Generating Legal Test Inputs for Object-Oriented Programs

Adam Kiezun, Shay Artzi, Michael Ernst, Carlos Pacheco, Jeff Perkins

MIT

M-TOOS’06
Automated Testing

- Goal: Automatically create a good test suite for an existing program with no specification

- Difficult
  - Complex object structures in programs are hard to create and test.
  - Specifications of object interaction are often not available.

- Our approach
  - Observe normal execution. Use information about actual call sequences to guide generation of tests.
Outline

- **Problem:** generating tests for complex structures

- **Technique**
  1. Create a model of legal calls / inputs
  2. Generate inputs using the model

- **Evaluation**
  - Test inputs for complex data structures
  - Coverage measurements
  - Observers as regression oracles

- **Conclusion**
Complex Test Inputs

- Test may require objects to be in certain states
- State can be defined by a sequence of mutator method calls

```java
RoadMap m1 = new RoadMap();
m1.init();
City c1 = new City("Portland");
c1.setMap(m1);
m1.addCity(c1);
```

Not all call sequences make sense:
- Some calls are only valid in certain states
  - e.g., must call `init()` before adding cities
- Interdependencies between arguments and/or receivers
  - e.g., map must be set before city is added
public class RoadMap {
    private Hashtable<City, Set<City>> cities;

    public static RoadMap genMap() {
        RoadMap m = new RoadMap();
        m.init(); return m;
    }

    public void init() {
        cities = new Hashtable<City, Set<City>>();
    }

    public void addCity(City c) {
        cities.put(c, new HashSet<City>());
    }

    public void addRoad(City c1, City c2) {
        addConnection(c1, c2);
        addConnection(c2, c1);
    }

    private void addConnection(City s, City t) {
        cities.get(s).add(t);
    }

    public int numNeighbors(City c) {
        return cities.get(c).size();
    }
}

public class City {
    private RoadMap map;
    private String name;

    public City(String name) {
        this.name = name;
    }

    public void setMap(RoadMap m) {
        this.map = m;
    }

    public void addRoad(City c) {
        map.addRoad(this, c);
    }

    public int numNeighbors() {
        return map.numNeighbors(this);
    }
}

public static void main(String[] a) {
    RoadMap m1 = RoadMap.genMap();
    City c1 = new City(“Portland”);
    c1.setMap(m1);
    City c2 = new City(“Seattle”);
    c2.setMap(m1);
    m1.addCity(c1);
    m1.addCity(c2);
    c1.addRoad(c2);
    c1.numNeighbors();
}
State Space Is Huge

- The state space is too large for exhaustive techniques
- Random selection is unlikely to quickly find many valid test inputs
- Specifications of object interaction are often not available
- Realistic classes are far more complex
Outline

- **Problem**: generating tests for complex structures
- **Technique**
  1. Create a model of legal calls / inputs
  2. Generate inputs using the model
- **Evaluation**
  - Test inputs for complex data structures
  - Coverage measurements
  - Observers as regression oracles
- **Conclusion**
Technique

- Program
  - Run
    - Execution trace
    - Infer model
      - Call sequence model
        - Generate tests
          - Test suite
Call Sequence Graphs

- Models of legal call sequences

- One directed, rooted graph per class
  - nodes are collections of object states (describe histories of method calls)
  - edges are method calls (including String and primitive arguments)

- Paths from root are legal call sequences

- Graph over-approximates sequences observed during execution
  - includes additional paths
  - under-specifies method arguments
Some values left unconstrained: _ = “don't care”
Inferring the Model

Steps:

1) Extract object histories from trace
   - Abstract away states for other objects
   - Filter out private and side-effect-free calls
2) Merge histories from objects of same class into a model for the class
Extracting Object Histories

Extract object histories from trace

- Abstract away states for other objects
- Filter out private and side-effect-free calls

m1 = genMap()
  m1 = new Map()
  m1.init()
cl = new City("Portland")
c1.setMap(m1)
c2 = new City("Seattle")
c2.setMap(m1)
m1.addCity(c1)
m1.addCity(c2)
c1.addRoad(c2)
  m1.addRoad(c1,c2)
  m1.addConnection(c1,c2)
  m1.addConnection(c2,c1)
c1.numNeighbors()
Extracting Object Histories

Extract object histories from trace

- Abstract away states for other objects
- Filter out private and side-effect-free calls

