Synthesizing Method Sequences for High-Coverage Testing

Based on "Synthesizing Method Sequences for High-Coverage Testing" by Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, Zhendong Su - OOPSLA'11

Prepared by Kamil Szarek
Agenda

- High-coverage testing and its difficulties
- Automatic test generation, existing tools
- New, Great Approach Called Seeker
  - Problem formulation
  - DynAnalyzer algorithm
  - StatAnalyzer algorithm
  - Examples
- Evaluation
- Future work
High-coverage testing

- Full or at least high code coverage is a desirable property of unit tests
- There are several types of coverage (e.g. structural coverage, data flow coverage)
- Here we focus on branch coverage: the percentage of branches that have been exercised by a test case suite
Branch coverage example

Client Code:
00: public static void foo(UDFSAlgorithm udfs) {
01:    ...
02:    if(udfs.GetIsComputed()) {
03:        ... // B6
04:    }
05:    // B7
06: }

// UDFS: UndirectedDepthFirstSearch
18: class UDFSAlgorithm {
19:    private IVEListGraph graph;
20:    private bool isComputed;
21:    public UDFSAlgorithm(IVEListGraph g) {
22:        ...
23:    public void Compute(IVertex s) {
24:        if(graph.GetEdges().Size() > 0) { // B4
25:            isComputed = true;
26:            foreach (Edge e in graph.GetEdges()) {
27:                ... // B5
28:            }
29:        }
30:    }
31: }

00: class AdjacencyGraph : IVEListGraph {
01:    private Collection edges;
02:    private ArrayList vertices;
03:    public void AddVertex(IVertex v) {
04:        vertices.Add(v); // B1
05:    }
06:    public Edge AddEdge(IVertex v1, IVertex v2) {
07:        if (!vertices.Contains(v1))
08:            throw new VNotFoundException("");
09:        // B2
10:        if (!vertices.Contains(v2))
11:            throw new VNotFoundException("");
12:        // B3
13:        // create edge
14:        Edge e = new Edge(v1, v2);
15:        edges.Add(e);
16:    }
17:}

Source: C# QuickGraph library
High-coverage testing

- Branch coverage can be quite hard to achieve
High-coverage testing

- Potential problems include tons of nested conditional statements and lack of knowledge about their conditions (e.g. no documentation for method whose return value is used to determine condition)
- Result: many hours spent on defining test environment for covering difficult branches
- Is there a better way...?
Yes, there is: automatic construction

- Two approaches: direct construction, sequence generation
- Program synthesis

Desired object state in form of conditional branch

Program synthesis

Method sequence that produces desired object state
Sequence generation challenges

- Large search space (multiple classes and methods)
- Primitive method parameters' values
- Object-oriented features such as encapsulation does not make it easier
Existing tools

- Pex
  - "Pex finds interesting input-output values of your methods, which you can save as a small test suite with high code coverage."
  - Dynamic Symbolic Execution
- Randoop
  - "Randoop generates unit tests using feedback-directed random test generation. In a nutshell, this technique randomly, but smartly, generates sequences of methods and constructor invocations for the classes under test, and uses the sequences to create tests."
New approach: "Seeker"

- Combines dynamic and static code analysis to reduce search space and generate appropriate primitive values
- Handles encapsulation properly
- Major challenges solved - awesome, but how does it work, precisely?
Problem formulation - definitions

- $C, M$ - sets of classes and methods
- $PrimTy, PrimVal$ - sets of primitive types and their values
- Method signature - $M \in M : C \times T_1 \times \ldots \times T_n \rightarrow T$, $T_i \in C \cup PrimTy$, $T \in C \cup PrimTy \cup \{ \text{void} \}$
Problem formulation - definitions

- **Method sequence (MCS):**
  Sequence of method calls \((m_1, \ldots, m_r)\) such that:
  - \(m_i = o.M_i(a_1, \ldots, a_n)\), where \(M_i \in M\)
  - \(o = \text{ret}(m_k)\) for some \(1 \leq k < i\) and for all \(1 \leq j \leq n\)
    \(a_j \in \text{PrimVal} \lor a_j = \text{null} \lor a_j = \text{ret}(m_l)\) for some \(1 \leq l < i\).

