The current State of Automated Debugging

Franz Wotawa
Technische Universität Graz
Institute for Software Technology
Inffeldgasse 16b/2, A-8010 Graz, Austria
wotawa@ist.tugraz.at

Outline

• Motivation
• Debugging techniques
  – Slicing-based debugging
  – Model-based debugging
  – Spectrum-based debugging
  – Mutation-based debugging
• Comparison
• Conclusion
MOTIVATION

Why debugging?

- Programs comprise bugs! Always! Yes, always!
- Testing & formal verifications might reduce the number of post-release bugs but there are limited resources in practice!
  - Not enough testing!
  - No complete formal verification!
Example: binary search

```java
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];
        if (midVal < key) {
            low = mid + 1;
        } else if (midVal > key) {
            high = mid - 1;
        } else {
            return mid; // key found
        }
    }
    return -(low + 1); // key not found.
}
```

Throws `ArrayIndexOutOfBoundsException`
Automated debugging – Why?

• It is a nice academic discipline!
• There are practical considerations!
  – Novices start programming / Tutoring systems for
    programming courses
  – Software Maintenance
  – Online during programming (like a grammar or
    spell checker)
  – Self-healing programs

But...

• Program size increasing
• Computational requirements
• One solution (bug candidate) might be not
  identifiable
• Multiple test cases
• Multiple bugs
• ...
What is required?

**Program (source code)**

1. public Data {
2.   public int min;
3.   public int max;
4.   public int result;
5.   public Data (int[]) input) {
6.     int i = 1;
7.     min = input[0];
8.     max = input[0];
9.     while (i < input.length) {
10.    if (input[i] < min) {
11.        min = input[i];
12.    } else if (input[i] > max) {
13.        max = input[i];
14.        i = i + 1;
15.    } result = min + max;
16.   }
}

**Test case(s)**

<table>
<thead>
<tr>
<th>TC</th>
<th>Input</th>
<th>Expected output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>input=[1]</td>
<td>result=2 min=1 max=1</td>
</tr>
<tr>
<td>B</td>
<td>input=[1,2]</td>
<td>result=3 min=1 max=2</td>
</tr>
<tr>
<td>C</td>
<td>input=[2,1,3,0]</td>
<td>result=3 min=0 max=3</td>
</tr>
<tr>
<td>D</td>
<td>input=[0,1,2,3]</td>
<td>result=3 min=0 max=3</td>
</tr>
<tr>
<td>E</td>
<td>input=[2,1]</td>
<td>result=3 min=1 max=2</td>
</tr>
</tbody>
</table>

Fault detection first!

**Program (source code)**

1. public Data {
2.   public int min;
3.   public int max;
4.   public int result;
5.   public Data (int[]) input) {
6.     int i = 2;
7.     min = input[0];
8.     max = input[0];
9.     while (i < input.length) {
10.    if (input[i] < min) {
11.        min = input[i];
12.    } else if (input[i] > max) {
13.        max = input[i];
14.        i = i + 1;
15.    } result = min + max;
16.   }
}

**Test case(s)**

<table>
<thead>
<tr>
<th>TC</th>
<th>Input</th>
<th>Computed output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>input=[1]</td>
<td>result=2 min=1 max=1</td>
</tr>
<tr>
<td>B</td>
<td>input=[1,2]</td>
<td>result=2 min=1 max=1</td>
</tr>
<tr>
<td>C</td>
<td>input=[2,1,3,0]</td>
<td>result=3 min=0 max=3</td>
</tr>
<tr>
<td>D</td>
<td>input=[0,1,2,3]</td>
<td>result=3 min=0 max=3</td>
</tr>
<tr>
<td>E</td>
<td>input=[2,1]</td>
<td>result=4 min=2 max=2</td>
</tr>
</tbody>
</table>
Fault localization and repair afterwards!

- But how?
  - Manually
  - Automated

Characteristics of debugging techniques

- Granularity (expressions, statements, methods,..)
- Kind of failure (wrong values, exceptions)
- Handling multiple faults or only single faults
- Requires one test case or many of them
- Fault localization only or with repair capabilities
What is a slice?

- A slice is a part of a program that behaves in the same way like the original program for a given set of variables at a certain location in the program. (Weiser, 1982)
- Static slicing vs. **dynamic slicing**
- Literature:
Dynamic slicing

- Based on the execution trace of a program enriched with:
  
  - *Data dependences*: A statement i depends on a statement j if there is a variable x defined in j that is used in i.
  
  - *Control dependences*: A statement i is control dependent on a test statement j (if, while,..) if the execution of j causes the execution of i.

