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:
public static void foo(UDFSAlgorithm udfs) {
    ...
    if(udfs.GetIsComputed()) {
        ... // B6
    }
    // B7
}
```

```
//UDFS:UndirectedDepthFirstSearch
class UDFSAlgorithm {
  private IVEListGraph graph;
  private bool isComputed;
  public UDFSAlgorithm(IVEListGraph g){
      ... }
  public void Compute(IVertex s){ ...
    if(graph.GetEdges().Size() > 0){ // B4
       isComputed = true;
       foreach (Edge e in graph.GetEdges()){
          ... // B5
       }
    }
  } ...
}
```

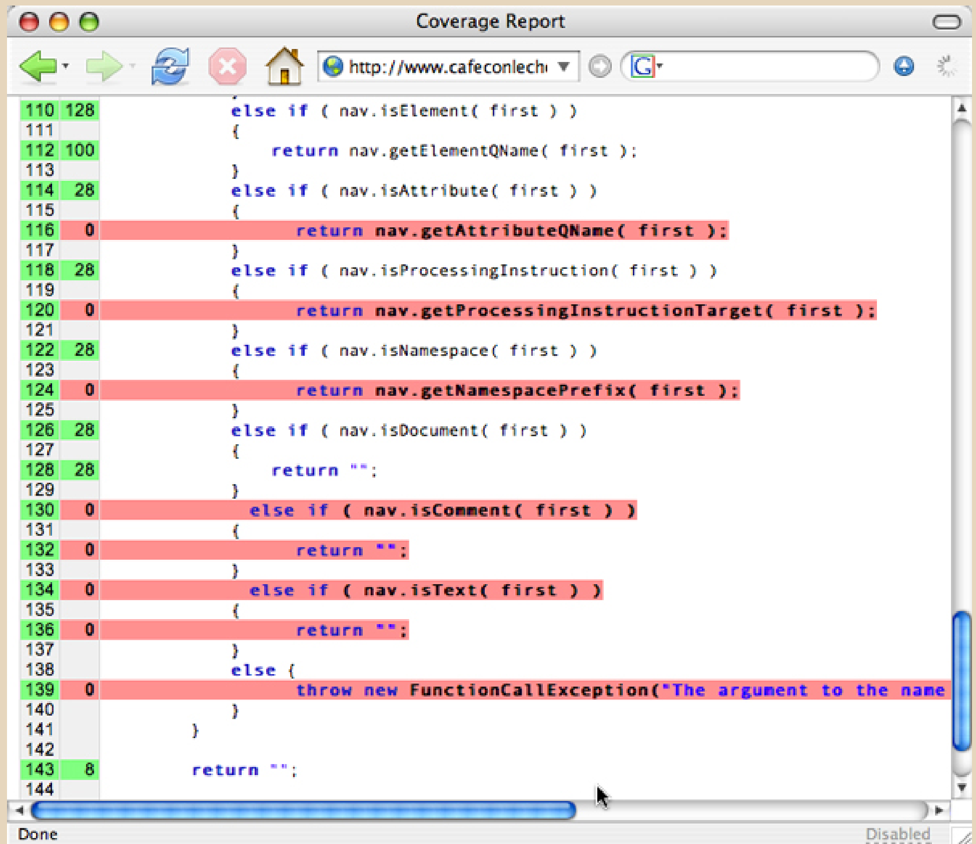
Source: C# QuickGraph library

```
class AdjacencyGraph : IVEListGraph {
  private Collection edges;
  private ArrayList vertices;
  public void AddVertex(IVertex v){
     vertices.Add(v); // B1
  }
  public Edge AddEdge(IVertex v1, IVertex v2){
     if (!vertices.Contains(v1))
        throw new VNotFoundException("");
     // B2
     if (!vertices.Contains(v2))
        throw new VNotFoundException("");
     // B3
     // create edge
     Edge e = new Edge(v1, v2);
     edges.Add(e);
  } ...
}
```

B6 B4 B3 B2 B1 →

High-coverage testing

- Branch coverage can be quite hard to achieve



Source: ibm.com

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 - $M \in M$:
  $C \times T_1 \times \ldots \times T_n \rightarrow T$, $T_i \in C \cup PrimTy$,
  $T \in C \cup PrimTy \cup \{ \mathrm{void} \}$

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 $M_i \in M$
  - $o = ret(m_k)$ for some $1 \leq k < i$ and for all $1 \leq j \leq n$

    $a_j \in PrimVal \vee a_j = null \vee a_j = ret(m_l)$ 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 ∈ *M* and a target branch *tb* within M, synthesize a method sequence (m_1, … , 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*

DynAnalyzer - ~~D~~SE

- 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 $J := \emptyset$	(intuitively, J is the set of already analyzed program inputs)
loop	
Choose program input $i \notin J$	(stop if no such i can be found)
Output i	
Execute $P(i)$; record path condition C	(in particular, $C(i)$ holds)
Set $J := J \cup C$	(viewing C as the set $\{i \mid C(i)\}$)
end loop	

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 StatAnalyzer(*tb*, *inpseq*)
6. Otherwise...

DynAnalyzer - ComputeDominants

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. Recursively invoke DynAnalyzer for each dominant branch and return method sequence if all dominant branches are covered along with *tb*
8. ...Fail otherwise

DynAnalyzer algorithm

Algorithm 1 *DynAnalyzer(tb, inpseq)*

Require: *tb* of type TB
Require: *inpseq* of type MCS
Ensure: *targetseq* of type MCS covering *tb* or `null`

```
Method m = GetMethod(tb)
SKT tmpskt = AppendMethod(inpseq, m)
DSE(tmpskt, tb, out targetseq, out CovB, out
NotCovB)
//Scenario 1
if tb ∈ CovB then
    return targetseq
end if

//Scenario 2
if tb ∈ NotCovB then
    return StatAnalyzer(tb, inpseq)
end if

//Scenario 3
if tb ∉ NotCovB then
    List<TB> tblist = ComputeDominants(tb)
    for all TB domtb ∈ tblist do
        inpseq = DynAnalyzer(domtb, inpseq)
        if inpseq == null then
            Break
        end if
    end for
    if inpseq ≠ null then
        return DynAnalyzer(tb, inpseq)
    end if
end if
return null
```

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*

StatAnalyzer - DetectField

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

StatAnalyzer - DetectField

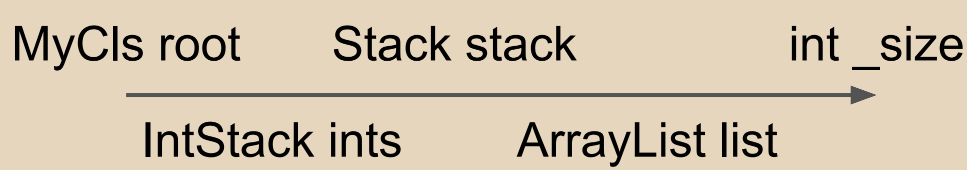
- There are two more results of DetectField, apart from *tfield*
- DetectField identifies the condition of *tfield* which is not satisfied - and should be in order to cover *tb*
- DetectField captures field hierarchy that includes objects from the one enclosing *tb* to *tfield*

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

```
public class IntStack {
   private Stack stack;
   public IntStack() {
      this.stack = new Stack; }
   public void Push(int item) {
      stack.Push(item); }
   public bool HasElements() {
      if(stack.size() > 0) { return true; }
      else { return false; }
   }
}
public class MyCls {
   private IntStack ints;
   public MyCls(IntStack ints) {
      this.ints = ints; }
   public void MyFoo() {
      if(ints.HasElements()) {
        ...// B8
      }
   }
}
```



StatAnalyzer

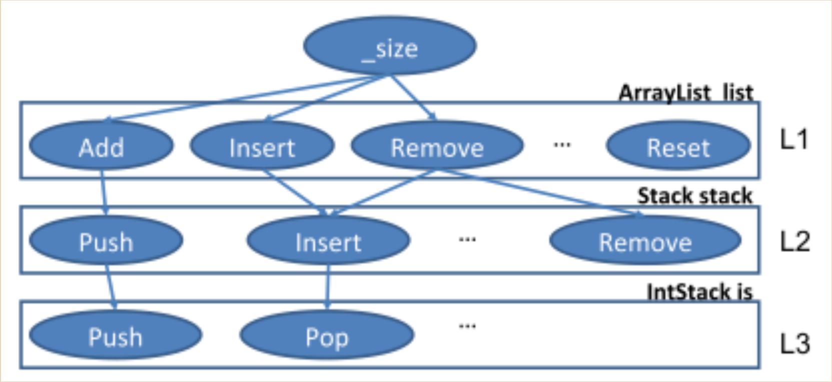
2. SuggestTargets
   - Identifies pre-target branches that need to be covered in order to cover *tb*

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

StatAnalyzer - SuggestTargets

- 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 *tfield* appears on the left side

StatAnalyzer - SuggestTargets

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

