Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests

[1] Tao Xie [2] Darko Marinov [1] David Notkin

[1] Dept. of Computer Science & Engineering, University of Washington,
[2] MIT Computer Science and Artificial Intelligence Laboratory (UIUC)

23 Sept. 2004
ASE 2004, Linz, Austria

Motivation

- Tool generated test cases
  - Many test cases
  - Important to reduce by eliminating “redundant” test cases
  - Need automation
- Common approach
  - Identify “similar” test cases and eliminate
  - Without reducing “quality” of test suite*
- Object-oriented programs
  - Test case is a sequence of method calls on an object
  - Note: Unit tests only

***Some reduction in fault detection may be tolerated!**

*Some reduction in fault detection may be tolerated!

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 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);
```

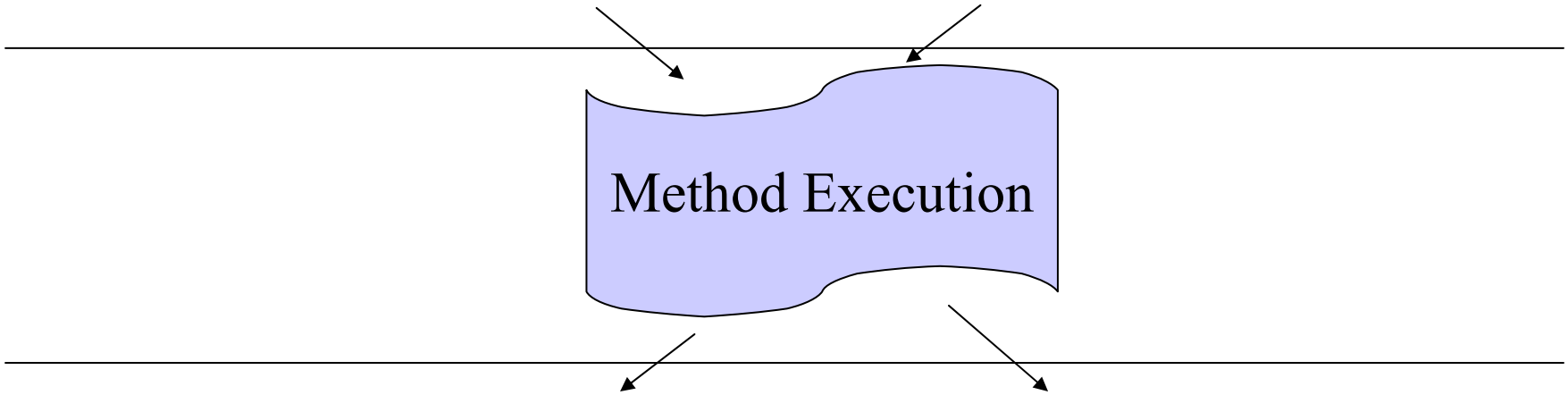
Test 3 (T3):

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

Same inputs ⇒ 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

How to represent object states?

Redundant Test Cases Defined

- Equivalent method executions
  - the same method names, signatures, and input (equivalent object states @entry and arguments)
- Redundant test case:
  - A test case is redundant for a test suite if the test suite has exercised method executions equivalent to all method executions exercised by the test case

Related Work

- State equivalence using observational equivalence [Bernot et al. 91, Doong&Frankl 94, Henkel&Diwan 03]
  - for verifying or inferring algebraic specifications
  - Expensive because of number of sequences
- State equivalence based on user-defined abstraction functions [Grieskamp et al. 02]
  - AsmLT tool for conformance testing
  - Need to define the function

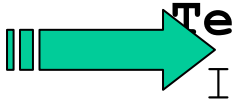
Five State-Representation Techniques

- 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]

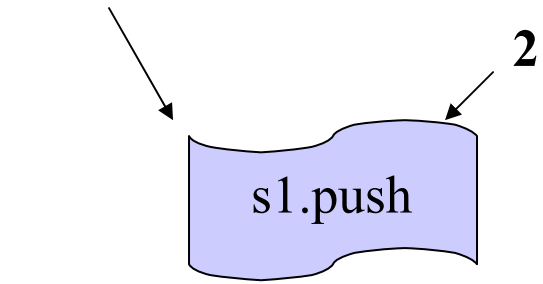
Test 1 (T1):

