Differential Symbolic Execution

Suzette Person
Dissertation Defense
July 8, 2009

PhD Committee
Matthew Dwyer, Advisor
Myra Cohen
Sebastian Elbaum
David Rosenbaum
Software Evolves

- Fix Faults
  - Re-test
  - Re-certify
  - Re-validate
  - V1.1

- Add New Features
  - Re-test
  - Re-certify
  - Re-validate
  - V2.0

- Refactor
  - Re-test
  - Re-certify
  - Re-validate
  - V2.1

...
Software Evolves

- Fix Faults
  - Re-test
  - Re-certify
  - Re-validate
  - ... → V1.1

- Add New Features
  - Re-test
  - Re-certify
  - Re-validate
  - ... → V2.0

- Refactor
  - Re-test
  - Re-certify
  - Re-validate
  - ... → V2.1

...
Motivation

while(CONDITION)
switch (CONDITION) {
...
    case 0x5c /* '\' */:
        if (CONDITION) return T_UNKNOWN;
switch (CONDITION{
    case 0x76 /* 'v' */:  tmp.append(0x0b /* 'v' */); break;
...
    default:
        if (CONDITION && CONDITION) {
...
            if (CONDITION && CONDITION && CONDITION) {
                nextByte();
            if (CONDITION && CONDITION && CONDITION) {
                nb = (byte)(nb * 8 + currByte() - 0x30);
            }

    ...
}
Motivation

```java
while(CONDITION)
    switch (CONDITION) {
        ...
        case 0x5c /* '\n' */:
            if (CONDITION) return T_UNKNOWN;
            switch (CONDITION { 
                case 0x76 /* 'v' */:  
                    tmp.append(0x0b /* '\v' */);  
                    break;
                ...
                default:
                    if (CONDITION && CONDITION) {
                        ...
                        if (CONDITION && CONDITION && CONDITION) {
                            ...
                            nextByte();
                            if (CONDITION && CONDITION && CONDITION) {
                                nb = (byte)(nb * 8 + currByte() - 0x30);
                            }
                        }
                    }
        ...
    }
}
Motivation

while(CONDITION)
    switch (CONDITION) {
        ...
        case 0x5c /* '\" */:
            if (CONDITION) return T_UNKNOWN;
            switch (CONDITION{
                case 0x76 /* 'v' */:
                    tmp.append(0x0b /* \v */); break;
                ...
                default:
                    if (CONDITION && CONDITION) {
                        ...
                        if (CONDITION && CONDITION && CONDITION) {
                            ...
                            nextByte();
                            if (CONDITION && CONDITION && CONDITION) {
                                nb = (byte)(nb * 8 + currByte() - 0x30); //check on this
                            }
                        }
                    }
                    ...
    }
}
Motivation
Differential Symbolic Execution

- Technique to detect and characterize the effects of program changes in terms of behavioral differences between program versions
  - Multi-stage analysis
  - Combines state-of-the-art symbolic execution techniques with over-approximating symbolic summaries
Differential Symbolic Execution

Source Line Diff:
50+% of source lines changed

vs.

DSE:
no functional differences found
Overview of Presentation

- DSE methodology
- Summarizing program behavior
- Notions of equivalence and deltas
- Applications of DSE
- Conclusions and future work
Symbolic Execution