Example

```java
1. public Data {
2.     public int min;
3.     public int max;
4.     public int result;
5.     public Data (int[] input) {
6.         int i = 2;
7.         min = input[0];
8.         max = input[0];
9.         while (i < input.length) {
10.            if (input[i] < min) {
11.                min = input[i];}
12.            if (input[i] > max) {
13.                max = input[i]; }
14.            i = i + 1; }
15.         result = min + max; }
```
Example (cont.)

```java
int i = 2;
min = input[0];
max = input[0];
while(i < input.length)
result = min + max;
```

Data dependences

Algorithm

- Slicing criterion (x,n,tc)
  - Variable x
  - Location/line number n
  - Test case tc
- “Classical” dynamic slicing algorithm:
  - Select node where x is defined the last time before executing line n. This node is part of the slice.
  - Traverse the graph backwards using the directed edges starting from that node. All nodes that are reachable are part of the slice.
But there is a problem...

```
int i = 2;
min = input[0];
max = input[0];
while(i<input.length)
result = min+max;
```

Slice for variable result does not comprise the real fault!!

Solution

- Consider also slices for test statements where the body comprise a statement defining a relevant variable, which has not been executed using the given test case.
Slicing with relevant variables

- `int i = 2;`
- `min = input[0];`
- `max = input[0];`
- `while(i<input.length)`
- `result = min+max;`

Slice for variable result considering relevant variables

Using slicing for debugging

- **Algorithm:**
  1. For all failing test cases and all variables where their stored computed value is contradicting the expected value compute a dynamic slice.
  2. Combine all dynamic slices.
- **But what means “combine”**?
  - Intersection
  - Union
Example (cont)

```java
1. public Data {
2.   public int min;
3.   public int max;
4.   public int result;
5.   public Data (int[] input) {
6.     int i = 2;
7.     min = input[0];
8.     max = input[0];
9.     while (i < input.length) {
10.        if (input[i] < min) {
11.            min = input[i];
12.        } if (input[i] > max) {
13.            max = input[i];
14.            i = i + 1;
15.        result = min + max; }
7.   }
8. }
```

- Slice for result: 6,7,8,9,10
- Slice for max: 6,8,9
- Intersection: 6,8,9
- Union: 6,7,8,9,10

Remarks on slicing

- Intersection computes smaller results than union.
- The intersection of slices can be empty (in cases of multiple faults)
- Slices can be computed fast
- Debugging restricted to statements
- Uses failing test cases only
DEBUGGING TECHNIQUES – MODEL-BASED

Basic idea behind model-based debugging

FAULTY
Program

Find causes
(Components)

EXECUTION

Discrepancies

Expected
behavior
(Test case(s))

CORRECT
Oracle
Specification
The model

- Represent a program using constraints or logic
- Use this representation for identifying the root cause
- Most important:
  - Introduce a predicate $AB / \neg AB$ stating that a statement or expression is faulty / correct respectively.

A small example

<table>
<thead>
<tr>
<th>Program</th>
<th>Test case(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $R = D / 2;$</td>
<td>• D=2, $A=\pi$, $C = 2\pi$</td>
</tr>
<tr>
<td>2. $A = R * R * \pi;$</td>
<td>• ...</td>
</tr>
<tr>
<td>3. $C = R * \pi;$</td>
<td></td>
</tr>
</tbody>
</table>
Assume Line 1 to be faulty (AB(1))

{D=2}
1. \[ R = \frac{D}{2}; \] {R=2}
2. \[ A = R \times R \times \pi; \] {A=4\pi} but {A=\pi}
3. \[ C = R \times \pi; \] {C=2\pi}

Assume Line 2 to be faulty (AB(2))

{D=2}
1. \[ R = \frac{D}{2}; \] {R=1}
2. \[ A = R \times R \times \pi; \] {A=\pi}
3. \[ C = R \times \pi; \] {C=\pi} but {C=2\pi}

INCONSISTENT!!
Assume Line 3 to be faulty (AB(3))

\[
\begin{align*}
1. & \quad R = D / 2; \\
   & \quad \{R=1\} \quad \text{CONSISTENT} \\
2. & \quad A = R * R * PI; \\
   & \quad \{A=PI\} \quad \text{and} \quad \{A=PI\} \\
3. & \quad C = R * PI; \\
   & \quad \{C=2PI\}
\end{align*}
\]

Diagnosis / root causes

- A diagnosis is a set of assumptions that statements / expressions fail that is CONSISTENT with the given test case(s).

- Simple algorithm:
  - Test all subset of the set of program statements for consistency.
Model extraction

- Program $\Rightarrow$ Loop-free representation $\Rightarrow$ Static single assignment form (SSA form) $\Rightarrow$ Constraint representation
- For statements add $\neg$AB predicates
- Example:
  - 6. $i_1 = 2$
  - $\neg$AB(6) $\Rightarrow$ $i_1 = 2$
- For more details see the presentation of Nica et al.

