Finding the Needles in the Haystack: Generating Legal Test Inputs for Object-Oriented Programs

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

Abstract

A test input for an object-oriented program typically consists of a sequence of method calls that use the API defined by the program under test. Generating legal test inputs can be challenging because, for some programs, the set of legal method sequences is much smaller than the set of all possible sequences; without a formal specification of legal sequences, an input generator is bound to produce mostly illegal sequences.

We propose a scalable technique that combines dynamic analysis with random testing to help an input generator create legal test inputs without a formal specification, even for programs in which most sequences are illegal. The technique uses an example execution of the program to infer a model of legal call sequences, and uses the model to guide a random input generator towards legal but behaviorally-diverse sequences.

We have implemented our technique for Java, in a tool called Palulu, and evaluated its effectiveness in creating legal inputs for real programs. Our experimental results indicate that the technique is effective and scalable. Our preliminary evaluation indicates that the technique can quickly generate legal sequences for complex inputs: in a case study, Palulu created legal test inputs in seconds for a set of complex classes, for which it took an expert thirty minutes to generate a single legal input.

1. Introduction

This paper addresses the challenge of automatically generating test inputs for unit testing object-oriented programs [23, 24, 16, 9, 21]. In this context, a test input is typically a sequence of method calls that creates and mutates objects via the public interface defined by the program under test (for example, `List l = new List(); l.add(1); l.add(2)` is a test input for a class that implements a list).

```
TextFileDriver d = new TextFileDriver();
Conn con = d.connect("jdbc:tinySQL",null);
Stmt s1 = con.createStmt();
s1.execute(
  "CREATE TABLE test (name char(25), id int)");
s1.executeUpdate(
  "INSERT INTO test(name, id) VALUES('Bob', 1)");
s1.close();
Stmt s2 = con.createStmt();
s2.execute("DROP TABLE test");
s2.close();
con.close();
```

Figure 1. Example of a manually written client code using the tinySQL database engine. The client creates a driver, connection, and statements, all of which it uses to query the database.

For many programs, most method sequences are illegal; for correct operation, calls must occur in a certain order with specific arguments. Techniques that generate unconstrained sequences of method calls are bound to generate mostly illegal inputs. For example, Figure 1 shows a test input for the tinySQL database server[1]. Before a query can be issued, a driver, a connection, and a statement must be created, and the connection must be initialized with a meaningful string (e.g., `"jdbc:tinySQL"`). As another example, Figure 7 shows a test input for a more complex API.

Model-based testing [10, 14, 20, 6, 12, 19, 13, 18, 7, 15] offers a solution. A model can specify legal method sequences (e.g., `close()` cannot be called before `open()`, or `connect()` must be called with a string that starts with `"jdbc:"`). But as with formal specifications, most programmers are not likely to write models (except perhaps for critical components), and thus non-critical code may not take advantage of model-based input generation techniques.

To overcome the problem of illegal inputs, we developed a technique that combines dynamic analysis and random testing. Our technique creates a model of method sequences from an example execution of the program under test, and uses the model to guide a random test input generator towards the creation of legal method sequences. Because the model's sole purpose is aiding a

[1] `http://sourceforge.net/projects/tinysql`

M-TOOS '06 Portland, OR, USA

random input generator, our model inference technique is different from previous techniques [8, 22, 1, 25] which are designed primarily to create small models for program understanding. Our models must contain information useful for input generation, and must handle complexities inherent in realistic programs (for example, nested method calls) that have not been previously considered. At the same time, our models need not contain any information that is useless in the context of input generation such as methods that do not mutate state.

A random generator uses the model to *guide* its input generation strategy. The emphasis on “guide” is key: to create behaviorally diverse inputs, the input generator may diverge from the model, which means that the generated sequences are similar to, but not identical to, the sequences used to infer the model. Generating such sequences is desirable because it permits our test generation technique to construct new behaviors rather than merely repeating the observed ones. Our technique creates diverse inputs by (i) generalizing observed sequences (inferred models may contain paths not observed during execution), (ii) omitting certain details from models (e.g., values of non-primitive, non-string parameters), and (iii) diverging from models by randomly inserting calls to methods not observed during execution. (Some of the generated inputs may be illegal—our technique uses heuristics that discard inputs that appear to be illegal based on the result of their execution [17].)

In this paper, we make the following contributions:

- We present a dynamic model-inference technique that infers call sequence models suitable for test input generation. The technique handles complexities present in real programs such as nested method calls, multiple input parameters, access modifiers, and values of primitives and strings.

- We present a random test-input generation technique that uses the inferred models, as well as feedback obtained from executing the sequences, to guide generation towards legal, non-trivial sequences.

- We present Palulu, a tool that implements both techniques for Java. The input to palulu is a program under test and an example execution. Palulu uses the example execution to infer a model, then uses the model to guide random input generation. Palulu’s output is a collection of test inputs for the program under test.

