



Output = object state @exit + Method return

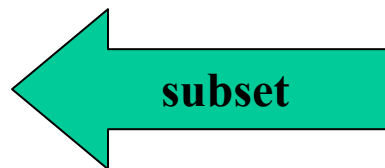
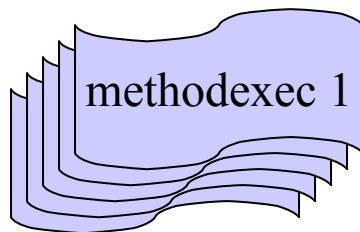
Testing a method with the same inputs is unnecessary

*We developed five techniques for
representing and comparing object states*

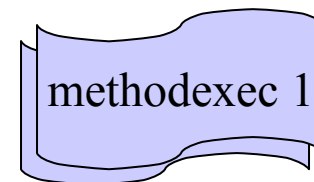
Redundant Tests Defined

- Equivalent method executions
 - the same method names, signatures, and input (equivalent object states @entry and arguments)
- Redundant test:
 - Each test produces a set of method executions
 - Test_j is redundant for a test suite (Test₁ ... Test_i)
 - if the method executions produced by Test_j is a subset of the method executions produced by Test₁ ... Test_i

Test₁ ... Test_i



Redundant Test_j



Comparison with Traditional Definition

- Traditionally redundancy in tests was largely based on structural coverage
 - A test was redundant with respect to a set of other tests if it added no additional structural coverage (no statements, no edges, no paths, no def-use edges, etc.)
- Unlike our new definition, this structural-coverage-based definition is not safe.
 - A redundant test (in the traditional definition) can expose new faults

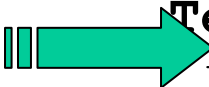
Five State-Representation Techniques

- Method-sequence representations
 - WholeSeq
 - The entire sequence
 - ModifyingSeq
 - Ignore methods that don't modify the state
- Concrete-state representations
 - WholeState
 - The full concrete state
 - MonitorEquals
 - Relevant parts of the concrete state
 - PairwiseEquals
 - **equals ()** method used to compare pairs of states

WholeSeq Representation

Method sequences that create objects

Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]



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

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

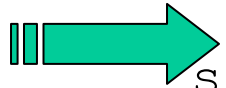
WholeSeq Representation

Method sequences that create objects

Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```



Test 3 (T3):

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

<init>().state

WholeSeq Representation

Method sequences that create objects

Notation: `methodName(entryState, methodArgs).state` [Henkel&Diwan 03]

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```



Test 3 (T3):

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

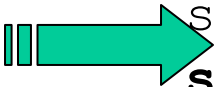
`isEmpty(<init>().state).state`

WholeSeq Representation

Method sequences that create objects

Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

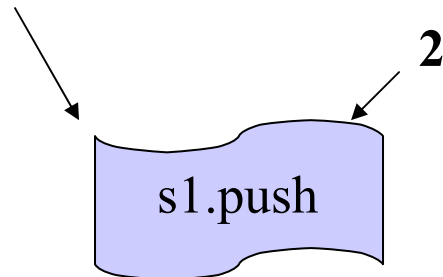
Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

Test 3 (T3):

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

push(isEmpty(<init>().state).state, 3).state



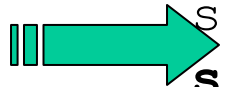
WholeSeq Representation

Method sequences that create objects

Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

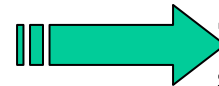
Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

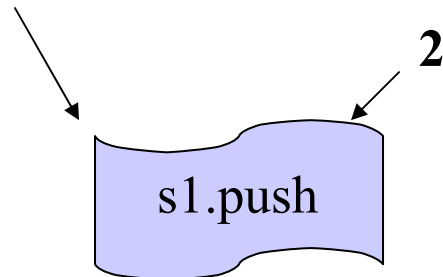


Test 3 (T3):

