Automatic Generation of Unit Regression Tests

Shay Artzi Adam Kieżun Carlos Pacheco Jeff Perkins
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139
{artzi,akiezun,cpacheco,jhp}@csail.mit.edu

Abstract

Software developers spend a significant amount of time modifying existing code. Regression testing (comparing previous behavior with current behavior) can be a powerful technique to check that modifications do not introduce unintended changes. This paper introduces a technique to automatically create class-specific regression tests from a program run. The technique is implemented for the Java language in Palulu.

In order to test a class, it may be necessary to place the class and other classes that it depends upon in specific states. To get to the desired state, methods may be need to be called in a particular order and with specific parameters. Palulu uses information from the program run to accomplish this. It builds a model for each class and uses the model to explore the possible behaviors of the class and build corresponding regression tests. It uses observer *methods (those that do not modify the state) to capture the result.*

In our experiments, Palulu succeeded in building tests for non-trivial programs and performed as well as or better than hand written tests.

1. Introduction

Tests for a unit of code require two parts: *test inputs* (a sequence of assignment statements and method calls) to exercise the code and an *oracle* to determine whether or not the result is correct. An oracle can be difficult to obtain. Regression testing uses a previous version of the code as an oracle, which allows a test to find when the behavior of the code has changed. Since much of software development is spent modifying code, regression testing can be quite valuable.

Our goal is to automate the creation of regression tests—both the creation of test inputs, and the creation of regression oracles. The introduction consists of three parts. To motivate the problem, we first describe the challenges that automated input creation techniques face. Next, we describe our approach to creating test inputs automatically. Finally, we describe our approach to creating regression oracles.

1.1 Creating Test Inputs is Difficult

Creating test inputs often requires placing an object in a particular state. We define the state of an object as the value of each of its (transitively reachable) fields. As the program progresses, mutable objects transition through various states. Each *mutator* method (one that changes the value of fields) may change the state of the object. The state of the object can thus also be defined by the sequence of mutator method calls (including their arguments) that have been made on the object. Consider the example Java classes in Figure 1. The sequence of calls

```
Graph g1 = new Graph();
g1.init();
Node n1 = new Node("NYC");
n1.setOwner(g1);
g1.addNode(n1);
```

defines a state for `g1` and `n1`.

Not all possible sequences of method calls are valid. For example, `Graph.addNode` is only valid after `Graph.init` has been called. `Node.addEdge` is only valid after `Node.setOwner` has been called. `Graph.addEdge` is only valid when `Graph-.addNode` has been called on both of its arguments. A method call is *invalid* when it violates the (implicit or explicit) invariants of the class. In all of the above examples, the invalid call throws an exception. Consider, however

```
n1.setOwner(g2)
g1.addNode(n1);
```

The call to `g1.addNode` is invalid because `n1`'s owner is `g2`. This does not throw an exception but is clearly not desirable (because `g1` now contains a node whose `owner` is not g1).

Additionally, not all of the possible values for primitive and string arguments will be valid for a method call. For example, strings often need to be in a particular format (such as a date, method signature, or URL). Numeric arguments may need to be in a range or have a particular mathematical relationship to one another.

A sequence of valid method calls is required to create a test. To achieve reasonable coverage, a set of test inputs must be created that explores the valid states. It is, however, challenging to create valid test inputs, as we have shown on example before. Specific calls must occur in a specific order with specific arguments. Specific relationships must be maintained between arguments. The search space of possible test inputs grows exponentially with the number of methods, possible arguments, and the length of the method sequence required. For example, if we presume 5 possible graph objects, 5 possible node objects, and a method sequence 10 elements in length, there are 235^{10} different possible sequences. An exhaustive search may be prohibitively expensive. Random exploration may not create interesting valid test inputs in a reasonable amount of time. Specifications may limit the search space, but specifications are often not available. Of course, non-trivial programs contain groups of inter-related classes with many more possible test inputs and more complex constraints than the ones shown in the example. We discuss a more complex example in Section 3.2.

1.2 Creating Test Inputs by Example

Our approach builds valid test inputs for classes with complex rules and inter-relationships by using information collected during an example program run to guide the construction of test inputs. A run of a program will use the classes of the program in a normal fashion. Valid sequences of method calls will be made on the various instances of each class. Arguments to those calls will be valid as well. For example, the `main` method in Figure 1 makes a sequence of valid method calls on `Graph` and `Node`.

Using information from the program run, the technique builds an *invocation model* for each class of interest. An invocation model describes the sequences of mutator method calls and arguments that occurred for the class. For example, the model for `Node` (shown in Figure 4) indicates that `setOwner` must be called before `addEdge`. The technique uses the model to build test inputs.

1.3 Regression Oracles via Observer Methods

Once a valid test input is created, the technique creates an oracle using a set of user-specified observer methods. The technique executes the test input (using the current version of the program) and then calls each observer. It creates an assertion using the result from the observer. Those assertions are combined with the test input to form a complete test. The complete test can be run on later versions of the program to ensure that the behavior did not change.

Figure 2 shows two sample tests generated by an implementation of our technique. Each executes a valid test input and checks the result using the `getDegree` observer method. These are not the same executions as in the `main` method. The invocation model guides the construction of test inputs, but does not limit them to only the ones in the original trace.

1.4 Contributions

In summary, we make the following contributions:

- We present *invocation models*, which describe valid sequences of method calls and their parameters, and a technique that dynamically infers them by tracking method calls in a program run.