- We evaluate Palulu on a set of real applications with constrained interfaces, showing that the inferred models assist in generating inputs for these programs. Our technique achieves better coverage than purely random test generation.

The remainder of the paper is organized as follows. Section 2 presents the technique. Section 3 describes an experimental evaluation of the technique. Section 4 surveys related work, and Section 5 concludes.

2. Technique

The input to our technique is an example execution of the program under test. The output is a set of test inputs for the program under test. The technique has two steps. First, it infers a model that summarizes the sequences of method calls (and their input arguments) observed during the example execution. Section 2.1 describes model inference. Second, the technique uses the inferred models to guide random input generation. Section 2.2 describes test input generation.

2.1 Model Inference

For each class observed during execution, our technique constructs a model called a *call sequence graph*. Call sequence graphs are rooted, directed, and acyclic. The edges represent method calls and their primitive and string arguments. Each node in the graph represents a collection of object states, each of which may be obtained by executing the method calls along some path from the root to the node. In other words, a node describes the history of calls. Each path starting at the root corresponds to a sequence of calls that operate on a specific object—the first method constructs the object, while the rest of the methods mutate the object (possibly as one of their parameters). Note that when two edges point to the same node, it does not necessarily mean that the underlying state of the program is the same.

For each class, the model inference algorithm constructs a model in two steps. First, it constructs a call sequence graph for each object of the class, observed during execution (Section 2.1.1). Second, it creates the model for the class by merging all call sequence graphs of objects of the class (Section 2.1.2). Thus, the call sequence graph for the class is a summary of call sequence graphs for all instances of the class.

For example, part (b) of Figure 2 shows the call sequence for $s1$, an object of class of `Stmt` in the program of Figure 1. Part (c) of Figure 2 shows the call sequence graph corresponding to the call sequence in part (b). The graph in part (c) indicates, for example, that it is possible to convert state A to state C either by calling `s1.execute()` or by calling `TS.parse(s1, DR)` and then calling `s1.setStmt(SQLStmt)`.

Figure 3 shows merging of call sequence graphs. The left and center parts show the graphs for `s1` and `s2`, while the right part shows the graph that merges the `s1` and `s2` graphs.

2.1.1 Constructing the Call Sequence Graph

A *call sequence* of an object contains all the calls in which the object participated as the receiver or a parameter, with the method nesting information for sub-calls. Figure 2(b) shows a call sequence. A *call sequence graph* of an object is a graph representation of the object’s call sequence—each call in the sequence has a corresponding edge between some nodes, and calls nested in the call correspond to additional paths between the same nodes. Edges are annotated with primitive and string arguments of the calls, collected during tracing. (Palulu records method calls, including arguments and return values, and field/array writes in a trace file created during the example execution of the program under test.)

The algorithm for constructing an object’s call sequence graph has three steps. First, the algorithm removes state-preserving calls from the call sequence. Second, the algorithm creates a call sequence graph from the call sequence. For nested calls, the algorithm creates alternative paths in the graph. Third, the algorithm removes non-public calls from the graph.

1. Removing state-preserving calls. The algorithm removes from the call sequence all calls that do not modify the state of the (Java) object.