What happens in case of our running example?

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.</td>
<td>int $i_1 = 2$;</td>
</tr>
<tr>
<td>7.</td>
<td>min$_1$ = input[0];</td>
</tr>
<tr>
<td>8.</td>
<td>max$_1$ = input[0];</td>
</tr>
<tr>
<td></td>
<td>cond = (i &lt; input.length);</td>
</tr>
<tr>
<td>9.</td>
<td>if (cond) {</td>
</tr>
<tr>
<td></td>
<td>.....</td>
</tr>
<tr>
<td></td>
<td>min$_n$ = phi(cond,min$_i$,min$_1$);</td>
</tr>
<tr>
<td></td>
<td>max$_n$ = phi(cond,max$_i$,max$_1$);</td>
</tr>
<tr>
<td>15.</td>
<td>result$_1$ = min$_n$ + max$_n$; }</td>
</tr>
</tbody>
</table>

- Test case B:
  - input=[1,2], min=1, max=2, result=3
- Diagnoses:
  - Statement 8
  - Statement 15
  - or Statement 6 and assuming cond to evaluate to true instead of false.
Literature


Remarks on model-based debugging

- Uses all information available for debugging
- High computational requirements
- Debugging not restricted to statements
- Uses failing test cases (positive test cases can be integrated under assumptions)
DEBUGGING TECHNIQUES – SPECTRUM-BASED

Basic idea

• Consider program runs for fault localization
• A statement is less likely to be a diagnosis candidate if it is executed in passing test cases (only)
• A statement is very likely to be faulty if it is executed in failing test cases (only)
• “Tarantula”
Execution traces for each test case

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i=2;</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>min = input[0];</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>max = input[0];</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>while(i&lt;input.length) {</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>if (input[i]&lt;min) {</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>min = input[i]; }</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>if (input[i]&gt;max) {</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>max=input[i]; }</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>i = i + 1; }</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>result = min + max;</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ERROR VECTOR</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Computing the rank

- Ochiai coefficient (R. Abreu et al. 2007):

\[
s_0(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j))(a_{11}(j) + a_{10}(j))}}
\]

\[
a_{pq}(i) = \left| \{ x_{ij} = p ^ \land e_i = q \} \right|
\]

### Execution traces with coefficients

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>(s_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i=2;</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.632</td>
</tr>
<tr>
<td>min = input[0];</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.632</td>
</tr>
<tr>
<td>max = input[0];</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.632</td>
</tr>
<tr>
<td>while(i&lt;input.length) {</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.632</td>
</tr>
<tr>
<td>if (input[i]&lt;min) {</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>min = input[i]; }</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>if (input[i]&gt;max) {</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>max=input[i]; }</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>i = i + 1; }</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>result = min + max;</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.632</td>
</tr>
<tr>
<td>ERROR VECTOR</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Remarks on spectrum-based debugging

- Computation fast and easy
- Provides good results in case of well structured programs
- Not always better than slicing
  - E.g. initialization procedures, ...
- Diagnosis at the statement level
- Uses positive and negative test cases
DEBUGGING TECHNIQUES – MUTATION-BASED

Basic idea

• Use principles of genetics / genetic programming for debugging
• Operators
  – Mutation operators (swap, delete, insert, change)
  – Re-combination / cross over
• Fitness function
  – Number of passing / failing test cases
Mutations – Change op.

6. int i = 2;
7. min = input[0];
8. max = input[0];
9. while (i < input.length) {
10. if (input[i] < min) {
11. min = input[i];
12. if (input[i] > max) {
13. max = input[i];
14. i = i + 1;
15. result = min + max; }

6. int i = 1;
7. min = input[0];
8. max = input[0];
9. while (i < input.length) {
10. if (input[i] < min) {
11. min = input[i];
12. if (input[i] > max) {
13. max = input[i];
14. i = i + 1;
15. result = min + max; }

Crossover

6. int i = 2;
7. min = input[0];
8. max = input[0];
9. while (i < input.length) {
10. if (input[i] < min) {
11. min = input[i];
12. if (input[i] > max) {
13. max = input[i];
14. i = i + 1;
15. result = min + max; }
Fitness function

- Guide search for mutant that passes all test cases
- Select only mutants that are better than the one computes so far wrt. the fitness function
- Possible fitness functions
  - Number of passing test cases for a mutant
    \[ \text{fitness}(P) = \left| \{ t \in \text{NegTC} \cup \text{PosTC} \land \text{pass}(P,t) \} \right| \]
  - Weighted sum, e.g.
    \[ \text{fitness}(P) = w_{pos} \times \left| \{ t \in \text{PosTC} \land \text{pass}(P,t) \} \right| + w_{neg} \times \left| \{ t \in \text{NegTC} \land \text{pass}(P,t) \} \right| \]

