Automated debugging – the past, the now, and the future

Franz Wotawa
TU Graz, Institute for Software Technology
wotawa@ist.tugraz.at
What is debugging?

• Fault detection, localization and correction in software and systems

• Fault detection: Testing and formal verification – at least partially automated

• Localization & Repair: Identifying the root cause of a detected misbehavior
0800 Andman started
0800 stopped - Andman
1000
13°C (033) MP - MC 1.38 x 10^5
(033) PRO 2.130476415
conv 2.130476415
Relays 6-2 in 033 failed special speed test
in flex arm.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multi-Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Andman started.
1700 Closed down.
Let's start with a small example

- Simple imperative language
- No conditionals!
- No loops!
- No recursive function!
- No function calls!
Example

"4-bit adder"

```
1. g1 = a1 && b1;
2. p1 = !(a1 && b1) || (!a1 && !b1));
3. g2 = a2 && b2;
4. p2 = !(a2 && b2) || (!a2 && b2));
5. g3 = a3 && b3;
6. p3 = !(a3 && b3) || (!a3 && !b3));
7. g4 = a4 && b4;
8. p4 = !(a4 && b4) || (!a4 && !b4));
9. z1 = !(c1 && p1) || (!c1 && !p1));
10. s11 = p1 && c1;
11. s12 = g1 || s11;
12. z2 = !(s12 && p2) || (!s12 && !p2));
13. s21 = c1 && p1 && p2;
14. s22 = g1 && p2;
15. s23 = (s21 || s22) || g2;
16. z3 = !(s23 && p3) || (!s23 && !p3));
17. s31 = c1 && p1 && p2 && p3;
18. s32 = g1 && p2 && p3;
19. s33 = g2 && p3;
20. s34 = ((s31 || s32) || s33) || g3;
21. z4 = !(s34 && p4) || (!s34 && !p4));
22. s41 = c1 && p1 && p2 && p3 && p4;
23. s42 = g1 && p2 && p3 && p4;
24. s43 = g2 && p3 && p4;
25. s44 = g3 && p4;
26. cn = (((g4 || s41) || s42) || s43) || s44;
```
a1 = true
a2 = false
a3 = true
a4 = false
b1 = true
b2 = false
b3 = false
b4 = false
c1 = false
z1 = false
z2 = true
z3 = true
z4 = false
cn = false

1. g1 = a1 && b1; // true
2. p1 = !((a1 && b1) || (!a1 && !b1)); // false
3. g2 = a2 && b2; // false
4. p2 = !((a2 && b2) || (!a2 && b2)); // true
5. g3 = a3 && b3; // false
6. p3 = !((a3 && b3) || (!a3 && !b3)); // true
7. g4 = a4 && b4; // false
8. p4 = !((a4 && b4) || (!a4 && !b4)); // false
9. z1 = !((c1 && p1) || (!c1 && !p1)); // false
10. s11 = p1 && c1; // false
11. s12 = g1 || s11; // true
12. z2 = !((s12 && p2) || (!s12 && !p2)); // false
13. s21 = c1 && p1 && p2; // false
14. s22 = g1 && p2; // true
15. s23 = (s21 || s22) || g2; // true
16. z3 = !((s23 && p3) || (!s23 && !p3)); // false
17. s31 = c1 && p1 && p2 && p3; // false
18. s32 = g1 && p2 && p3; // true
19. s33 = g2 && p3; // false
20. s34 = ((s31 || s32) || s33) || g3; // true
21. z4 = !((s34 && p4) || (!s34 && !p4)); // true
22. s41 = c1 && p1 && p2 && p3 && p4; // true
23. s42 = g1 && p2 && p3 && p4; // false
24. s43 = g2 && p3 && p4; // false
25. s44 = g3 && p4; // false
26. cn = (((g4 || s41) || s42) || s43) || s44; // false
Debugging

