Mutually Enhancing Test Generation and Specification Inference

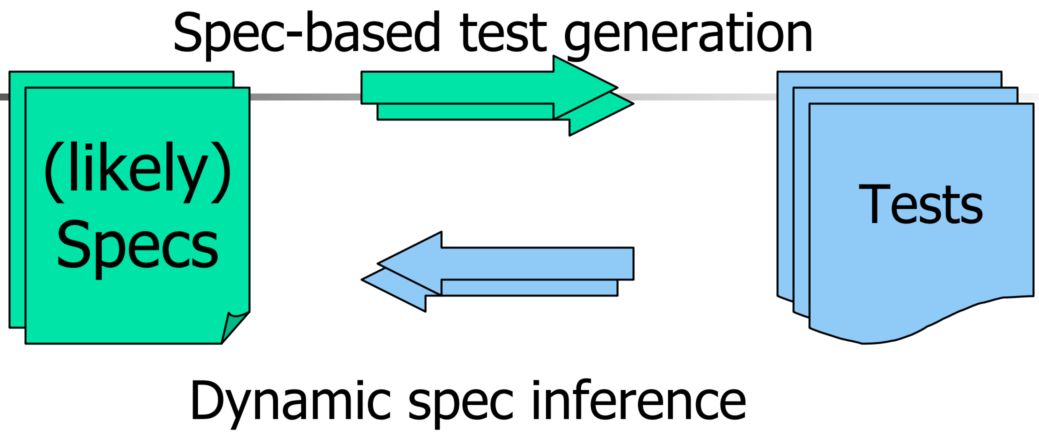
Tao Xie David Notkin

Department of Computer Science & Engineering
University of Washington

August 15th, 2003

Foundations of Software Engineering, Microsoft Research

Synopsis



- Need specs for (many kinds of) test generation
- Need tests for dynamic spec inference
- We have applied feedback loop between these approaches that
  - aids in test generation (improving specs and helping in producing oracles)
  - aids in spec inference (improving the underlying test suites)

Outline

- Background
- Feedback Loop between Test Generation and Spec Inference
    - Axiomatic Spec Inference and Test Generation
    - Algebraic Spec Inference and Test Generation
- Conclusion

Background – Test Generation

- White-Box Test Generation
  - Jtest [ParaSoft] …

  + Cover structural entities, e.g. statement, branch, path.

  - Test oracle problem
    - Rely on uncaught runtime exceptions

- Black-Box Test Generation
  - Korat [Boyapati et al.02], AsmL [Grieskamp et al. 02], Jtest…

  + Use specs to guide test generation

  + Use specs as test oracles

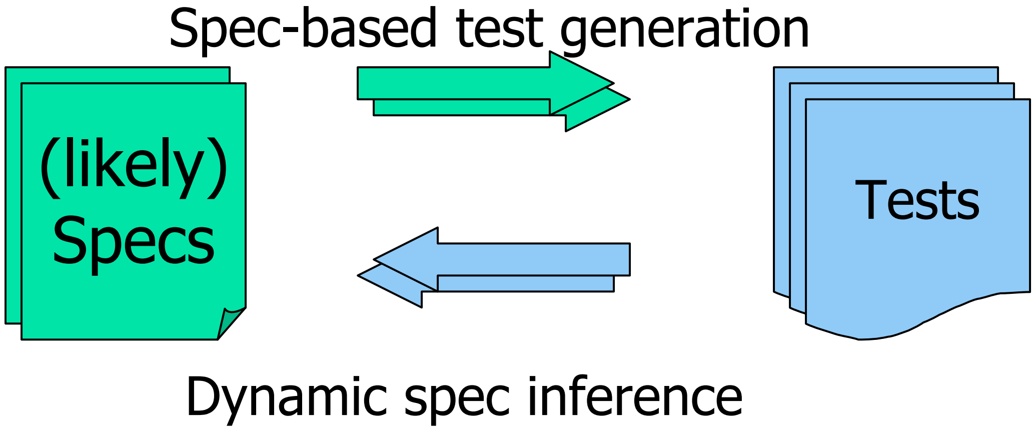
  - Require *a priori* specs

Background – Dynamic Spec Inference

- Axiomatic specification inference
  - Daikon [Ernst et al. 01]
- Algebraic specification inference
  - [Henkel & Diwan 03]
- Protocol specification inference
  - Strauss [Ammons et al. 02], Hastings [Whaley et al. 02]