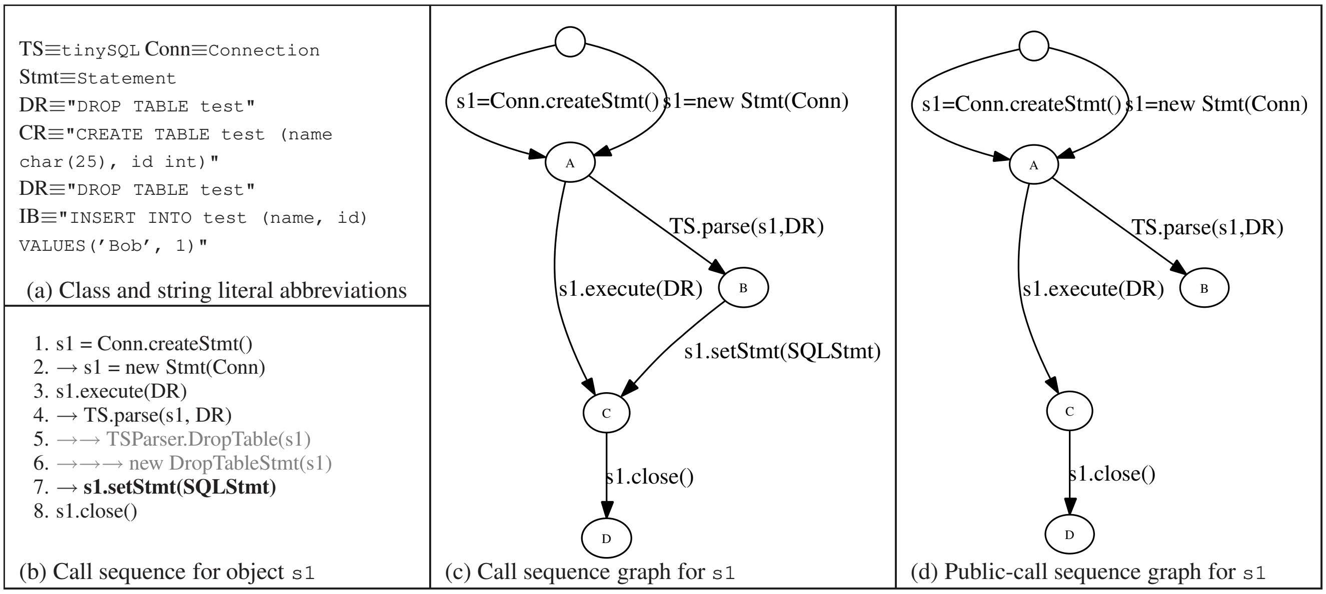

Figure 2. Constructing a call sequence graph for an object. (a) Abbreviations used in Figures 2 and 3. (b) Call sequence involving object `s1` in the code from Figure 1. Indented lines (marked with arrows) represent nested calls, shaded lines represent state-preserving calls, and lines in bold face represent non-public calls. (c) Call sequence graph for `s1` inferred by the model inference phase; it omits state-preserving calls. The path A-B-C represents two calls (lines 4 and 7) nested in the call in line 3. (d) Public call sequence graph, after removing from (b) an edges corresponding to a non-public call.

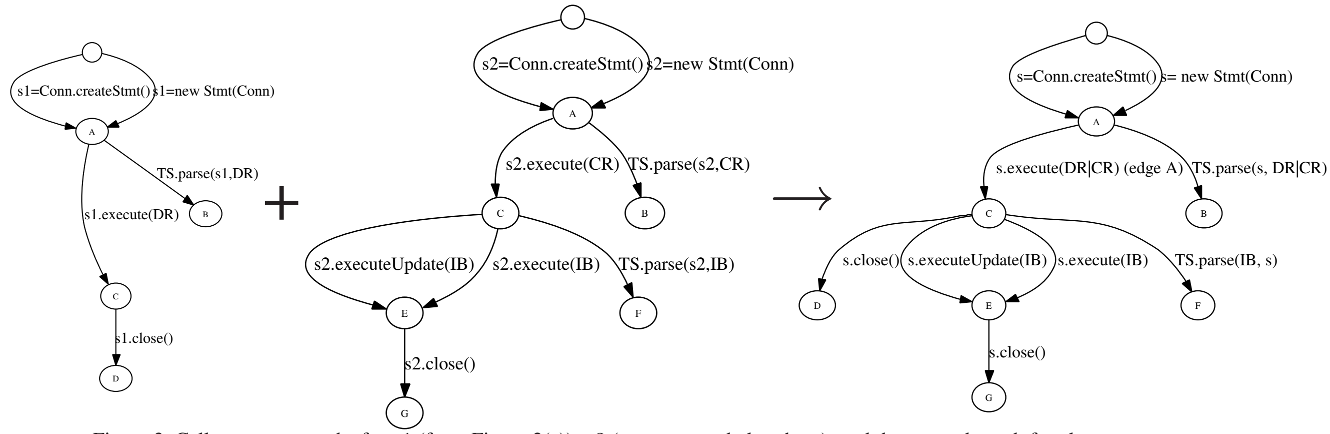

Figure 3. Call sequence graphs for $s1$ (from Figure 2(c)), $s2$ (not presented elsewhere), and the merged graph for class `Statement`.

State-preserving calls are of no use in constructing inputs, and omitting them reduces model size and search space without excluding any object states. Use of a smaller model containing only state-changing calls makes test generation more likely to explore many object states (which is one goal of test generation) and aids in exposing errors. State-preserving calls can, however, be useful as oracles for generated inputs, which is another motivation for identifying them. For example, the call sequence graph construction algorithm ignores the calls in lines 5 and 6 in Figure 2(b).

To discover state-preserving calls, the technique performs a dynamic immutability analysis [3] on the example execution. A method parameter (including the receiver) is considered immutable if no execution of the method changes the state of the object passed to the method as the actual parameter. The "state of the object" is the part of the heap that is reachable from the object by following field references.

2. Constructing call sequence graph. The call sequence graph construction algorithm is recursive and is parameterized by the call sequence, a starting node, and an ending node. The top-level invocation (for the whole history of an object) uses the root as the starting node and a dummy as the ending node[2].