```
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]

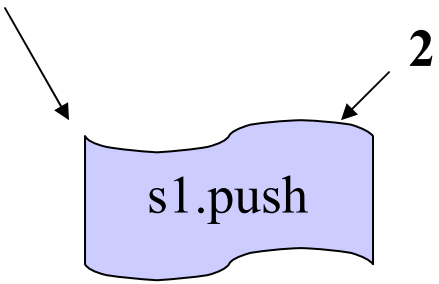
Test 1 (T1):

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

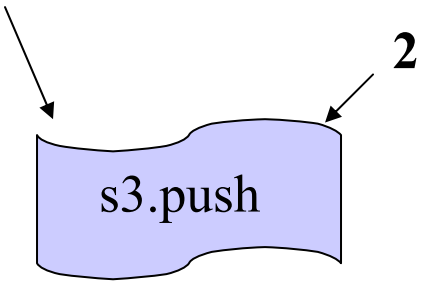
Test 3 (T3):

```
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

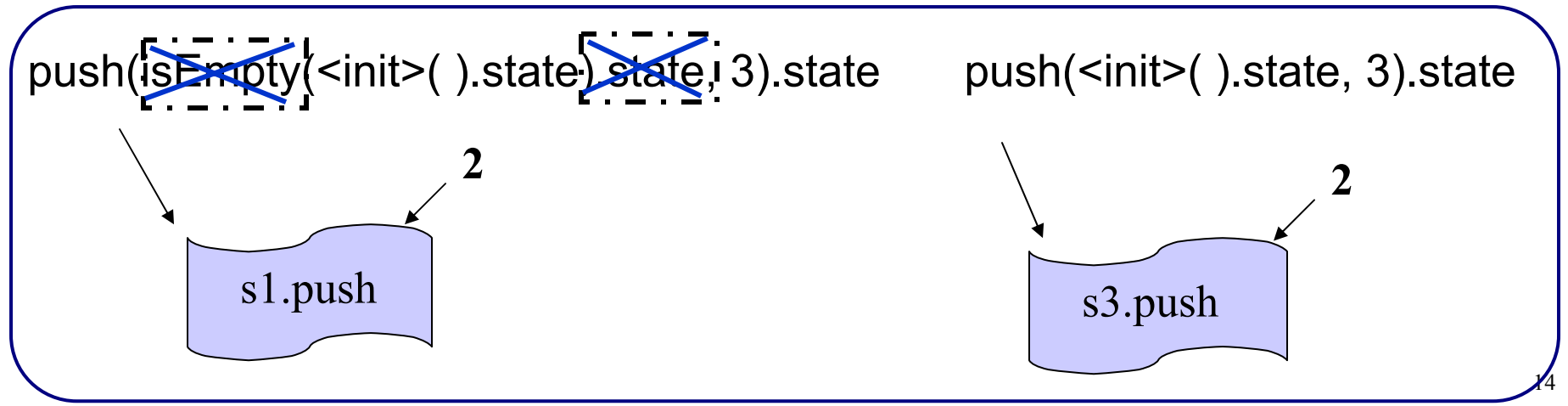
State-modifying method sequences that create objects

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();
```



WholeState Representation

The entire concrete state reachable from the object

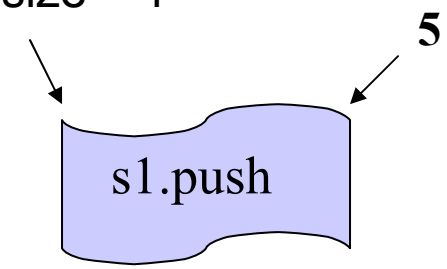
Test 1 (T1):

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

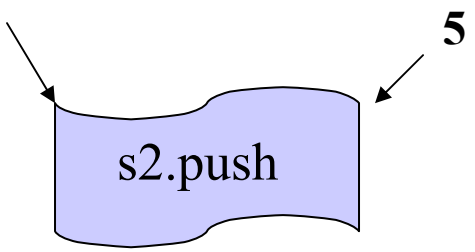
Test 2 (T2):