- We use observer methods to create an oracle for regression testing.

```
public class Main {
  public static void main(String[] args) {
    Graph g1 = Graph.generateGraph();
    Node n1 = new Node("NYC");
    n1.setOwner(g1);
    Node n2 = new Node("Boston");
    n2.setOwner(g1);
    g1.addNode(n1);
    g1.addNode(n2);
    n1.addEdge(n2);
    n1.getDegree();
  }
}

public class Graph {
  private Map<Node, Set<Node>> adjMap;

  public static Graph generateGraph() {
    Graph g = new Graph();
    g.init();
    return g;
  }
  public void init() {
    adjMap = new HashMap<Node, Set<Node>>();
  }

  public void addNode(Node n) {
    adjMap.put(n, new HashSet<Node>());
  }

  public void addEdge(Node n1, Node n2) {
    addToNode(n1,n2);
    addToNode(n2,n1);
  }

  private void addToNode(Node source, Node target) {
    Set<Node> succ = adjMap.get(source);
    succ.add(target);
  }

  public int getDegree(Node n) {
    return adjMap.get(n).size();
  }
}

public class Node {
  private Graph owner;
  private String name;

  public Node(String name) {
    this.name = name;
  }
  public void setOwner(Graph a) {
    this.owner = a;
  }
  public void addEdge(Node n) {
    owner.addEdge(this, n);
  }
  public int getDegree() {
    return owner.getDegree(this);
  }
}
```

Figure 1. Example program. The following implicit rules must be followed to create a valid `Graph`: `Graph.init()` must be called before any of the other methods of `Graph`; `Node.setOwner()` must be called before any of the other methods of `Node`; `Graph.addNode()` must be called for a node before `Graph.addEdge()` is called for that node.

```
public void test13() {
    Node var327 = new Node("NYC");
    Graph var328 = new Graph();
    var327.setOwner(var328);
    var328.init();
    var328.addNode(var327);
    var327.addEdge(var327);
    assertEquals(1, var327.getDegree());
}

public void test22() {
    Graph var494 = Graph.generateGraph();
    Node var496 = new Node("Boston");
    Graph var497 = new Graph();
    var496.setOwner(var497);
    var497.init();
    var494.addNode(var496);
    var494.addNode(var496);
    var497.addNode(var496);
    var494.addEdge(var496, var496);
    assertEquals(0, var496.getDegree());
}
```

Figure 2. Two test cases from the regression suite generated by Palulu for the classes in Figure 1. Each of these follows the implicit rules for `Graph` and `Node`.

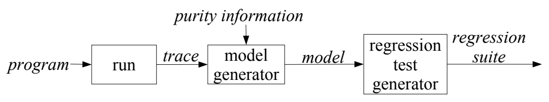

Figure 3. Technique overview. The inputs to the technique (left) are (1) a program under test and (2) a concrete execution of the program. The output (right) is a regression suite.

- We evaluate the effectiveness of our technique by comparing the regression tests it generates with (1) random input generation that does not use an invocation model, and (2) manually-written regression tests.

- We perform a case of study of our technique's ability to generate complex test inputs.

The remainder of the paper is organized as follows. Section 2 describes our technique. Section 3 contains the description and result of the experiments we performed to evaluate Palulu. Finally, Section 5 surveys related work.

2. Technique

Figure 3 shows a high-level view of the technique. Given the program and an example run, the technique automatically creates regression unit tests for selected classes in the program. The technique has four steps.

1. **Trace method calls.** Execute an instrumented version of the program that traces the method calls (and parameters/return values to the calls) that occur during the execution. Tracing is described in Section 2.1.

2. **Infer a model.** Use the trace to generate a model that abstracts the proper uses of the program's components. Model inference is described in Section 2.2.

```
Graph.generateGraph() -> g1
   new Graph() -> g1
   g1.init()
      edges = new HashMap()
new Node("NYC":String) -> n1
n1.setOwner(Graph)
new Node("Boston":String) -> n2
n2.setOwner(Graph)
g1.addNode(n1)
   new HashSet() -> hs1
   edges.put(n1,hs1) ->  null
g1.addNode(n2)
   new HashSet() -> hs1
   edges.put(n2,hs2) ->  null
n1.addEdge(n2)
   g1.addEdge(n1,n2)
      g1.addToNode(n1,n2)
         edges.get(n1) ->  hs1
         hs1.add(n2) ->  true
      g1.addToNode(n2,n1)
         edges.get(n2) ->  hs2
         hs1.add(n1) ->  true
n1.getDegree() -> 1
   g1.getDegree(n1) ->  1
     edges.get(n1) ->  hs1
     hs1.size() -> 1
```

Figure 5. The simplified trace created by running the example's main method (line 2 in Figure 1) using our tracing tool. The right hand side of the arrow refers to the return value.

3. **Generate test inputs.** Use the model to generate method call sequences representing test inputs. This step is described in Section 2.3.1.

4. **Create a regression oracle.** Execute the inputs and create a regression test oracle that captures their behavior. We capture the behavior of each method invocation by recording its return value, as well as the state of the receiver and parameters before and after the method is invoked. To capture an object's state, we call its observer methods and record their return values. Regression oracle creation is described in Section 2.3.2.

The output of the technique is a test suite, in which each test case consists of a method sequence (test input) and assertions about the state of the objects at each point during execution (regression oracle).

2.1 Tracing

Tracing collects the history of method invocations. Figure 5 shows the trace obtained when running the example program from Figure 1. Tracing is implemented by dynamically instrumenting class files as they are loaded. Each method call is instrumented at the call site, which allows calls to the JDK to be instrumented without instrumenting the JDK itself. An entry is written immediately before and after the call so that it can be determined how calls are nested. The receiver, each argument and the return value for the call are logged. Primitives and strings are logged with their value. Objects are logged with a unique ID.