Figure 4 shows a pseudo-code implementation of the algorithm. The algorithm processes the call sequence call by call, while keeping track of the last node it reached. When a call is processed, a new edge and node are created and the newly created node becomes the last node. The algorithm annotates the new edge with primitive and string arguments of the call.

Nested calls are handled by recursive invocations of the construction algorithm and give rise to alternative paths in the call sequence graph. After a call to method c is processed (i.e., an edge between nodes n_1 and n_2 is added to the graph), the algo-

[2] Dummy nodes are not shown in Figures 2 and 3.

```
// Insert sequence cs between nodes start and end.
createCallSequenceGraph(CallSequence cs,
                        Node start, Node end) {
  Node last = start;
  for (Call c : cs.topLevelCalls()) {
    Node next = addNewNode();
    addEdge(c, last, next); // add "last --c--> next"
    CallSequence nestedCalls = cs.getNestedCalls(c);
    createCallSequenceGraph(nestedCalls, next, last);
    last = next;
  }
  replaceNode(last, end); // replace last by end
}
```

Figure 4. The call sequence graph construction algorithm written in Java-like pseudo-code. The algorithm is recursive, creating alternative paths in the graph for nested calls.

rithm creates a path in the graph starting from n_1 and ending in n_2, containing all calls invoked by c.

For example, part (c) of Figure 2 contains two paths from state A to state C. This alternative path containing `TS.parse(s1, DR)` and `s1.setStmt(SQLStmt)` was added because the call to `s1.execute()` (line 3) in part (b) of Figure 2 invokes those two calls (lines 4 and 7).

3. Removing non-public calls. After constructing the object's call sequence graph, the algorithm removes from the graph each edge that corresponds to a non-public method. Thus, each path through the graph represents a sequence of method calls that a client (such as a test case) could make on the class. Ignoring non-public calls in the same way as state-preserving calls would not yield a graph with the desired properties.

For example, in part (c) of Figure 2, the edge corresponding to the non-public method `s1.setStmt(SQLStmt)` gets removed, which results in the graph presented in part (d) of Figure 2.

2.1.2 *Merging Call Sequence Graphs*

After the algorithm creates call sequence graphs for all observed objects of a class, it merges them into the class's model as follows. First, merge their root nodes. Whenever two nodes are merged, merge any pair of outgoing edges (and their target nodes) if (i) the edges record the same method, and (ii) the object appears in the same parameter positions (if the object is the receiver of the first method it must be the receiver of the second, similarly for the parameters); other parameters, including primitives and strings may differ. When two edges are merged, the new edge stores their combined set of primitives and strings.

For example, the call graphs for `s1` and `s2` can be found in left and center parts of Figure 3, while the combined model is on the right. The edges corresponding to `s1.execute(DR)` and `s2.execute(CR)` are merged to create the edge `s.execute(DR|CR)`.

2.2 Generating Test Inputs

The input generator uses the inferred call sequence models to guide generation towards legal sequences. The generator has three arguments: (1) a set of classes for which to generate inputs, (2) call sequence models for a subset of the classes (those for which the user wants test inputs generated using the models), and (3) a time limit. The result of the generation is a set of test inputs for the classes under test.

The input generator works by mixing pure random generation and model-based generation, as we explain below. The generator is incremental: it maintains an (initially empty) *component set* of previously-generated method sequences, and creates new sequences by extending sequences from the component set with new method calls.

Generating test inputs works in two phases, each using a specified fraction of the overall time limit. In the first phase, the generator does not use the models and creates test inputs in a random way. The purpose of this phase is initializing the component set with sequences that can be used during model-based generation. This phase may create sequences that do not follow the models, which allows for creation of more diverse test inputs. In the second phase, the generator uses the models to guide the creation of new test inputs.

An important challenge in our approach is creating tests that differ sufficiently from observed execution. Our technique achieves this goal by (i) generalizing observed sequences (inferred models may contain paths not observed during execution), (ii) omitting certain details from models (e.g., values of non-primitive, non-string parameters), and (iii) diverging from models by randomly inserting calls to methods not observed during execution (such sequences are created in the first, random, phase of generation and may be inserted in the second, model-based, phase).

2.2.1 *Phase 1: Random generation*

In this phase, the generator executes the following three steps in a loop, until the time limit expires [17].

1. **Select a method.** Select a method $m(T_0, \ldots, T_K)$ at random from among the public methods declared in the classes under test (T_0 is the type of the receiver). The new sequence will have this method as its last call.

2. **Create a new sequence.** For type T_i of each parameter of method m, attempt to find, in the component set, an argument of type T_i for method m. The argument may be either a primitive value or a sequence s_i that creates a value of type T_i. There are two cases:

   - If T_i is a primitive (or string) type, then select a primitive value at random from a pool of primitive inputs (our implementation seeds the pool with inputs like `0`, `1`, `-1`, `'a'`, `true`, `false`, `""`, etc.).
   - If T_i is a reference type, then use `null` as the argument, or select a random sequence s_i in the component set that creates a value of type T_i, and use that value as the argument. If no such sequence exists, go back to step 1.