1. \( g_1 = a_1 \&\& b_1; \)
2. \( p_1 = !((a_1 \&\& b_1) || (!a_1 \&\& !b_1)); \)
3. \( g_2 = a_2 \&\& b_2; \)
4. \( p_2 = !((a_2 \&\& b_2) || (!a_2 \&\& b_2)); \)
5. \( g_3 = a_3 \&\& b_3; \)
6. \( p_3 = !((a_3 \&\& b_3) || (!a_3 \&\& !b_3)); \)
7. \( g_4 = a_4 \&\& b_4; \)
8. \( p_4 = !((a_4 \&\& b_4) || (!a_4 \&\& !b_4)); \)
9. \( z_1 = !((c_1 \&\& p_1) || (!c_1 \&\& !p_1)); \)
10. \( s_{11} = p_1 \&\& c_1; \)
11. \( s_{12} = g_1 || s_{11}; \)
12. \( z_2 = !((s_{12} \&\& p_2) || (!s_{12} \&\& !p_2)); \)
13. ...

\[
\begin{align*}
  a_1 &= \text{true} \\
  a_2 &= \text{false} \\
  a_3 &= \text{true} \\
  a_4 &= \text{false} \\
  b_1 &= \text{true} \\
  b_2 &= \text{false} \\
  b_3 &= \text{false} \\
  b_4 &= \text{false} \\
  c_1 &= \text{false} \\
  z_1 &= \text{false} \\
  z_2 &= \text{true} \\
  z_3 &= \text{true} \\
  z_4 &= \text{false} \\
  c_n &= \text{false}
\end{align*}
\]

- "\( z_2 \) has to be true"
- "This can only be the case if \( s_{12} \) and \( p_2 \) have different values"
- "Assume \( p_2 \) to be \text{false}"
- "This can only be the case if \( \neg a_2 \&\& b_2 \) is \text{true}"
- "Hence \( b_2 \) should be \( \neg b_2 \)"
- "I have to check by running the program again"
Tool demo
VISIONS (FROM THE PAST)
Mark Weiser (1952-1999)
Chief scientist at Xerox PARC

“Ubiquitous computing names the third wave in computing, just now beginning. First were mainframes, each shared by lots of people. Now we are in the personal computing era, person and machine staring uneasily at each other across the desktop. Next comes ubiquitous computing, or the age of calm technology, when technology recedes into the background of our lives.”
Program Slicing
MARK WEISER

Abstract—Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow. Starting from a subset of a program’s behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a “slice,” is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of behavior.

DEFINITIONS
Some properties of slices are presented. In particular, finding statement-minimal slices is in general unsolvable, but using data flow analysis is sufficient to find approximate slices. Potential applications include automatic slicing tools for debugging and parallel processing of slices.

Definition: A digraph is a structure (N, E), where N is a set

ness shows up in testing, parallel processor distribution, maintenance, and especially debugging. A previous study showed experienced programmers mentally slicing while debugging, based on an informal definition of slice [22]. Our concern here is with 1) a formal definition of slices and their abstract

propositions of behavior from the program being decomposed.

Unlike procedures and data abstractions, slices are designed to be found automatically after a program is coded. Their usefulness shows up in testing, parallel processor distribution, maintenance, and especially debugging. A previous study showed experienced programmers mentally slicing while debugging, based on an informal definition of slice [22]. Our concern here is with 1) a formal definition of slices and their abstract

slices of \( P \), \( \text{DEF}(n) \) is the set of variables whose values are used at \( n \), and \( \text{DEF}(n) \) is the set of variables whose values are changed at \( n \). A state trajectory of a program is just a trace of its execution which snapshots all the variable values just before executing each statement.

Definition: A state trajectory of length \( k \) of a program \( P \) is a finite list of ordered pairs

\[(n_1, s_1)(n_2, s_2) \cdots (n_k, s_k)\]

where each \( n \) is in \( N \) (the set of nodes in \( P \)) and each \( s \) is a function mapping the variables in \( V \) to their values. Each \( (n, s) \) gives the values of \( V \) immediately before the execution of \( n \).

Our attention will be on programs which halt, so infinite state trajectories are specifically excluded.

