# 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?

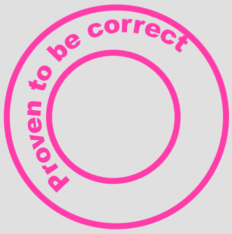
- Programs comprise bugs! Always! Yes, always!
- Testing & formal verifications might reduce the number of post-release bugs but there are limited resources in practice!
  - Not enough testing!
  - No complete formal verification!

# Example: binary search

```
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`

| | |
|---|---|
| **Bug ID:** | 5045582 |
| **Votes** | 0 |
| **Synopsis** | (coll) binarySearch() fails for size larger than 1<<30 |
| **Category** | java:classes_util |
| **Reported Against** | tiger-beta |
| **Release Fixed** | mustang(b83) |
| **State** | 10-Fix Delivered, Verified, bug |
| **Priority:** | 2-High |
| **Related Bugs** | 6412541 , 6437371 , 5050278 , 4306897 |
| **Submit Date** | 11-MAY-2004 |
| **Description** | |

```
FULL PRODUCT VERSION :
java version "1.5.0-beta"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-beta-b32c)
Java HotSpot(TM) Client VM (build 1.5.0-beta-b32c, mixed mode)


ADDITIONAL OS VERSION INFORMATION :
Linux freeway 2.4.21-4-686 #1 Sat Aug 2 23:27:25 EST 2003 i686 GNU/Linux


A DESCRIPTION OF THE PROBLEM :
java.util.Arrays.binarySearch() will throw an ArrayIndexOutOfBoundsException if the array
is large.  This is caused by overflow in the calculation:

          int mid = (low + high) >> 1;

The correct calculation uses unsigned shift:

          int mid = (low + high) >>> 1;

There are similar problems in Collections, and TreeMap also includes the faulty calculation

          int mid = (lo + hi) / 2;

There may be others.
```

## 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)

```
public Data {
    public int min;
    public int max;
    public int result;
    public Data (int[] input) {
        int i = 1;
        min = input[0];
        max = input[0];
        while (i < input.length) {
            if (input[i] < min) {
                min = input[i];}
            if (input[i] > max) {
                max = input[i]; }
            i = i + 1; }
        result =min + max; } }
```

## Test case(s)

| TC | Input | Expected output |
|---|---|---|
| A | input=[1] | result=2<br>min=1<br>max=1 |
| B | input=[1,2] | result=3<br>min=1<br>max=2 |
| C | input=<br>[2,1,3,0] | result=3<br>min=0<br>max=3 |
| D | input=<br>[0,1,2,3] | result=3<br>min=0<br>max=3 |
| E | input=[2,1] | result=3<br>min=1<br>max=2 |

# Fault detection first!

```
public Data {
    public int min;
    public int max;
    public int result;
    public Data (int[] input) {
        int i = 2;
        min = input[0];
        max = input[0];
        while (i < input.length) {
            if (input[i] < min) {
                min = input[i];}
            if (input[i] > max) {
                max = input[i]; }
            i = i + 1; }
        result =min + max; } }
```

| TC | Input | Computed output |
|---|---|---|
| A | input=[1] | result=2<br>min=1<br>max=1 |
| B | input=[1,2] | result=2<br>min=1<br>max=1 |
| C | input=<br>[2,1,3,0] | result=3<br>min=0<br>max=3 |
| D | input=<br>[0,1,2,3] | result=3<br>min=0<br>max=3 |
| E | input=[2,1] | result=4<br>min=2<br>max=2 |

## 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

# DEBUGGING TECHNIQUES - SLICING

## 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:
  - Mark Weiser, *Programmers Use Slices when Debugging*, Communication of the ACM, 25(7), 1982.
  - Frank Tip, *A Survey of Program Slicing Techniques*, Journal of Programming Languages, 3(3), 1995.
  - Richard A. DeMillo and Hsin Pan and Eugene H. Spafford, *Critical Slicing for Software Fault Localization*, International Symposium on Software Testing and Analysis (ISSTA), 1996.

# Dynamic slicing

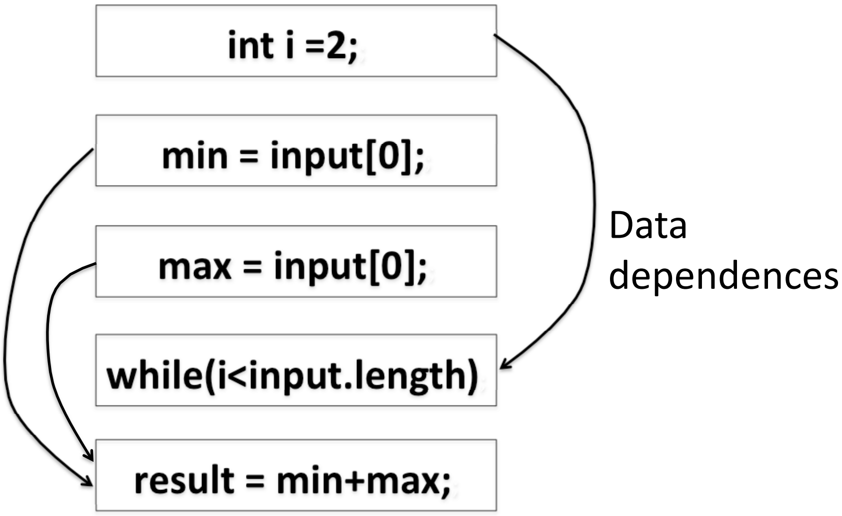
- Based on the execution trace of a program enriched with:
  - *Data dependences*: A statement i depends on a statement j if there is a variable x defined in j that is used in i.
  - *Control dependences*: A statement i is control dependent on a test statement j (if, while,..) if the execution of j causes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

# Example