   Create a new sequence by concatenating the s_i sequences and appending the call of m (with the chosen parameters) to the end.

3. **Add the sequence to the component set.** Execute the new sequence (our implementation uses reflection to execute sequences). If executing the sequence does not throw an exception, then add the sequence to the component set. Otherwise, discard the sequence. Sequences that throw exceptions are not useful for further input generation. For example, if the one-method input `a = sqrt(-1);` throws an exception because the input argument must be non-negative, then there is no sense in building upon it to create the two-method input `a = sqrt(-1); b = log(a);`.

Example. We illustrate random input generation using the `tinySQL` classes. In this example, the generator creates test inputs for classes `Driver` and `Conn`. In the first iteration, the generator selects the static method `Conn.create(Stmt)`. There are no sequences in the component set that create a value of type `Stmt`, so the generator goes back to step 1. In the second iteration, the generator selects the constructor `Driver()` and creates the sequence `Driver d = new Driver()`. The generator executes the sequence, which throws no exceptions. The generator adds the sequence to the component set. In the third iteration, the generator selects the method `Driver.connect(String)`. This method requires two arguments: the receiver or type `Driver` and the argument of type `String`. For the receiver, the generator uses the sequence `Driver d = new Driver();` from the component set. For the argument, the generator randomly selects `""` from the pool of primitives. The new sequence is `Driver d = new Driver(); d.connect("")`. The generator executes the sequence, which throws an exception (i.e., the string `""` is not valid a valid argument). The generator discards the sequence.

2.2.2 *Phase 2: Model-based generation*

Model-based generation is similar to random generation, but the generator uses the model to guide the creation of new sequences. We call the sequences that the model-based generator creates *modeled sequences*, which are distinct from the sequences generated by the random generator. The model-based generator keeps two (initially empty) mappings. Once established, the mappings never change for a given modeled sequence. The mo (modeled object) mapping maps each modeled sequence to the object, for which the sequence is being constructed. The cn (current node) mapping maps each modeled sequence to the node in the model that represents the current state of the sequence's mo-mapped object.

Similarly to the random generator from Phase 1 (Section 2.2.1), the model-based generator attempts to create a new sequences by repeatedly extending (modeled) sequences from the component set. The component set is initially populated with the sequences created in the random generation phase. The model-based generator repeatedly performs one of the following two actions (randomly selected), until the time limit expires.

- **Action 1: create a new modeled sequence.** Select a class C and an edge E that is outgoing from the root node in the model of C (select both class and edge at random). Let $m(T_0, \ldots, T_k)$ be the method that edge E represents. Create a new sequence s' that ends with a call to m, in the same manner as random generation (Section 2.2.1)—concatenate sequences from the component set to create the arguments for the call, then append the call to m at the end. Execute s' and add it to the component set if it terminates without throwing an exception. Create the mo mapping for s'—the s' sequence mo-maps to the return value of the call to m (model inference ensures that m does have a return value). Finally, create the initial cn mapping for s'—the s' sequence cn-maps to the target node of the E edge.

- **Action 2: extend an existing modeled sequence.** Select a *modeled* sequence s from the component set and an edge E outgoing from the node $cn(s)$ (i.e., from the node to which s maps by cn). These selections are done at random. Create a new sequence s' by extending s with a call to the method that edge E represents (analogously to Action 1). If a parameter of m is of a primitive or string type, randomly select a value from among those that decorate edge E. Execute s' and add it to the component set if it terminates without throwing an exception. Create the mo mapping for s'—the s' sequence mo-maps to the same value as sequence s. This means that s' models an object of the same type as s. Finally, create the cn mapping for s'—the s' sequence cn-maps to the target node of the E edge.

Example. We use `tinySQL` classes to show an example of how the model-based generator works. The generator in this example uses the model presented in the right-hand side of Figure 3. In the first iteration, the generator selects Action 1, and method `createStmt`. The method requires a receiver, and the generator finds one in the component set populated in the random generation phase (Section 2.2.1). The method executes with no exception thrown and the generator adds it to the component set. The following shows the newly created sequence together with the mo and cn mappings.

sequence s	$mo(s)$	$cn(s)$
`Driver d = new Driver();` `Conn c = d.connect("jdbc:tinySQL");` `Statement st = c.createStmt();`	`st`	A

In the second iteration, the generator selects Action 2 and method `execute`. The method requires a string parameter and the model is decorated with two values for this call (denoted by `DR` and `CR` in the right-most graph of Figure 3). The generator randomly selects `CR`. The method executes with no exception thrown and the generator adds it to the component set. The following shows the newly created sequence together with the mo and cn mappings.

sequence s	$mo(s)$	$cn(s)$
`Driver d = new Driver();` `Conn c = d.connect("jdbc:tinySQL");` `Statement st = c.createStmt();` `st.execute("CREATE TABLE test name\` `          char(25), id int)");`	`st`	C

3. Evaluation

This section presents an empirical evaluation of Palulu's ability to create test inputs. Section 3.1 shows that Palulu yields better coverage than undirected random generation. Section 3.2 illustrates how Palulu can create a test input for a complex data structure.