Example: extracting history for `c1`

Calls involving `c1`

```javascript
m1 = genMap()
m1 = new Map()
m1.init()

\texttt{c1} = new City(“Portland”)
\texttt{c1}.setMap(m1)

\texttt{c2} = new City(“Seattle”)
\texttt{c2}.setMap(m1)

m1.addCity(\texttt{c1})
m1.addCity(c2)

\texttt{c1}.addRoad(c2)
\texttt{m1}.addRoad(\texttt{c1},c2)

\texttt{m1}.addConnection(\texttt{c1},c2)
\texttt{m1}.addConnection(c2,\texttt{c1})

\texttt{c1}.numNeighbors()
```
Extracting Object Histories

Extract object histories from trace

- Abstract away states for other objects
- Filter out private and side-effect-free calls

```javascript
m1 = genMap()
m1 = new Map()
m1.init()
c1 = new City("Portland")
c1.setMap(m1)
c2 = new City("Seattle")
c2.setMap(m1)
m1.addCity(c1)
m1.addCity(c2)
c1.addRoad(c2)
m1.addRoad(c1,c2)
m1.addConnection(c1,c2)
m1.addConnection(c2,c1)
c1.numNeighbors()
```
Extracting Object Histories

Extract object histories from trace

- Abstract away states for other objects
- Filter out private and side-effect-free calls

```
m1 = genMap()
m1 = new Map()
m1.init()
c1 = new City("Portland")
c1.setMap(m1)
c2 = new City("Seattle")
c2.setMap(m1)
m1.addCity(c1)
m1.addCity(c2)
c1.addRoad(c2)
m1.addRoad(c1, c2)
g1.addConnection(c1, c2)
g1.addConnection(c2, c1)
c1.numNeighbors()
```

```
c1 = new City("Portland")
_.addCity(c1)
c1.addRoad(_)
_.addRoad(c1, _)
_.addConnection(c1, _)
_.addConnection(_, c1)
c1.numNeighbors()
```
Extracting Object Histories

Extract object histories from trace

- Abstract away states for other objects
- Filter out private and side-effect-free calls
Extracting Object Histories

object history for \( c_1 \)

\[
\begin{align*}
c_1 &= \text{new City("Portland")} \\
c_1 &= \text{setMap(\_)} \\
_\_ &= \text{addCity(c1)} \\
c_1 &= \text{addRoad(\_)} \\
_\_ &= \text{addRoad(c1,\_)}
\end{align*}
\]

object history for \( c_2 \)

\[
\begin{align*}
c_2 &= \text{new City("Seattle")} \\
c_2 &= \text{setMap(\_)} \\
_\_ &= \text{addCity(c2)} \\
_\_ &= \text{addRoad(c2)} \\
_\_ &= \text{addRoad(\_,c2)}
\end{align*}
\]
Merging Object Histories

- Incrementally incorporate histories into the models

- When adding an object history:
  - Merge prefixes (reuse existing nodes and edges)
  - Record primitives and Strings passed as parameters
  - Add nested calls as alternative paths
Merging: example

c = new City(“Portland”)
c.setMap(_)
_.addCity(c)
c.addRoad(_)
_.addRoad(c,_)

c = new City(“Portland”)
c.setMap(_)
_.addCity(c)
c.addRoad(_)  
_.addRoad(c,_)
Merging: example

c = new City("Portland")
c.setMap(_)
_.addCity(c)
c.addRoad(_)
_.addRoad(c,_)
Merging: example

c = new City("Portland")
c.setMap(_)
_.addCity(c)
c.addRoad(_)
_.addRoad(c, _)

\[\text{c = new City("Seattle")}
\text{c.setMap(_)}
\text{_.addCity(c)}
\text{_.addRoad(c)}
\text{_.addRoad(_, c)}\]
Merging: example

c = new City("Portland")
c.setMap(_)
_.addCity(c)
c.addRoad(_)
_.addRoad(c, _)

c = new City("Seattle")
c.setMap(_)
_.addCity(c)
_.addRoad(c)
_.addRoad(_, c)

_._addRoad(c, _)

c = new City("Portland"||"Seattle")
c.setMap(_)
_.addCity(c)
_.addRoad(_, c)

_.addRoad(_, _)

c.addRoad(_)

_.addRoad(_, _)

