Generating Source Inputs for Metamorphic Testing Using Dynamic Symbolic Execution

Eman Alatawi, Tim Miller and Harald Søndergaard

Department of Computing and Information Systems
The University of Melbourne
Motivation

Our goal

Investigate the use of dynamic symbolic execution (DSE) to work jointly with MT to generate source test inputs.

DSE <-> MT

• **Dynamic Symbolic Execution (DSE)**
  Generates tests automatically from source code using symbolic reasoning and constraint solving.

• **Metamorphic testing (MT)**
  Uses domain-specific properties to check program behaviour to alleviate the oracle problem.

**Our goal**

Investigate the use of dynamic symbolic execution (DSE) to work jointly with MT to generate source test inputs.
Symbolic Execution

Key idea
Execution of programs using symbolic input values instead of concrete data values.

```c
int twice ( int v) {
    return 2 * v ;
}

void function ( int x, int y) {
    z = twice (y);
    if (z == x)
        if (x > y+10)
            ERROR;
}```
Symbolic Execution

```c
int twice ( int v) {
    return 2 * v ;
}

void function ( int x, int y) {
    z = twice (y);
    if (z == x) {
        if (x > y+10)
            ERROR;
    }
}
```

```plaintext
X=x0 , Y = y0
PC = true

X=x0 , Y = y0 ,
z = 2*y0
PC : x0 != 2* y0

X=x0 , Y = y0 ,
z = 2*y0
PC: x0 = 2* y0

X=x0 , Y = y0 , z = 2*y0
PC: x0 != 2* y0

X=x0 , Y = y0 , z = 2*y0
PC: x0 = 2* y0

X=x0 , Y = y0 , z = 2*y0
PC: x0 = 2* y0
\( \land (x0 > y0 + 10) \)

X=x0 , Y = y0 , z = 2*y0
PC: x0 = 2* y0
\( \land (x0 \leq y0 + 10 ) \)

ERROR!
```
Symbolic Execution

```c
int twice ( int v) {
    return 2 * v ;
}

void function ( int x, int y) {
    z = twice (y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

```
X=x0, Y = y0
PC = true
```

```
X=x0, Y = y0 ,
z = 2*y0
PC : x0 != 2*y0
```

```
X=x0, Y = y0, z = 2*y0
PC: x0 = 2* y0
x = 0, y = 1
```

```
X=x0 , Y = y0 , z = 2*y0
PC: x0 = 2* y0
\land (x0 > y0 + 10)
```

```
ERROR!  x = 42 , y = 21
```

```
X=x0 , Y = y0 , z = 2*y0
PC: x0 = 2* y0
\land (x0 \leq y0 + 10 )
```

```
X=x0 , Y = y0 , z = 2*y0
PC: x0 = 2* y0
x = 1, y = 1
```
Dynamic Symbolic Execution (DSE)

```c
int twice (int v) {
    return 2*v;
}

void function ( int x, int y) {
    z = twice (y);
    if (z == x)
        if (x > y+10)
            ERROR;
    }
```

```
X=x0 , Y = y0
PC = true
```

```
X=x0 , Y = y0 ,
z = 2* y0
PC : x0 != 2* y0
```

```
X=x0 , Y = y0 ,
z = 2* y0
PC : x0 = 2* y0
```

```
X=x0 , Y = y0 , z = 2* y0
PC: x0 = 2* y0
∧ (x0 > y0 +10)
```

```
X=x0 , Y = y0 , z = 2* y0
PC: x0 = 2* y0
∧ (x0 ≤ y0 +10 )
```

ERROR!
int twice (int v) {
    return 2 * v;
}

void function (int x, int y) {
    z = twice(y);
    if (z == x)
        if (x > y + 10)
            ERROR;
}
Dynamic Symbolic Execution (DSE)

```c
int twice (int v) {
    return 2 * v;
}

void func(int x, int y) {
    z = twice(y);
    if (z == x)
        if (x > y + 10)
            ERROR;
}
```

Concrete execution:
- X = 1, Y = 2
- Z = 4
- ...
- ...
- ...