- *In another words: compiler wouldn't complain (much).*
Problem formulation - definitions

- **Sequence skeleton (SKT):** Just like MCS, except that we do not require values of primitive type arguments. Useful when we don't know what values we're going to need.

- **Target branch (TB):** Branch of conditional statement to be covered. Input of the algorithm.
Problem formulation - definitions

- Method sequence synthesis:
  Given a method under test $M \in M$ and a target branch $tb$ within $M$, synthesize a method sequence $(m_1, \ldots, m_r)$ that constructs the receiver object and arguments of $M$ and drives $M$ to successfully cover $tb$. 
Seeker algorithm overview

- Two algorithms in feedback loop: DynAnalyzer and StatAnalyzer - dynamic and static analysis
- Dynamic analysis attempts to generate target sequence
- If it fails, static analysis starts, utilizing information from dynamic analysis
- Then dynamic analysis explores static analysis results and filters them out
DynAnalyzer

- Input: target branch $tb$, input sequence $inpseq$

1. Identify method $m$ containing $tb$
2. Append $m$ to $inpseq$ as sequence skeleton (no primitive values) producing $tmpskt$
3. Run DSE subroutine to explore $tmpskt$ for generating target sequence that covers $tb$
Symbolic Execution allows method to have non-concrete (symbolic) parameters

When program executes conditional branch where condition has symbolic parameter, symbolic execution considers both branches

Constraints on symbols are collected

Constraint solver or theorem prover is used to obtain concrete values
DynAnalyzer - DSE

- Dynamic Symbolic Execution generates simple inputs instead of symbols
- After an execution, constraint solver is used to change inputs in order to cover different branches

**Algorithm 2.1. Dynamic symbolic execution**

Set $J := \emptyset$

loop
  Choose program input $i \notin J$
  Output $i$
  Execute $P(i)$; record path condition $C$
  Set $J := J \cup C$
end loop

(intuitively, $J$ is the set of already analyzed program inputs)
(stop if no such $i$ can be found)
(in particular, $C(i)$ holds)
(viewing $C$ as the set $\{i \mid C(i)\}$)
DynAnalyzer - DSE

- DSE outputs targetseq (null when it fails), CovB (set of covered branches) and NotCovB (set of not covered branches)
- Depending on DSE results:
  4. If targetseq is not null, we're done
  5. If targetseq is null and \( tb \in NotCovB \), we return \( \text{StatAnalyzer}(tb, inpseq) \)
  6. Otherwise…
6. **ComputeDominants**
   a. Prime dominant: branch whose alternative branch is covered by DSE,
   b. All other dominant branches of $tb$ between the prime dominant and $tb$.

7. **Recurisely invoke DynAnalyzer for each dominant branch and return method sequence if all dominant branches are covered along with $tb$**

8. ...Fail otherwise
DynAnalyzer algorithm

\begin{algorithm}
\caption{DynAnalyzer}(tb, inpseq)
\begin{algorithmic}
\Require \texttt{tb} of type TB
\Require \texttt{inpseq} of type MCS
\Ensure targetseq of type MCS covering \texttt{tb} or null
\Statex
\State Method \texttt{m} = GetMethod(\texttt{tb})
\State \texttt{SKT} \texttt{tmpskt} = AppendMethod(\texttt{inpseq, m})
\State DSE(\texttt{tmpskt}, \texttt{tb}, out targetseq, out CovB, out NotCovB)
\State //Scenario 1
\If {\texttt{tb} \in CovB}
\State \Return targetseq
\EndIf
\State //Scenario 2
\If {\texttt{tb} \in NotCovB}
\State \Return StatAnalyzer(\texttt{tb}, \texttt{inpseq})
\EndIf
\State //Scenario 3
\If {\texttt{tb} \notin NotCovB}
\State List<TB> \texttt{tlist} = ComputeDominants(\texttt{tb})
\ForAll {TB \texttt{domtb} \in \texttt{tlist}}
\State \texttt{inpseq} = DynAnalyzer(\texttt{domtb, inpseq})
\If {\texttt{inpseq} == null}
\State Break
\EndIf
\EndFor
\If {\texttt{inpseq} \neq null}
\State \Return DynAnalyzer(\texttt{tb, inpseq})
\EndIf
\EndIf
\State \Return null
\end{algorithmic}
\end{algorithm}
**StatAnalyzer**

- Input same as for DynAnalyzer
- Main purpose: to identify other branches that can help cover $tb$

1. **DetectField**
   - Identifies member field $tfield$ that needs to be modified to produce object state for covering $tb$
   - This is trivial if condition directly refers to field
   - Otherwise, we use execution trace from DSE which includes statements executed in each method
StatAnalyzer - DetectField

- DetectField starts from method call involved in \( tb \) and proceeds backwards.
- Denote \( retvar \) as variable/value associated with the return statement in method call.

a. If \( retvar \) is member field, \( tfield \) is \( retvar \)

b. If \( retvar \) is data-dependent on member field, that field is \( tfield \)
c. If `retvar` is data-dependent on return of nested method call, `DetectField` is repeated with that method call

d. If `retvar` is control-dependent on member field, that field is `tfield`

e. If `retvar` is control-dependent on return of nested method call, `DetectField` is repeated with that method call
There are two more results of DetectField, apart from *tfield*

DetectField identifies the condition of *tfield* which is not satisfied - and should be in order to cover *tb*

DetectField captures field hierarchy that includes objects from the one enclosing *tb* to *tfield*
StatAnalyzer - DetectField - example

- B8 is \( tb \)
- DetectField for ints. HasElements()
- (e): DetectField for stack. size()
- (a): _size field in ArrayList is \( tfield \)
- Detected condition: stack.size() > 0

```
00: public class IntStack {
01:     private Stack stack;
02:     public IntStack() {
03:         this.stack = new Stack();
04:     }
05:     public void Push(int item) {
06:         stack.Push(item);
07:     }
08:     public bool HasElements() {
09:         if(stack.size() > 0) { return true; }
10:     }
11: }
12: public class MyCls {
13:     private IntStack ints;
14:     public MyCls(IntStack ints) {
15:         this.ints = ints;
16:     }
17:     public void MyFoo() {
18:         if(ints.HasElements()) {
19:             ...// B8
20:         }
21: }
```

MyCls root  Stack stack  int _size
IntStack ints  ArrayList list
2. SuggestTargets
   ○ Identifies pre-target branches that need to be covered in order to cover \( tb \)

a. Find \( tobject \) - object that is nearest to \( tfield \) in field hierarchy and can be modified directly or by public method

b. Identify methods (and pre-target branches within) that help produce a desired value
The latter part is non-trivial; there might be intermediate objects between tobject and tfield.

- Method-call graph
- Root represents tfield, other nodes are methods and form layers of classes
- Edge from root to first layer if method modifies tfield.
- Edge between layers on method calls
StatAnalyzer - SuggestTargets

- Graph generation based on field hierarchy
- We're looking for statements where \textit{tfield} appears on the left side
c. Traverse method-call graph to identify methods that can be invoked on `tobject` to achieve desired value for `tfIELD` and pre-target branches within these methods

```
00: public class IntStack {
01:     private Stack stack;
02:     public IntStack() {
03:         this.stack = new Stack; }
04:     public void Push(int item) {
05:         stack.Push(item);
06:     }     
07:     public bool HasElements() { 
08:         if(stack.size() > 0) { return true; }
09:         else { return false; }
10:     }
11: }
12: public class MyCls {
13:     private IntStack ints;
14:     public MyCls(IntStack ints) {
15:         this.ints = ints;
16:     }     
17:     public void MyFoo() {
18:         if(ints.HasElements()) {
19:             ...//BB
20:         }
21:     }
22: }
```

- `ints` is `tobject`
- `IntStack.Push` is identified as method which modifies `tfIELD`
- We need to cover branch in `Push method` to cover `tb`
3. Try to cover branches recognized by SuggestTargets using DynAnalyzer and use new sequences to cover \(tb\)
   - Static analysis may suggest branches that not necessarily help us in our quest, e.g. IntStack.Pop
   - If dynamic analysis successfully covers pre-target branch, we're using generated sequence to try to cover the original target branch \(tb\) using DynAnalyzer
Algorithm 2 StatAnalyzer(tb, inpseq)