Quality of analysis depends on quality of tests

Background – Circular Dependency



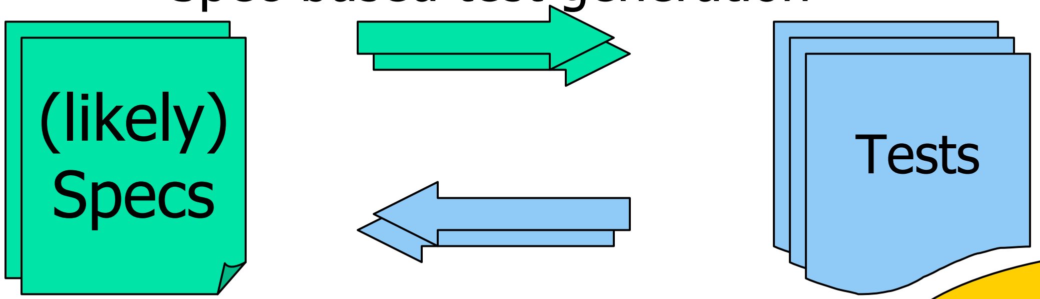
- Circular dependency: test generation and spec inference
- Win-win feedback loop:
  - Better spec ←→ better tests?

Outline

- Background
- Feedback Loop between Test Generation and Spec Inference
  - Axiomatic Spec Inference and Test Generation
  - Algebraic Spec Inference and Test Generation
- Conclusion