X = x0, Y = y0, z = 4
PC: x0 ! = 4

X = x0, Y = y0, z = 4
PC: x0 = 4

X = x0, Y = y0, z = 2 * y0
PC: x0 = 4
\( \land (x0 > y0 + 10) \)

X = x0, Y = y0, z = 2 * y0
PC: x0 = 4
\( \land (x0 \leq y0 + 10) \)

ERROR!
Dynamic Symbolic Execution (DSE)

Intelligent exploration of a program’s execution paths (“white-box fuzzing”).

Given a program and input, trace execution; simultaneously perform symbolic execution, collecting and storing constraints at branch points.

Each such “path constraint” (a conjunction) explicates input conditions under which the given path is followed.

Negation of one or more conjuncts corresponds to new paths; a constraint solver can generate new input exercising these.
DSE advantages

**High code coverage**
The use of a constraint solver helps reasoning about reachability considerably.

**Small size test suites**
**DSE disadvantages**

**Only generic properties are checked**
Many deviations from the expected behaviour are not found; for that, a test oracle would be needed.

**Complex constrains and path explosion**
Some types of constraints (on floating point values, or certain non-linear constraints, or constraints on references/pointers) are too hard for constraint solvers to handle.
1. **Permutation:**
   IF \( x' = \text{reverse}(x) \),
   Then
   \( \text{Sort}(x) = \text{Sort}(x') \)

2. **Addition:**
   IF \( c > 0 \land x' = [x_1+c,...,x_n+c] \),
   Then
   \( \text{sort}(x) \neq \text{sort}(x') \land \forall i \in [1, n] : (x[i] + c = x'[i]) \)

**Follow-up Test inputs**
1. \( x' = [0,1] \)

**Evaluation**
1. \([0,1] = [0,1] \) ✔

Comparing Source test output & Follow-up test outputs

**Test inputs**

1. \( x' = [0,1] \)
2. \( x' = [2,1] \)

**Test outputs**

1. \( [0,1] = [0,1] \)
2. \([0,1] \neq [1,2] \) ✔
Research Questions

**Q1** To what extent can using dynamic symbolic execution (DSE) to generate source test inputs improve the effectiveness of MT, compared with random testing?

**Q2** Can the use of MRs to generate follow-up tests be an effective technique to enhance the inputs generated by either DSE or random testing?
Automated source test generation using DSE

Key idea

MT can gain leverage from the strength of DSE in generating test inputs, compared with random testing

• Owing to its ability to
  • detect more generic faults,
  • achieve high code coverage,
  • keep the number of tests small.
Automated source test generation using DSE

Example:

```csharp
int signalsReachedThreshold (int[] signals) {
    int count = 0, threshold = 50;
    for (int i = 0; i < signals.Length; i++) {
        if (signals[i] == threshold) {
            count--; // logical error
        }
    }
    return count;
}
```

DSE generated test inputs

- Pex generates a test suite with seven test inputs:
  - null, {}, {0}, {50}, {50,0},{50,50}, {50,0,0}
- with 100% block coverage
Key idea

Using MRs to extend test suites generated randomly or by DSE has potential to cover additional parts of the code and detect more faults than either random or DSE based test suites.

• We refer to this process of generating additional tests using passed source test inputs and according to a set of MRs as “metamorphic test generation”.

Metamorphic test input generation
**Metamorphic test input generation**

**Example:**

```c
int abs(int num) {
    if (num > 0)
        return num;
    return num;  // '-' inadvertently omitted
}
```

**DSE generated test inputs**
- Pex generates a test suite with two test inputs: 0, 1
- with 100% block coverage.

**Follow-up tests using MT**
- MR \( m = -n \)  \( \Rightarrow \)  \( O(m) = O(n) \)
- Generates test inputs: 0, -1
- Input -1 reveals the error
Experiment design

**Objects of Analysis**
The five subject programs are written in C#.

**Test suites**
- Random based test suites
- DSE based test suites

**Test oracles**
- No oracle
- Metamorphic relations
- Golden oracle

Faulty versions of the programs under test
- Mutation analysis
- CREAM mutation creator tool for C#

