Fitness-Guided Path Exploration in Automated Test Generation

Tao Xie
Department of Computer Science
North Carolina State University
http://ase.csc.ncsu.edu/

Joint work with Nikolai Tillmann, Peli de Halleux, Wolfram Schulte from Microsoft Research

Motivation

- Testing is tedious
- Too easy to miss cases
- Old tests get stale
- Too much legacy code – what does it do?

Automated Software Testing to help
Outline

- Parameterized Unit Tests and Pex
- Dynamic Symbolic Execution
- Fitness-Guided Path Exploration
- Evaluation
- Conclusion

Unit Testing Today

A unit test is a small program with assertions.

```csharp
void AddTest()
{
    HashSet set = new HashSet();
    set.Add(7);
    set.Add(3);
    Assert.IsTrue(set.Count == 2);
}
```

Many developers write such unit tests by hand.
Parameterized Unit Testing

```csharp
void AddSpec(int x, int y)
{
    HashSet set = new HashSet();
    set.Add(x);
    set.Add(y);

    Assert.AreEqual(x == y, set.Count == 1);
    Assert.AreEqual(x != y, set.Count == 2);
}
```

Parameterized Unit Tests separate two concerns:
(1) The specification of externally visible behavior (assertions)
(2) The selection of internally relevant test inputs (coverage)

Dynamic Symbolic Execution

Code to generate inputs for:

```csharp
void CoverMe(int[] a)
{
    if (a == null) return;
    if (a.Length > 0)
        if (a[0] == 1234567890)
            throw new Exception("bug");
}
```

<table>
<thead>
<tr>
<th>Constraints to solve</th>
<th>Input</th>
<th>Observed constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>a==null</td>
<td></td>
<td>a==null &amp; &amp;</td>
</tr>
<tr>
<td>a!=null</td>
<td>{}</td>
<td>a!=null &amp; &amp;</td>
</tr>
<tr>
<td>a[0]==1234567890</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a==null &amp; &amp; a.Length&gt;0 &amp; a[0]==1234567890</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a==null &amp; &amp; a.Length&gt;0 &amp; &amp; a[0]==1234567890</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a==null &amp; &amp; a[0]==123...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[0]==123...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Negated condition

Done: There is no path left.
Parameterized Unit Test in Pex

Write \([\text{PexMethod}]\) with parameters, invoke Analysis

http://research.microsoft.com/Pex

Pex Exploration Results

Table: data and results
Generated Unit Tests

Real World Example: ResourceReader

Actual code from .NET base class libraries
Takes stream of bytes, extracts 'resource' chunks

```csharp
[SecurityPermissionAttribute(SecurityAction.LinkDemand, Flags=SecurityPermissionFlag.SerializationFormatter)]
public static class ResourceReader
{
    if (stream == null)
    throw new ArgumentNullException("stream");
    if (!stream.CanRead)

    _resCache = new Dictionary<string, ResourceLocation>(FastResourceComparer.Default);
    _store = new BinaryReader(stream, Encoding.UTF8);
    // We have a faster code path for reading resource files from an assembly.
    _res = stream as UnmanagedMemoryStream;

    BCLDebug.Log("RES:\FILE\FUGLY", "ResourceReader ctor\Stream\UnmanagedMemoryStream: *=| uns|=null");
    ReadResources();
}
```
ResourceReader

```csharp
private byte[] readResources();

if (.NET Framework 4.0)
    _defaultBinder = new TypeLimitingBinder();
else
    _defaultBinder = new TypeLimitingBinder();

// Read ResourceManager header

public virtual int ReadInt32()

if (m_stream != null)
    return m_stream.ReadByte();
else
    return 0;

return (n1 >> 24) | (n2 >> 16) | (n3 >> 8) | n4;
```

Pexing ResourceReader

Test input, generated by Pex

```
TestProject1 | TestDriver | TestDriver
TestProject1 | TestDriver | TestDriver
```
Division of Testing Labor

Parameterized Unit Tests (PUTs) separate two concerns:

• The specification of external behavior (i.e., assertions)

• The selection of internal test inputs (i.e., coverage)

PUTs == Algebraic Specifications

• A PUT can be read as a universally quantified, conditional axiom.
  \[ \forall \text{int name, int data. } \text{name} \neq \text{null } \land \text{data} \neq \text{null } \Rightarrow \text{equals(ReadResource(name, WriteResource(name, data)), data)} \]

• Teaching/training of writing specs is challenging but we do have success with teaching PUT/Pex