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

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



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



MonitorEquals Representation

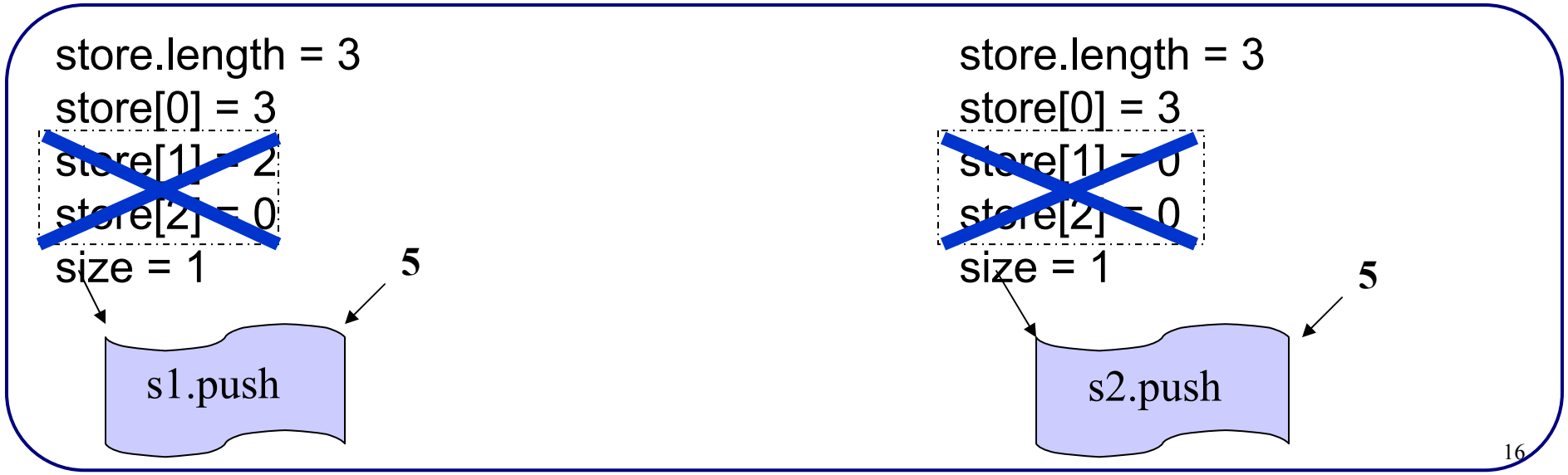
The relevant part of the concrete state defined by *equals* (invoking obj.*equals*(obj) and monitor field accesses)

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);
```



PairwiseEquals Representation

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

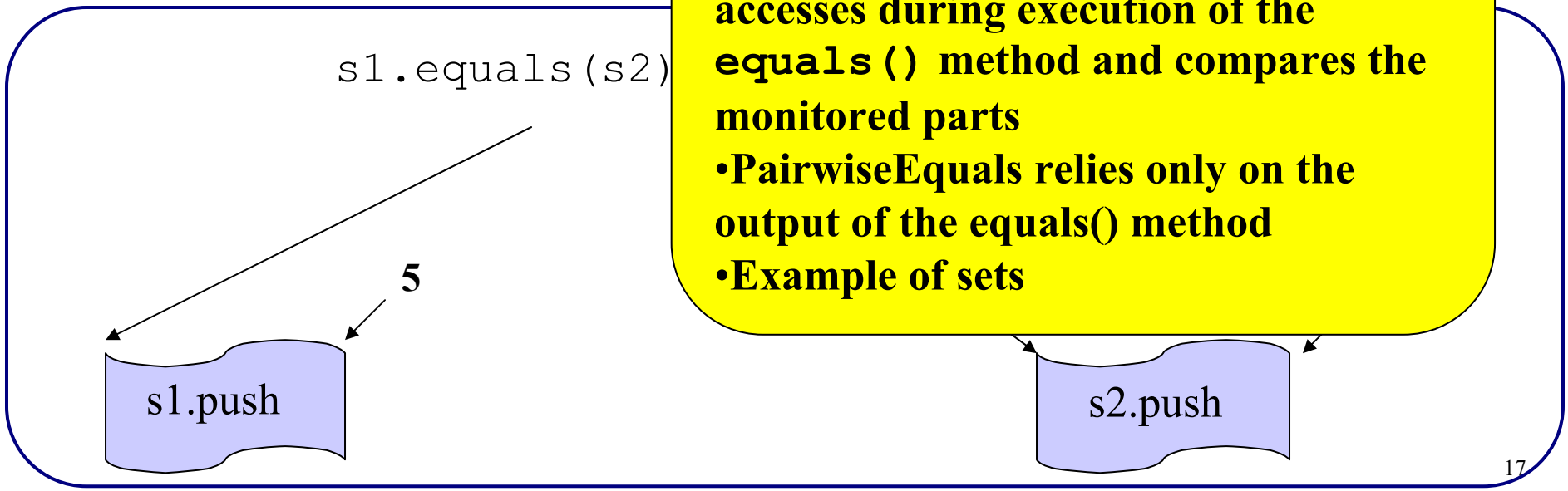
Test 1 (T1):