Slices reproduce a projection from the behavior of the original program. This projection must be the values of certain variables as seen at certain statements.

Manuscript received August 27, 1982; revised June 28, 1983. This work was supported in part by the National Science Foundation under Grant MCS-80-18294 and by the U.S. Air Force Office of Scientific Research under Grant F49620-80-C-001. A previous version of this paper was presented at the 5th International Conference on Software Engineering, San Diego, CA, 1981.

The author is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

0098-5589/84/0700-0352$01.00 © 1984 IEEE
Ehud Shapiro (1955- )
Prof. at Weizmann Institute of Science
Algorithmic Program Debugging

Ehud Y. Shapiro

1982 ACM Distinguished Dissertation
Other work

“Visions”

• Formalizing mental processes of debugging

• Provide a theory behind debugging

• Reduce user interaction

• Provide debugging methods for different purposes, e.g., supporting:
  – Experience programmers
  – Programming novices
Common characteristics

• These methods are based on:
  – Data dependencies
  – Control dependencies
  – Executions and tests (ako dynamic programming)
  – Planning and explanation

• Usually semi-automatic requiring (heavy) user interaction
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
... A PERSONAL VIEW
public int isTriangle(int a, int b, int c) {
    // ASSUME: a <= b <= c
    int result;
    if (c >= a + b)
        result = 1; // NO TRIANGLE
    else {
        if (a == b && b == c)
            result = 4; // EQUILATERAL
        else if (a == b || b == c)
            result = 3; // ISOSCELES
        else
            result = 2; // SCALENE
    }
    return result;
}
```java
public int isTriangle(int a, int b, int c) {
    // ASSUME: a <= b <= c
    int result;
    if (c >= a + b) {
        result = 1; // NO TRIANGLE
    } else {
        if (a == b && b == c) {
            result = 4; // EQUILATERAL
        } else if (a == b || b == c) {
            result = 3; // ISOSCELES
        } else {
            result = 2; // SCALENE
        }
    }
    return result;
}
```

<table>
<thead>
<tr>
<th>TC</th>
<th>Input</th>
<th>Computed output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a = 1, b = 1, c = 1</td>
<td>result = 4</td>
</tr>
<tr>
<td>B</td>
<td>a = 2, b = 2, c = 3</td>
<td>result = 3</td>
</tr>
<tr>
<td>C</td>
<td>a = 2, b = 3, c = 4</td>
<td>result = 3</td>
</tr>
<tr>
<td>D</td>
<td>a = 2, b = 3, c = 5</td>
<td>result = 1</td>
</tr>
</tbody>
</table>
Method 1: Slicing

• 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:
public int isTriangle(int a, int b, int c) {
    // ASSUME: a <= b <= c
    int result;
    if (c >= a + b) {
        result = 1; // NO TRIANGLE
    } else {
        if (a == b && b == c) {
            result = 4; // EQUILATERAL
        } else if (a == b || b == c) {
            result = 3; // ISOSCELES
        } else {
            result = 2; // SCALENE
        }
    }
    return result;
}

TC | Input            | Computed output
---|------------------|------------------
C  | a = 2, b = 3, c = 4 | result = 3
public int isTriangle(int a, int b, int c) {
    // ASSUME: a <= b <= c
    int result;
    if (c >= a+b) {
        result = 1; // NO TRIANGLE
    } else {
        if (a == b && b == c)
            result = 4; // EQUILATERAL
        else if (a == b || b == c)
            result = 3; // ISOSCELES
        else
            result = 2; // SCALENE
    }
    return result;
}
Method 2: Model-based debugging

• Represent a program using constraints or logic
• Use this representation for identifying the root cause
• Most important:
  – Introduce a predicate AB / ¬AB stating that a statement or expression is faulty / correct respectively.
• A diagnosis is a set of assumptions that statements / expressions fail that is CONSISTENT with the given test case(s).
```java
public int isTriangle(int a, int b, int c) {
    // ASSUME: a <= b <= c
    int result;
    if(c >= a+b)
        result = 1; // NO TRIANGLE
    else {
        if(a == b && b == c)
            result = 4; // EQUILATERAL
        else if(a == b || b == c)
            result = 3; // ISOSCELES
        else
            result = 2; // SCALENE
    }
    return result;
}
```