[http://sites.google.com/site/teachpex](http://sites.google.com/site/teachpex)
Dynamic Symbolic Execution

*Dynamic* symbolic execution (DSE) combines static and dynamic analysis:

- Execute a program multiple times with different inputs
  - build *path condition*: input constraints for the execution path on the side
  - plug in *concrete results* of operations which cannot reasoned about symbolically
- Use a constraint solver to obtain new inputs
  - solve a constraint system that represents an execution path not seen before

![Diagram of Dynamic Symbolic Execution](image)
Initially, choose Arbitrary

\[ a[0] = 0; \]
\[ a[1] = 0; \]
\[ a[2] = 0; \]
\[ a[3] = 0; \]
\[ \cdots \]

Run Test and Monitor

Constraint System

Choose an Uncovered Path

Test Inputs

Execution Path

Known Paths

Record Path Condition

Choose an Uncovered Path

Dynamic Symbolic Execution

// Check for magic number
int magicNum = _store.ReadINC32();
if (magicNum != MagicNumber)
throw new ArgumentException();

Dynamic Symbolic Execution
Dynamic Symbolic Execution

Initially, choose Arbitrary

Test Inputs

Execution Path

Constraint System

known Paths

Choose an Uncovered Path

Run Test and Monitor

Know Paths

Record Path Condition

Computer Science
Dynamic Symbolic Execution

Initially, choose Arbitrary Test Inputs
Solve Constraint System
Choose an Uncovered Path

Run Test and Monitor Execution Path
Record Path Condition

Known Paths

Test Loop Example - Loop

```csharp
public bool TestLoop(int x, int[] y) {
    if (x == 90) {
        for (int i = 0; i < y.Length; i++)
            if (y[i] == 15)
                x++;
        if (x == 110)
            return true;
    }
    return false;
}
```

Path condition: !(x == 90)

New path condition: (x == 90)

New test input: TestLoop(90, {0})
DSE Example - Loop

```csharp
public bool TestLoop(int x, int[] y) {
    if (x == 90) {
        for (int i = 0; i < y.Length; i++)
            if (y[i] == 15)
                x++;
        if (x == 110)
            return true;
    }
    return false;
}
```

Path condition:
(x == 90) && !(y[0] == 15)

↓

New path condition:
(x == 90) && (y[0] == 15)

↓

New test input:
TestLoop(90, {15})

Challenge in DSE - Loop

```csharp
public bool TestLoop(int x, int[] y) {
    if (x == 90) {
        for (int i = 0; i < y.Length; i++)
            if (y[i] == 15)
                x++;
        if (x == 110)
            return true;
    }
    return false;
}
```

Path condition:
(x == 90) && !(y[0] == 15) && !(x+1 == 110)

↓

New path condition:
(x == 90) && (y[0] == 15) && (x+1 == 110)

↓

New test input:
No solution!?
A Closer Look

```csharp
public bool TestLoop(int x, int[] y) {
    if (x == 90) {
        for (int i = 0; i < y.Length; i++)
            if (y[i] == 15)
                x++;
        if (x == 110)
            return true;
    }
    return false;
}
```

**Path condition:**

$$\text{Path condition: } (x == 90) \&\& (y[0] == 15) \&\& (0 < y.\text{Length}) \&\& ! (1 < y.\text{Length}) \&\& ! (x + 1 == 110)$$

**New path condition:**

$$\text{New path condition: } (x == 90) \&\& (y[0] == 15) \&\& (0 < y.\text{Length}) \&\& (1 < y.\text{Length})$$

$$\Rightarrow \text{Expand array size}$$

---

A Closer Look

```csharp
public bool TestLoop(int x, int[] y) {
    if (x == 90) {
        for (int i = 0; i < y.Length; i++)
            if (y[i] == 15)
                x++;
        if (x == 110)
            return true;
    }
    return false;
}
```

**Path condition:**

$$\text{Path condition: } (x == 90) \&\& (y[0] == 15) \&\& (0 < y.\text{Length}) \&\& ! (1 < y.\text{Length}) \&\& ! (x + 1 == 110)$$

**New path condition:**

$$\text{New path condition: } (x == 90) \&\& (y[0] == 15) \&\& (0 < y.\text{Length}) \&\& (1 < y.\text{Length})$$

$$\Rightarrow \text{Expand array size}$$

---

**We can have infinite paths!**

**Manual analysis → need at least 20 loop iterations to cover the target branch**

**Exploring all paths up to 20 loop iterations is infeasible:**

$$2^{20} \text{ paths}$$
Fitnex: Fitness-Guided Exploration

public bool TestLoop(int x, int[] y) {
    if (x == 90) {
        for (int i = 0; i < y.Length; i++)
            if (y[i] == 15)
                TestLoop(90, {15, 0})
                TestLoop(90, {15, 15})
            if (y[i] == 15)
                x++;
    }
    if (x == 110)
        return true;
    return false;
}