3.1 Coverage

We compared using our call sequence models to using universal models (that allow any method sequence and any parameters) to guide test input generation in creating inputs for programs that define constrained APIs. Our hypothesis is that tests generated by following the call sequence models will be more effective, since the test generator is able to follow method sequences and use input arguments that emulate those seen in an example input. We measure effectiveness via block and class coverage, since a test suite with greater coverage is generally believed to find more errors. (In the future, we plan to extend our analysis to include an evaluation of error detection.)

Program	tested classes	classes for which technique generated at least one input		block coverage	
		Universal model	Call sequence model	Universal model	Call sequence model
tinySQL	32	19	30	19%	32%
HTMLParser	22	22	22	34%	38%
SAT4J	22	22	22	27%	36%
Eclipse	70	46	46	8.0%	8.5%

Figure 5. Classes for which inputs were successfully created, and coverage achieved, by using following call sequence models and universal models.

3.1.1 *Subject programs*

We used four Java programs each of which contains a few classes with constrained APIs, requiring specific method calls and input arguments to create legal input.

- **tinySQL**[3] (27 kLOC) is a minimal SQL engine. We used the program's test suite as an example input.
- **HTMLParser**[4] (51 kLOC) is real-time parser for HTML. We used our research group's webpage as an example input.
- **SAT4J**[5] (11 kLOC) is a SAT solver. We used a file with a non-satisfiable formula, taken from DIMACS[6], as an example input.
- **Eclipse compiler**[7] (98 kLOC) is the Java compiler supplied with the Eclipse project. We wrote a 10-line program for the compiler to process, as an example input.

3.1.2 *Methodology*

As the set of classes to test, we selected from the program's public non-abstract classes, those classes that were touched during the sample execution. For classes not present in the execution, call sequence models are not created and therefore the input generated by the two techniques will be the same.

The test generation was run in two phases. In the first phase, seeding, it generated components for 20 seconds using universal models for all the classes in the application. In the second phase, test input creation, it generated test inputs for 20 seconds for the classes under test using either the call sequence models or the universal models.

Using the generated tests, we collected block and class coverage information with emma[8].

3.1.3 *Results*

Figure 5 shows the results. The test inputs created by following the call sequence models achieve better coverage than those created by following the universal model.

The class coverage results differ only for tinySQL. For example, without the call sequence models, a valid connection or a properly-initialized database are never constructed, because of the required initialization methods and specific input strings.

The block coverage improvements are modest for Eclipse (6%, representing 8.5/8.0) and HTMLParser (12%). SAT4J shows a 33% improvement, and tinySQL, 68%. We speculate that programs with more constrained interfaces, or in which those interfaces play a more important role, are more amenable to the technique. Future research should investigate these differences further in order to characterize the programs for which the technique works best, or to improve its performance on other programs.

The results are not dependent on the particular time bound chosen. For example, generation using the universal models for 100 seconds achieved less coverage than generation using the call sequence models for 10 seconds.

[3]`http://sourceforge.net/projects/tinysql`
[4]`http://htmlparser.sourceforge.net`
[5]`http://www.sat4j.org`
[6]`ftp://dimacs.rutgers.edu`
[7]`http://www.eclipse.org`
[8]`http://emma.sourceforge.net`

Class	**Description**	**Requires**
`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 ($y = ax + b$) over two scalar variables	`PptSlice2`
`BinaryCore`	Helper class	`LinearBinary`

Figure 6. Some of the classes needed to create a valid test input for Daikon's `BinaryCore` class. For each class, the **requires** column contains the types of all valid objects one needs to construct to create an object of that class.

3.2 Constructing a Complex Input

To evaluate the technique's ability to create structurally complex inputs, we applied it to the `BinaryCore` class within Daikon [11], 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 to keep track of the valid invariants at each program point.

An undirected input generation technique (whether random or systematic) would have little chance of generating a valid BinaryCore instance. Some of its constraints are (see Figure 6):

- The constructor to a `BinaryCore` takes an argument of type `Invariant`, which has to be of run-time type `LinearBinary` or `PairwiseLinearBinary`, subclasses of `Invariant`. Daikon contains 299 classes that extend `Invariant`, so the state space of type-compatible but incorrect 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. The `VarInfo` objects passed to `PptSlice2` must be a subset of those passed to `PptTopLevel`. 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.