```
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);
```

- **Fundamental difference between MonitorEquals and PairwiseEquals**
- **MonitorEquals monitors field accesses during execution of the `equals()` method and compares the monitored parts**
- **PairwiseEquals relies only on the output of the equals() method**
- **Example of sets**



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 w.r.t. T1*
WholeSeq	
ModifyingSeq	T3
WholeState	T3
MonitorEquals	T3, T2
PairwiseEquals	T3, T2

Experiment: Evaluated Test Generation Tools

- ParaSoft Jtest 4.5
  - A commercial Java testing tool
  - Generates tests with method-call lengths up to three
- JCrasher 0.2.7
  - An academic robustness testing tool
  - Generates tests with method-call lengths of one

Questions to Be Answered

- How much do we benefit after applying Rostra on tests generated by Jtest and JCrasher?

- Does redundant-test removal decrease test suite quality?

Experimental Subjects

class	*methods*	*public methods*	*ncnb loc*	*Jtest tests*	*JCrasher tests*
IntStack	5	5	44	94	6
UBStack	11	11	106	1423	14
ShoppingCart	9	8	70	470	31
BankAccount	7	7	34	519	135
BinSearchTree	13	8	246	277	56
BinomialHeap	22	17	535	6205	438
DisjSet	10	7	166	779	64
FibonacciHeap	24	14	468	3743	150
HeapMap	27	19	597	5186	47
LinkedList	38	32	398	3028	86
TreeMap	61	25	949	931	1000

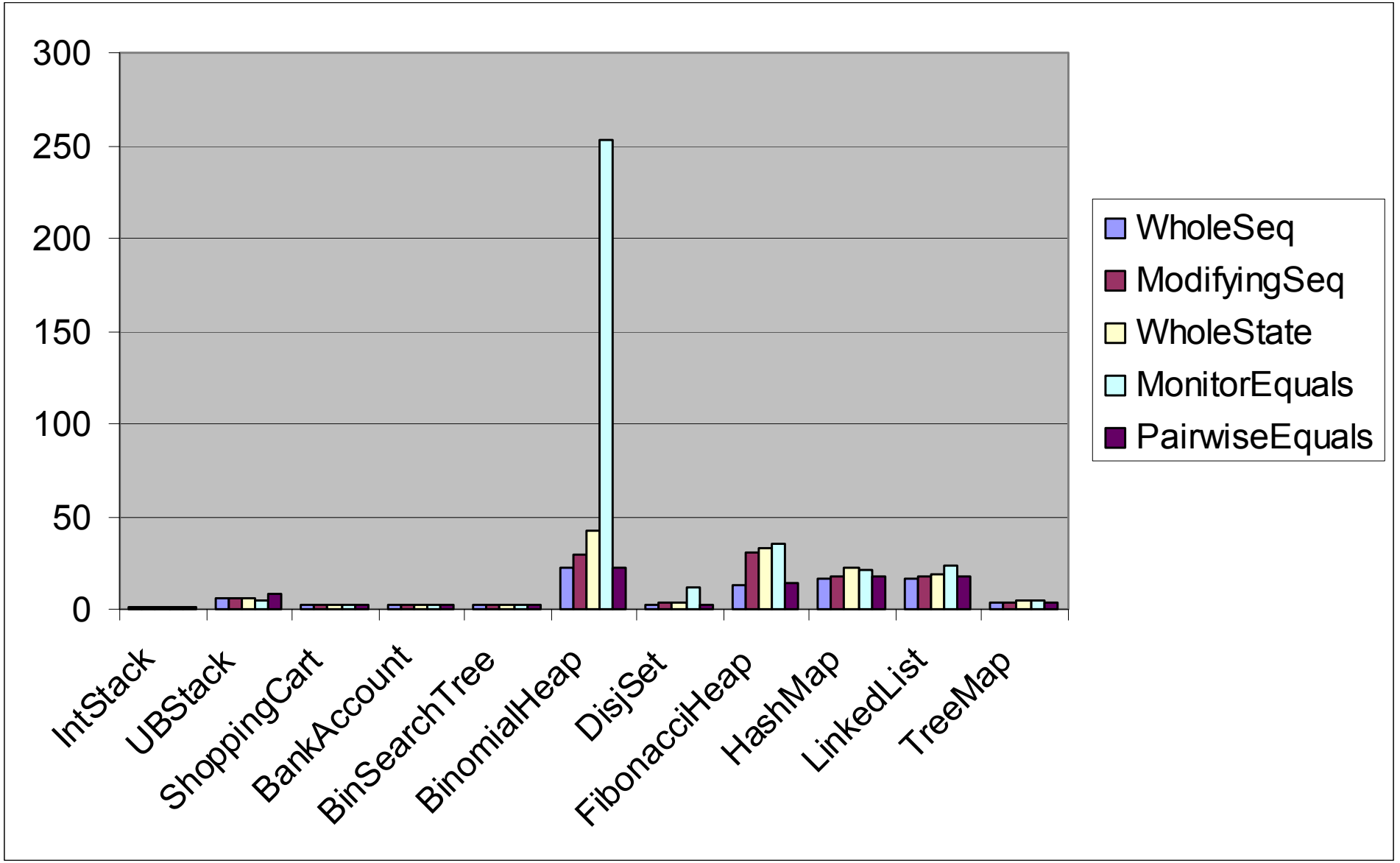
Assumptions About Subjects

- Method-sequence representations assume that each method does not modify argument state
- MonitorEquals and PairwiseEquals representations assume a user-defined `equals()`

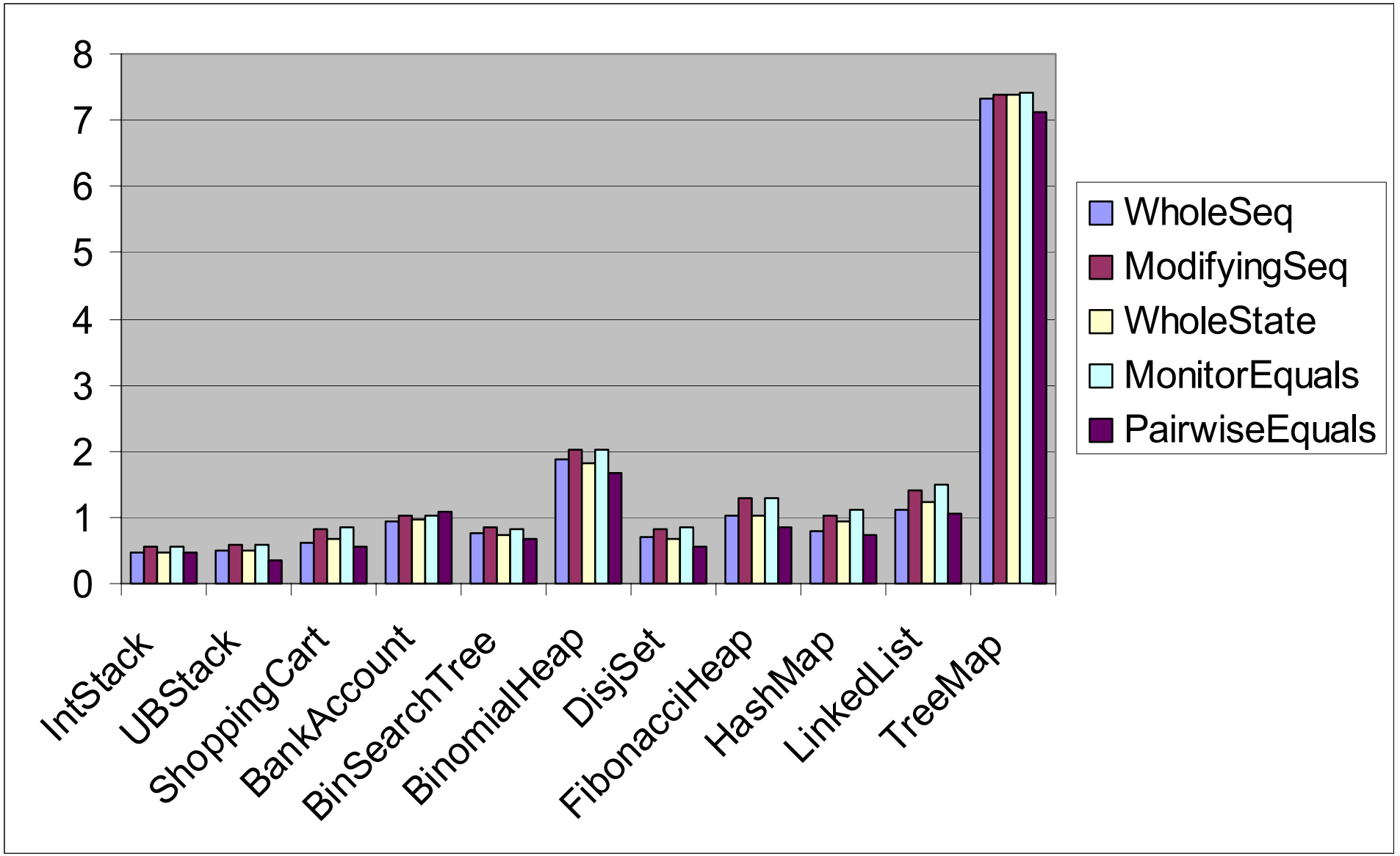
Quality of Original Test Suites

	Jtest-generated tests	JCrasher-generated tests
Avg num uncaught exceptions	4	2
Avg Branch cov	77%	52%
Avg mutant killing ratio (600 mutants)	53%	30%