_.addRoad(c, _)
Merging: example

c = new City("Portland")
c.setMap(_)
_.addCity(c)
c.addRoad(_)
_.addRoad(c, _)

c = new City("Seattle")
c.setMap(_)
_.addCity(c)
_.addRoad(c)
_.addRoad(_, c)
_.addRoad(c)
Outline

- **Problem:** generating tests for complex structures

- **Technique**
  1. Generate a model of legal calls / inputs
  2. Create inputs using the model

- **Evaluation**
  - Test inputs for complex data structures
  - Coverage measurements
  - Observers as regression oracles

- **Conclusion**
Test Input Generator

Two Phases:

1. Random Generation
   . Allows calling methods not observed during execution
   . Generates random sequences of method calls

2. Model-Based Generation
   . Model is under-constrained
     . Alternative paths in model
     . Underspecified method arguments
   . Generation is randomized: faced with a choice, generator picks one randomly
Model-Based Input Generator

Example: generate a test input for RoadMap

```java
m = new RoadMap
m.init()
_.setMap(m)
_.setMap(m)
c = City("Portland","Seattle")
c.setMap(_)
_.addCity(c)
_.addRoad(c, _)
c.addRoad(_)
```

…...
Model-Based Input Generator

Example: generate a test input for RoadMap

```java
RoadMap m1 = new RoadMap();
m1.init();
```

```java
RoadMap m = new RoadMap();
m = genMap();
_.setMap(m);
_.setMap(m);
c = City("Portland"|"Seattle");
c.setMap(_);
_.addCity(c);
_.addRoad(c, _);
c.addRoad(_);
```
Model-Based Input Generator

Example: generate a test input for RoadMap

```java
RoadMap m1=new RoadMap();
m1.init();
__.setMap(m1);
```

```java
m = new RoadMap
m.init()
__.setMap(m)
__.setMap(m)
c = City("Portland"|"Seattle")
c.setMap(_)
__.addCity(c)
__.addRoad(c,_)```

```java
. . .
```

```java
. . .
```

```java
. . .
```

```java
. . .
```

```java
. . .
```

```java
. . .
```

```java
. . .
```

```java
. . .
```

```java
. . .
```

```java
. . .
```

```java
. . .
```
Model-Based Input Generator

Example: generate a test input for RoadMap

```java
RoadMap m = new RoadMap;
   m.init();
   .setMap(m);
   m = genMap();
   m = new RoadMap;
   m.init();
   .setMap(m);
   c = City("Portland"|"Seattle");
   c.setMap(_);
   .addCity(c); c.addRoad(c, _);
   .addRoad(c, _);
   c.addRoad(_);
```
Model-Based Input Generator

Example: generate a test input for RoadMap

```java
RoadMap m1 = new RoadMap();
m1.init();
City c1 = new City("Seattle");
c1.setMap(m1);

RoadMap m = new RoadMap();
m = genMap();
c = City("Portland","Seattle");
City c1 = new City("Seattle");
c1.setMap(m1);
```

```
RoadMap m1 = new RoadMap();
m1.init();
City c1 = new City("Seattle");
c1.setMap(m1);
```
Model-Based Input Generator

Example: generate a test input for RoadMap

```
RoadMap m1 = new RoadMap();
m1.init();
City c1 = new City("Seattle");
c1.setMap(m1);
_.setMap(m1);
```

```
RoadMap m = new RoadMap();
m = genMap();
c = City("Portland","Seattle");
c1 = new City("Seattle");
c1.setMap(m1);
_.setMap(m1);
c.addRoad(_);
_.addRoad(c, _);
_.addCity(c);
_.addRoad(c, _);
c.addRoad(_);
```
Model-Based Input Generator

Example: generate a test input for RoadMap

```java
RoadMap m1 = new RoadMap();
m1.init();
City c1 = new City("Seattle");
c1.setMap(m1);
```

Generation is randomized--may insert redundant calls
Outline

● **Problem:** generating tests for complex structures

● **Technique**
  1. Generate a model of legal calls / inputs
  2. Create inputs using the model

● **Evaluation**
  – Test inputs for complex data structures
  – Coverage measurements
  – Observers as regression oracles

● **Conclusion**
Evaluation: creating complex inputs

- Daikon invariant detector (ca 185 kLOC)
  - Internal data structures constructed and used in very specific ways
  - Static type information is not enough to generate valid structures
  - Example: LinearBinaryCore
Creating a valid LinearBinaryCore