2.2 Invocation Model Inference

The trace contains many legal sequences of calls. In fact the calls involving each object contain legal call sequences on instances

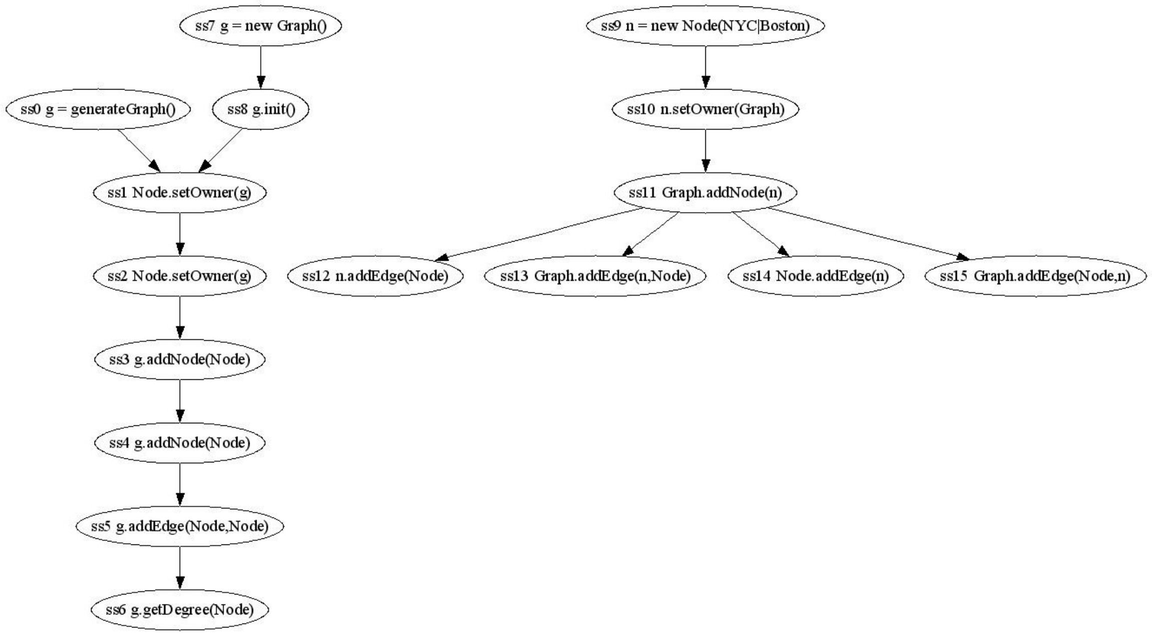

Figure 4. Invocation models for the `Graph` (left) and `Node` (right) from Figure 1.

of the object's class. We use the term *object history* to denote the sub-trace of calls that involve an object. Object histories follow the implicit rules of the classes. Our technique used those rules to create test cases. For objects of a given class, many histories may exist. However, many of them are likely to share prefixes of calls. Our technique summaries object histories into invocation models, to use this sharing.

Each invocation model is a directed graph. One model is created for each class for which instances are found in the trace[1]. The nodes in each graph are method signatures and an edge between two nodes indicate that the source method should be called before the target method on an instance of the class.

Figure 4 shows invocation models for the `Graph` and `Node` classes from the example in Figure 1. The models capture many of the implicit rules of those classes. For example, we noted in the introduction that `Graph.init` must be called before a node can be added to the graph by `Graph.addNode`. This is visible from the model of class `Graph` on the left-hand-side in Figure 4: the node for `Graph.init` precedes the node for `Graph.addNode` in the graph.

The algorithm for inferring invocation models has three steps:

1. Extract object histories from the trace (Section 2.2.1).
2. Filter calls from the histories (Section 2.2.2)
3. Create a model for each class by merging all the histories of instances of this class (Section 2.2.3).

2.2.1 Extracting Object Histories

The first step in the algorithm for inferring invocation models is to extract object histories for all the objects observed in the trace. An object's history consists of all invocations in the trace that includes the object as a receiver, parameter, or return value. An object's history describes the sequence of observable operations performed on an object. An object can take part in an operation without being explicitly passed as a parameter; our technique does not currently track this information.

Figure 6 shows the histories of the `Graph` and `Node` instances derived from the example trace. The indentation serves to indicate method call nesting.

2.2.2 Filtering Calls from Object Histories

The second step of the algorithm that infers invocation models is applying the following three filters on each object's history:

- **Non-Mutators Filter** Since non-mutator method calls do not change the object's state they are less important in test cases creation. This filter removes all calls to non-mutator methods as well as any call nested inside them (which should also be a non-mutator). The technique uses argument specific non-mutator information. A method is non-mutator over an argument/receiver if it does not modify any (transitively reachable) fields in that argument/receiver. It is a non-mutator over a return value if the return value is not created within the method or if it has no nested non-mutator method calls. Non-mutators can either be supplied by the user or computed via static analysis [12, 10]. Figure 7 contains the list of non-mutators in the example classes from Figure 1.

  An additional reason for removing calls to non-mutator methods is optimization. We found out in one of our experiments[2] that filtering non-mutators removed 73% of all calls. This filtering helps creating more manageable models.

- **Return Value Filter** This filter removes a call from an object history if the object is only the return value of the call.

[1]The user can specify a subset of those classes.

[2]This observation was made in the experiment described in Section 3.1.

History for the `Graph` instances:

```
Graph.generateGraph() -> g1
   new Graph() -> g1
   g1.init()
n1.setOwner(g1)
n2.setOwner(g1)
g1.addNode(n1)
g1.addNode(n2)
   g1.addEdge(n1,n2)
      g1.addToNode(n1,n2)
      g1.addToNode(n2,n1)
g1.getDegree(n1) -> 2
```

History for one of the `Node` instances:

```
new Node("NYC") -> n1
n1.setOwner(g1)
g1.addNode(n1)
   edges.put(n1, hs1)
n1.addEdge(n2)
   g1.addEdge(n1,n2)
      g1.addToNode(n1,n2)
         edges.get(n1) -> hs1
      g1.addToNode(n2,n1)
         hs1.add(n1)
n1.getDegree() -> 1
   g1.getDegree(n1) -> 1
      edges.get(n1) -> hs1
```

Figure 6. Simplified object histories for instances of `Graph` and `Node` obtained from the trace.

```
Node.getDegree() : receiver
Map.put(Object,Object) : parameters
Map.get(Object) : receiver, parameters
Set.add(Object) : parameters
```

Figure 7. List of non-mutator methods in the classes in Figure 1. Each method is followed by a list of arguments that it does not mutate.