Measures
- Block coverage
- Mutation score
## Subject programs used in experiments

<table>
<thead>
<tr>
<th>Subject</th>
<th>LOC</th>
<th>DSE source tests</th>
<th>MRs</th>
<th>Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>S: Soundex</td>
<td>60</td>
<td>21</td>
<td>5</td>
<td>41</td>
</tr>
<tr>
<td>E: Edit distance</td>
<td>15</td>
<td>5</td>
<td>2</td>
<td>104</td>
</tr>
<tr>
<td>A: Approx string matching</td>
<td>48</td>
<td>12</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>T: TCAS</td>
<td>78</td>
<td>11</td>
<td>4</td>
<td>231</td>
</tr>
<tr>
<td>B: BigInt</td>
<td>201</td>
<td>5</td>
<td>3</td>
<td>995</td>
</tr>
</tbody>
</table>
Results

### Code coverage

Using DSE base increased coverage by 16% on average for fixed test size.

The exception is BigInt which Pex struggles to handle.

### Mutation score

Using DSE base increased fault finding except for one case (BigInt).

<table>
<thead>
<tr>
<th>Subj</th>
<th>ST</th>
<th>Coverage %</th>
<th>Mutation score %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>B</td>
<td>M</td>
</tr>
<tr>
<td>S</td>
<td>DSE</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>R_5</td>
<td>90</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>R_{10n}</td>
<td>92</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>R_{100n}</td>
<td>92</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>R_{1000n}</td>
<td>96</td>
<td>100</td>
</tr>
<tr>
<td>E</td>
<td>DSE</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>R_5</td>
<td>94</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>R_{10n}</td>
<td>94</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>R_{100n}</td>
<td>94</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>R_{1000n}</td>
<td>94</td>
<td>100</td>
</tr>
<tr>
<td>A</td>
<td>DSE</td>
<td>45</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>R_5</td>
<td>41</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>R_{10n}</td>
<td>42</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>R_{100n}</td>
<td>34</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>R_{1000n}</td>
<td>47</td>
<td>99</td>
</tr>
<tr>
<td>T</td>
<td>DSE</td>
<td>97</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>R_5</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>R_{10n}</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>R_{100n}</td>
<td>28</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>R_{1000n}</td>
<td>36</td>
<td>58</td>
</tr>
<tr>
<td>B</td>
<td>DSE</td>
<td>43</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>R_5</td>
<td>59</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>R_{10n}</td>
<td>59</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>R_{100n}</td>
<td>59</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>R_{1000n}</td>
<td>59</td>
<td>65</td>
</tr>
<tr>
<td>Avg</td>
<td>DSE</td>
<td>77</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>R_5</td>
<td>61</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>R_{10n}</td>
<td>61</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>R_{100n}</td>
<td>62</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>R_{1000n}</td>
<td>67</td>
<td>84</td>
</tr>
</tbody>
</table>
Conclusion

A1 Using DSE generated source test inputs improves the effectiveness of MT compared to random test input generation.

A2 Using metamorphic test input generation to extend a test suite generated automatically using DSE or random testing can increase the code coverage considerably.

- but the improvement in the fault detection rate can be marginal and depends mainly on the strength of the used MRs.
Appendix : Used programs and MR

- **Soundex algorithm for phonetic indexing (SNDX)**
  An algorithm for computing the phonetic soundex code of a given word by collecting its consonants into classes based on the sounding similarity between them. Assume that $X = \{x_1, x_2, \cdots \}$ is the given word, and the output $O(X)$ will the soundex code of X. We derived the following MRs:

  - MR1. Inclusion
    $z \in \{a, e, i, o\} \Rightarrow O(x_1x_2\cdots) = O(x_1zx_2z\cdots z)$
  
  - MR2. Duplication
    $O(x_1x_2\cdots x_n) = O(x_1x_2\cdots x_nx_n)$
  
  - MR3. Reversal
    $|X| > 4 \land X \neq \text{reverse}(X) \Rightarrow O(X) \neq O(\text{reverse}(X))$
  