```c
int computeFine(int balance) {
    if (balance <= 0) {
        return 10;
    } else {
        return 0;
    }
}
```

<table>
<thead>
<tr>
<th>balance (decimal)</th>
<th>RETURN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-100</td>
<td>10</td>
</tr>
<tr>
<td>-99</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

vs.

<table>
<thead>
<tr>
<th>condition</th>
<th>RETURN</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance &lt;= 0</td>
<td>10</td>
</tr>
<tr>
<td>!(balance &lt;= 0)</td>
<td>0</td>
</tr>
</tbody>
</table>

UNIVERSITY OF
Nebraska
Lincoln
Differential Symbolic Execution

Δ_{old}  common  Δ_{new}

Locations

Behaviors
DSE Methodology

Diagram:

DSE Step 1
- $V_i$ source
- $V_i$ common
- $V_i$ source
- diff

DSE Step 2
- Pre-computed Summaries
- Extended Symbolic Execution
- $V_i$ Symbolic Summary
- Extended Symbolic Execution
- $V_j$ Symbolic Summary

DSE Step 3
- Generate Deltas
- Check Equivalence
- $V_i \equiv V_j$
- Client Analyses
  - Impact Analysis
  - Test Suite Evolution
  - Refactoring Assurance
  - Change Characterization
  - Selective Re-certification
  - ...
DSE Methodology

DSE Step 1
- $V_i$ source
- $V_i \cdot V_j$ common
- $V_j$ source
- diff

Pre-computed Summaries

DSE Step 2
- Extended Symbolic Execution
- $V_i$ Symbolic Summary
- $V_j$ Symbolic Summary

DSE Step 3
- Generate Deltas
- Check Equivalence
- $V_i \equiv V_j$

Client Analyses
- Impact Analysis
- Test Suite Evolution
- Refactoring Assurance
- Change Characterization
- Selective Re-certification

...
DSE Methodology
DSE Methodology
Overview of Presentation

- DSE methodology
- Summarizing program behavior
- Notions of equivalence and deltas
- Applications of DSE
- Conclusions and future work
Summaries of Program Behaviors

- It is not always possible to compute complete summaries
  - Non-linear arithmetic
  - Loops and recursion
Incomplete Summaries

- Explicitly define the input space covered by the summary...

\[ i_1 \lor i_2 \lor i_3 \]
Incomplete Summaries

- ...then, focus subsequent analysis tool on behaviors not covered

![\![i_1 \lor i_2 \lor i_3]![](image)
Abstract Summaries on Common Code

Abstract Symbolic Summary:
Read: \{x, tmp\}
Write: \{tmp, old\}

void test(...){ // V_i
    s_1;
    s_2;
    s_3;

    //begin unchanged code
    ...
    for (int i=0; i<len; i++){
        tmp = tmp + x[i];
    }
    old = tmp;
    //end unchanged code
    s_n;
    s_{n+1};
    ...
}

void test(...){ // V_j
    s_a;
    s_b;
    s_c;

    //begin unchanged code
    ...
    for (int i=0; i<len; i++){
        tmp = tmp + x[i];
    }
    old = tmp;
    //end unchanged code
    s_m;
    s_{m+1};
    ...
}
Abstract Summaries on Common Code

Abstract Symbolic Summary:
Read: \{x, tmp\}
Write: \{tmp, old\}

```
void test(...){  // V_i
    S_1;
    S_2;
    S_3;

    // begin unchanged code
    ...
    for (int i=0; i<len; i++){
        tmp = tmp + x[i];
    }
    old = tmp;
    // end unchanged code
    S_n;
    S_{n+1};
    ...
}
```

```
void test(...){  // V_j
    S_a;
    S_b;
    S_c;