The only exception to this filter is if the object is constructed inside the call. Those calls are either constructors or static factory method like the `Graph.generateGraph()` (first line in Figure 6) that can be used to create instances.

- **Non-Public Filter** This filter removes calls to non-public methods. This filter is applied, because non-public methods cannot be called from a test.

  Our technique assumes that methods performing field-writes do not call other methods. This can be easily achieved by replacing all field-writes with `set` methods.

  If a non-public `set-` method is nested in a filtered call, then filtering out non-public method calls can leave the object is an unknown state. In this case, the technique filters all calls after the call to the `set-` method (in the same method).

Figure 8 shows the filtered object histories for the instances of class `Node`.

2.2.3 *Merging Objects Histories into Models*

The last step of the model inference algorithm combines the histories of all instances of a class to create the class's model. For each class, the algorithm starts from an empty model and incrementally incorporates the histories of all instances of that class to the class's model.

```
new Node("NYC") -> n1
n1.setOwner(g1)
g1.addNode(n1)
n1.addEdge(n2)
   g1.addEdge(n1,n2)
```

```
new Node("Boston") -> n2
n2.setOwner(g1)
g1.addNode(n2)
n1.addEdge(n2)
   g1.addEdge(n1,n2)
```

Figure 8. Object histories for instances of the `Node`, after filtering.

Figure 4 contains the models inferred from the trace of Figure 5. The model for each class is a directed graph where nodes are method calls and edges define permitted order of method calls. Nodes also contain the position of the object in the call. The source nodes (nodes with no incoming edges) are methods that are used to create new instances (constructors and static factories).

When it adds a history to the model, the algorithm reuses nodes and edges from an existing prefix of calls (up to variable renaming) and creates new nodes and edges if no shared prefix exists. When searching for an existing prefix, nodes (method calls) and object positions are compared. In addition, whenever a primitive or a String parameter is encountered, its value is stored. Thus, primitive values are directly embedded on a per-parameter basis.

Assuming all field writes are done through setters (method that do not call any other methods), the sequence of method calls nested in each method call is an alternative, legal sequence for the same object. The algorithm is concerned with legal call sequences. Therefore, it adds, as alternative paths to each call, all the paths that are created from considering the calls nested in it (if any exist). For example, consider the method `Graph.generateGraph` (line 18 in Figure 1). This method calls two other methods: `Graph` constructor and the `Graph.init` method. The model for `Graph` (left in Figure 4 contains two paths from the source nodes to the `Graph.addNode` method. One path is through the method `Graph.generateGra` and the second path is through its nested events.

2.3 Generating Test Cases

The test case generator takes as input:

a. An invocation model for the classes in the tested program P.

b. A list of *tested classes*: a subset of the classes in P to generate test cases for.

c. For each tested class, a list of observer methods: pure methods that return a value [12, 10]. As examples, methods `Node.getDegree` and `Graph.getDegree` (in lines 41 and 59 of Figure 1, respectively) are observer methods. The list can be a subset of the purity list provided to the model inferencer. (Our implementation restricts observer methods to be parameter-less methods that return a primitive, which avoids the difficulties of providing parameters to the observer methods and of recording a return value which is an object. Addressing these difficulties is future work.)

d. A time limit.

The test case generator iterates through the tested classes. At each iteration, it tries to create a test case for the currently tested class C. The process stops when the time limit is exhausted. A test case consists of a test input and an oracle; Section 2.3.1 describes test inputs creation, and Section 2.3.2 describes regression oracle creation.

2.3.1 Generating Test Inputs

In the context of this paper, a test input for class C is a sequence of method calls that create, and possibly mutate, an object. of class C. For each test input, we also keep track of the list of objects contained in the test input, and the current state (from the invocation model) of each object. Figure 9 shows a test input and the state of each of its objects.

The test input generator maintains a database of previously-created inputs (i.e., method sequences). The database is initially empty. To create a new test input, the generator retrieves from the database uniformly at random an existing input that contains at least one object o of type C (the base case is the empty sequence).

Next, the generator extends the test input by appending a method call allowed by the model for object o. If there are several possible method calls, the generator creates several new inputs, one per call.

As Figure 4 shows, the model specifies what some of the parameters to the method call are. For the parameters that the model does not specify, the generator finds parameters to the call using the following algorithm.

1. If the parameter is a primitive, randomly select a primitive value from the possible values specified in the model.

2. If the parameter is of type C_h, an instance of that type must be found. If there is already an instance of C_h in the sequence, use that instance. If not, one must be created. If C_h appears more than once in the parameter list, do not reuse the same instance.

3. If there is no applicable reference, select one of two actions (at random):

   a. recursively try to create an input for type C_h, or

   b. retrieve an existing input from the database that creates an object of the desired parameter type. Call this a *helper* input.

4. Append the helper input to the end of the sequence of calls; the sequence now contains an object of type C_h. Go to step 3.

When all parameters are filled, the test input is executed[3]. If execution leads to an exception, the input is discarded. Our technique conservatively assumes that exceptional behavior is indicative of an illegal state, and does not attempt to create regression tests out of such states (recall that the model is incomplete; this incompleteness can lead to creating illegal inputs, even when the model is followed).