Feedback Loop

Spec-based test generation



Dynamic spec inference

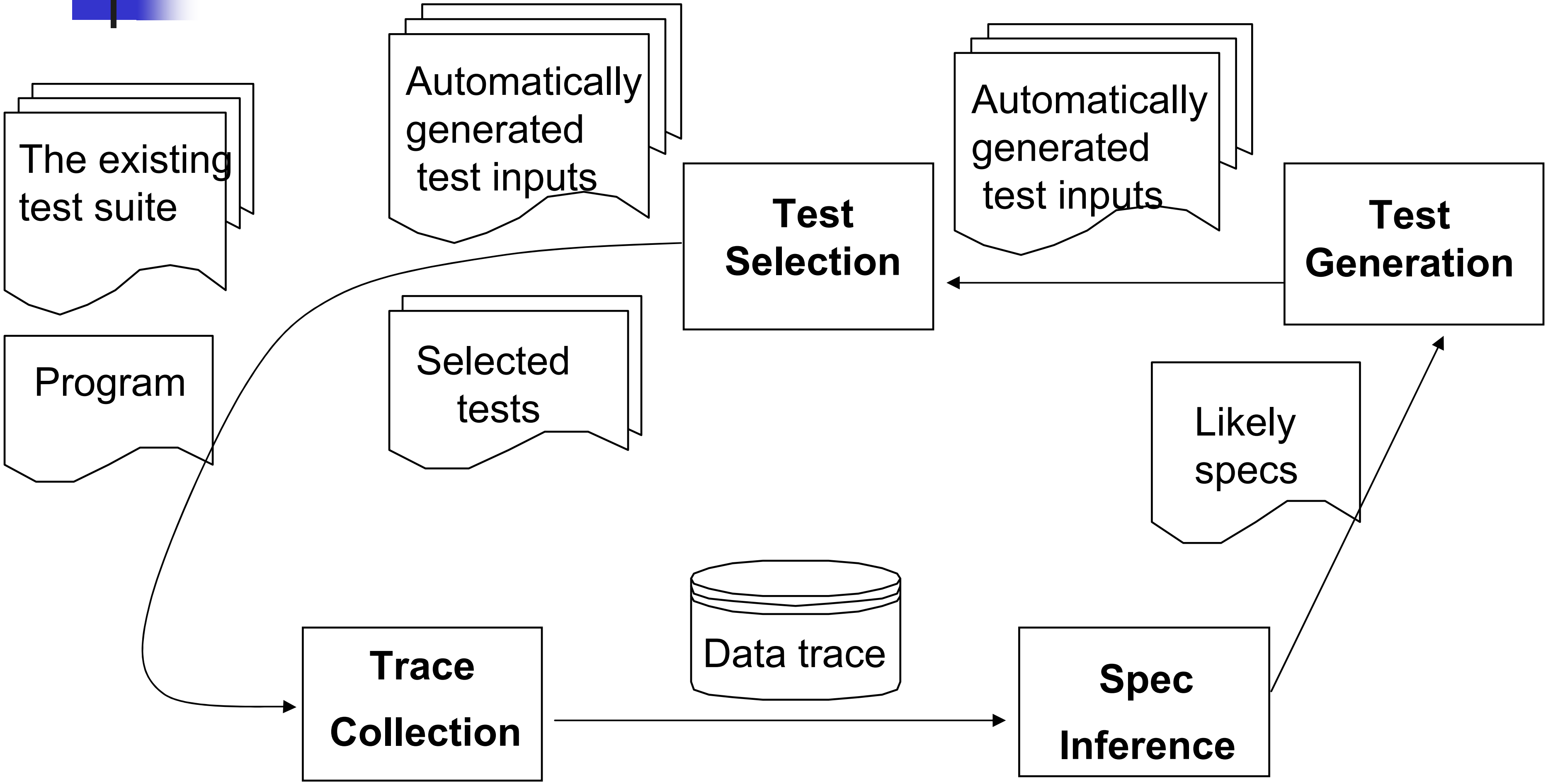
- Inferred Specs → Test Generation
  - Reduce the scope of analysis
- Generated Tests → Spec Inference
  - Verify/refine the inferred specs
- Spec-Violating Tests → Test Selection
  - Inspection and test augmentation

Lack of Specs Problem

Insufficient Test Problem

Test Oracle Problem

Feedback Loop Framework



Outline

- Background
- Feedback Loop between Test Generation and Spec Inference
    - Axiomatic Spec Inference and Test Generation
    - Algebraic Spec Inference and Test Generation
- Conclusion

Feedback Loop between Axiomatic Spec Inference and Test Generation

[ASE 03]

- Trace collection (Daikon Java front-end)
- Spec inference (Daikon)
- Test generation (Jtest)
- Test selection
- Iterations

Trace Collection & Axiomatic Spec Inference

- Trace collection
  - Method entry point: args, obj fields
  - Method exit point: return, updated args, obj fields
- Spec inference
  - Look for patterns and relationships among values, e.g. $x < a ` a < x < b ` y / a x \rightarrow b \triangleright$
  - Preconditions, postconditions, and class invariants

Axiomatic Spec-Based Test Generation

- Black-box test generation based on Design by Contract (DbC) comments (Jtest)
  - Generates and executes test inputs
    - Ex: for a 11-method `uniqueBoundedStack` class with 47 LOC
      - Call length 1: 14 tests (63% statement cov.)
      - Call length 2: 96 tests (86% statement cov.)
      - Call length 3: 1745 tests (86% statement cov.)
  - Problem suppression for inputs violating the preconditions
  - Both preconditions and postconditions have impacts on test generation

Test Generation Issue: Over-Constrained Preconditions

- Too restrictive preconditions may leave (maybe important) legal inputs untested
- Solution: precondition guard removal
- New problem: allow illegal inputs
  - But only report postcondition-violating or exception-throwing illegal inputs
- Alternatives: precondition guard relaxation?