Require: A target branch tb
Require: A sequence inpseq
Ensure: A sequence targetseq covering tb

1: Field tfield = DetectField(tb)
2: List<TB> tlist = SuggestTargets(tfield)
3: for all TB pretb ∈ tlist do
4:     MCS targetseq = DynAnalyzer(pretb, inpseq)
5:     if targetseq ≠ null then
6:         targetseq = DynAnalyzer(tb, targetseq)
7:         if targetseq ≠ null then
8:             return targetseq
9:     end if
10: end if
11: //Try other alternative target branches
12: end for
13: return null
Seeker algorithm

- That's all! ...
- DynAnalyzer and StatAnalyzer are mutually recursive
- We start with DynAnalyzer($tb$, null)
- An example follows
Seeker - example

Client Code:
00: public static void foo(UDFSAlgorithm udfs) {
01:     ...
02:     if(udfs.GetIsComputed()) {
03:         ... // B6
04:     }
05:     // B7
06: }

//UDFS:UndirectedDepthFirstSearch
18: class UDFSAlgorithm {
19:     private IVertexListGraph graph;
20:     private bool isComputed;
21:     public UDFSAlgorithm(IVertexListGraph g) {
22:         ...
23:     }
24:     public void Compute(IVertex s) {
25:         if (graph.GetEdges().Size() > 0) { // B4
26:             isComputed = true;
27:             foreach (Edge e in graph.GetEdges()) {
28:                 ... // B5
29:             }
30:         } ...
31:     }

00: class AdjacencyGraph : IVertexListGraph {
01:     private Collection edges;
02:     private ArrayList vertices;
03:     public void AddVertex(IVertex v) {
04:         vertices.Add(v); // B1
05:     }
06:     public Edge AddEdge(IVertex v1, IVertex v2) {
07:         if (!vertices.Contains(v1))
08:             throw new VNotFoundException(""),
09:         // B2
10:         if (!vertices.Contains(v2))
11:             throw new VNotFoundException("");
12:         // B3
13:         // create edge
14:         Edge e = new Edge(v1, v2);
15:         edges.Add(e);
16:     }
17: }

Source: C# QuickGraph library
Seeker - example

- **DynAnalyzer**(B6, null)
  - $B6 \in NotCovB$ after DSE
- **StatAnalyzer**(B6, null)
  - isComputed is $tfield$, $B4$ is pre-target branch
- **DynAnalyzer**(B4, null)
  - $B4 \in NotCovB$ after DSE
- **StatAnalyzer**(B4, null)
- ...
Seeker - example

- DynAnalyzer(B1, null)

```java
01: Vertex s1 = new Vertex(0);
02: AdjacencyGraph ag = new AdjacencyGraph();
03: ag.AddVertex(s1);
```

- DynAnalyzer(B2, S2)

```java
01: Vertex s1 = new Vertex(0);
02: AdjacencyGraph ag = new AdjacencyGraph();
03: ag.AddVertex(s1);
04: ag.AddEdge(s1, null);
```

- ...And so on

```java
01: Vertex s1 = new Vertex(0);
02: AdjacencyGraph ag = new AdjacencyGraph();
03: ag.AddVertex(s1);
04: ag.AddEdge((IVertex)s1, (IVertex)s1);
05: UDFSAlgorithm ud = new UDFSAlgorithm(ag);
06: ud.Compute((IVertex)null);
```
Implementation

- Heavily based on Pex API (not that surprising)
- Pex is launched multiple times to synthesize target sequences
- Results are cached and shared between subsequent launches
- Open-source prototype is available to download
## Evaluation

- Authors compared Seeker with Pex, Randoop and manually written tests (total of 28K lines of code)

