NC STATE UNIVERSITY Computer Science

Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking

Tao Xie

Department of Computer Science

North Carolina State University, USA

http://ase.csc.ncsu.edu/

Motivation

- A test case consists of:
  - Test inputs (e.g., method-call sequences)
  - Test oracles (e.g., assertions, specifications)
- Test oracles help expose
  - Faults in the current program version
  - Regression faults in future program versions
- Automatically generated tests complement manually written tests
  - But automatically generated tests lack test oracles

Orstra Approach

- Input: an initial unit-test suite
- Output: an augmented unit-test suite
  to guard against regression faults

- Capture/assert procedure
  - Execute an initial test suite
  - Capture state representations of exercised
    - Receiver objects
    - Method returns
  - Create assertions for asserting
    - Receiver objects
    - Method returns

Example: Bounded Stack Class

```
public class UBStack {
   private Comparable[] elems;
   private int size;
   public UBStack() { ... }
   //standard stack operations
   public void push(Comparable i) { ... }
   public void pop() { ... }
   //stack observer methods
   public int getNumberOfElems() { ... }
   public boolean isFull() { ... }
   public boolean isEmpty() { ... }
   public boolean isMember(Comparable i) {... }
   public MyInput top() { ... }
}
```

Example: Automatically Generated Test

```
public void test1() {
   UBStack s1 = new UBStack();
   MyInput i1 = new MyInput(3);
   s1.push(i1);
   s1.top();
   s1.pop();
   s1.isMember(i1);
}
```

```
public class MyInput implements Comparable {
 private int o;
 public MyInput(int i) { o = i; }
 public boolean equals(Object that) {...}
 ...
}
```

Example: Augmented Test

```
public void testAug1() {
  UBStack s1 = new UBStack();
  ...
  MyInput i1 = new MyInput(3);
  s1.push(i1);
  //start inserting new assertions for observers
  assertEquals(s1.isEmpty(), false);
  assertEquals(s1.isFull(), false);
  assertEquals(s1.getNumberOfElements(), 1);
  MyInput temp_i1 = new MyInput(3);
  assertEquals(s1.isMember(temp_i1), true);
  //finish inserting new assertions for observers

  assertEquals(Runtime.genStateStr(s1.top()), "o:3;");
  s1.pop();
    ...
  assertEquals(s1.isMember(i1), false);
}
```

Outline

- Motivation & overview
- Object-state representations
- Method-execution-behavior assertions
- Regression-oracle augmentation
- Experiment
- Conclusion

Method-Sequence Representation

- Method sequence that produces the object

```
UBStack s1 = new UBStack();
MyInput i1 = new MyInput(3);
s1.push(i1);
```

```
s1 state:
  push(UBStack<init>().state,
                        myInput<init>(3).state).state
```

first argument: receiver of non-static, non-constructor method

`.state`: the state of the receiver after the invocation

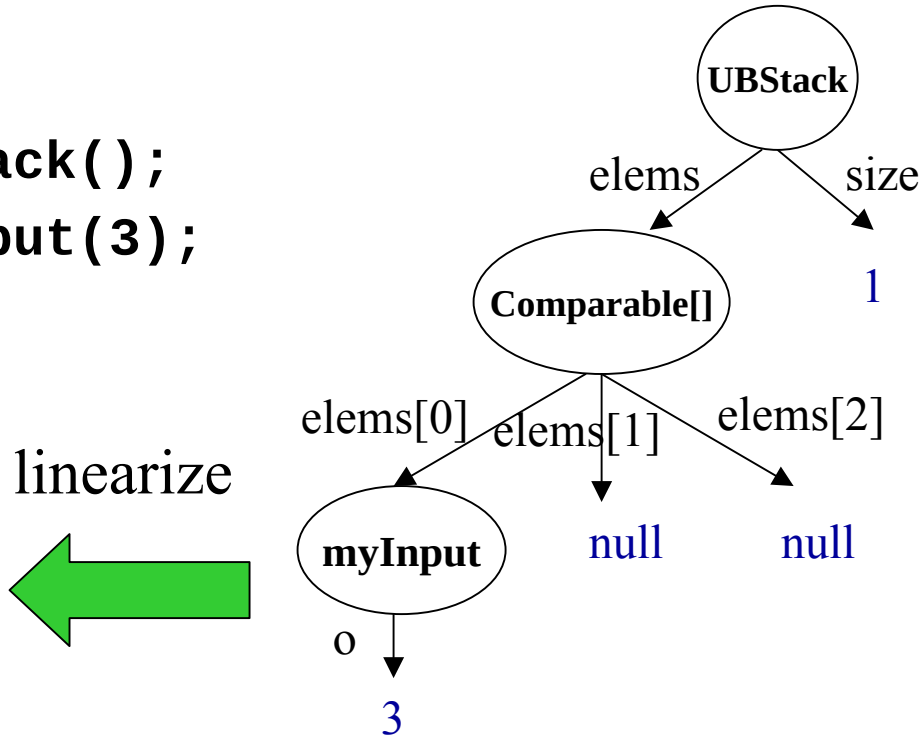
`.retval`: the return of the invocation

[Henkel&Diwan ECOOP'03, Xie+ ASE'04]