Test Selection

- Select tests violating at least one inferred postcondition.
- Inspect them:
  - illegal inputs:
    - Adding preconditions or defensive programming
  - legal inputs:
    - Fault exposure: bug fixing and regression test suite augmentation
    - Normal, but new feature exercising: regression test suite augmentation

- Complementary technique: Select tests exercising at least one new structural entity.

Specification Violation - Example

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

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

```
top: @post: [($result == -1) == (this.numberOfElements == 0)]
is violated by input:
 uniqueBoundedStack THIS = new uniqueBoundedStack ();
 THIS.push (-1);
 int RETVAL = THIS.top ();
```

Iterations

- Iterates until reaching a fixed point (no violations)

- In the next iteration, spec inference is based on:
  - the existing test suite augmented by
    - new violating tests
    - all generated tests

Experiment – Subject Programs

Jtest method call length: 2

Programs	#Public Methods	#LOC	#Manual-tests	#Jtest-tests
UB-Stack (JUnit)	11	47	8	96
UB-Stack (JAX)	11	47	15	96
RatPoly-1	13	161	24	223
RatPoly-2	13	191	24	227
RatPolyStack-1	13	48	11	128
RatPolyStack-2	12	40	11	90
BinaryHeap	10	31	-	166
BinarySearchTree	16	50	-	147
DisjSets	4	11	-	24
QueueAr	7	27	-	120
StackAr	8	20	-	133
StackLi	9	21	-	99

Experiment – Results

With Preconds: basic tech W/O Preconds: precond removal tech

#SelT: #Selected tests #FRT: #Fault-revealing tests

Programs	Iteration 1				Iteration 2				Iteration 3			
	With Preconds		W/O Preconds		With Preconds		W/O Preconds		With Preconds		W/O Preconds	
	#SelT	#FRT	#SelT	#FRT	#SelT	#FRT	#SelT	#FRT	#SelT	#FRT	#SelT	#FRT
UBS (JUnit)	1		15	5	2		6	1			1	
UBS (JAX)	3		25	9			4					
RatPoly-1	2	2	1	1								
RatPoly-2	1	1	1	1	1	1						
RatPolyStack-1			12	8			5	2			1	
RatPolyStack-2	1		10	7			2					
……												
Median of #FRT/ #SelT	20%		68%		0%		17%		_		0%	

- #Selected tests are not too large (affordable to inspect)
- #Selected tests have high probability of exposing a fault or indicating a necessary precondition
- A couple of iterations are good enough

Outline

- Background
- Feedback Loop between Test Generation and Spec Inference
  - Axiomatic Spec Inference and Test Generation
  - Algebraic Spec Inference and Test Generation
- Conclusion

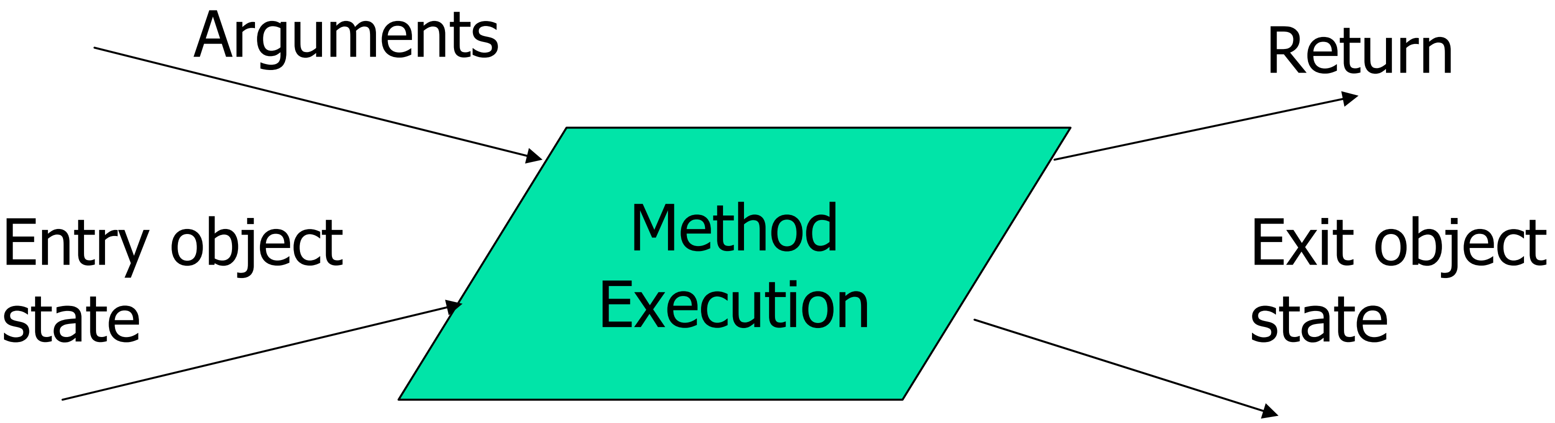
Feedback Loop between Algebraic Spec Inference and Test Generation