<table>
<thead>
<tr>
<th>Subject</th>
<th>Namespace</th>
<th># Branches</th>
<th>Randoop # Tests</th>
<th>Randoop Cov</th>
<th>Randoop Time</th>
<th>Pex # Tests</th>
<th>Pex Cov</th>
<th>Pex Time</th>
<th>Seeker # Tests</th>
<th>Seeker Cov</th>
<th>Seeker Time</th>
<th>Manual # Tests</th>
<th>Manual Cov</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickGraph</td>
<td>OVERALL</td>
<td>1119</td>
<td>10140</td>
<td>51.2</td>
<td>0.2</td>
<td>334</td>
<td>31.6</td>
<td>4.4</td>
<td>1923</td>
<td>68.2</td>
<td>3.2</td>
<td>21</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Algorithms</td>
<td>572</td>
<td>-</td>
<td>38.1</td>
<td>-</td>
<td>-</td>
<td>24.8</td>
<td>-</td>
<td>-</td>
<td>52.1</td>
<td>-</td>
<td>-</td>
<td>24.8</td>
</tr>
<tr>
<td></td>
<td>Collections</td>
<td>269</td>
<td>-</td>
<td>87.7</td>
<td>-</td>
<td>-</td>
<td>17.8</td>
<td>-</td>
<td>-</td>
<td>94.0</td>
<td>-</td>
<td>-</td>
<td>11.2</td>
</tr>
<tr>
<td></td>
<td>... (5 more)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dsa</td>
<td>OVERALL</td>
<td>665</td>
<td>10493</td>
<td>14.9</td>
<td>1.0</td>
<td>552</td>
<td>83.8</td>
<td>3.7</td>
<td>961</td>
<td>90</td>
<td>0.9</td>
<td>298</td>
<td>93.2</td>
</tr>
<tr>
<td></td>
<td>Algorithms</td>
<td>198</td>
<td>-</td>
<td>41.9</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>88.3</td>
</tr>
<tr>
<td></td>
<td>DataStructures</td>
<td>433</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>76.7</td>
<td>-</td>
<td>-</td>
<td>86.4</td>
<td>-</td>
<td>-</td>
<td>90.8</td>
</tr>
<tr>
<td></td>
<td>... (2 more)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xUnit</td>
<td>OVERALL</td>
<td>2379</td>
<td>10148</td>
<td>24.9</td>
<td>6.1</td>
<td>1265</td>
<td>38.6</td>
<td>4.5</td>
<td>1360</td>
<td>41.1</td>
<td>2.0</td>
<td>282</td>
<td>62.7</td>
</tr>
<tr>
<td></td>
<td>Gui</td>
<td>432</td>
<td>-</td>
<td>34.3</td>
<td>-</td>
<td>-</td>
<td>40.8</td>
<td>-</td>
<td>-</td>
<td>46.1</td>
<td>-</td>
<td>-</td>
<td>17.8</td>
</tr>
<tr>
<td></td>
<td>Sdk</td>
<td>706</td>
<td>-</td>
<td>25.1</td>
<td>-</td>
<td>-</td>
<td>35.6</td>
<td>-</td>
<td>-</td>
<td>40.2</td>
<td>-</td>
<td>-</td>
<td>86.3</td>
</tr>
<tr>
<td></td>
<td>... (6 more)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUnit</td>
<td>Util</td>
<td>1810</td>
<td>10129</td>
<td>16.1</td>
<td>1.7</td>
<td>816</td>
<td>35.3</td>
<td>7.5</td>
<td>1804</td>
<td>43.5</td>
<td>3.7</td>
<td>319</td>
<td>63.9</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>5973</td>
<td>40910</td>
<td>26</td>
<td>9.0</td>
<td>2967</td>
<td>41.3</td>
<td>20.1</td>
<td>6048</td>
<td>52.3</td>
<td>9.8</td>
<td>920</td>
<td>59.2</td>
</tr>
</tbody>
</table>

*Table 2. Branch coverage achieved by Randoop, Pex, Seeker, and manually written tests.*
Evaluation

- Defects detection

Defects detected include `OverflowException`, `IndexOutOfRangeException`, or even infinite loop in `QuickGraph`
Future work

- Loop-based sequences

```
00: public static void foo1(IntStack ints) {
01:     if(ints.size() > 3) {
02:         ... // B9
03:     }
04: }
```

- Abstract classes, interfaces and callback methods
A few links

- Seeker: http://research.csc.ncsu.edu/ase/projects/seeker/
- Seeker prototype: http://pexase.codeplex.com/releases/view/50822
- Randoop: http://code.google.com/p/randoop/
Thank you