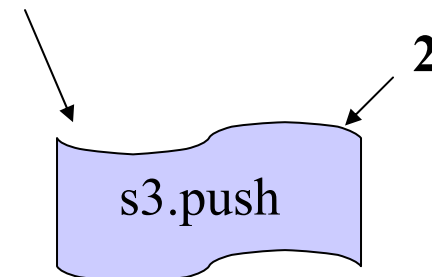
```
IntStack s3 =  
    new IntStack();  
s3.push(3);  
s3.push(2);  
s3.pop();
```



push(isEmpty(<init>().state).state, 3).state



push(<init>().state, 3).state

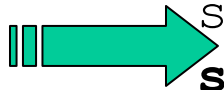


ModifyingSeq Representation

State-modifying method sequences that create objects

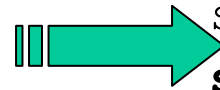
Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

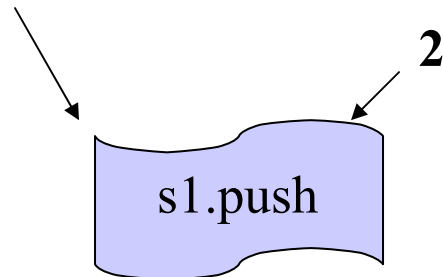


Test 3 (T3):

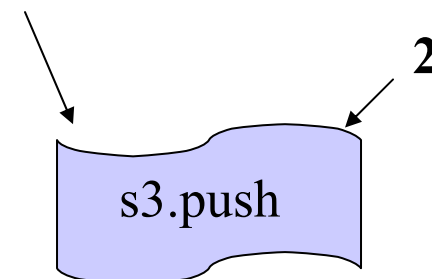
```
IntStack s3 =  
    new IntStack();  
s3.push(3);  
s3.push(2);  
s3.pop();
```



~~push(~~isEmpty~~(~~<init>~~().state, ~~state~~, 3).state~~



push(~~<init>~~().state, 3).state

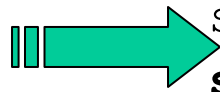


WholeState Representation

The entire concrete state reachable from the object

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

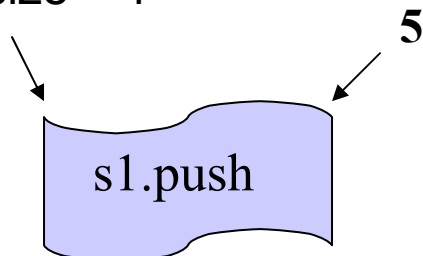


Test 2 (T2):

```
IntStack s2 =  
    new IntStack();  
s2.push(3);  
s2.push(5);
```

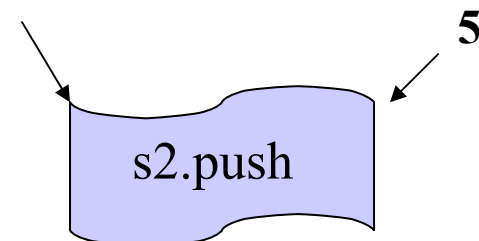


store.length = 3
store[0] = 3
store[1] = 2
store[2] = 0
size = 1



**Comparison by
isomorphism**

store.length = 3
store[0] = 3
store[1] = 0
store[2] = 0
size = 1



MonitorEquals Representation

The relevant part of the concrete state defined by *equals* (invoking *obj.equals(obj)* and monitor field accesses)

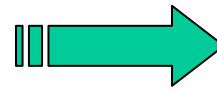
Test 1 (T1):

```
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);
```



store.length = 3

store[0] = 3

~~store[1] = 2~~

~~store[2] = 0~~

size = 1

5

s1.push

Comparison by
isomorphism

store.length = 3

store[0] = 3

~~store[1] = 0~~

~~store[2] = 0~~

size = 1

5

s2.push

PairwiseEquals Representation

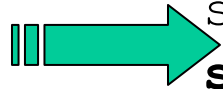
The results of *equals* invoked to compare pairs of states

Test 1 (T1):

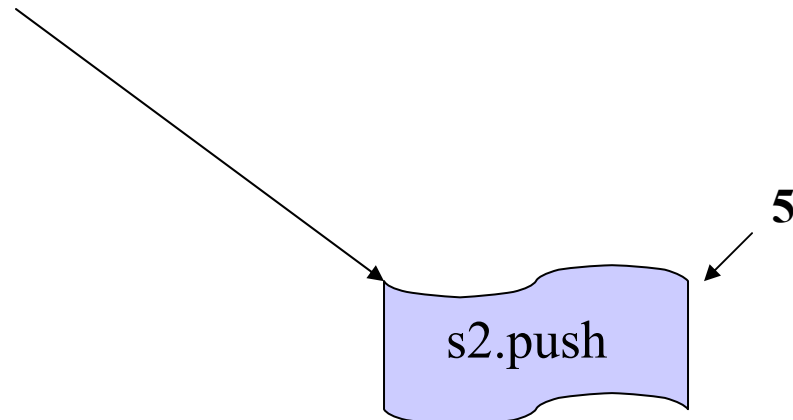
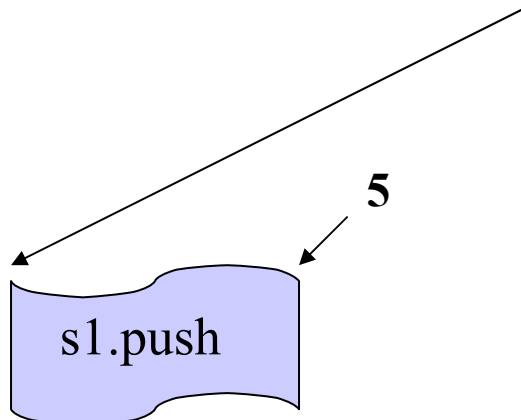
```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

Test 2 (T2):

```
IntStack s2 =  
    new IntStack();  
s2.push(3);  
s2.push(5);
```



`s1.equals(s2) == true`



Redundant-Test Detection

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

Test 3 (T3):

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

Redundant-Test Detection

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```



Test 3 (T3):

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

Using last four techniques:

ModifyingSeq, WholeState, MonitorEquals, PairwiseEquals

Redundant-Test Detection

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

Test 3 (T3):

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



Using last four techniques:

ModifyingSeq, WholeState, MonitorEquals, PairwiseEquals

Redundant-Test Detection

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

Test 3 (T3):

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



Using last four techniques:

ModifyingSeq, WholeState, MonitorEquals, PairwiseEquals

Redundant-Test Detection

Test 1 (T1):

```
IntStack s1 =  
    new IntStack();  
s1.isEmpty();  
s1.push(3);  
s1.push(2);  
s1.pop();  
s1.push(5);
```

Test 3 (T3):

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

Test 3 is redundant w.r.t Test 1

Using last four techniques:

ModifyingSeq, WholeState, MonitorEquals, PairwiseEquals

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();
```

<i>technique</i>	<i>detected redundant tests w.r.t. T1</i>
WholeSeq	
ModifyingSeq	T3
WholeState	T3
MonitorEquals	T3, T2
PairwiseEquals	T3, T2

Experiment:

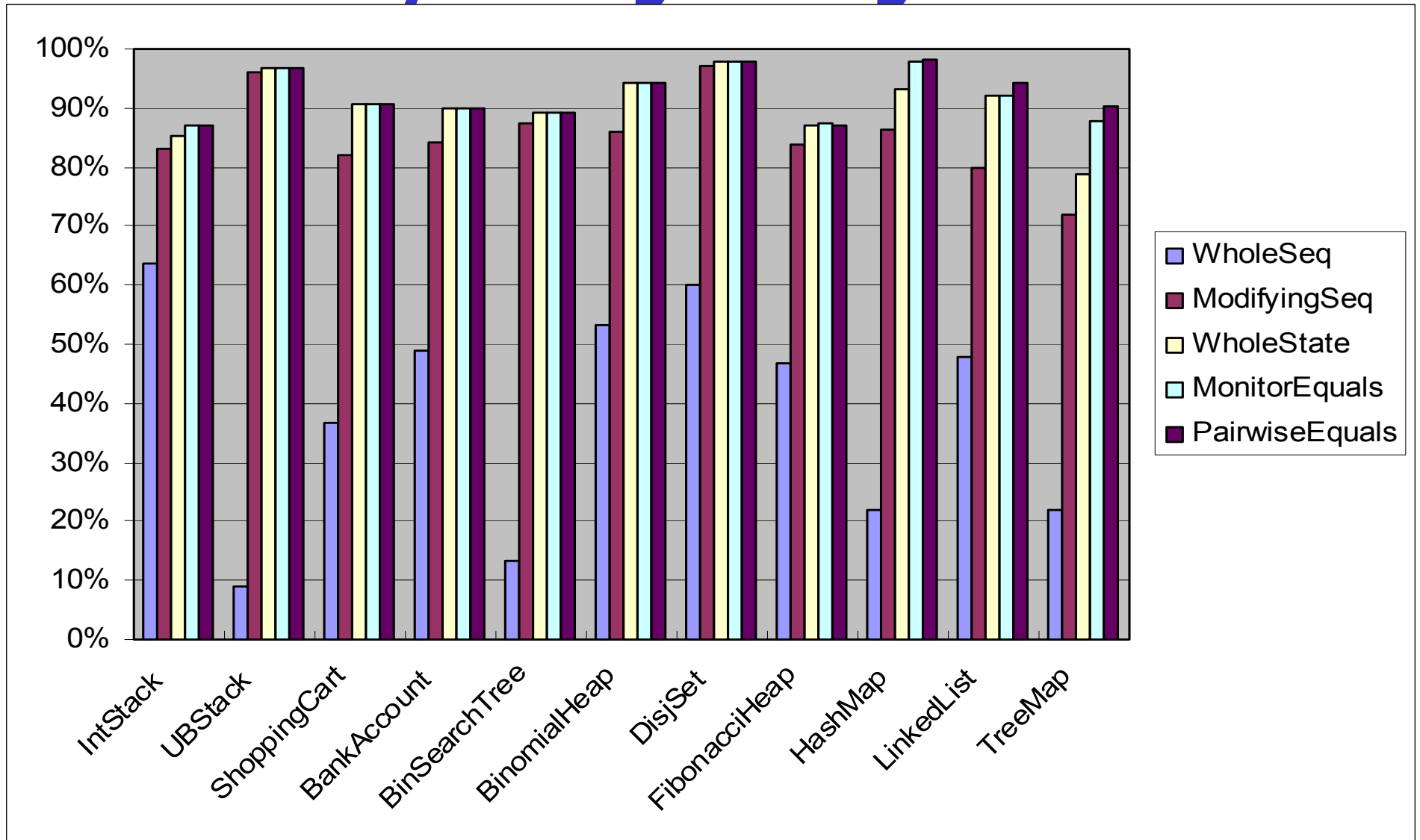
Evaluated Test Generation Tools

- ParaSoft Jtest 4.5 (both black and white box testing)
 - A commercial Java testing tool
 - Generates tests with method-call lengths up to three
- JCrasher 0.2.7 (robustness testing)
 - An academic Java testing tool
 - Generates tests with method-call lengths of one
- Use them to generate tests for 11 subjects from a variety of sources
 - Most are complex data structures

Answered Two Questions

- How much do we benefit after applying Rostra on tests generated by Jtest and JCrasher?
 - The last three techniques detect around
 - **90%** redundant tests for Jtest-generated tests
 - **50%** on half subjects for JCrasher-generated tests.
 - Detected redundancy in increasing order for five techniques
- Does redundant-test removal decrease test suite quality?
 - The first three techniques preserve both branch cov and mutation killing capability
 - Two equals-based techniques have very small loss

Redundancy among Jtest-generated Tests



- The last three techniques detect around 90% redundant tests
- Detected redundancy in increasing order for five techniques

Overview

- Motivation
- Redundant-test detection based on object equivalence
- Test selection based on operational violations
- Conclusions

Operational Abstraction Generation

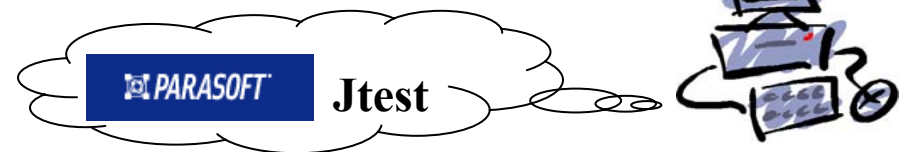
[Ernst et al. 01]

- 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

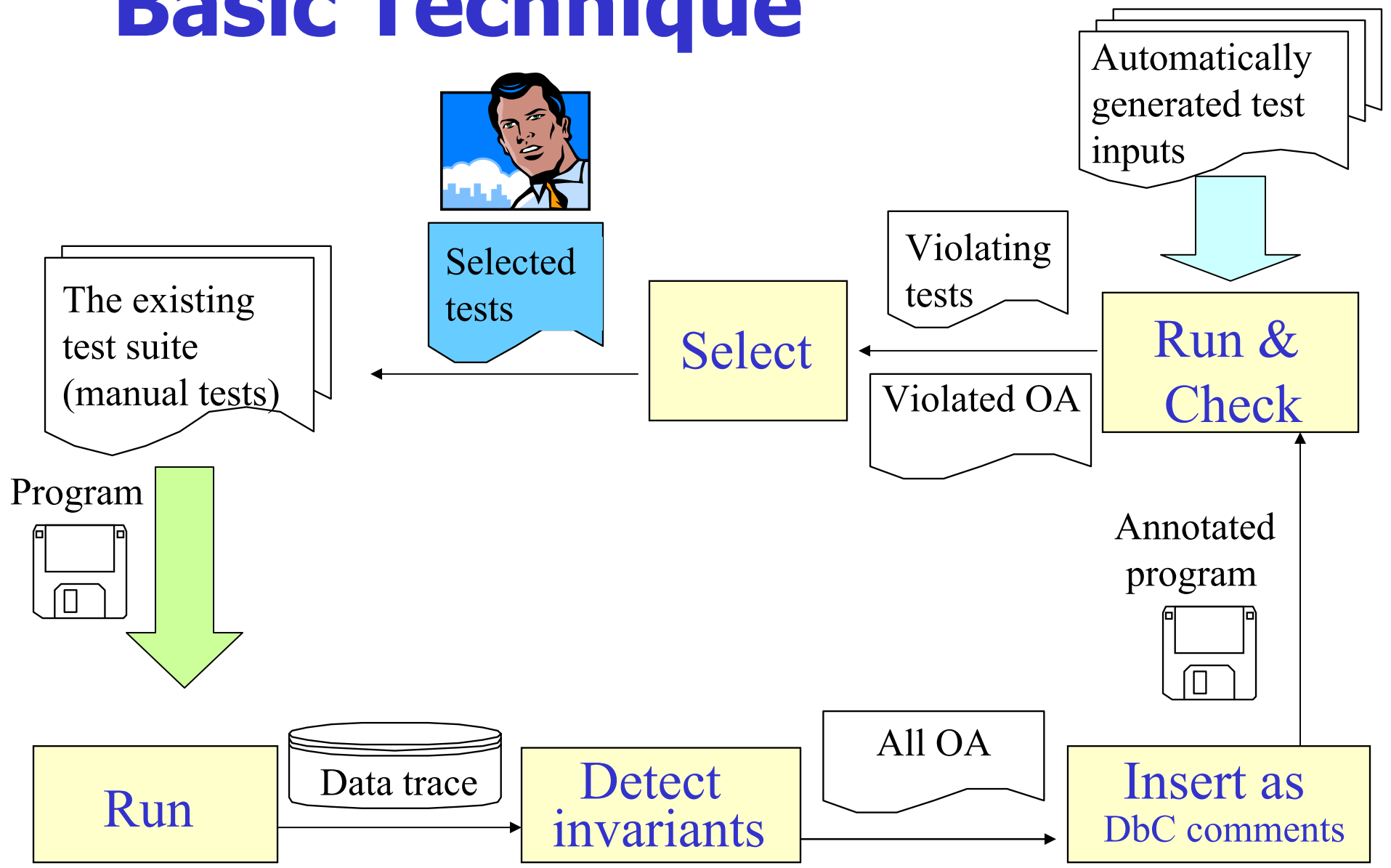


Specification-Based Testing

- Goal: generate test inputs and test oracles from specifications
- Tool: **ParaSoft Jtest** (both **black** and white box testing)
- Approach:
 1. Annotate Design by Contract (DbC) [Meyer 97]
 - Preconditions/Postconditions/Class invariants
 2. Generate test inputs that
 - Satisfy preconditions
 3. Check if test executions
 - Satisfy postconditions/invariants

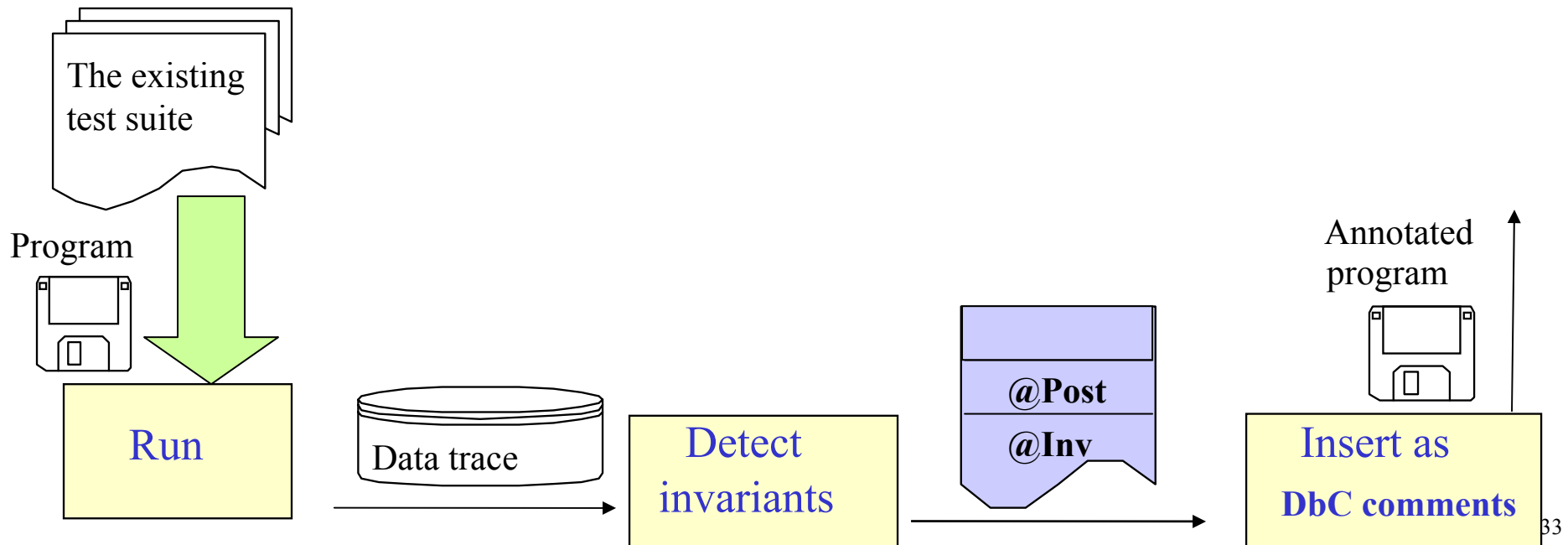


Basic Technique



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:

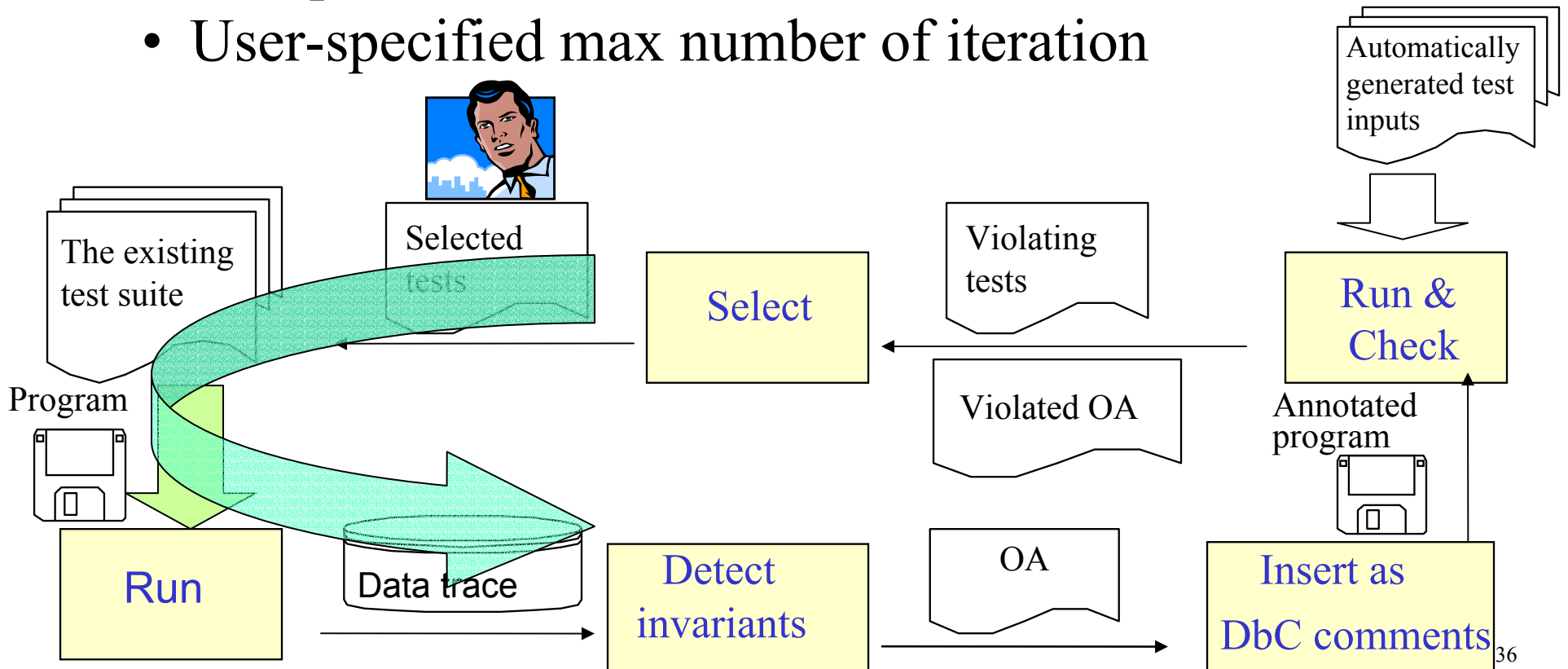
```
@post: [($result == -1) ⇔ (this.numberOfElements == 0)]
```

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 operational abstractions
- Iterates until
 - No operational violations
 - User-specified max number of iteration



Experiment: Subject Programs Studied

- **12** programs from assignments and texts
(standard data structures)
- Accompanying manual test suites
 - **~94%** branch coverage

Answered Questions

- Is the number of tests selected by our approach **small** enough?
 - if yes, **affordable** inspection effort
 - Range(0...25) Median(3)
- Do the selected tests by our approach have a **high** probability of exposing abnormal behavior?
 - if yes, select a **good subset** of generated tests
 - Iteration 1: **20%** (Basic) **68%** (Pre_Removal)
 - Iteration 2: **0%** (Basic) **17%** (Pre_Removal)

More Strategic Approaches-I

- How do we generate “useful” tests automatically?
 - Exhaustively exercise N arguments up to N method call length N
 - Breadth-first search of concrete-object state space: (limit: $N = 6$) [UW-CSE-04-01-05]
 - Breadth-first search of symbolic-object state space: (limit: $N = 8$) using symbolic execution to build up symbolic states [UW-CSE-04-10-02]
 - Longer method call length
 - Higher branch coverage
 - Generate representative arguments automatically

More Strategic Approaches - II

- How do we know the program runs incorrectly in the absence of uncaught exceptions?
 - **Test selection:** infer universal and common properties and identify common and special tests [OOPSLA Companion 04, UW-CSE-04-08-03]
 - **Test abstraction:** recover succinct object-state-transition information for inspection [ICFEM 04, SAVCBS 04]
 - **Regression testing:** detect behavior deviation of two versions by comparing value spectra (defined based on program states) [ICSM 04]

Conclusions

- Specifications can help automated software testing
 - However, specifications are often not written in practice
- Developed strategic approaches to enjoy some benefits of specification-based testing by using inferred program properties
 - Redundant test detection
 - Test generation
 - Test selection
 - Test abstraction
 - Regression testing

Questions?

<http://www.cs.washington.edu/homes/taoxie/publications.htm>