Key observations: with respect to the coverage target
- not all paths are equally promising for branch-node flipping
- not all branch nodes are equally promising to flip

• Our solution:
  - Prefer to flip branch nodes on the most promising paths
  - Prefer to flip the most promising branch nodes on paths
  - Fitness function to measure “promising” extents

Fitness Function

• Compute fitness value (distance between the current state and the goal state)
• Search tries to minimize fitness value

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Fitness function</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>$F(a == b)$</td>
<td>$</td>
</tr>
<tr>
<td>$F(a &gt; b)$</td>
<td>$(b - a) + K$</td>
</tr>
<tr>
<td>$F(a &gt;= b)$</td>
<td>$(b - a)$</td>
</tr>
<tr>
<td>$F(a &lt; b)$</td>
<td>$(a - b) + K$</td>
</tr>
<tr>
<td>$F(a &lt;= b)$</td>
<td>$(a - b)$</td>
</tr>
<tr>
<td>$F(P_1 &amp; &amp; P_2)$</td>
<td>$F(P_1) + F(P_2)$</td>
</tr>
<tr>
<td>$F(P_1 | P_2)$</td>
<td>$(F(P_1) + F(P_2))(F(P_1) + F(P_2))$</td>
</tr>
</tbody>
</table>

[Tracey et al. 98, Liu et al. 05, …]
Fitness Function for \((x == 110)\)

```java
public bool TestLoop(int x, int[] y) {
if (x == 90) {
    for (int i = 0; i < y.Length; i++)
        if (y[i] == 15)
            x++;
    if (x == 110)
        return true;
}
return false;
}
```

Fitness function: \(|110 - x|\)

Compute Fitness Values for Paths

```java
public bool TestLoop(int x, int[] y) {
if (x == 90) {
    for (int i = 0; i < y.Length; i++)
        if (y[i] == 15)
            x++;
    if (x == 110)
        return true;
}
return false;
}
```

<table>
<thead>
<tr>
<th>(x, y)</th>
<th>Fitness Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(90, {0})</td>
<td>20</td>
</tr>
<tr>
<td>(90, {15})</td>
<td>19</td>
</tr>
<tr>
<td>(90, {15, 0})</td>
<td>18</td>
</tr>
<tr>
<td>(90, {15, 15})</td>
<td>18</td>
</tr>
<tr>
<td>(90, {15, 15, 0})</td>
<td>17</td>
</tr>
<tr>
<td>(90, {15, 15, 15})</td>
<td>16</td>
</tr>
<tr>
<td>(90, {15, 15, 15, 0})</td>
<td>16</td>
</tr>
<tr>
<td>(90, {15, 15, 15, 15})</td>
<td>15</td>
</tr>
</tbody>
</table>