<table>
<thead>
<tr>
<th>TC</th>
<th>Input</th>
<th>Expected output</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>a = 2, b = 3, c = 4</td>
<td>result = 2</td>
</tr>
</tbody>
</table>

ab₄ ∨ (cond₄ = (c ≥ a+b))
ab₅ ∨ (cond₄ → result=1)
ab₇ ∨ (cond₇ = (a=b ∧ b=c))
ab₈ ∨ ((¬cond₄ ∧ cond₇) → result=4)
ab₉ ∨ (cond₉ = (a=b ∧ b=b))
ab₁₀ ∨ ((¬cond₄ ∧ ¬cond₇ ∧ cond₉) → result=3)
ab₁₂ ∨ ((¬cond₄ ∧ ¬cond₇ ∧ ¬cond₉) → result=2)
Diagnosis

• Note: $ab_i$ belongs to statement $i$

• Diagnosis = All solutions of the corresponding constraint satisfaction problem

• Diagnosis = Set of statements where the corresponding $ab$’s are set to true lead to a satisfiable constraint system
\[ a \lor (\text{cond}_4 = (c \geq a+b)) \]
\[ b \lor (\text{cond}_4 \rightarrow \text{result}=1) \]
\[ c \lor (\text{cond}_7 = (a=b \land b=c)) \]
\[ d \lor ((\neg\text{cond}_4 \land \text{cond}_7) \rightarrow \text{result}=4) \]
\[ e \lor (\text{cond}_9 = (a=b \land b=b)) \]
\[ f \lor ((\neg\text{cond}_4 \land \neg\text{cond}_7 \land \text{cond}_9) \rightarrow \text{result}=3) \]
\[ g \lor ((\neg\text{cond}_4 \land \neg\text{cond}_7 \land \neg\text{cond}_9) \rightarrow \text{result}=2) \]

\[ a=2 \]
\[ b=3 \]
\[ c=4 \]
\[ \text{ab}_9=\text{true or ab}_{10}=\text{true lead to a satisfiable constraint system} \]

```java
public int isTriangle(int a, int b, int c) {
    // ASSUME: a <= b <= c
    int result;
    if (c >= a+b)
        result = 1; // NO TRIANGLE
    else {
        if (a == b && b == c)
            result = 4; // EQUILATERAL
        else if (a == b || b == c)
            result = 3; // ISOSCELES
        else
            result = 2; // SCALENE
    }
    return result;
}
```

14.06.17 HSST 2017 Halmstad, Sweden
Remarks

• Diagnosis results depend on models
  – Strength
  – Level of abstraction, e.g., using result=2 or not_zero(result)

• High computational complexity (test all subsets of ab’s)
Literature

Method 3: Spectrum-based Fault Localization

• 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”
```c
int result;
if (c >= a + b) { result = 1; }
if (a == b && b == c) { result = 4; }
else if (a == b || b == c) { result = 3; }
else { result = 2; }
return result;
```
Computing the rank

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

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

\[ a_{pq}(i) = \left| \{ j \mid x_{ij} = p \land e_j = q \} \right| \]

```c
int result;

if (c >= a + b)
    result = 1;

if (a == b && b == c)
    result = 4;

else if (a == b || b == c)
    result = 3;

else
    result = 2;

return result;
```
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
Method 4: Mutation-based debugging