```
public Data {
    public int min;
    public int max;
    public int result;
    public Data (int[] input) {
        int i = 2;
        min = input[0];
        max = input[0];
        while (i < input.length) {
            if (input[i] < min) {
                min = input[i];}
            if (input[i] > max) {
                max = input[i]; }
            i = i + 1; }
        result =min + max; } }
```

- Test case B:
  - input=[1,2], min=1, max=2, result=3

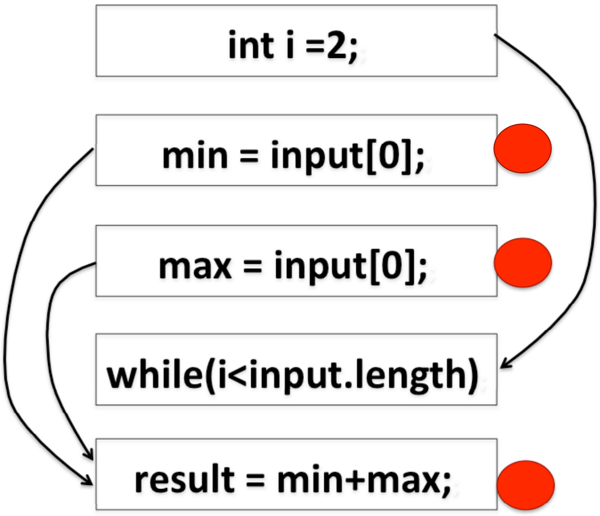
# Example (cont.)




# 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…

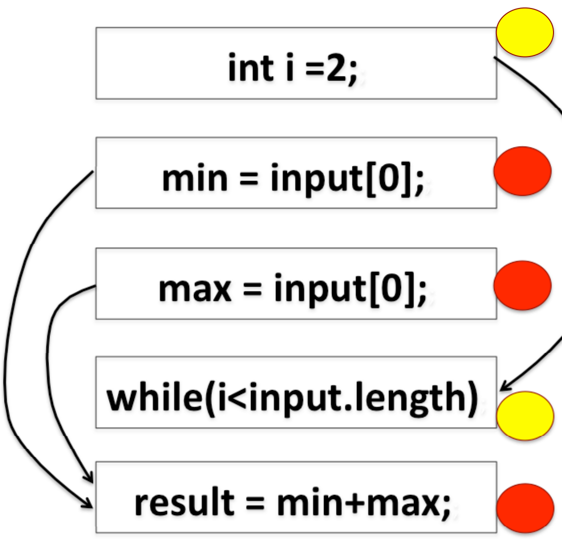



Slide for variable result does not comprise the real fault!!

# 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




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)

```
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; } }
```

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

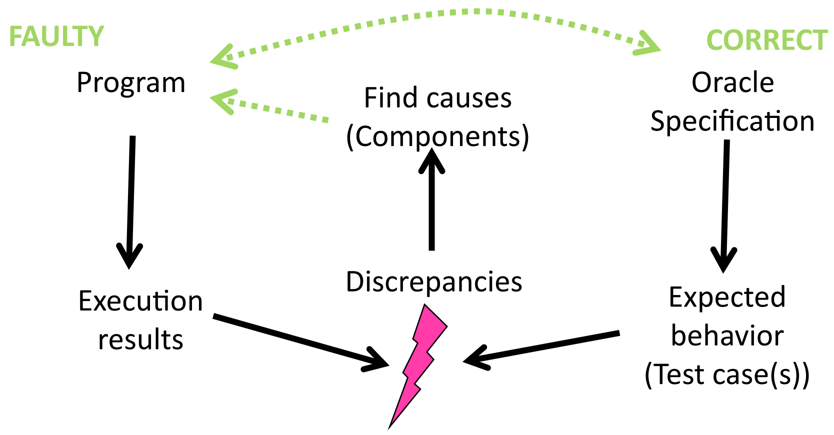
- Intersection:
  6,8,9
- Union:
  6,7,8,9,10

# 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



# The model

- 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 small example

**Program**

```
R = D / 2;
A = R * R * PI;
C = R * PI;
```

**Test case(s)**

- D=2, A=$\pi$, C = 2$\pi$
- ...

# