Tool-Assisted Unit Test Selection Based on Operational Violations

Tao Xie David Notkin

Department of Computer Science & Engineering
University of Washington, Seattle, WA

Oct. 8th, 2003

ASE 2003, Montreal, Canada

Synopsis

- Context: Automatic white-box test generation has many benefits
  - + Lots of tests generated for coverage and robustness
- Problems:
  - – Oracles not generated for correctness checking
  - – Lots of tests generated impractical for inspection to add oracles
- Goal:
  - From generated tests, select best candidates for manual inspection to add oracles

Synopsis (cont.)

- Solution: Use dynamic invariant detector to generate properties (a.k.a operational abstractions) observed from existing test executions
  - Guide test selection for inspection
  - Guide better test generation

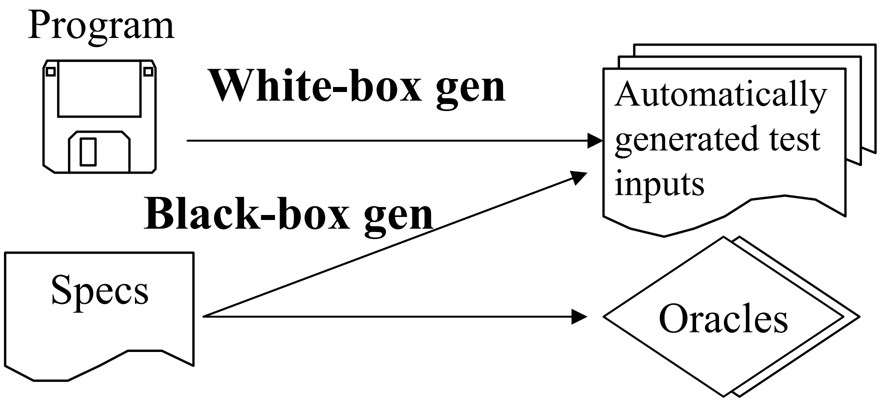
Benefits of specification-based testing can be obtained without the pain of writing the specifications!

Outline

- Motivation
- Operational Violation Approach
- Experiment
- Related Work
- Conclusion

Automatic Unit Test Generation

- White-box test generation
  - + Cover structural entities, e.g. statement, branch
  - – **Test oracle problem**
- Black-box test generation
  - + Guide test generation
  - + Produce test oracles
  - – **Require *a priori* specs**



Program

White-box gen

Automatically generated test inputs

Black-box gen

Specs

Oracles

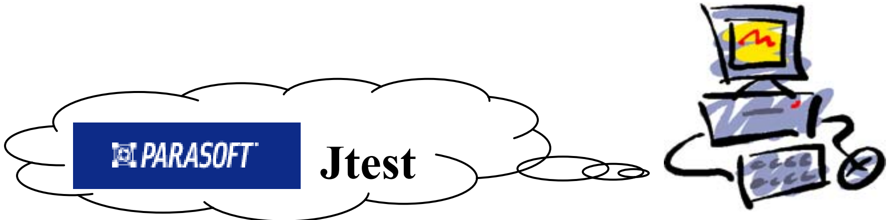
PARASOFT **Jtest**

http://www.parasoft.com/

Specification-Based Testing

- Goal: generate test inputs and test oracles from specifications
- Tool: ParaSoft Jtest
- 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

Up to range(1…3) method calls in a test



Operational Abstraction Generation

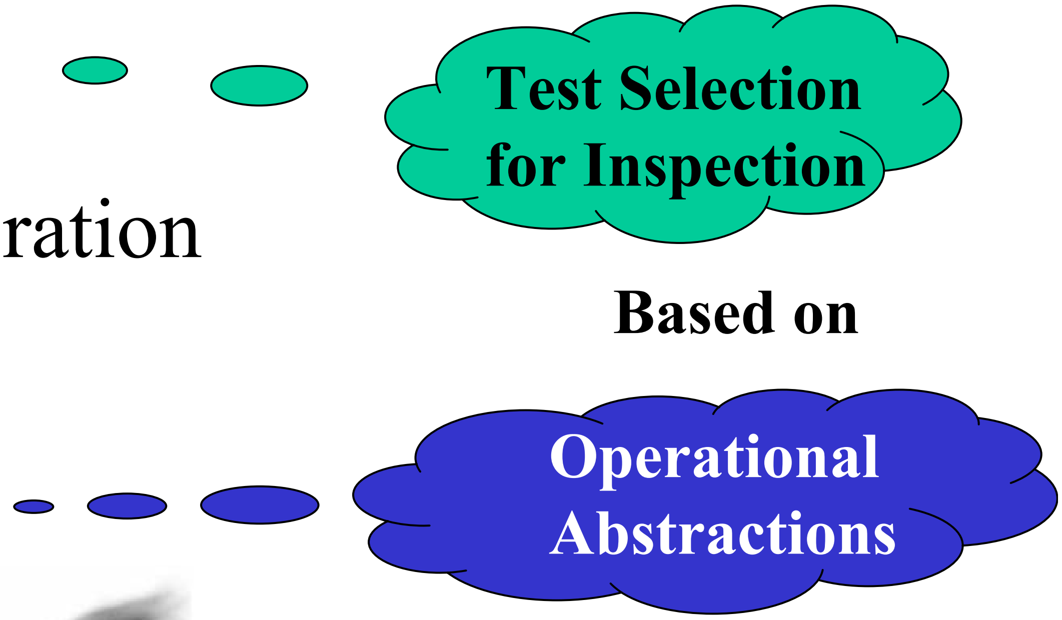
[Ernst et al. 01]

- 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

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

Automatic Unit Test Generation

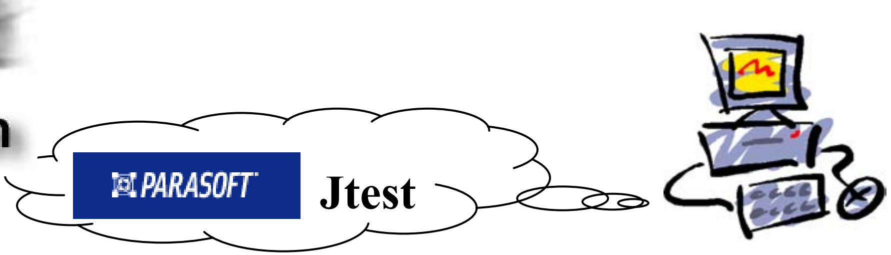
- White-box test generation
  - + Cover structural entities, e.g. statement, branch
  - – **Test oracle problem**
- Black-box test generation
  - + Guide test generation
  - + Produce test oracles
  - – **Require *a priori* specs**



Test Selection for Inspection

Based on

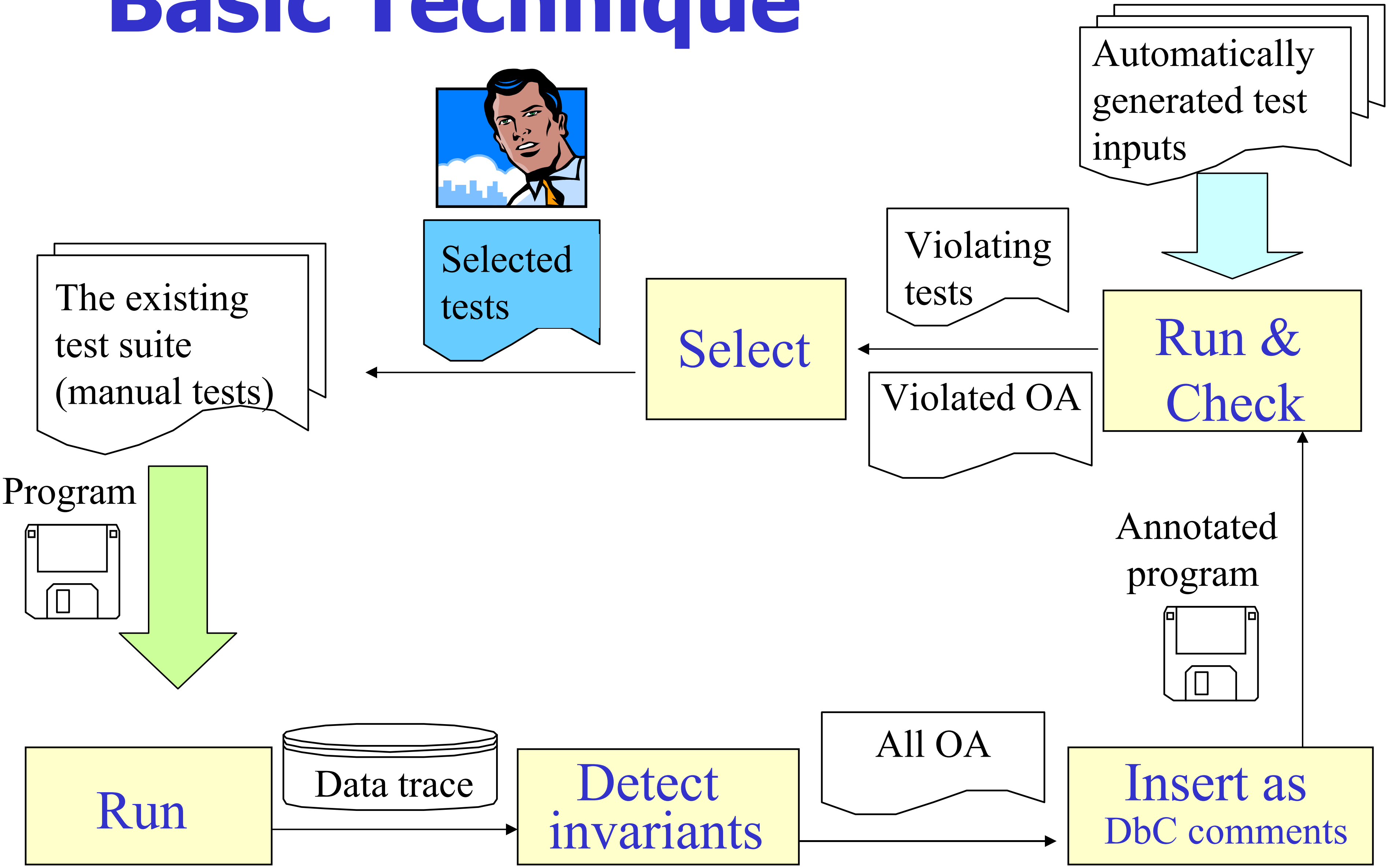
Operational Abstractions



Outline

- Motivation
- Operational Violation Approach
- Experiment
- Related Work
- Conclusion

Basic Technique

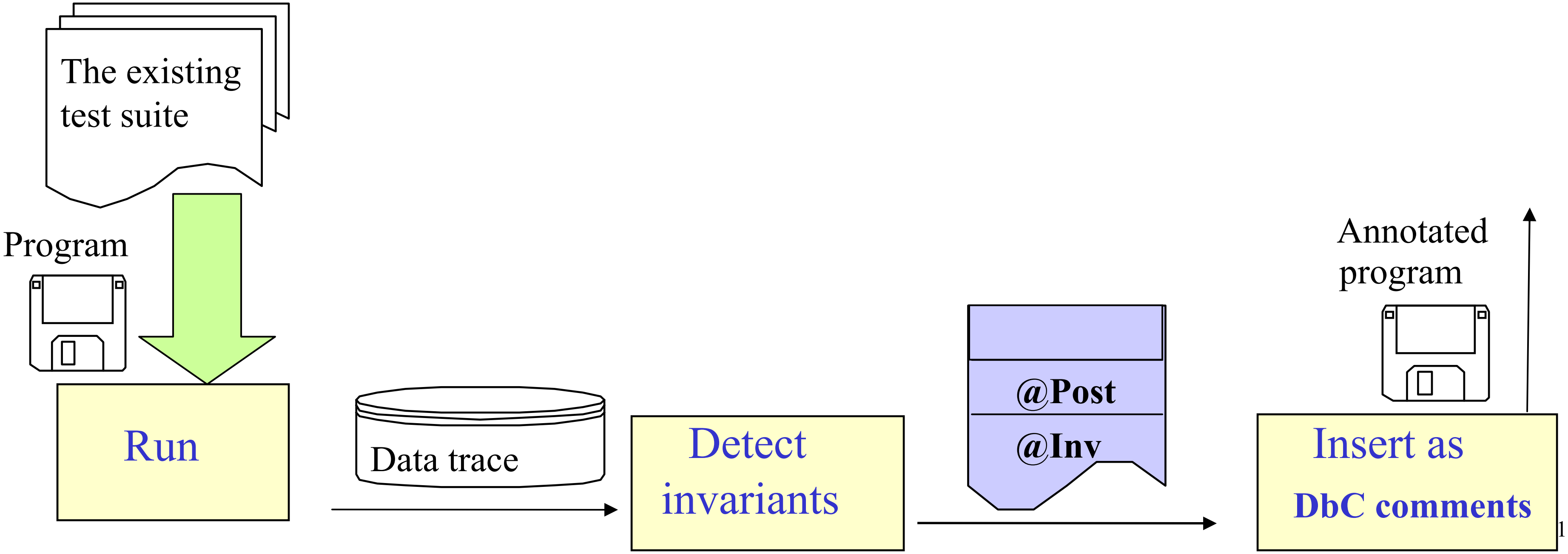


OA: Operational Abstractions

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:

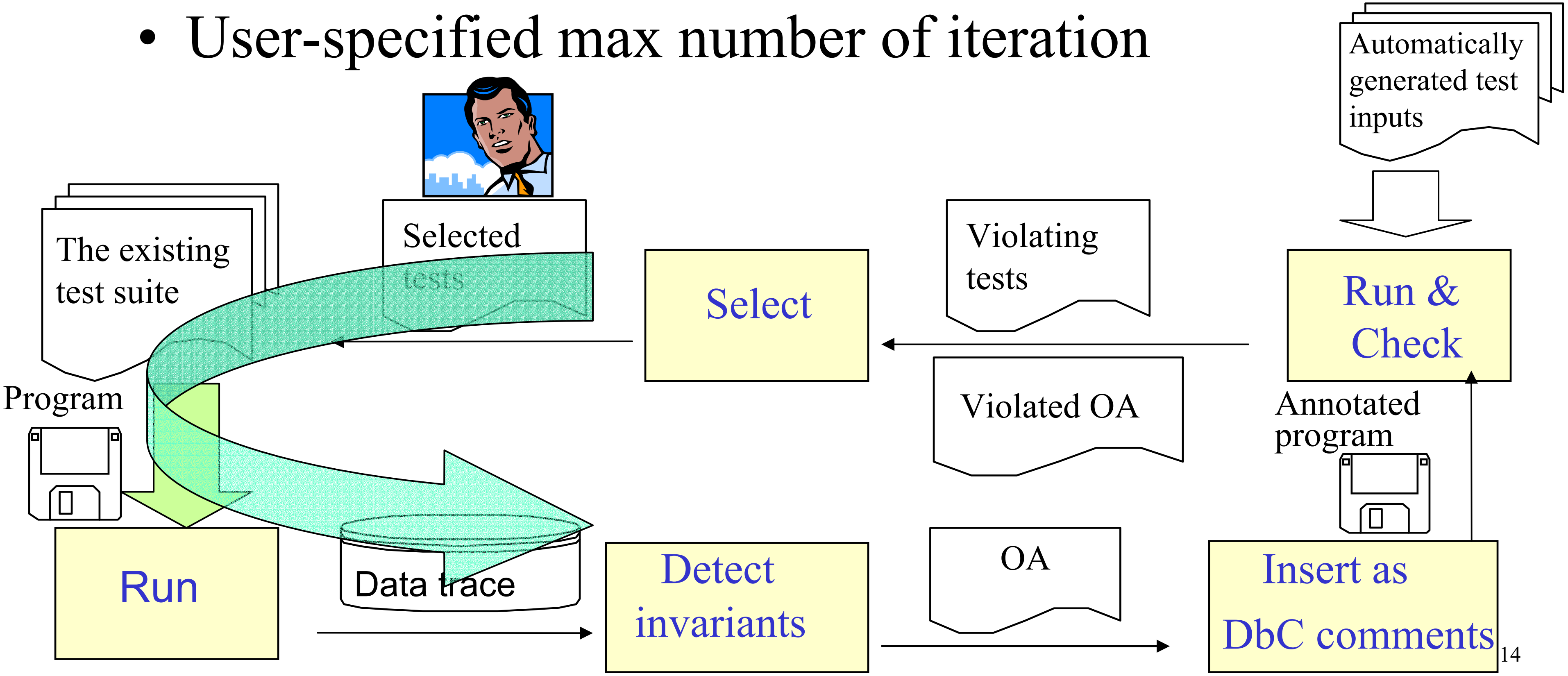
```
@post: [($result == -1) ⇔ (this.numberOfElements == 0)]
```

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



Outline

- Motivation
- Operational Violation Approach
- Experiment
- Related Work
- Conclusion

Subject Programs Studied

- **12** programs from assignments and texts (standard data structures)
  - Total **775** executable LOC in **127** methods

- Accompanying manual test suites
  - ~**94**% branch coverage

Questions to Be Answered

- Is the number of automatically generated tests large enough?
  - if yes, need test selection

- Is the number of tests selected by our approach small enough?
  - if yes, affordable inspection effort

Questions to Be Answered (cont.)

- Do the selected tests by our approach have a high probability of exposing faults?
  - if yes, select a good subset of generated tests

- How does our approach compare with structural test selection approach?
  - Structural approach: select tests that exercise new branch

Measurements

- The number of generated tests without operational abstractions
- The number of selected tests by our approach/structural approach
- The percentage of fault-revealing selected tests by our approach/structural approach
  - Human inspection to determine
  - Also counting illegal inputs that exhibit abnormal behavior, e.g. pop on empty stack leading to invalid object state

Experiment Results

- The number of 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]

- Relatively large for inspection
- Need test selection

Experiment Results (cont.)

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

- Relatively small for inspection
- Require affordable inspection effort
- Our approach selects more tests than structural approach

Experiment Results (cont.)

- The percentage of fault-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

- Relatively high (our approach)
- Select good subset of generated tests
- Our approach complements structural approach

Experiment Results (cont.)

- Jtest's running time on test generation and execution dominates
  - Most programs ~**5** mins
  - But 3 programs **10~20** mins

– Running Jtest several times within each iteration

+ Class- and method-centric

+ Automatic except for human inspection in the end

Experiment Results (cont.)

- Many fault-revealing tests not generated by Jtest without operational abstractions

- Operational abstractions guide the tool to better generate tests

Threats to Validity

- Representative of true practice
  - Subject programs, faults, and tests

- Instrumentation effects that bias the results
  - Faults on tools (integration scripts, Daikon, Jtest)

Outline

- Motivation
- Operational Violation Approach
- Experiment
- Related Work
- Conclusion

Related Work

- Use of operational abstractions
  - Operational Difference [Harder et al. 03] – regression testing
  - DIDUCE [Hangal & Lam 02] – detect the sources of errors
- Specification-based test selection [Chang & Richardson 99]
- Structural test selection/prioritization
  - Residual/additional structural coverage techniques [Pavlopoulou & Young 99][Rothermel et al. 01][Srivastava & Thiagarajan 02]
  - Execution profile clustering/sampling [Dicknson et al. 01]

Outline

- Motivation
- Operational Violation Approach
- Experiment
- Related Work
- Conclusion

Conclusion

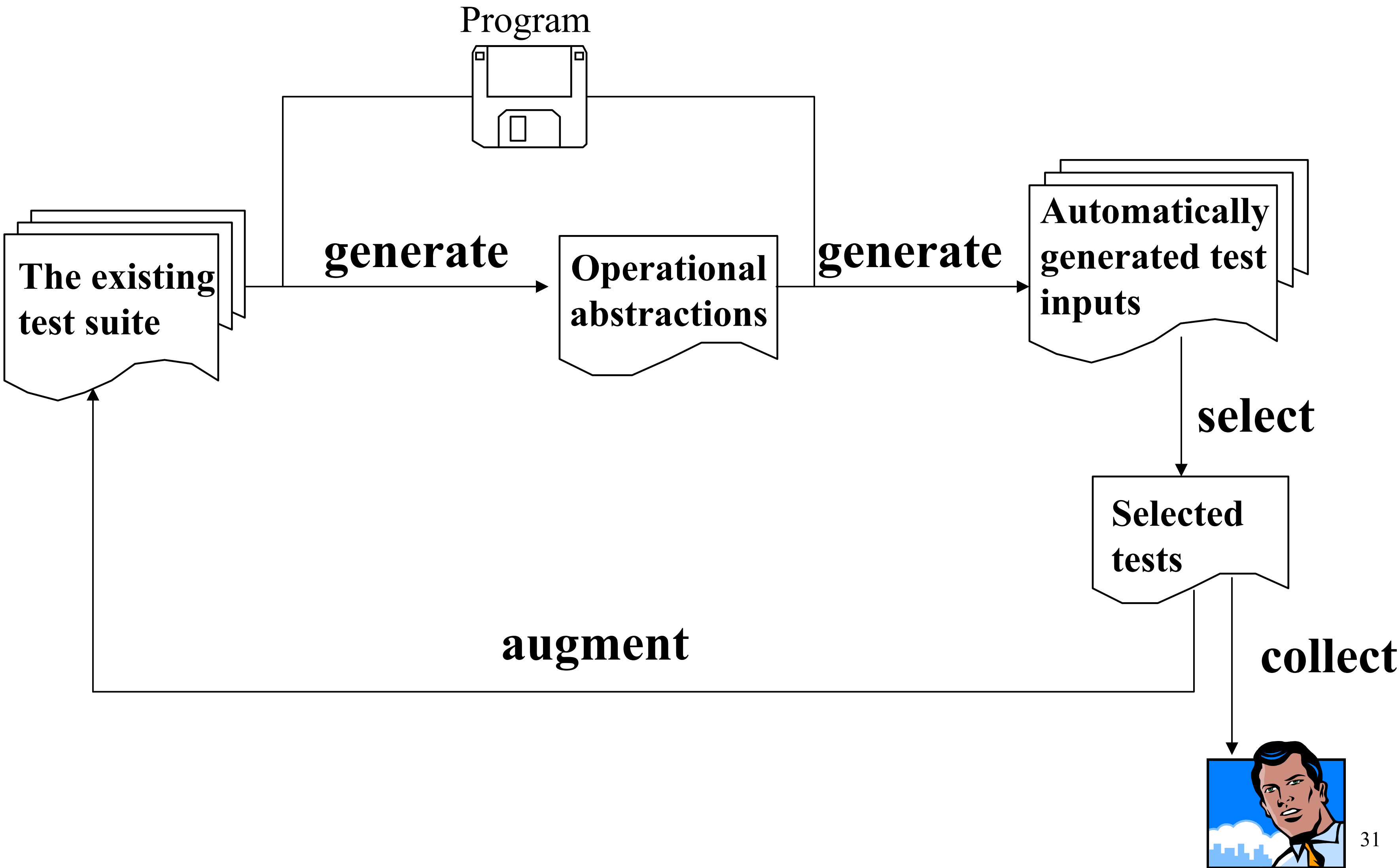
- Operational Abstractions guide Test Generation and Selection for human inspection
  - Basic technique, Precondition removal technique, Iterations
  - Experiment demonstrates its usefulness

In future work:

- Investigate sources of variations affecting cost-effectiveness
- Feedback loop between specification inference and test generation
- Protocol specifications and algebraic specifications

Questions?