Algorithm (sketch)

1. Let \( M \) be \( \{ P_{\text{orig}} \} \).
2. Minimize the set \( M \) wrt. the fitness function.
3. Let \( M' \) be the empty set.
4. For all \( P \) in \( M \) do:
   a) if \( P \) is a solution (or optimal wrt. the fitness function), return \( P \) as result.
   b) Otherwise, add all MUTATIONS\((P)\) do \( M' \) if the fitness function provides a better value than for \( P \).
   c) Select some \( P' \) from \( M \) and add CROSSOVER\((P,P')\) to \( M' \).
5. Let \( M \) be \( M' \) and go to 2.
Results

• Weimer et al. 2009 presented empirical results at ICSE using genetic programming (using a more sophisticated algorithm)
  – Programs varied from 22 to 21,553 LOC
  – Diagnosis time from 149 to 533 seconds
  – Success rate from 5 to 100 %

Remarks – Mutation-based debugging

• Fault localization and repair!
• Uses positive and negative test cases
• Granularity: Statement and Expressions
• High computational requirements
• Focusing using most probable statements (using spectrum-based methods,..)
• Literature:
## COMPARISON

### Summary of methods

<table>
<thead>
<tr>
<th></th>
<th>Slicing</th>
<th>Model-based</th>
<th>Spectrum-based</th>
<th>Mutation-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Granularity</td>
<td>Stmnts</td>
<td>Stmnts/Expr</td>
<td>Stmnts/Module</td>
<td>Stmnts/Expr</td>
</tr>
<tr>
<td>Single/Multiple</td>
<td>Both</td>
<td>Both</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>Faults</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Computational</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Type of fault</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># test cases</td>
<td>&gt;=1</td>
<td>&gt;=1</td>
<td>&gt;&gt;1</td>
<td>&gt;&gt;1</td>
</tr>
<tr>
<td>Localization/</td>
<td>Localization</td>
<td>Localization</td>
<td>Localization</td>
<td>Repair</td>
</tr>
<tr>
<td>Repair</td>
<td></td>
<td>(Repair)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some results

- Only average values from results obtained using 9 different programs
- Model-based debugging (VBM, AIM) requires from 3 to 377 seconds (avg. 28 for VBM and 185 for AIM)

<table>
<thead>
<tr>
<th>LoC</th>
<th>Tests</th>
<th>SSlice</th>
<th>DSlice</th>
<th>Exec</th>
<th>VBM</th>
<th>AIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>55.44</td>
<td>17.78</td>
<td>0.412</td>
<td>0.576</td>
<td>0.532</td>
<td>0.686</td>
<td>0.866</td>
</tr>
</tbody>
</table>

Comparison

- Every method has advantages and disadvantages
- Methods with high higher computational requirements deliver better diagnosis results
- Integration of methods to improve the overall capabilities while retaining a low computational profile required
Slicing – Model-based

- Previous work proved that slicing can be integrated into model-based reasoning
- Slices = Conflicts (a slice comprise those statements that lead to an inconsistency)
- Better results than using slicing alone (when considering the union of slices). The results are similar when using the intersection operator.
- Literature:

Spectrum-based – Model-based

- Consider execution traces as conflicts and use the coefficients of spectrum-based debugging for computing a likelihood value for the computed diagnosis.
- See:
Spectrum-based – Mutation-based

• Use information that some statements are more likely (spectrum-based)
• Only these statements are considered for mutation

CONCLUSION
Conclusion

- Focus on debugging for experienced programmers (during implementation or maintenance)
- There is no best / most accurate / optimal debugging method
- Results are encouraging but improvements are still necessary
- Integration into IDEs is still missing

Remarks

- There are other methods for debugging
  - Tutoring systems
  - Checking (of syntactical rules)
  - Delta Debugging
- More knowledge lead to better results (formal specifications,...)
Open research questions

• Comparison of methods still missing
• Integration of methods (model-based and mutation-based debugging)
• Handling of object-oriented languages
• Quality of obtained results should be improved (e.g. less candidates)
• How to obtain lower computational requirements (while not increasing the number of diagnosis candidates)

Open research questions (cont.)

• Combining testing, i.e., test case generation, and debugging
  – How to obtain a test case that distinguishes candidates?
• Abstraction and debugging (partially solved, i.e., initial work available)
• Integration of verification, testing and debugging
QUESTIONS?