- Trace collection
- Spec inference
- Test generation
- Test selection

- Iterations

Trace Collection

- Object = data + operations



- Trace data:
  - Method entry point: args, **entry object state**
  - Method exit point: return, **exit object state**

Object State Collection - Challenges

- Simply outputting (all) object field values doesn’t work
  - Which object fields of ancestor classes are relevant?
  - Which object fields of the current class are relevant?
  - How deep shall we track referencing object fields?

Object State Collection - Solution

- We developed a tracing front-end based on BCEL
- Require a pre-defined "equals" method
  - Instrument *"this.equals(this)*" at public method entry and exit points.
  - Collect the object field values accessed within *"this.equals(this)*".
  - Sort these object field values by their field names and treat non-null reference field values as "Non-null".

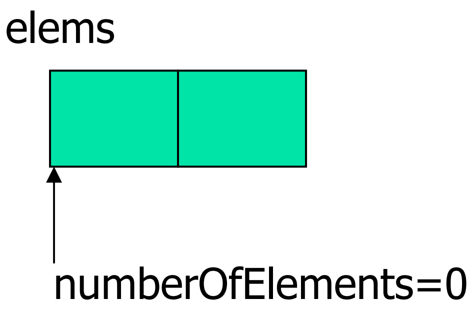
- **1389** (of 1745) Jtest-tests produce **12713** method executions, but only **63** distinct entry object states/args.

Object State Collection - Example

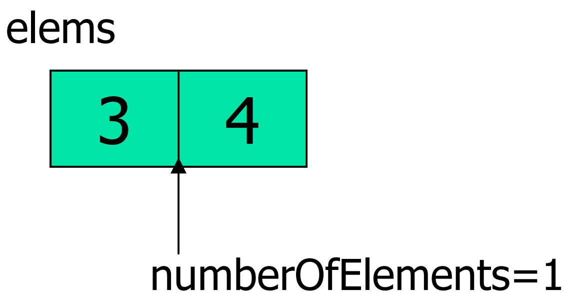
```
public class uniqueBoundedStack {
  private int[] elems;
  private int numberOfElements;
  public uniqueBoundedStack() {
      numberOfElements = 0;
      max = 2;
      elems = new int[max];
  }
  …
}
```

```
public boolean equals(uniqueBoundedStack s) {
   if (s.maxSize() != max)
         return false;
  if (s.getNumberOfElements() != numberOfElements)
        return false;
  int [] sElems = s.getArray();
  for (int j=0; j<numberOfElements; j++)          {
       if ( elems[j] != sElems[j])
            return false;
  }
  return true;
}
```

- stack =new uniqueBoundedStack()



- stack.push(3); stack.push(4); stack.pop();



Exit state: (this.euqals(this))

elems = Non-null

elems[0] = 3

max = 2

numberOfElements = 1

Algebraic Spec Inference

- 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*
- Look for equality patterns among args, return, entry state, exit state of either method in a pair
  - Based on axiom templates