Manually-written test input (written by an expert)	Palulu-generated test input
`VarInfoName namex = VarInfoName.parse("x");` `VarInfoName namey = VarInfoName.parse("y");` `VarInfoName namez = VarInfoName.parse("z");`	`VarInfoName name1 = VarInfoName.parse("return");` `VarInfoName name2 = VarInfoName.parse("return");`
`ProglangType inttype = ProglangType.parse("int");` `ProglangType filereptype = ProglangType.parse("int");` `ProglangType reptype = filereptype.fileToRepType();`	`ProglangType type1 = ProglangType.parse("int");` `ProglangType type2 = ProglangType.parse("int");`
`VarInfoAux aux = VarInfoAux.parse("");`	`VarInfoAux aux1 =` ` VarInfoAux.parse(" declaringClassPackageName=, ");` `VarInfoAux aux2 =` ` VarInfoAux.parse(" declaringClassPackageName=, ");`
`VarComparability comp =` `VarComparability.parse(0, "22", inttype);`	`VarComparability comp1 =` ` VarComparability.parse(0, "22", type1);` `VarComparability comp2 =` ` VarComparability.parse(0, "22", type2);`
`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 v1 =` ` new VarInfo(name1, type1, type1, comp1, aux1);` `VarInfo v2 =` ` new VarInfo(name2, type2, type2, comp2, aux2);`
`VarInfo[] slicevis = new VarInfo[] {v1, v2};` `VarInfo[] pptvis = new VarInfo[] {v1, v2, v3};`	`VarInfo[] vs = new VarInfo[] {v1, v2};`
`PptTopLevel ppt =` ` new PptTopLevel("StackAr.StackAr(int):::EXIT33",` ` pptvis);`	`PptTopLevel ppt1 =` ` new PptTopLevel("StackAr.push(Object):::EXIT", vs);`
`PptSlice2 slice = new PptSlice2(ppt, slicevis);`	`PptSlice slice1 = ppt1.gettempslice(v1, v2);`
`Invariant proto = LinearBinary.getproto();` `Invariant inv = proto.instantiate(slice);`	`Invariant inv1 = LinearBinary.getproto();` `Invariant inv2 = inv1.instantiate(slice1);`
`BinaryCore core = new BinaryCore(inv);`	`BinaryCore lbc1 = new BinaryCore(inv2);`

Figure 7. The first code listing is a test input written by an expert developer of Daikon. It required about 30 minutes to write. The second listing is a test input generated by the model-based test generator when following the call sequence models created by a sample execution of Daikon. 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).

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

We used our technique to generate test inputs for `BinaryCore`. To create the model, we used a trace from an example supplied with the Daikon distribution. 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 7 (left) 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 are not aware of a simpler way to obtain a `BinaryCore`.

Figure 7 (right) shows one of the three inputs that Palulu generated for `BinaryCore`. For ease of comparison between the inputs generated manually and automatically, we renamed automatically-named variables and reordered method calls when the reordering did not affect the results. Palulu successfully generated all the helper classes involved. Palulu generated some objects in a way slightly different from the manual input; for example, to generate a `Slice`, Palulu used the return value of a method in `PptTopLevel` instead of the class's constructor.

4. Related Work

Palulu combines dynamic call sequence graph inference with test input generation. This section discusses related work in each area in more detail.

4.1 Dynamic Call Sequence Graph Inference

There is a large literature on call sequence graph inference; we discuss some techniques most closely related to our work. Cook and Wolf [8] generate a FSM from a linear trace of atomic, parameter-less events using grammar-inference algorithms [2]. Whaley and Lam [22] combine dynamic analysis of a program run and static analysis of the program's source to infer pairs of methods that cannot be called consecutively. Ammons et al. [1] use machine learning to generate the graph; like our technique, Ammon's is inexact (i.e., the inferred state machine allows more behaviors than those observed in the trace).

In all the above techniques, the intended consumer of the inferred graphs is a person wanting to gain program understanding. Our end goal is generating test inputs for object-oriented APIs; the consumer of our graphs is a mechanical test input generator, and the model is only as good as it is helpful in generating inputs. This fact imposes special requirements that our inference technique addresses. To be useful for real programs, our call sequence graph inference technique must handle program traces that include methods with multiple input parameters, nested calls, private calls, primitive parameters, etc. On the other hand, the size of the graph is less crucial to us. In addition, the models of the above techniques mostly discover rules affecting one object (for instance, opening a connection before using it). In contrast, our model inference discovers rules consisting of many objects and method calls.