If the test input executes normally, a regression oracle is created for it, as described in the next section. In addition, the input is added to the database, where it can later be selected to be extended, or used as a helper input.

[3]In Palulu, we use Java's reflection mechanism to execute inputs.

2.3.2 Creating a Regression Oracle

The regression oracle creation takes as input a sequence, and the run-time objects resulting from its execution. For each object, it calls all observer methods specified for the class of the object, and records their return values. Each invocation of an observer method can be converted into an assertion in a straighforward way. The sequence together with the assertions comprise a unit regression test. Figure 2 contains examples of test unit regression tests for classes in Figure 1.

3. Evaluation

In this Section, we present the empirical evaluation of:

- the effectiveness of using observer methods in creating regression oracles (the experiment is described in Section 3.1), and

- the effectiveness of our technique in creating complex test inputs (the experiment is described in Section 3.2).

3.1 Observer Methods Experiment

In this experiment, we evaluated how the usage of test oracles based on observer methods affects the error-detection effectiveness of generated test suites. We used **RatPoly**, an assignment given in the MIT's software engineering class (6.170). **RatPoly** has the following salient characteristics:

- It has many different versions, each implementing the same interface.

- One of the versions of the program is a reference implementation, to which we can compare all the other versions. We make the simplifying assumption that the reference implementation contains no errors. It is justified by our goal of creating regression errors, which reveal inconsistencies between versions of the program.

- Some of the other versions contain errors. This is known because a manually-written test suite that reveals those errors is available.

The students were given a set of interfaces defining a one-variable rational-numbered coefficient polynomial library, and a test suite written using those interfaces. Their task was to submit an implementation of the interfaces that passed the test suite. (Not all students turned in assignments that passed the test suite, despite the fact that they were given the suite as part of the assignment.) The reference implementation (to which the students did not have access), and the manually-written test suite were written by the course staff.

We created the suite that we used in our experiments, which we call in this paper the Gold suite, by enhancing the staff-written suite with tests that were generated by our techniques and that we have manually verified to be fault-revealing. We also present, for reference, the results of running the original, staff-written test suite and compare them to tests suites generated automatically by our techniques.

Recall that our techniques require an example execution to create test suites. In this experiment, we used a 3-line program that performs a sequence of 9 simple operations on polynomials. We intentionally used a short example program to see if our technique

line	sequence	objects and states after each line
1	`Node var1 = new Node("NYC");`	`var1` (state ss6)
2	`Graph var2 = new Graph();`	`var1` (state ss6), `var2` (state ss4)
3	`var1.setOwner(var2);`	`var1` (state ss7), `var2` (state ss4)
4	`var2.init();`	`var1` (state ss7), `var2` (state ss5)

Figure 9. A test input derived from models in Figure 4. The input has 4 lines and creates two objects, a graph and a node. The graph and node go through different states of the model. The state names refer to node labels in the graphs in Figure 4.

could create useful tests even with a small amount of run-time data. The trace collected was over this small example and the reference (staff-provided) implementation.

We used and compared 6 different test generation techniques. They were a combination of 3 input generation strategies and 2 oracle creation strategies. The input generation strategies were:

a. **Unconstrained:** strategy that allows invocation of any method at any time with any parameter (of a valid type).

b. **Primitive constraints:** strategy that allows invocation of any method at any time but the constrains the arguments to those seen in the example execution.

c. **Primitive and method sequencing constraints:** strategy based on a model that uses method sequencing constraints, such as those discussed in Section 2.3.1.

The oracle creation strategies were:

a. **Exceptions:** a test suite fails if executing a test case in it leads to an exception being thrown (recall from Section 2.3.1 that we do not use tests inputs that lead to exceptions in the original program).

b. **Exceptions+observers:** a test suite fails if executing a test case in it leads to an exception being thrown. Additionally, the suite fails if a call to an observer method returns a value that is different than in the reference program (when run on the same input).

We repeated the following experimental process for each of the test generation techniques:

1. Execute the example program on the reference implementation to create the trace (as described in Section 2.1). The Unconstrained input generation strategy does not require tracing.

2. Create the model required for test input generation.

3. Generate the test suite using the model and the reference implementation. We bounded the execution time of this phase to 7 seconds.

4. Run the test suite on each of the 143 student programs for **RatPoly** and record the result. The result is either **pass** or **fail**. The **pass** result is given only when each test case in the test suite succeeds.

5. Compare the result to those of the Gold suite, for each student program. There are four possible outcomes (summarized in Figure 10):

   a. Both the generated suite and the Gold suite fail. In this case the generated suite correctly identifies an error in the program.

	Gold passes	Gold fails
generated passes	OK Pass	False Negative
generated fails	False Positive	OK Fail

Figure 10. Four possible outcomes of running a generated test suite and Gold test suite on a program.

   b. Both the generated the Gold suite pass. In this case, the generated suite correctly identifies a non-erroneous program.

   c. The generated suite passes, but the Gold suite fails. In this case, the generated suite misses an error (i.e., it is a false negative).

   d. The generated suite fails, but the Gold suite passes. In this case, the reported a false error (i.e., it is a false positive).

3.1.1 Results

Figures 11 and 12 present the results of the **RatPoly** experiment. The experiment was performed on a Pentium 4 machine, running Debian Linux, with 3.6 GHz CPU; the heap space was limited to 1 GB. Rows in the tables contain results for the 6 different generated tests suites. Additionally, we include, for reference, results for the original, staff-written test suite.

Columns in Figure 11 indicate, in order: the suite's name, tracing time, modelling time, test generation time, number of created sequences and test running time (averaged over all students).

The first 4 columns in Figure 12 show the results of executing the test suites on student programs. Let T denote the test suite. Then, $OKPass(T)$ is the number of students for which both T and the Gold test suite pass. Similarly, $OKFail(T)$ is the number of student programs for which both test suites fail and $FalsePositives(T)$ is the number of programs for which the Gold suite passes and T fails. Finally, $FalseNegatives(T)$ is the number of programs for which the Gold suite fails and T passes.