Algebraic Spec Inference – Axiom Templates - I

- *foo2(foo1(S, arg1), arg2) = const*
  - *isEmpty(push(Stack, element)) == false*
- *foo2(foo1(S, arg1), arg2) = arg1 or arg2*
  - *top (push(Stack, element)) == element*
- *foo2(foo1(S, arg1), arg2) = foo1(S, arg1)*
  - *equals (pop(uniqueBoundedStack()), uniqueBoundedStack())*
- *foo2(foo1(S, arg1), arg2) = S*
  - *equals (pop (push (Stack, element)), S)*
- *foo2(foo1(S, arg1), arg2) = foo1(foo2(S, arg2), arg1)*
  - *equals (push(push(Stack, element1), element2) ,*
    push(push(Stack, element2), element1)
- *foo1(S, arg1) = const*
  - *maxSize(Stack) == 2*
- *foo1(S, arg1) = S*
  - *equals (print(Stack), Stack)*

Algebraic Spec Inference – Axiom 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*

Algebraic Spec-Based Test Generation

- Parameter generation
  - Collect non-referencing parameter values exercised by existing tests
  - Collect method call traces from test class to handle referencing parameters
- Object state setup
  - Collect object states exercised by existing tests
- Method sequence generation
  - LHS and RHS of Inferred axioms

- Test code generation based on the Danish tool [Hughes & Stotts 96]