• 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 operators

```java
1. public int isTriangle(int a, int b, int c) {  
2.     // ASSUME: a <= b <= c  
3.     int result;  
4.     if(c >= a+b)  
5.         result = 1; // NO TRIANGLE  
6.     else {  
7.         if(a == b && b == c)  
8.             result = 4; // EQUILATERAL  
9.         } else if(a == b || b == c)  
10.            result = 3; // ISOSCELES  
11.            else  
12.            result = 2; // SCALENE  
13.         }  
14.         return result;  
15.     }
```

```java
1. public int isTriangle(int a, int b, int c) {  
2.     // ASSUME: a <= b <= c  
3.     int result;  
4.     if(c >= a+b+1)  
5.         result = 1; // NO TRIANGLE  
6.     else {  
7.         if(a == b && b == c)  
8.             result = 4; // EQUILATERAL  
9.         } else if(a == b || b == c)  
10.            result = 3; // ISOSCELES  
11.            else  
12.            result = 2; // SCALENE  
13.         }  
14.         return result;  
15. }
```
public int isTriangle(int a, int b, int c) {
    // ASSUME: a <= b <= c
    int result;
    if (c >= a+b) {
        result = 1; // NO TRIANGLE
    } else {
        if (a == b && b == c) {
            result = 4; // EQUILATERAL
        } else if (a == b || b == c) {
            result = 3; // ISOSCELES
        } else {
            result = 2; // SCALENE
        }
    }
    return result;
}
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
    \[ \text{fitness}(P) = \left| \left\{ t \mid t \in \text{NegTC} \cup \text{PosTC} \land \text{pass}(P,t) \right\} \right| \]
  – Weighted sum, e.g.
    \[ \text{fitness}(P) = w_{pos} \times \left| \left\{ t \mid t \in \text{PosTC} \land \text{pass}(P,t) \right\} \right| + w_{neg} \times \left| \left\{ t \mid t \in \text{NegTC} \land \text{pass}(P,t) \right\} \right| \]
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:
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 Faults</td>
<td>Both</td>
<td>Both</td>
<td>(Both)</td>
<td>Both</td>
</tr>
<tr>
<td>Computational costs</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/Repair</td>
<td>Localization</td>
<td>Localization / (Repair)</td>
<td>Localization</td>
<td>Repair</td>
</tr>
</tbody>
</table>
Other methods

• Delta debugging:

• Bayesian networks:
APPLICATIONS & TOOLS
Available tools

• There are not so many
  – Static slicing, e.g., Wala
  – Dynamic slicing, e.g., JSlice
  – Spectrum-based fault localization, e.g., GZolta
  – ....

• Most of the tools are research tools and not public available
Welcome to the T.J. Watson Libraries for Analysis (WALA)

The T. J. Watson Libraries for Analysis (WALA) provide static analysis capabilities for Java bytecode and related languages and for JavaScript. The system is licensed under the Eclipse Public License, which has been approved by the OSI (Open Source Initiative) as a fully certified open source license. The initial WALA infrastructure was independently developed as part of the DOMO research project at the IBM T.J. Watson Research Center. In 2006, IBM donated the software to the community.

For recent updates on WALA, join the mailing list, or follow WALA on Twitter or Google+.

Core WALA Features

WALA features include:

- Java type system and class hierarchy analysis
- Source language framework supporting Java and JavaScript
- Interprocedural dataflow analysis (RHS solver)
- Context-sensitive tabulation-based slicer
**JSlice**

**Latest version: 2.0 (April 15, 2008)**

**JSlice** is a dynamic slicing tool for Java programs. It collects and analyzes an execution trace (for slicing) in a compressed form.

- **What is dynamic slicing?**

  Dynamic slicing is a technique for program debugging and understanding. Given a program $P$, the programmer provides a slicing criterion of the form $(I, L, V)$, where $I$ is a program input, $L$ is a set of some statement instances during execution of program $P$ with input $I$, and $V$ is a set of variables referenced by $L$. The purpose of slicing is to find out statements in $P$ which have affected the values of $V$ at $L$ during execution, via dynamic control or data dependencies. So, if during program execution, the values of $V$ at $L$ were "unexpected", the corresponding slice can be inspected to explain the reason for the unexpected values. More on dynamic slicing can be found in research paper [2].

- **Why collect, and analyze an execution trace in compressed form?**

  Dynamic slicing is performed on an execution trace by detecting dynamic control and data dependencies. However, the size of an execution trace may be huge. Consequently, it is important to compactly represent the execution trace and perform program analysis (e.g. dynamic slicing) over the compact representation.