Concrete-State Representation

- Parts of the program heap that are reachable from the object

```
UBStack s1 = new UBStack();
MyInput i1 = new MyInput(3);
s1.push(i1);
```



linearize

```
s1 state:
 elems:o:3;,n,n;size:1
```

[Boyapati+ ISSTA'02, Xie+ ASE'04]

Observer-Abstraction Representation

- Return values of observer invocations on the object
  - an observer: method whose return type is not void

```
UBStack s1 = new UBStack();
MyInput i1 = new MyInput(3);
s1.push(i1);
```

s1 state:

observer invocation	ret value
isEmpty()	false
isFull()	false
getNumberOfElems()	1
isMember(new MyInput(3))	true
top()	"o:3;"

[Xie&Notkin ICFEM'04]

Test Execution

- The execution of a test produces a sequence of method executions
- A method execution comprises
  - Method name (+signature)
  - Inputs at method entry
    - Receiver-object state
    - Argument-object state
  - Outputs at method exit
    - Receiver-object state S_{exit}
    - Method-return value r

```
UBStack s1 = new UBStack();
MyInput i1 = new MyInput(3);
        s1.push(i1);
```

Asserting Receiver-Object State S_{exit} – I

- If another method sequence can be found to produce an object state $S' \equiv S_{exit}$, create an assertion to compare the state representations of S' and S_{exit}.

```
UBStack s1 = new UBStack();
MyInput i1 = new MyInput(3);
s1.push(i1);
s1.top();
s1.pop();
//start inserting new assertions for state equivalence
UBStack temp_s1 = new UBStack();
EqualsBuilder.reflectionEquals(s1, temp_s1);
```

Asserting Receiver-Object State S_{exit} – II

- If there is an observer method *ob*, create an assertion to compare the return of an *ob* invocation on S_{exit} with the expected value

```
UBStack s1 = new UBStack();
MyInput i1 = new MyInput(3);
s1.push(i1);
//start inserting new assertions for observers
assertEquals(s1.isEmpty(), false);
assertEquals(s1.isFull(), false);
assertEquals(s1.getNumberOfElements(), 1);
MyInput temp_i1 = new MyInput(3);
assertEquals(s1.isMember(temp_i1), true);
assertEquals(Runtime.genStateStr(s1.top()), "o:3;");
```

Asserting Method-Return Value *r*– I

- If *r* is of a primitive type, create an assertion to compare `r` with the expected primitive value

```
UBStack s1 = new UBStack();
MyInput i1 = new MyInput(3);
s1.push(i1);
s1.top();
s1.pop();
//start inserting new assertion for isMember
assertEquals(s1.isMember(i1), false);
```

Asserting Method-Return Value r– II

- If r is of the class-under-test type, create an assertion by using the ways of asserting a receiver-object state S_{exit}

```
UBStack s1 = new UBStack();
//start inserting new assertions for observers
assertEquals(s1.isEmpty(), true);
assertEquals(s1.isFull(), false);
assertEquals(s1.getNumberOfElements(), 0);
MyInput temp_i1 = new MyInput(3);
assertEquals(s1.isMember(temp_i1), false);
assertEquals(s1.top(), null);
```

Asserting Method-Return Value *r*– III

- If *r* is of a non-primitive type *R*, create an assertion to compare *r*'s concrete-state representation string with the expected value

```
UBStack s1 = new UBStack();
MyInput i1 = new MyInput(3);
s1.push(i1);
//start inserting new assertion for isMember
assertEquals(Runtime.genStateStr(s1.top()), "o:3;");
```

Automatic Regression-Oracle Augmentation

- State-capturing phase
  - Instrument the class under test & test class
  - Run the test class
    - Collect three types of state representations for objects
- Assertion-building phase
  - Insert assertions into the test code
  - Output the augmented test code in JUnit

Experiment –Question to be Answered

- Can Orstra improve the fault-detection capability of an automatically generated test suite?