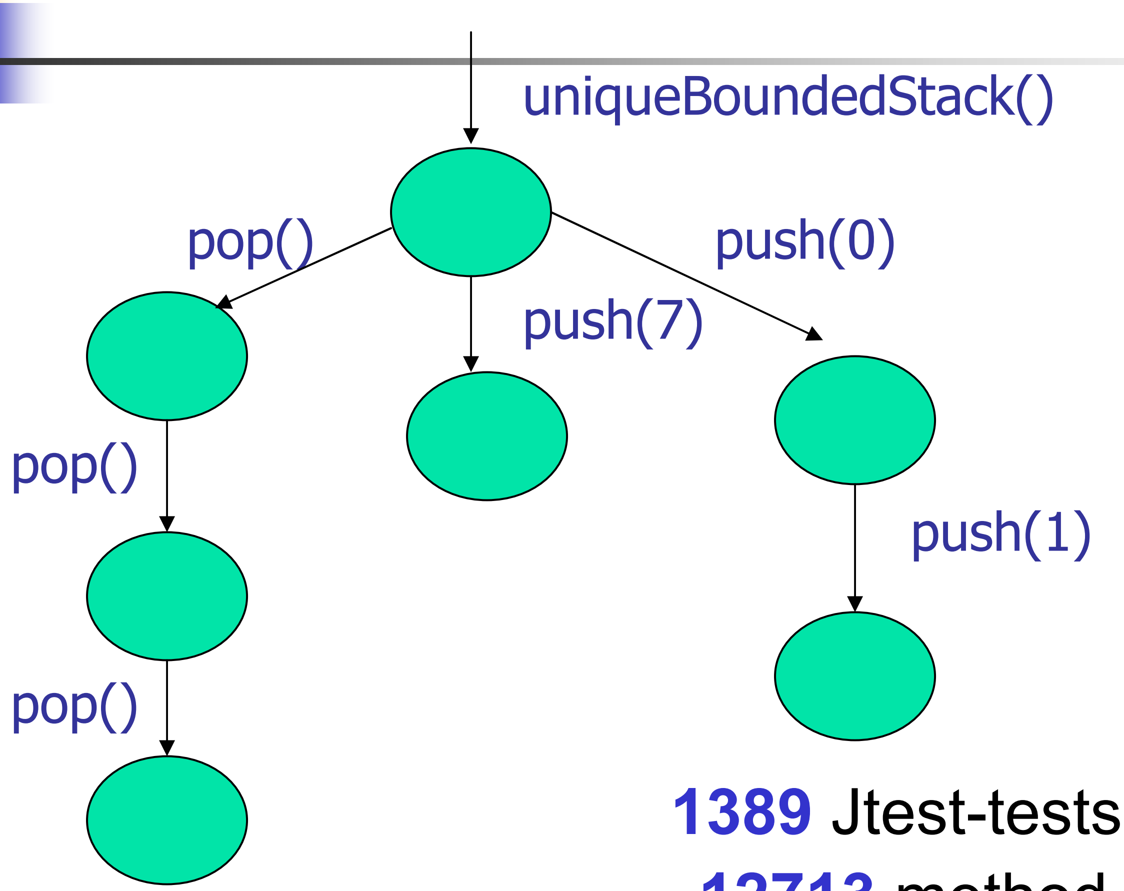
Test Selection

- Test selection
  - Axiom-violating tests
    - LHS != RHS for axiom LHS = RHS
  - Minimum tests contributing to inference of a new axiom
- Complementary technique: Select tests exercising at least one new structural entity.

Iterations

- Iterations stop until reaching fixed point or terminating conditions are satisfied, e.g. size = max_size
- Not all possible method pairs can be composed
  - In the first iteration, dummy axioms are generated
- Grow parameters
  - When the return of a method is the same type as a parameter
- Grow object states
  - Construct object state tree, only new object states are added to the tree

Object State Tree Growth - Example



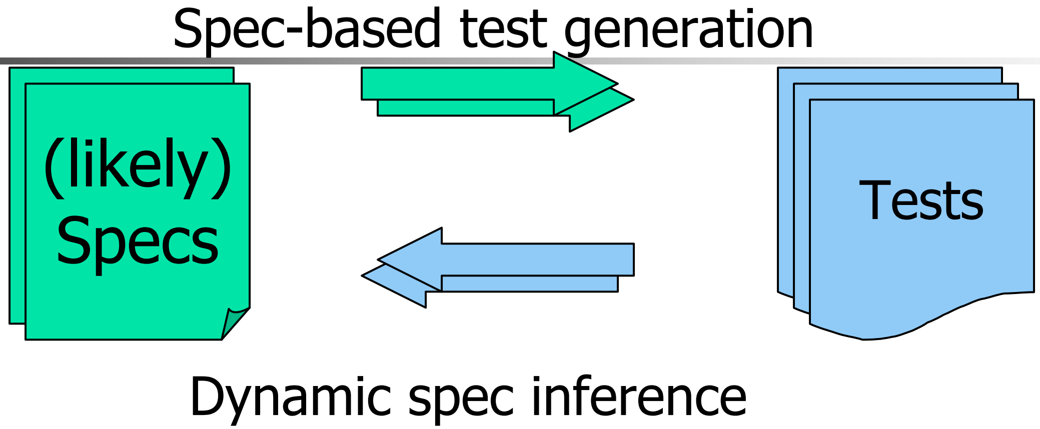
1389 Jtest-tests produces

- **12713** method executions
- **63** distinct entry object states/args
- **7** distinct object states!!

Outline

- Background
- Feedback Loop between Test Generation and Spec Inference
  - Axiomatic Spec Inference and Test Generation
  - Algebraic Spec Inference and Test Generation
- Conclusion

Conclusion



- Feedback loop between test generation and spec inference
  - Axiomatic specs (integration of Daikon and Jtest)
  - Algebraic specs
- Aids in test generation (improving specs and helping in producing oracles)
- Aids in spec inference (improving the underlying test suites)

Questions?

Object State Collection - Complications

- “equals” may call other public methods
  - Keep track of call depth
- Object field’s object fields might be accessed
  - Tracked objects include “*this*”, referencing object fields transitively accessed from “this”.
  - Collect an object field value if its object is tracked
- More
  - Array element’s order doesn’t matter – access count heuristics
  - “equals(C obj)” method contains shortcut (if this == obj return true) – replace “return true” with nop