The last two columns indicate precision and recall and are calculated as follows:

- **Error Detection Recall** measures how often the generated test suite was successful in detecting an error.

$$Recall(T) = \frac{OKFail(T)}{FalseNegatives(T) + OKFail(T)}$$

- **Error Detection Precision** measures how often errors reported by the generated test suite were actual errors.

$$Precision(T) = \frac{OKFail(T)}{FalsePositives(T) + OKFail(T)}$$

Test Suite	Tracing time	Modelling time	Test gen time	# sequences	Test run time avg
Unconstrained	n/a	n/a	7	2016	7.00
Unconstrained + observers	n/a	n/a	7	2016	10.90
Primitive constraints	2.9	2.6	7	1753	5.39
Primitive constraints + observers	2.9	2.6	7	1753	5.95
Method seq constraints	2.9	2.6	7	1465	8.25
Method seq constraints + observers	2.9	2.6	7	1465	9.51
Original Manual	n/a	n/a	n/a	190	0.08

Figure 11. Time characteristics of the RatPoly experiment. Times are in seconds.

Test Suite	OKPass	OKFail	FalsePositives	FalseNegatives	Precision	Recall
Unconstrained	74	21	1	47	0.95	0.31
Unconstrained + observers	74	41	1	27	0.98	0.60
Primitive constraints	66	31	9	37	0.78	0.46
Primitive constraints + observers	47	57	28	11	0.67	0.84
Method seq constraints	75	32	0	36	1.00	0.47
Method seq constraints + observers	74	63	1	5	0.98	0.93
Original Manual	75	14	0	54	1.00	0.21

Figure 12. Results of the RatPoly experiment. The 4 middle columns refer to the possible outcomes of running a test suite, as described in Figure 10.

3.1.2 *Discussion of the results*

Results presented in Figure 12 show that, in this experiment,

- Using observer methods to create test oracles was effective. Error detection recall was almost 2 times higher when observer methods were used (0.31 vs. 0.6, 0.46 vs. 0.84 and 0.47 vs. 0.93). At the same time, using observers did not result in a significant loss of precision (0.95 vs. 0.98, 0.78 vs. 0.67 and 1.00 vs. 0.98).

- Generated tests suites outperformed the manually-written test suite in error detection recall (i.e., revealed more errors). The best-performing generated test suite had recall $0.93/0.21 = 4.43$ times higher than the manual test suite. By manual inspection, we verified that many of the errors that the staff-written test suite missed were due to incorrect handling of special cases (e.g., the zero polynomial) and unexpected mutation of objects (The specification required the classes to be immutable. Some student programs failed to meet that requirement—our generated test suites found some of those errors.)

- Using method sequencing constraints was of only limited effectiveness. This is visible by comparing the results for ‘Primitive constraints + observers’ and ‘Method seq constraints + observers’. Method sequencing constraints were not necessary to create tests for this experiment, because the objects were intended to be immutable.

- Using primitive values found in the execution increased the effectiveness of generated test suites, both in recall and precision. This is visible by comparing the first 2 rows of the table (i.e., ‘Unconstrained’ and ‘Unconstrained + observers’) with the next 2 rows (i.e., ‘Primitive constraints’ and ‘Primitive constraints + observers’). To understand why this was effective, consider the `RatPoly.parse(String)` method. A valid argument for this method must represent a polynomial, e.g., `"1/2*x^2-2*x+1"`. Random exploration of strings is unlikely to quickly find many valid inputs for this method. Using the string values observed in the trace narrows the search space.

Class	Description	References
VarInfoName	Variable Name	
VarInfo	Variable Description	VarInfoName PptTopLevel
PptSlice2	Two variables from a program point	VarInfo PptTopLevel Invariant
PptTopLevel	Program point	PptSlice2 VarInfo
LinearBinary	Linear Invariant over two scalar variables	PptSlice2
BinaryCore	Linear helper class	LinearBinary

Figure 13. Some of the classes needed to create a valid test input for Daikon’s BinaryCore class

3.2 Complex Inputs Experiment

To evaluate our technique’s effectiveness in creating legal *and* structurally complex inputs, we applied it to the `BinaryCore` class within Daikon [5], a tool that infers program invariants. BinaryCore is a helper class that calculates whether or not the points passed to it form a line. Daikon maintains a complex data structure involving many classes (see Figure 13) to keep track of the valid invariants at each program point. Some of the constraints in creating a valid BinaryCore instance are:

- The constructor to a `BinaryCore` takes an object of type `Invariant`, which has to be of run-time type `LinearBinary` or `PairwiseLinearBinary`, subclasses of `Invariant`. Daikon contains dozens of classes that extend `Invariant`, so the state space of incorrect but type-compatible possibilities is very large.

- To create a legal `LinearBinary`, one must first create a legal `PptTopLevel` and a legal `PptSlice2`. Both of these classes require an array of `VarInfo` objects. In addition, the constructor for `PptTopLevel` requires a string in a specific format; in Daikon, this string is read from a line in the input file.

- The constructor to `VarInfo` takes five objects of different types. Similar to `PptTopLevel`, these objects come from constructors that take specially-formatted strings.

- None of the parameters involved in creating a `BinaryCore` or any of its helper classes can be null.

4. BinaryCore: Manual vs. Generated

Figure 14 shows a test input that creates a `BinaryCore` object. This test was written by a Daikon developer, who spent about 30 minutes writing the test input.

We used our technique to generate test inputs for `BinaryCore`.

We gave the input generator a time limit of 10 seconds. During this time, it generated 3 sequences that create `BinaryCore` objects, and about 150 helper sequences.