- **How does the compression scheme work in the tool?**

  The compactness of the trace representation is owning to several factors. First, bytecodes which do not correspond to memory access or control transfer are not traced. Second, the sequences of addresses used by memory access or control transfer bytecodes are stored separately. Since these sequences typically have high repetition of patterns, we exploit such repetition to save space. We modify a well-known lossless data compression algorithm called SEQUITUR [1] for this purpose. More technical details about this tool have been discussed in our paper [3].

- **Are there similar tools available?**

  To the best of our knowledge, there is no other dynamic slicing tool available for Java. If you are interested, you can find static slicing tools for C and Java [here](#).
Automatic Testing & Debugging using Spectrum-based Fault Localization (SFL) for Eclipse™.
Why are there not so many tools?

• Adaptation effort
  – “Every programming language is different”

• Maybe not so useful
  – Programmers know their code well (during development)
  – Quick response required
  – User interface should be well adapted
But there are niches!

• End user programming
  – Testing & debugging for spreadsheets

• Debugging support for nightly builds
  – Who to blame?

• Debugging support for maintenance
  – Who to blame?
  – Support programmer

• Teaching programming ako Tutoring Systems
... and more...

• Integration of fault localization into testing tools

• Improve the user interface

• Combine different debugging methods
WHAT’S NEXT?
What we have now!
Integration of techniques

• Integration with testing
  – Debugging tools closely interlinked with testing tools

• Integration of debugging tools
  – Improve quality of fault localization results
  – Reduce computation time
Find the right niches!
End user programming

• Programmers without experience in programming & SE
  – Testing
  – Fault localization & correction

• High error rate & often wrong corrections
Motivation

• DEOS project
  (http://spreadsheets.ist.tugraz.at/)
  – Automated fault localization for spreadsheet programs
  – (End) user support ➔ interactive debugging
  – Investigate on foundations

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Cardiogenic Shock Estimator</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><strong>End Diastolic Volume</strong></td>
<td>120 mL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>60 mL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>72 bpm</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 m2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><strong>Stroke Volume</strong></td>
<td>2 mL</td>
<td>=B2/B3</td>
</tr>
<tr>
<td>4</td>
<td><strong>Cardiac Output</strong></td>
<td>144 mL/min</td>
<td>=B6*B4</td>
</tr>
<tr>
<td>5</td>
<td><strong>Cardiac Index</strong></td>
<td>72 mL/min/m2</td>
<td>=B7/B5</td>
</tr>
</tbody>
</table>

2.160 mL/min/m2

Fault: "/" instead of "-"
Motivation

• Panko (1): estimation of human error rate to be between 3% to 5% (formulas).
• Hofer et al. (2): Wrong user input influences fault localization!

Fig. 4. Effect histograms for different types of misclassification. \( W_{rel} < 0 \) indicates an improvement of the ranking of the faulty cell. \( W_{rel} > 0 \) indicates a worsening of the ranking of the faulty cell. All outliers (i.e. data sets with \( W_{rel} > 10 \)) are summarized in the rightmost bar.
WHAT COMES AFTER THE NICHEs?
What do we want?

• Real automated debugging!

• Automated!
  – Fault detection
  – Fault localization
  – Fault correction

• Almost no user-interaction
Today’s failure handling

Microsoft Outlook has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Restart Microsoft Outlook

Please tell Microsoft about this problem.

Microsoft Outlook created an error report that you can send to help us improve Microsoft Outlook. We will treat this report as confidential and anonymous.
But wouldn’t it be nice to see the following?

Microsoft Outlook has encountered a problem and needs to close. We are correcting the fault now and restart the application in 2 minutes. Thank you for being patient.
But there are challenges!

• Need to know more:
  – program’s purpose,
  – requirements, and
  – the underlying domain

• Keep computational costs low
• Integration into current processes and development tools
What we need!

• Knowledge about the program, its purpose and its environment!