    // begin unchanged code
    ...
    for (int i=0; i<len; i++){
        tmp = tmp + x[i];
    }
    old = tmp;
    // end unchanged code
    S_m;
    S_{m+1};
    ...
}
```
Update Path condition

\[
\text{PC}_{V_i} \land \text{PC}_B(X_i, T_i)
\]

& Program state

\[
\nu[\text{old}]: \text{old}_B(X_i, T_i) \\
\nu[\text{tmp}]: \text{tmp}_B(X_i, T_i)
\]
Abstract Summaries on Common Code

<table>
<thead>
<tr>
<th>void test(...) { // Vi</th>
<th>void test(...) { // Vj</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1;</td>
<td>s0;</td>
</tr>
<tr>
<td>s2;</td>
<td>s1;</td>
</tr>
<tr>
<td>s3;</td>
<td>s2;</td>
</tr>
<tr>
<td></td>
<td>s3;</td>
</tr>
<tr>
<td></td>
<td>s4;</td>
</tr>
<tr>
<td></td>
<td>s5;</td>
</tr>
<tr>
<td></td>
<td>s6;</td>
</tr>
<tr>
<td>//begin unchanged code</td>
<td>//begin unchanged code</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>for (int i=0; i&lt;len; i++)</td>
<td>for (int i=0; i&lt;len; i++)</td>
</tr>
<tr>
<td></td>
<td>tmp = tmp + x[i];</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>old = tmp;</td>
<td>old = tmp;</td>
</tr>
<tr>
<td>//end unchanged code</td>
<td>//end unchanged code</td>
</tr>
<tr>
<td>sn;</td>
<td>sm;</td>
</tr>
<tr>
<td>sn+1;</td>
<td>sm+1;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Abstract Symbolic Summary:
Read: {x, tmp}
Write: {tmp, old}

Traditional symbolic execution
Suspend symbolic execution & Instantiate symbolic summary
Abstract Summaries on Common Code

Update
Path condition
PC_{V_j}: \ldots \land PC_B(X_j, T_j)
&
Program state
v[old]: old_B(X_j, T_j)
v[tmp]: tmp_B(X_j, T_j)
Overview of Presentation

- DSE methodology
- Summarizing program behavior
- Notions of equivalence and deltas
- Applications of DSE
- Conclusions and future work
Checking Equivalence

- What does “equivalent” mean?
  - Refactoring assurance: are the externally observable behaviors the same?
  - Change characterization: given a set of inputs, how are the computed values affected?
- How do we describe the differences and common behaviors?
Checking Equivalence

//old
int computeFine(int balance) {
1: if (balance < 0)
2: return 10;
3: else if (balance == 0)
4: return 10;
5: else
6: return 0;
}

//new
int computeFine(int balance) {
1: if (balance <= 0)
2: return 10;
3: else
4: return 0;
}
Checking Equivalence

\[ \text{old}_{\text{sum}} = \begin{array}{|c|c|} \hline \text{balance} < 0 & \text{RETURN} = 10 \\ \hline \text{balance} == 0 & \text{RETURN} = 10 \\ \hline !\text{(balance} < 0) \&\& !\text{(balance} == 0) & \text{RETURN} = 0 \\ \hline \end{array} \]

\[ \text{new}_{\text{sum}} = \begin{array}{|c|c|} \hline \text{balance} <= 0 & \text{RETURN} = 10 \\ \hline !\text{(balance} <= 0) & \text{RETURN} = 0 \\ \hline \end{array} \]
Functional Equivalence

\[
((\text{balance} < 0) \land \text{RETURN} = 10) \lor
((\text{balance} == 0) \land \text{RETURN} = 10) \lor
(\neg(\text{balance} < 0) \land \neg(\text{balance} == 0) \land \text{RETURN} = 0)
\]

\[\equiv\]

\[
((\text{balance} <= 0) \land \text{RETURN} = 10) \lor
(\neg(\text{balance} <= 0) \land \text{RETURN} = 0)
\]

Functionally Equivalent? √
Functional Equivalence

\[ \text{old}_{\text{sum}} \land \neg \text{new}_{\text{sum}} \]

\[ \Delta_{\text{old}} \]

\[ \text{old} \cap \text{new} \]

\[ \Delta_{\text{new}} \]

\[ \text{old}_{\text{sum}} \land \neg \text{new}_{\text{sum}} \]