Experiment – Test Generation Tools

- ParaSoft Jtest 4.5
  - A commercial Java testing tool
  - Generates tests with method-call lengths up to three
  - Relies on uncaught exceptions and simple assertions on primitive returns of observers
- JCrasher 0.2.7
  - An academic robustness testing tool
  - Generates tests with method-call lengths of one
  - Relies only on uncaught exceptions

Experiment – Benchmarks

class	meths	public meths	ncnb loc	Jtest tests	JCrasher tests	faults
IntStack	5	5	44	94	6	83
UBStack	11	11	106	1423	14	305
ShoppingCart	9	8	70	470	31	120
BankAccount	7	7	34	519	135	42
BinSearchTree	13	8	246	277	56	309
BinomialHeap	22	17	535	6205	438	310
DisjSet	10	7	166	779	64	307
FibonacciHeap	24	14	468	3743	150	311
HashMap	27	19	597	5186	47	305
LinkedList	38	32	398	3028	86	298
TreeMap	61	25	949	931	1000	311

Experiment – Measures

- Fault-exposure ratio (FE):

$$\frac{\#\text{faults detected by the test suite}}{\#\text{total faults.}}$$

- Improvement factor

$$\frac{FE_{aug} - FE_{orig}}{FE_{orig}}$$

Experiment – Results

class	JCrasher-gen tests			Jtest-gen tests		
	orig	aug	improve	orig	aug	improve
IntStack	9%	40%	3.36	47%	47%	0.00
UBStack	39%	53%	0.36	60%	60%	0.00
ShoppingCart	0%	48%	∞	56%	56%	0.00
BankAccount	0%	98%	∞	98%	98%	0.00
BinSearchTree	8%	20%	1.58	20%	27%	0.34
BinomialHeap	18%	95%	4.19	85%	95%	0.12
DisjSet	23%	31%	0.36	26%	43%	0.65
FibonacciHeap	9%	96%	9.28	55%	96%	0.74
HashMap	14%	76%	4.30	22%	76%	2.43
LinkedList	7%	35%	3.73	45%	45%	0.01
TreeMap	2%	89%	54.40	12%	89%	6.29
Average	12%	62%	9.06	48%	67%	0.96
Median	9%	53%	3.55	47%	60%	0.12

Related Work

- Specifications serve as test oracles [Richardson 94, Cheon&Leavens 02, Boyapati+ 02, …]
- Program crashes or uncaught exceptions as fault symptoms [Csallner&Smaragdakis 04/05]
- Automatic test factoring [Saff+ 05] and selective capture&replay [Orso&Kennedy 05]
- Program-spectrum comparison [Harrold+ 00, Xie&Notkin 05]
- Test selection for inspection [Xie&Notkin 03]
- Observer abstraction for test-behavior inspection [Xie&Notkin 04]

Conclusion

- Test oracles help expose
  - Faults in the current program version
  - Regression faults in future program versions
- Automatically generated tests complement manually written tests
  - But automatically generated tests lack test oracles
- Orstra augments generated tests by inserting assertions that guard against regression faults
- Experimental results show that Orstra can effectively increase the fault-detection capability of generated tests

Questions?

Discussion

- Analysis cost
  - `|assertions| = O(|nonEqvStates| × |observers|+ |statesEqvToAnother|)`
- Fault-free behavioral changes, e.g.,
  - bug-fixing changes
  - nondeterministic or different behaviors across runs
- (Un)availability of observers
  - Still can compare receiver-object states
- Iterations of augmentation
  - invoking state-modifying observer methods → new object states

Discussion (cont.)

- Quality of automatically generated unit-test suites
  - Higher quality → higher improvement factor
- Augmentation of other types of test suites
  - Integration tests
  - Manually generated test suite
- Incorporation of oracle augmentation in test generation
  - State capturing phase (test generation+execution)
  - Assertion-building phase

Comparison of State Representations

- Method-sequence representation strictly subsumes concrete-state representation.
- Concrete-state representation strictly subsumes the observer-abstraction representation
- Among different observers, the representation resulting from the toString() observer
  - often subsumes the representation resulting from other observers
  - is often equivalent to the concrete-state representation.

Comparison of State Representations (cont.)

- Concrete-state representation exposes more implementation details than the other two representations
- Concrete-state representation is sensitive to changes on the object's field structure or the semantic of its fields, even if these changes do not cause any behavioral difference in the object's interface.

Test Execution

- The execution of a test produces a sequence of method executions
- A method execution comprises

```
UBStack s1 = new UBStack();
MyInput i1 = new MyInput(3);
       s1.push(i1);
```

  - Method name (+signature) • `UBStack.push(MyInput)`
  - Method entry (inputs)
    - Argument-object state • `o:3;`
    - Receiver-object state • `elems:n,n,n;size:0`
  - Method exit (outputs)
    - Receiver-object state S_{exit} • `elems:o:3;,n,n;size:1`
    - Argument-object state • `o:3;`
    - Method-return value *r* • `N/A`