• Artificial Intelligence methods may be applicable!
  – Machine Learning (because of data gained during execution)
  – Model-based / Logic-based approaches (because of properties, requirements, etc.)
Combination of Machine Learning and Reasoning is necessary (and a hot future topic)

John Launchbury, Director I2O, DARPA
https://www.youtube.com/watch?v=-O01G3tSYpU
What do take with us?

• Machine Learning for model extraction
  – IBM Watson, NELL, ...
• Combination of Statistics-based methods and reasoning
• Orientation towards explanations
  – Natural language processing & understanding

• For debugging:
  – Learn specification / requirements from executions
  – Use programs for reasoning
  – Background knowledge necessary
We also need...

• ... a new execution architecture including
  – A monitoring system for fault detection based on properties or user feedback
  – A fault localization module for identifying the root cause of the failure
  – A fault correction module for correcting the program using stored test cases for ensuring meeting requirements
  – An automatic test system that stores tests and executes them.
public int isTriangle(int a, int b, int c) {
    // ASSUME: a <= b <= c
    int result;
    if (c >= a+b)
        result = 1; // NO TRIANGLE
    else
        if(a == b && b == c)
            result = 4; // EQUILATERAL
        else if(a == b || b == c)
            result = 3; // ISOSCELES
        else
            result = 2; // SCALENE
    return result;
}
Self-* programs

• Fault correction should be transparent to the user
  – Fault detection may not be communicated to the user
  – Correction process might not be visible

• Quality assurance has to be guaranteed
  – Past (correct) computations should not be influenced
  – Only correction of faults
Summary

- In the past debugging research focuses on providing a theory

- Automated debugging support is possible!

- Currently, there are not enough tools available!

- Tool integration is needed
Summary

• There are some next big things

  – Niches for automated debugging tools (e.g. spreadsheets)

  – Self-* programs that correct themselves
Recent surveys

• For more information:

http://paris.utdallas.edu/qrs17/contest.html

INTRODUCTION
Software plays an integral part in our lives and in our technological society. Therefore, taking a test driven approach is critical.

To promote advanced software testing skills, the First International Conference on Software Quality, Reliability, and Security (QRS 2017) is organizing a Software Testing Contest on July 26 in Prague, Czech Republic.

CONTEST GUIDELINES
All applications have to be submitted by 7/20/2017.

The contest has two parts:
- The first part is a 1-hour tutorial in the morning of July 26. During this tutorial, we will help contestants properly configure their laptops, and explain how to use MoocTest.net platform. Each contestant is required to have basic knowledge about Java programming and JUnit.
- The second part is a 3-hour contest where contestants are required to design test cases based on the given specifications to test three Java programs. The quality of these test cases is determined based on the code coverage (statement and branch) and mutation score achieved.

RUNNING ENVIRONMENT
The laptop used for this contest must install the following software:
- Java JDK 1.7 (cannot be other JDK versions)
- Eclipse Luna (cannot be other Eclipse versions)
- A working web browser

If you cannot find the installation package of the above software, you can download them from the contest website. We will also help you set up the environment during the Tutorial session.

HOW TO APPLY
Please send emails to Ruizhi Gao at youtianzui.nju@gmail.com if you want to apply for this contest. Please provide your first name, last name, and affiliation in the application email.

AWARDS
A panel of judges will rank all contestants based on their achieved code coverage and mutation score (detailed evaluation criteria is yet to be decided) and select a grand prize winning contestant that will receive $1,200 US dollars. Contestants in the 2nd-3th place will receive $800 and $500. Contestants in the 4th-10th place will receive $300 US dollars. All contestants may receive a cash support according to their travel expenses. The detailed policy is yet to be determined.

CONTEST VENUE
The First International Software Testing Contest will be held at the Czech Technical University in Prague, Faculty of Information Technology (Thákurova 9, 166 00 Praha 6 Czech Republic GPS: 50.105120, 14.390232)

IMPORTANT DATES
• 7/20/2017: Application deadline
• 7/26, 2017: Tutorial & Contest
• 7/27, 2017: Ceremony
END OF PART 1

QUESTIONS?