Figure 14 also shows one of the three inputs that Palulu generated for `BinaryCore`. For ease of comparison, we renamed variables (Palulu assigned names like `var4722` to all variables), reordered method calls when the reordering did not affect the results, and added some whitespaces. Palulu successfully generated all the helper classes involved. It generated some objects in a way slightly different from the manual input; for example, to generate a `Slice`, it used the return value of a method in `PptTopLevel` instead of the class's constructor.

Without an invocation model, an input generation technique would have little chance of generating this sequence; the specific primitive parameters, the fact that a `BinaryCore` requires a `LinearBinary`, not just any `Invariant`, are all crucial pieces of information without which a search through the space of possibilities would be infeasible. Moreover, the path to a legal `BinaryCore` is highly constrained: there is not an easier way to obtain a `BinaryCore`.

5. Related Work

There is a large literature on automatic test generation; the most relevant to our work centers around what we call the *method sequence generation problem*: generating sequences of method calls that create and mutate objects for use as test inputs. Our survey centers around three topics: (1) modeling legal method sequences, (2) generating method sequences from the models, and (3) creating regression oracles for the sequences.

5.1 Modeling Legal Method Sequences

In this section, we survey techniques that automatically extract finite state machines (FSMs) from a program trace [3, 14, 9]. In these techniques, as in ours, nodes represent method calls, and a transition from node m_1 to node m_2 means that a call of m_1 can be followed by a call of m_2.

Cook and Wolf [3] generate FSMs for software processes. They consider only linear traces of atomic, parameter-free events (an event is just a label), which allows them to directly apply and extend existing grammar-inference methods [1]. Their focus is on evaluating the effectiveness of different inference methods in leading to a model that a human deems reasonable (for example, one of their case studies characterizes differences in the models inferred for a correct and an incorrect modification to a software artifact). The grammar-inference techniques that they use are not directly applicable to object-oriented programs, which produce nested—not linear—traces, and where an event represents a method call, which takes multiple parameters and affects more than one object.

Our modeling technique addresses the more complex traces that arise from modeling object-oriented systems. We deal with nested calls by creating alternate paths in an FSM and cutting off events for nested private calls. We deal with multiple-parameter events by inferring an FSM over a set of single-object histories, which consist of calls that affected a single object o, ignoring the identity of all other objects participating in the calls.

Whaley *et al.* [14] and Meghani *et al.* [9] also infer method-sequence FSMs for object-oriented systems; they also ignore important modeling issues such as parameters (other than the receiver) and nested calls, which we address. Whaley and Lam [14] use a dynamic analysis of a program run to records observed legal method sequences, and a static analysis that infers pairs of methods that cannot be called consecutively. Meghani and Ernst [9] improve on Whaley's technique by using a program trace to dynamically infer pre/postconditions on methods, and create a transition from m_1 to m_2 when the preconditions of m_2 are consistent with the postconditions of m_1. The main motivation (and evaluation) in both works is a better understanding of the system being modeled; they do not use their models to generate test inputs. Our main goal in deriving a model is to generate test inputs; this led us to augment our models with additional information such as primitive parameters.

5.2 Generating Test Inputs

Previous work on generating inputs from a specification of legal method sequences[2, 4, 8, 13] expect the user to write a model by hand. These techniques have been tested largely on toy examples, such as linked lists, stacks, etc. that, in addition to testing small structures, have few or no methods that take other objects as parameters. This makes input generation easier—there are fewer decisions to make, such as how to create an object to pass as a parameter.

Since we use an automatically-generated model and apply our technique to realistic applications, our test input generator must account for any lack of information in the generation model and still be able to generate inputs for complex data structures. Randomization helps here: whenever the generator faces a decision (typically due to under-specification in the generated model), a random choice is made. As our evaluation shows, the randomized approach leads to legal inputs. Of course, this process can also lead to illegal structures being created. In future work, we would like to investigate techniques that can help us detect illegal structures resulting from generating inputs using an under-specified model.

5.3 Capturing Behavior

Several techniques have been proposed to capture a program's intended behavior from an execution trace, ranging from deriving operational abstractions [5, 6], to algebraic specifications [7]. These techniques aim to generalize the trace into properties that are (with reasonable confidence) invariant across different program runs. Daikon [5], for example, infers program invariants from a program trace.

Our technique differs in that it does not attempt to derive generalizable properties, but simply records specific observed values for each input, using a predefined set of observer methods (we do not address the problem of finding observer methods, but we could use developed static analyses [12, 10]). The danger in capturing observations specific to one input is that they may reflect implementation details that are not important for correctness. Looking at the observations created for both RatPoly and Daikon, we found them to be close to what a programmer would have written as an oracle. Augmenting our input-specific oracles with dynamically-inferred invariants may lead to better oracles, and is a topic of future work.

5.4 Creating Unit Tests from System Tests

As mentioned in the last section our technique captures exhibited behavior at the unit level. Our technique can be used to create a set of unit tests from a large (potentially long-running) system test. Another way to think of this process is as a refactoring of the system test into a set of smaller unit tests.

Saff *et al.* propose tests factoring [11] as a different technique to create unit tests from a system test. Test factoring captures the interactions between tested and untested components in the system, and creates mock objects for the untested portions of the interaction, yielding a unit test where the environment is simulated via mocks. Test factoring accurately reproduces the execution of the entire system test. Our technique, on the other hand, uses the system test as only as a guide to create new sequences of method calls. Both techniques share the goal of creating focused, fast unit tests where there were none.

Test factoring creates tests that exactly mimic the original execution, so it never creates illegal method sequences; due to the incompleteness in the model,our technique can create illegal method sequences. On the other hand, our technique has several advantages over test factoring:

- **Tests are self-contained.** The end result of our technique is a compilable JUnit test suite. Test factoring creates mock objects and requires a special infrastructure to rerun a factored test.