```
public class IntStack {
   private Stack stack;
   public IntStack() {
      this.stack = new Stack; }
   public void Push(int item) {
      stack.Push(item); }
   public bool HasElements() {
      if(stack.size() > 0) { return true; }
      else { return false; }
   }
}
public class MyCls {
   private IntStack ints;
   public MyCls(IntStack ints) {
      this.ints = ints; }
   public void MyFoo() {
      if(ints.HasElements()) {
        ...// B8
      }
   }
}
```

- ints is *tobject*
- IntStack.Push is identified as method which modifies *tfield*
- We need to cover branch in Push method to cover *tb*

StatAnalyzer

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

StatAnalyzer - algorithm

Algorithm 2 $StatAnalyzer(tb, inpseq)$

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

```
Field tfield = DetectField(tb)
List<TB> tblist = SuggestTargets(tfield)
for all TB pretb ∈ tblist do
    MCS targetseq = DynAnalyzer(pretb, inpseq)
    if targetseq ≠ null then
        targetseq = DynAnalyzer(tb, targetseq)
        if targetseq ≠ null then
            return targetseq
        end if
    end if
    //Try other alternative target branches
end for
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:
public static void foo(UDFSAlgorithm udfs) {
    ...
    if(udfs.GetIsComputed()) {
        ... // B6
    }
    // B7
}
```

```
//UDFS:UndirectedDepthFirstSearch
class UDFSAlgorithm {
  private IVEListGraph graph;
  private bool isComputed;
  public UDFSAlgorithm(IVEListGraph g){
     ... }
  public void Compute(IVertex s){ ...
    if(graph.GetEdges().Size() > 0){ // B4
      isComputed = true;
      foreach (Edge e in graph.GetEdges()){
        ... // B5
      }
    }
  } ...
}
```

```
class AdjacencyGraph : IVEListGraph {
  private Collection edges;
  private ArrayList vertices;
  public void AddVertex(IVertex v){
    vertices.Add(v); // B1
  }
  public Edge AddEdge(IVertex v1, IVertex v2){
    if (!vertices.Contains(v1))
      throw new VNotFoundException("");
    // B2
    if (!vertices.Contains(v2))
      throw new VNotFoundException("");
    // B3
    // create edge
    Edge e = new Edge(v1, v2);
    edges.Add(e);
  } ...
}
```

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)

```
Vertex s1 = new Vertex(0);
AdjacencyGraph ag = new AdjacencyGraph();
ag.AddVertex(s1);
```

- DynAnalyzer(B2, S2)

```
Vertex s1 = new Vertex(0);
AdjacencyGraph ag = new AdjacencyGraph();
ag.AddVertex(s1);
ag.AddEdge(s1, null);
```

- ...And so on

```
Vertex s1 = new Vertex(0);
AdjacencyGraph ag = new AdjacencyGraph();
ag.AddVertex(s1);
ag.AddEdge((IVertex)s1, (IVertex)s1);
UDFSAlgorithm ud = new UDFSAlgorithm(ag);
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)

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

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

Evaluation

- Defects detection

Subject	Randoop			Pex			Seeker		
	AT	FT	D	AT	FT	D	AT	FT	D
QuickGraph	6956	456	10	334	14	11	1923	117	34
Dsa	687	17	3	552	34	15	961	61	20
xUnit	112	0	0	1265	12	5	1360	12	5
NUnit	528	76	3	816	10	7	1804	16	13
Total	**8283**	**549**	**11**	**2967**	**70**	**38**	**6048**	**206**	**72**

AT: All Tests, FT: Failing Tests, D: Defects

Table 6. Defects detected by all approaches.

- Defects detected include OverflowException, IndexOutOfRangeException, or even infinite loop in QuickGraph

Future work

- Loop-based sequences

```
public static void foo1(IntStack ints) {
    if(ints.size() > 3) {
        ... // B9
    }
}
```

- 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
- Pex: http://research.microsoft.com/en-us/projects/pex/
- Randoop: http://code.google.com/p/randoop/
- DSE: http://people.cs.umass.edu/~yannis/dysy-icse08.pdf
- Program Synthesis: http://research.microsoft.com/en-us/um/people/sumitg/pubs/synthesis.html

Thank you