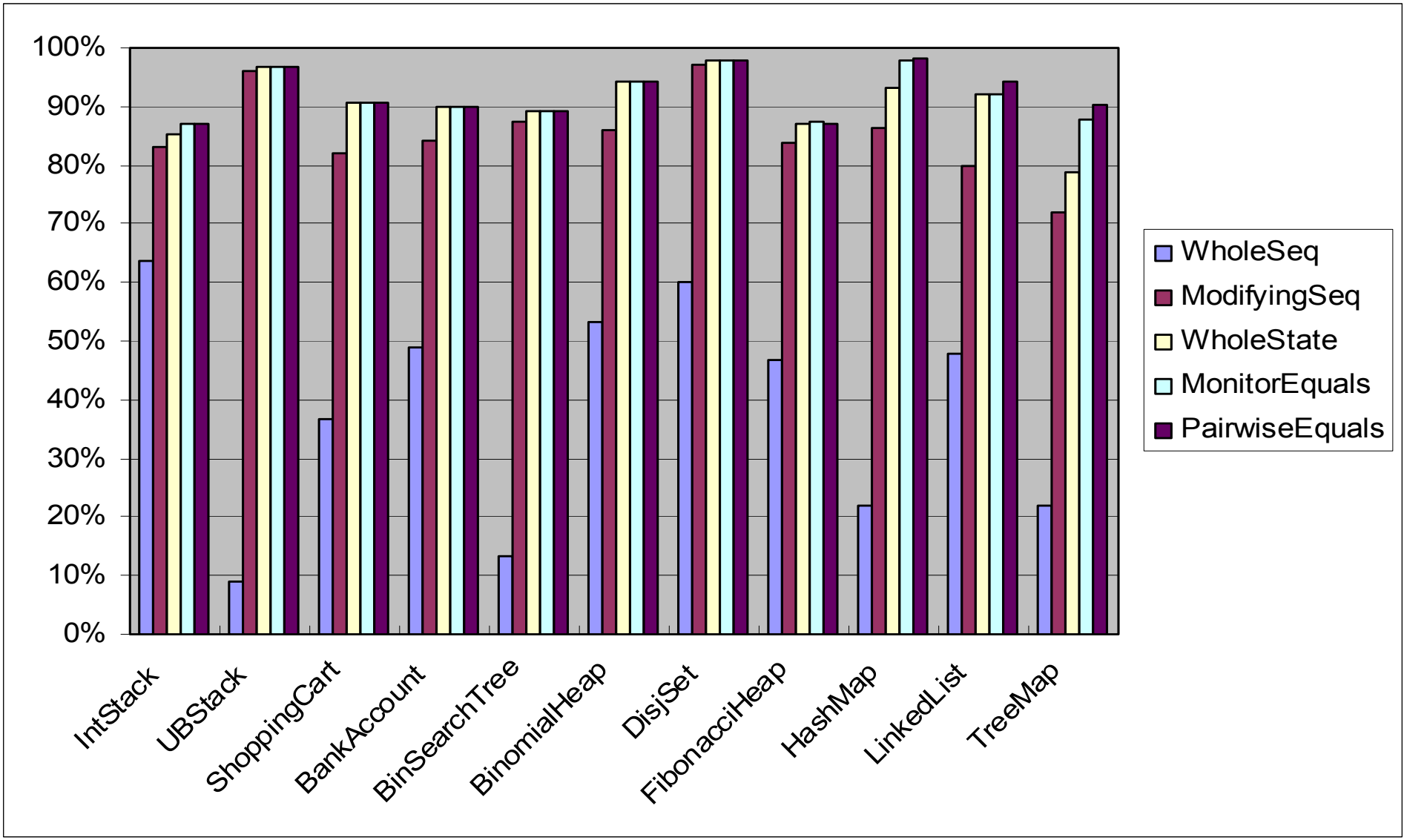
Elapsed Real Time in Minimizing Jtest-Generated Tests (in secs)



Elapsed Real Time in Minimizing JCrasher-Generated Tests (in secs)

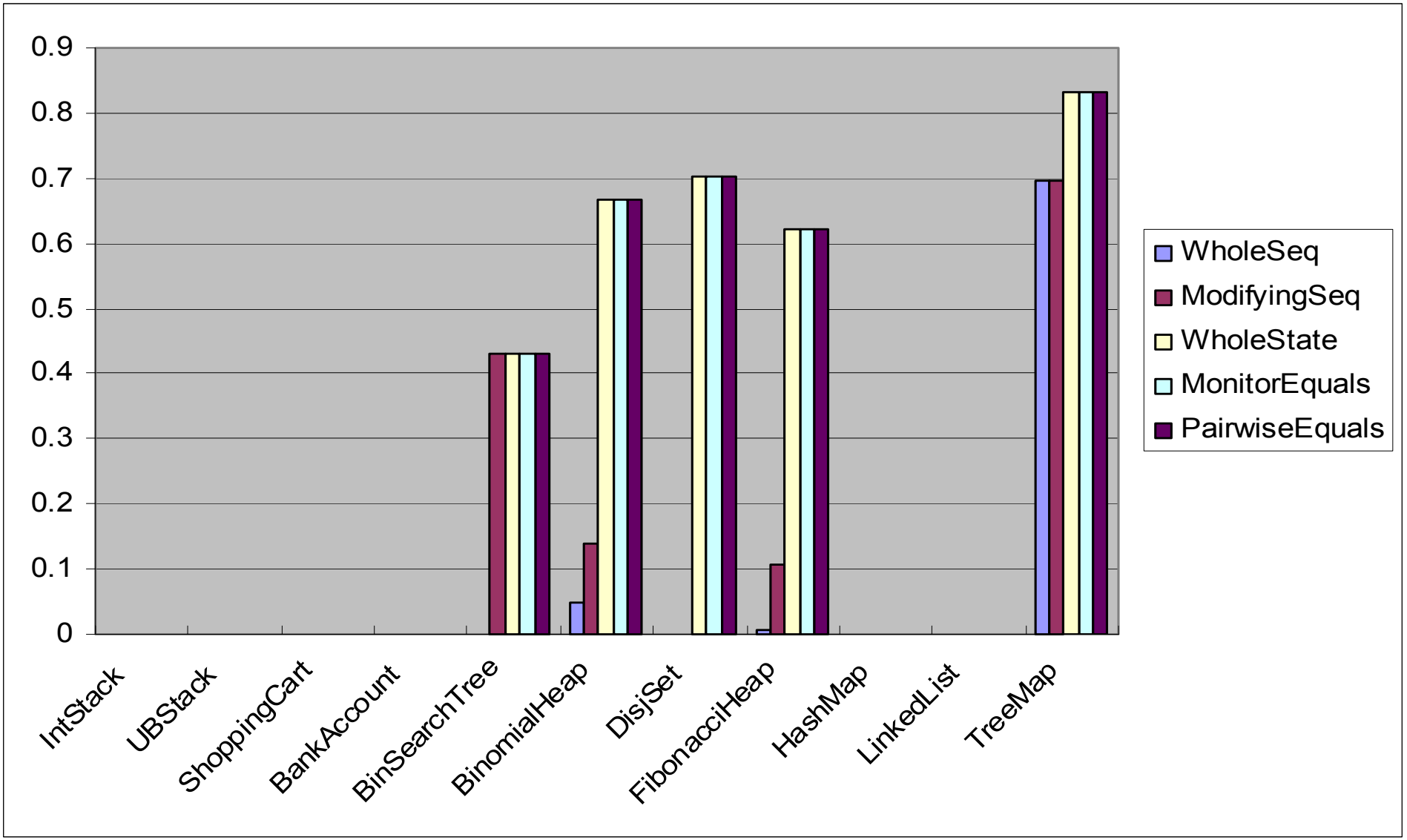


Redundancy among Jtest-generated Tests



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

Redundancy among JCrasher-generated Tests



- **The last three techniques detect over 50% on half subjects**
- **JCrasher generates fewer tests and shorter tests**

Quality of Minimized Test Suites

- All five techniques on JCrasher preserve all measurements
- The first three techniques on Jtest preserve all measurements.
- Two *equals* techniques on Jtest decrease (with only small loss in 2 programs)
  - in branch cov %
  - in mutant killing %

Comparison of Five Techniques

- Time and space taken to find redundant tests
  - from a couple of seconds to several minutes across subjects