- **The technique creates a regression oracle; test factoring must be provided one.** Test factoring takes a system test that already has an oracle, and makes a unit test case out of it, where the oracle is the same as the original. Our technique creates a regression oracle automatically.

- **The technique explores behavior beyond the system test.** Our model loses much of the information contained in the system test, which leads to generating objects that are likely to differ from those seen in the system test. Test factoring exactly mimics the system test, so it will catch no more (or less) than the errors caught by the original test. Our technique's ability to create tests that explore behaviors not explored in the system test means that our technique can detect errors that the system test missed. On the flip side, our technique may also miss errors that the original test caught; an experimental evaluation is necessary to determine how many errors we miss that the original test catches, and vice-versa.

6. Conclusion

We have presented a technique that automatically generates regression test suites at the unit level. To generate test inputs, the technique creates an invocation model from an example run of the program, and uses the model to generate legal, structurally-complex inputs that would be difficult to create without knowledge of the program's operation. For each test input, it creates a regression oracle by capturing the state of objects using observer methods. Our experiments suggest that the resulting regression tests are capable of revealing errors, including errors not caught by an extensive test suite. Our experiments also demonstrate the ability of the technique to generates structurally-complex inputs for real applications.

7. References

[1] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15(3):237–269, Sept. 1983.

[2] H. Y. Chen, T. H. Tse, and T. Y. Chen. Taccle: a methodology for object-oriented software testing at the class and cluster levels. *ACM Trans. Softw. Eng. Methodol.*, 10(1):56–109, 2001.

[3] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[4] R.-K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 165–177, New York, NY, USA, 1991. ACM Press.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[6] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, Orlando, Florida, May 22–24, 2002.

[7] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, Darmstadt, July 2003. Springer.

[8] D. Hoffman and P. Strooper. Classbench: A methodology and framework for automated class testing, 1997.

[9] S. V. Meghani and M. D. Ernst. Determining legal method call sequences in object interfaces, May 2003.

[10] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. *Lecture Notes in Computer Science*, 2027:20+, 2001.

[11] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE 2005: Proceedings of the 21st Annual International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, USA, Nov. 9–11, 2005.

[12] A. Sălcianu and M. C. Rinard. Purity and side effect analysis for java programs. In *VMCAI*, pages 199–215, 2005.

[13] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 302–310, Washington, DC, USA, 1993. IEEE Computer Society.

[14] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 218–228, Rome, Italy, July 22–24, 2002.

Manually-written test input (written by an expert)

```java
VarInfoName namex = VarInfoName.parse ("x");
VarInfoName namey = VarInfoName.parse ("y");
VarInfoName namez = VarInfoName.parse ("z");

ProglangType inttype = ProglangType.parse ("int");
ProglangType filereptype = ProglangType.repparse ("int");
ProglangType reptype = filereptype.fileTypeToRepType();

VarInfoAux aux = VarInfoAux.parse ("");

VarComparability comp = VarComparability.parse (0, "22", inttype);

VarInfo v1 = new VarInfo (namex, inttype, reptype, comp, aux);
VarInfo v2 = new VarInfo (namey, inttype, reptype, comp, aux);
VarInfo v3 = new VarInfo (namez, inttype, reptype, comp, aux);

VarInfo[] slicevis = new VarInfo[] {v1, v2};
VarInfo[] pptvis = new VarInfo[] {v1, v2, v3};

PptTopLevel ppt = new PptTopLevel
  ("DataStructures.StackAr.StackAr(int):::EXIT33", pptvis);

PptSlice2 slice = new PptSlice2 (ppt, slicevis);
Invariant proto = LinearBinary.getproto();
Invariant inv = proto.instantiate (slice);

BinaryCore core = new BinaryCore (inv);

core.addmodified (1, 1, 1);
core.addmodified (2, 2, 1);
core.addmodified (3, 3, 1);
core.addmodified (4, 4, 1);
core.addmodified (5, 5, 1);
```

Palulu-generated test input (formatted for ease of comparison)

```java
VarInfoName name1 = VarInfoName.parse("return");
VarInfoName name2 = VarInfoName.parse("return");

ProglangType type1 = ProglangType.parse("int");
ProglangType type2 = ProglangType.parse("int");

VarInfoAux aux1 = VarInfoAux.parse("␣declaringClassPackageName=DataStructures,␣");
VarInfoAux aux2 = VarInfoAux.parse("␣declaringClassPackageName=DataStructures,␣");

VarComparability comparability1 = VarComparability.parse(0, "22", type1);
VarComparability comparability2 = VarComparability.parse(0, "22", type2);

VarInfo info1 = new VarInfo(name1, type1, type1, comparability1, aux1);
VarInfo info2 = new VarInfo(name2, type2, type2, comparability2, aux2);
VarInfo info3 = new VarInfo(info2);
VarInfo info4 = VarInfo.origVarInfo(info3);
VarInfo info5 = VarInfo.origVarInfo(info2);
VarInfo info6 = VarInfo.origVarInfo(info3);

VarInfo[] infos = new VarInfo[] { info1, info2};

PptTopLevel ppt1 = new PptTopLevel("DataStructures.StackAr.push(java.lang.Object):::EXIT", infos);

PptSlice slice1 = ppt1.gettempslice(info1, info2);

Invariant inv1 = LinearBinary.getproto();
Invariant inv2 = inv1.instantiate(slice1);

BinaryCore lbc1 = new BinaryCore(inv2);
```

Figure 14. The first code listing is a test input written by a developer of Daikon. It required about 30 minutes to write. The second listing is a test input generated by our technique. For ease of comparison, we renamed automatically-generated variable names and grouped method calls related to each class (but we preserved any ordering that affects the results).