\[ \neg \text{old}_{\text{sum}} \land \text{new}_{\text{sum}} \]
Partition-effects Equivalence

$$((\text{balance} < 0) \wedge \text{RETURN} = 10) \lor
((\text{balance} == 0) \wedge \text{RETURN} = 10) \lor
(\neg(\text{balance} < 0) \wedge \neg(\text{balance} == 0) \wedge \text{RETURN} = 0)$$

$$\equiv$$

$$((\text{balance} <= 0) \wedge \text{RETURN} = 10) \lor
(\neg(\text{balance} <= 0) \wedge \text{RETURN} = 0)$$

Partition-effects Equivalent?
Partition-effects Equivalence

$m$

$m'$
Partition-effects Equivalence

\[ m \]

\[ m' \]

\[ \Delta m' \]
### Partition-effects Equivalence

<table>
<thead>
<tr>
<th>Expression</th>
<th>Condition</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_{\text{old}}$</td>
<td>$\text{balance} &lt; 0$</td>
<td>$\text{RETURN} = 10$</td>
</tr>
<tr>
<td></td>
<td>$\text{balance} == 0$</td>
<td>$\text{RETURN} = 10$</td>
</tr>
<tr>
<td>Common</td>
<td>$(\neg \text{balance} &lt; 0) \land (\neg \text{balance} == 0)$</td>
<td>$\text{RETURN} = 0$</td>
</tr>
<tr>
<td>$\delta_{\text{new}}$</td>
<td>$\text{balance} &lt;= 0$</td>
<td>$\text{RETURN} = 10$</td>
</tr>
</tbody>
</table>
Overview of Presentation

- DSE methodology
- Summarizing program behavior
- Notions of equivalence and deltas
- Applications of DSE
- Conclusions and future work
Applications of DSE

- Applied to artifacts from Software-artifact Infrastructure Repository (http://sir.unl.edu)
  - JMeter
  - Siena
- Client applications
  - Refactoring assurance
  - Test suite evolution
  - Change characterization
Change Characterization

//Siena version 3
public static boolean match(byte[] x, byte[] y){
    if (x.len != y.len) return false;
    for(int i=0; i<x.len; ++i)
        if (x[i] != y[i]) return false;
    return true;
}

//Siena version 4
public static boolean match(byte[] x, byte[] y){
    if (x == null && y == null) return true;
    if (x == null || y == null || x.len != y.len) return false;
    for(int i=0; i<x.len; ++i)
        if (x[i] != y[i]) return false;
    return true;
}
### match() Version 3

<table>
<thead>
<tr>
<th>Input Partition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>X==null</td>
<td>RETURN == EXCEPTION</td>
</tr>
<tr>
<td>Y==null</td>
<td>RETURN == EXCEPTION</td>
</tr>
<tr>
<td>!(X==null) &amp;&amp; !(Y==null) &amp;&amp; (X.l != Y.l)</td>
<td>RETURN == FALSE</td>
</tr>
<tr>
<td>!(X==null) &amp;&amp; !(Y==null) &amp;&amp; (X.l != Y.l) &amp;&amp; PC_{B_1}(T, X, Y)</td>
<td>RETURN == RET_{B_1}(T, X, Y)</td>
</tr>
</tbody>
</table>

### match() Version 4

<table>
<thead>
<tr>
<th>Input Partition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>X==null &amp;&amp; Y==null</td>
<td>RETURN == TRUE</td>
</tr>
<tr>
<td>X==null &amp;&amp; !(Y==null)</td>
<td>RETURN == FALSE</td>
</tr>
<tr>
<td>!(X==null) &amp;&amp; Y==null</td>
<td>RETURN == FALSE</td>
</tr>
<tr>
<td>!(X==null) &amp;&amp; !(Y==null) &amp;&amp; (X.l != Y.l)</td>
<td>RETURN == FALSE</td>
</tr>
<tr>
<td>!(X==null) &amp;&amp; !(Y==null) &amp;&amp; (X.l != Y.l) &amp;&amp; PC_{B_1}(T, X, Y)</td>
<td>RETURN == RET_{B_1}(T, X, Y)</td>
</tr>
</tbody>
</table>
## Change Characterization

<table>
<thead>
<tr>
<th>On input</th>
<th>match() Version 3</th>
<th>match() Version 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x == null \land y == null$</td>
<td>throws NPE</td>
<td>RETURN $== \text{TRUE}$</td>
</tr>
<tr>