  - MR4. Deletion
    $|X| > 4 \land X[n-1] \in \{a, e, i, o\} \Rightarrow O(X) = O(X - X[n-1])$
  
  - MR5. Character case conversion
    $O(X) = O(\text{toLowerCase}(X))$
Appendix : Used programs and MR

- **Edit distance algorithm**
  An implementation of Edit distance algorithm. It calculates the total minimum number of edit operations that transforms one given string $x$ into another one $y$. We use the following MRs:

  - MR1. Limit case
    $O(X,\ '\ ') = O(\ '\ ', X) = |X|$
  - MR2. Concatenation
    $X' = X++R \land Y' = Y++S \Rightarrow$
    $O(X', Y') = \min(O(X, Y), O(X', Y) + 1, O(X, Y') + 1)$
    where $R$ and $S$ are random strings.
Appendix : Used programs and MR

- **Approximate string matching**
  A library called Fuzzy String that determines the approximate equality between two strings using 12 different algorithms such as Longest Common Subsequence, and Sørensen-Dice Distance. Assume that $X = x_1x_2 \cdots$ is the source string to be matched, $Y = y_1y_2 \cdots$ is the target string to be matched with, $t \in \{\text{strong, weak, normal}\}$ is the tolerance level, $op$ is the algorithm used, and output $O(X, Y, t, op)$ is true if and only if the two strings are approximately matched according to tolerance level $t$. We use the following MRs:
  
  - MR1. Reversal
    \[ X' = reverse(X) \& Y' = reverse(Y) \Rightarrow O(X, Y, t, op) = O(X', Y', t, op) \]
  
  - MR2. Limit cases
    \[ t \neq \text{weak} \Rightarrow O(X', t, op) = false \land O(X, X, t, op) = true \]
  
  - MR3. Monotonicity
    \[ O(X, Y, \text{strong}, op) = true \Rightarrow O(X, Y, \text{normal}, op) = O(X, Y, \text{weak}, op) = true \]
Appendix : Used programs and MR

- **Traffic Collision Avoidance System (TCAS)**
  A small aircraft conflict avoidance system. It takes 12 parameters and calculates whether there will be a conflict between a current aircraft and an approaching aircraft. Possible outputs include 0 (no action), 1 (upward), or 2 (downward). We adopted the following MRs from [17]:
  
  - MR1. Changing parameters randomly
    By randomly changing the parameters values, and \( c\text{us} = \) current vertical separation between two aircrafts at the closest point is set to be greater than 600, the following relations should hold:
    \[
    c\text{us} > 600 \land O = 0 \Rightarrow O' \in \{0, 1, 2\}
    \]
    \[
    c\text{us} > 600 \land O \neq 0 \Rightarrow O' \neq O
    \]
  - MR2. Changing aircraft’s intention
    For otherwise fixed parameters, let \( O \) be the output given \( \text{aircraft_intent} = \text{true} \) and let \( O' \) be the output given \( \text{aircraft_intent} = \text{false} \). Then \( O' = O \).
  - MR3. Changing the status of reports validity
    For otherwise fixed parameters, let \( O \) be the output given \( \text{reports_valid} = \text{true} \) and let \( O' \) be the output given \( \text{reports_valid} = \text{false} \). Then \( O' = O \).
  - MR4: Changing TCAS confidence indicator
    For otherwise fixed parameters, let \( O \) be the output given \( \text{TCAS_has_high_confidence} = \text{true} \) and let \( O' \) be the output given \( \text{TCAS_has_high_confidence} = \text{false} \). Then \( O' = O \).

- **Big Integers**
  An implementation of BigInteger class that supports large integer arithmetic operations that are not supported natively by the compilers. We focus on the addition and multiplication operations. We use the following MRs for addition (and similar ones for multiplication):
  
  - MR1. Commutation
    \( O(x, y) = O(y, x) \)
  - MR2. Association
    \( O(O(x + y), z) = O(x, O(y + z)) \)
  - MR3. Distribution
    \( z \times O(x, y) = O((x \times z), (y \times z)) \)
Thanks