Iterations



Program

The existing test suite

generate

Operational abstractions

generate

Automatically generated test inputs

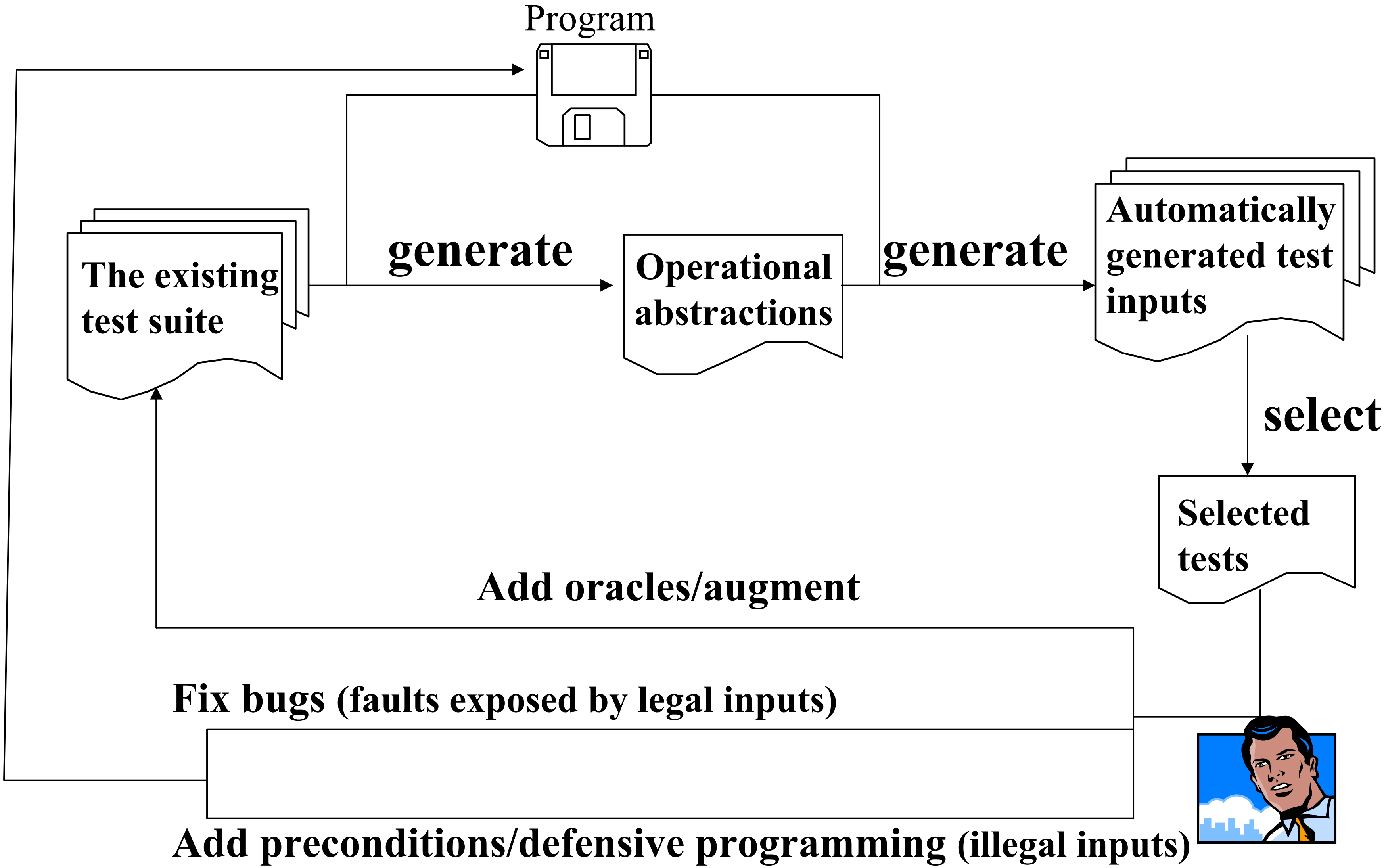
select

Selected tests

collect

augment

Iterations


Program
The existing test suite
generate
Operational abstractions
generate
Automatically generated test inputs
select
Selected tests
Add oracles/augment
Fix bugs (faults exposed by legal inputs)
Add preconditions/defensive programming (illegal inputs)