Give preference to flip paths with better fitness values
We still need to address which branch node to flip on paths ...
## Compute Fitness Gains for Branches

```csharp
public bool TestLoop(int x, int[] y) {
    if (x == 90) {
        for (int i = 0; i < y.Length; i++)
            if (y[i] == 15)
                x++;
        if (x == 110)
            return true;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>(x, y)</th>
<th>Fitness Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(90, [{0})</td>
<td>20</td>
</tr>
<tr>
<td>(90, {15})</td>
<td>flip b4 19</td>
</tr>
<tr>
<td>(90, {15, 0})</td>
<td>flip b2 19</td>
</tr>
<tr>
<td>(90, {15, 15})</td>
<td>flip b4 18</td>
</tr>
<tr>
<td>(90, {15, 15, 0})</td>
<td>flip b2 18</td>
</tr>
<tr>
<td>(90, {15, 15, 15})</td>
<td>flip b4 17</td>
</tr>
<tr>
<td>(90, {15, 15, 15, 0})</td>
<td>flip b2 17</td>
</tr>
<tr>
<td>(90, {15, 15, 15, 15})</td>
<td>flip b4 16</td>
</tr>
<tr>
<td>(90, {15, 15, 15, 15, 0})</td>
<td>flip b2 16</td>
</tr>
<tr>
<td>(90, {15, 15, 15, 15, 15})</td>
<td>flip b4 15</td>
</tr>
</tbody>
</table>

- Flipping Branch b4 (b3) gives us average 1 (-1) fitness gain (loss)
- Flipping branch b2 (b1) gives us average 0 fitness gain (loss)

### Compute Fitness Function

- Branch b1: i < y.Length
- Branch b2: i >= y.Length
- Branch b3: y[i] == 15
- Branch b4: y[i] != 15

## Compute Fitness Gain for Branches cont.

1. For a flipped node leading to Fnew, find out the old fitness value Fold before flipping
   - Assign Fitness Gain (Fold - Fnew) for the branch of the flipped node
   - Assign Fitness Gain (Fnew - Fold) for the other branch of the flipped node
2. Compute the average fitness gain for each branch over time
Search Frontier

- Each branch node candidate for being flipped is prioritized based on its composite fitness value:
  - (Fitness value of node - Fitness gain of its branch)
- Select first the one with the best composite fitness value
- To avoid local optimal or biases, the fitness-guided strategy is integrated with Pex’s previous search strategies

Evaluation Subjects

- A collection of micro-benchmark programs routinely used by the Pex developers to evaluate Pex’s performance, extracted from real, complex C# programs

<table>
<thead>
<tr>
<th>Subject</th>
<th>Phrase blocks</th>
<th>Phrase blocks</th>
<th>Phrase blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>96</td>
<td>17</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>29</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>19</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>28</td>
<td>21</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>21</td>
<td>22</td>
<td>25</td>
</tr>
<tr>
<td>8</td>
<td>34</td>
<td>23</td>
<td>19</td>
</tr>
<tr>
<td>9</td>
<td>29</td>
<td>24</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>25</td>
<td>25</td>
<td>44</td>
</tr>
<tr>
<td>11</td>
<td>27</td>
<td>26</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>27</td>
<td>27</td>
<td>9</td>
</tr>
<tr>
<td>13</td>
<td>27</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>14</td>
<td>39</td>
<td>29</td>
<td>21</td>
</tr>
<tr>
<td>15</td>
<td>38</td>
<td>30</td>
<td>69</td>
</tr>
</tbody>
</table>
Techniques under Comparison

- **Pex with the Fitnex strategy**
- **Pex without the Fitnex strategy**
  - Pex’s previous default strategy
- **Random**
  - a strategy where branch nodes to flip are chosen randomly in the already explored execution tree
- **Iterative Deepening**
  - a strategy where breadth-first search is performed over the execution tree

Evaluation Results

Pex w/o Fitnex on average by a factor of 1.9 improvement over Random
Pex w/ Fitnex on average by a factor of 5.2 improvement over Random
Impact

• Since Sept 17, 2008, Pex releases’ default exploration strategy integrates Fitnexion
  - http://research.microsoft.com/Pex
• Fitnexion is released as open source
  - http://www.codeplex.com/Pex

• Download counts of Pex in early Nov 2008
  - About 4000 after available for about half a year.
  - About 1000 of Pex for Visual Studio 2010 Community Technology Preview (Microsoft Incubation Software) after available for about two weeks

Case Study on Pex [TAP 2008]

• A previous version of Pex was applied on a core .NET component
  - Already extensively tested for several years
  - Assertions written by developers
  - >10,000 public methods
  - >100,000 basic blocks
• Found a significant number of benign bugs, e.g. NullReferenceException, IndexOutOfRangeException, ...
• 17 unique bugs involving
  - violation of developer-written assertions,
  - exhaustion of memory,
  - other serious issues.
Ongoing/Future Work

- Method sequence generation
- Regression test generation
- String generation (e.g., regular expressions)
- Environment mocking
- Test generalization
- Guidance from tool users
- ...

Conclusion

- Parameterized Unit Tests separate
  - Manual specification of external behavior
  - Pex’s selection of internal test inputs
- Dynamic Symbolic Execution enables Pex to deal with various complications
- Real-world challenges of path explosion call for guided path exploration
  - Fitness values of explored paths
  - Fitness gains of branches’ past flipping
- Evaluation results show the effectiveness of the new Fitnex strategy
- Fitnex has been integrated in Pex’ default strategy
Constraint Solving: Z3

- SMT-Solver (“Satisfiability Modulo Theories”)
  - Decides logical first order formulas with respect to theories
  - SAT solver for Boolean structure
  - Decision procedures for relevant theories:
    - uninterpreted functions with equalities,
    - linear integer arithmetic, bitvector arithmetic,
    - arrays, tuples
- Model generation for satisfiable formulas
  - Models used as test inputs
- Incremental solving
  - Enables efficient model minimization