<td>$x == null \land y ! = \text{null}$</td>
<td>throws NPE</td>
<td>RETURN $== \text{FALSE}$</td>
</tr>
<tr>
<td>$x \neq \text{null} \land y == \text{null}$</td>
<td>throws NPE</td>
<td>RETURN $== \text{FALSE}$</td>
</tr>
</tbody>
</table>
Test Suite Evolution

- Fix Faults
  - Re-test
  - Re-certify
  - Re-validate
  - V1.1

- Add New Features
  - Re-test
  - Re-certify
  - Re-validate
  - V2.0

- Refactor
  - Re-test
  - Re-certify
  - Re-validate
  - V2.1

+ New Test Cases

- Obsolete Test Cases
Automated Test Suite Evolution

- Automatically categorize existing test cases
  - Redundant test cases
  - Invalid (obsolete) test cases
- Provide developer hints to support test case re-use
- Identify missing test cases
Automated Test Suite Evolution

\[ \Delta_{\text{old}} \quad \text{old} \cap \text{new} \quad \Delta_{\text{new}} \]

- Possibly Invalid Test Cases
- Valid Test Cases
- New Test Cases

\[ \text{old}_{\text{sum}} \quad \text{new}_{\text{sum}} \]
Overview of Presentation

- DSE methodology
- Summarizing program behavior
- Notions of equivalence and deltas
- Applications of DSE
- Conclusions and future work
Summary & Conclusions

- Contributions – Theoretical Foundations
  - Multi-stage, differential program analysis
  - Uses abstract symbolic summaries to leverage program commonalities
    - Enables client analyses to focus on the *behavioral differences*
    - Avoids fundamental limitations of symbolic execution
Summary & Conclusions

- Contributions – Theoretical Foundations
  - Two notions of equivalences and deltas
    - Functional
    - Partition-effects
  - Support for generalized symbolic execution of heap-manipulating Java programs*
  - Support for summaries of common code sequences*

*Tasks not listed in proposal
Summary & Conclusions

- Contributions – Implementation Framework
  - Text-based diff tool
  - AST-based diff tool
  - Extended symbolic execution (built on JPF & interfaces with CVC3 theorem prover)
  - Equivalence checker (interfaces with CVC3 theorem prover)
Summary & Conclusions

- Contributions – Practical Application
  - Change characterization
  - Refactoring assurance
  - Automated test suite evolution (ATSE)
Future Work

- Extend Differential Symbolic Execution
  - Apply DSE to other imperative models
  - Further explore abstract symbolic summaries
    - Partially interpreted uninterpreted functions
    - Multiple uninterpreted functions
    - Use only “as necessary”
Future Work

- Create hybrid analyses
  - Light-weight analyses
    - Impact analysis to drive symbolic execution into only the parts of the state space that are changed
  - Other techniques
    - Demand-driven symbolic execution
Future Work

- Explore new client applications of Differential Symbolic Execution
  - Software re-certification
  - Change characterization and metrics
  - Refactoring assurance
  - ...
- Perform empirical studies of cost and effectiveness of DSE
Differential Symbolic Execution

Suzette Person
Dissertation Defense
July 8, 2009

PhD Committee
Matthew Dwyer, Advisor
Myra Cohen
Sebastian Elbaum
David Rosenbaum
Related Work

- Currie et al. (Int’l Journal Par. Prog. ‘06)
- Bryant et al. (TCL ‘01)
- Jackson et al. (ICSM ‘94)
- Neamtiu et al. (MSR ‘05)
- Apiwattanapong et al. (ASE ‘07)
- Santelices et al., Apiwattanapong et al. (ASE ‘08, TAIC PART ‘06)
- Siegel et al. (ISSTA ’06, PVM/MPI ‘08)