Another related project is Terracotta [25], which dynamically infers temporal properties from traces, such as "event E_1 always happens before E_2." Our call sequence graphs encode specific event sequences, but do not generalize the observations. Using inferred temporal properties could provide even more guidance to a test input generator.

After we publicized our algorithm and experimental results [4], Yuan and Xie [26] presented a very similar algorithm that creates per-object state machines, then combines them. However, they do not present any experimental results.

4.2 Generating Test Inputs with a Model

A large body of existing work addresses the problem of generating test inputs from a specification or model; below we survey the most relevant.

Most of the previous work on generating inputs from a specification of legal method sequences [10, 14, 20, 6, 12, 19, 13, 18, 7, 15] expects the user to write the specification by hand, and assumes that all inputs derived from the specification are legal. In addition, many of these techniques are designed primarily for testing reactive systems or single classes such as linked lists, stacks, etc. whose methods typically can take any objects as parameters. This greatly simplifies input generation—there are fewer decisions to make, such as how to create an object to pass as a parameter.

Like Palulu, the Agedis [13] and Jartege [15] tools use random test input generation; Agedis requires the user to write a model as a UML diagram, and Jartege requires the user to provide JML specifications. The tools can generate random inputs based on the models; the user also provides an oracle to determine whether an input is legal, and whether it is fault-revealing. Compared to Palulu, these tools represent a different trade-off in user control versus automation.

Since we use an automatically-generated model and apply our technique to realistic programs, our test input generator must account for any lack of information in the generated model and still be able to generate inputs for 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 creation of illegal structures. In future work, we plan to investigate techniques to minimize this problem.

An alternative approach to creating objects is via direct heap manipulation (e.g., Korat [5]). Instead of using the public interface of an object's class, Korat constructs an object by directly setting values of the object's fields (public and private). To ensure that this approach produces legal objects, the user provides a detailed object invariant specifying legal objects. Our approach does not require a manually-written invariant to create test inputs. Instead, it infers a model and uses it to guide the random search towards legal object states.

5. Conclusion

We have presented a technique that automatically generates structurally complex inputs for object oriented programs.

Our technique combines dynamic model inference with randomized, model-based test input generation to create high-quality test suites. The technique is targeted for programs that define constrained APIs for which random generation alone cannot generate useful tests that satisfy the constraints. It guides random generation with a model that summarizes method sequencing and method input constraints seen in an example execution.

We have implemented our technique for Java. Our experimental results show that test suites generated by our tool achieve better coverage than randomly generated ones. Our technique is capable of creating legal tests for data structures that take a human significant effort to test.

6. References

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, Jan. 2002.

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

[3] S. Artzi, M. D. Ernst, D. Glasser, and A. Kieżun. Combined static and dynamic mutability analysis. Technical Report MIT-CSAIL-TR-2006-065, MIT CSAIL, Sept. 18, 2006.

[4] S. Artzi, A. Kieżun, C. Pacheco, and J. Perkins. Automatic generation of unit regression tests. `http://pag.csail.mit.edu/6.883/projects/unit-regression-tests.%pdf`, Dec. 16, 2005.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, pages 123–133, July 2002.

[6] H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: A methodology for object-oriented software testing at the class and cluster levels. *IEEE TSE*, 10(1):56–109, 2001.

[7] Conformiq. Conformiq test generator. `http://www.conformiq.com/`.

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

[9] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431, May 2005.

[10] R.-K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *TAV4*, pages 165–177, Oct. 1991.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, Feb. 2001.

[12] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA*, pages 112–122, July 2002.

[13] A. Hartman and K. Nagin. The AGEDIS tools for model based testing. In *ISSTA*, pages 129–132, July 2004.

[14] D. Hoffman and P. Strooper. Classbench: a framework for automated class testing. *Software: Practice and Experience*, 1997.

[15] C. Oriat. Jartege: A tool for random generation of unit tests for Java classes. In *QoSA/SOQUA*, pages 242–256, Sept. 2005.

[16] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, July 2005.

[17] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. Technical Report MSR-TR-2006-125, Microsoft Research, Sept. 2006.

[18] Reactive Systems, Inc. Reactis. `http://www.reactive-systems.com/`.

[19] J. Tretmans and E. Brinksma. TorX: Automated model based testing. In *1st European Conference on Model Driven Software Engineering*, pages 31–43, 2003.

[20] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *ICSM*, pages 302–310, Sept. 1993.

[21] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *ISSTA*, pages 37–48, July 2006.

[22] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, pages 218–228, July 2002.

[23] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE*, pages 196–205, Nov. 2004.

[24] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, Apr. 2005.

[25] J. Yang and D. Evans. Automatically inferring temporal properties for program evolution. In *ISSRE*, pages 340–351, Nov. 2004.

[26] H. Yuan and T. Xie. Substra: A framework for automatic generation of integration tests. In *AST Workshop*, pages 64–70, May 2006.