  - in roughly increasing order except for pairwiseEquals (being the least expensive)
- The number of redundant tests found
  - in increasing order

Conclusions

- Redundant tests add cost without any benefit
- Existing test generation tools can be potentially improved (by incorporating Rostra framework)
- The experimental results have shown
  - High redundancy among their generated tests
  - Removing them does not decrease test suite quality
- Rostra framework useful in test minimization, assessment, selection, and generation

Evolution of ParaSoft Jtest

- Version 4.5 (released in March 2002) allows method-call lengths (1 — 3) [studied in this work]
- Version 5.0 (released in Feb 2004) allows method-call length of only 1
- ParaSoft notified the authors last week that Version 6.0 (internal version, not yet released) has addressed the test redundancy issue identified by the authors and added back the option to generate long call sequence

Questions?

Threats to Validity

- Representative of true practice?
  - Subject programs, third-party test generation tools
- Instrumentation effects that bias the results
  - Faults on tools (Rostra, Jtest, JCrasher, measurement tools)

Applications

- **Assessment:** compare the quality of different test suites.
- **Selection:** select a subset of automatically generated tests to augment an existing test suite.
- **Minimization:** minimize an automatically generated test suite for correctness inspection and regression executions.
- **Generation:** avoid generating and executing redundant tests

Related Work

- Test selection or minimization with some loss in the quality of test suites [Rothermel et al. 98, Chang&Richardson 99, Harder et al. 03, Xie&Notkin 03]
  - for regression testing or test inspection