```java
VarInfoName name_x = VarInfoName.parse("x");
VarInfoName name_y = VarInfoName.parse("y");
VarInfoName name_z = VarInfoName.parse("z");

ProglangType int_type = ProglangType.parse("int");  //string must denote a type
ProglangType file_rep_type = ProglangType.rep_parse("int");  //string must denote a type
ProglangType rep_type = file_rep_type.fileTypeToRepType();  //required call

VarInfoAux aux = VarInfoAux.parse(""");

VarComparability comp = VarComparability.parse(0, "22", int_type);  //param "22" must be a number

VarInfo v1 = new VarInfo(name_x, int_type, rep_type, comp, aux);
VarInfo v2 = new VarInfo(name_y, int_type, rep_type, comp, aux);
VarInfo v3 = new VarInfo(name_z, int_type, rep_type, comp, aux);

VarInfo[] ppt_vis = new VarInfo[] {v1, v2, v3};
VarInfo[] slice_vis = new VarInfo[] {v1, v2};  //must be a 2-elem subset of ppt_vis

PptTopLevel ppt = new PptTopLevel("DataStructures.StackAr.StackAr(int):::EXIT33", ppt_vis);  //string must be in a specific format

PptSlice2 slice = new PptSlice2(ppt, slice_vis);
Invariant proto = LinearBinary.get_proto();
Invariant inv = proto.instantiate(slice);

LinearBinaryCore lbc = new LinearBinaryCore(inv);  //one of 2 specific subtypes of Invariant (299 total)
```

- At every step, there are hundreds of other possible calls
- Our tool was able to create 3 different, legal, LinearBinaryCores in 10 seconds
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=, ");
VarInfoAux aux2 = VarInfoAux.parse(" declaringClassPackageName=, ");
VarComparability comp1 = VarComparability.parse(0, "22", type1);
VarComparability comp2 = VarComparability.parse(0, "22", type2);
VarInfo v1 = new VarInfo(name1, type1, type1, comp1, aux1);
VarInfo v2 = new VarInfo(name2, type2, type2, comp2, aux2);

VarInfo[] vs = new VarInfo[] {v1, v2};

PptTopLevel ppt1 = new PptTopLevel("StackAr.push(Object):::EXIT", vs);

PptSlice slice1 = ppt1.gettempslice(v1, v2);
Invariant inv1 = LinearBinaryCore.getproto();
Invariant inv2 = inv1.instantiate(slice1);
LinearBinaryCore lbc = new LinearBinaryCore(inv2);
Coverage Experiment

- Experiment
  - 4 subject programs (11-98 kLOC)
  - Measured block coverage achieved
    - using our model-based approach, vs
    - random generation

- Results
  - Model-based generation improved coverage 6% to 68%
  - Largest improvement for programs with more constrained interfaces.
Regression Experiment

- Experiment: MIT 6.170 assignment, 143 students
- Existing staff solution and staff-written test suite

- Regression oracle: fail if exception or different values returned by observer methods (using staff solution as reference implementation).
- We compared generated suite to suite written by course staff.

  - Results: generated test suite caught 4.5 times more faulty implementations than staff-written one
    - Staff-written suite detects 14 faulty implementations
    - Generated suite detects 63
    - Randomly generated suite detects 41
Next Steps

– Compare to exhaustive testing techniques (software model checking)
– Categorize programs on which the technique works best
– Investigate enhancing the models with additional constraints on object states
– Investigate using the models in anomaly detection
Contributions

– Created a model-based technique for automatic creation of test suites from a run of a program.
– Our tool created valid inputs for a complex data structure from a large application.
– Using our tool improves coverage of test suites.
– In our experiment, generated suite had almost 5 times better error detection than suite written by hand (and minimal false-positive rate).
Additional Slides
Creating regression tests

Two-step process:

1) Generate test inputs from model
   Explores model; uses randomization

2) Create a regression oracle for each input
   Uses observer methods

input + regression oracle = regression test
Generating a regression oracle

- Given: a newly-created input
- Goal: create a regression oracle for input
  - Execute input
  - Call observer methods on resulting objects
  - Record return values

```java
Map m = new Map();
m.init();
City c = new City("Seattle");
c.setMap(m);
c.setMap(m);
City c2 = new City("Portland");
c2.setMap(m);
assertTrue(m.numCities() == 2);
assertTrue(c.numNeighbors() == 0);
assertTrue(c2.numNeighbors() == 0);
```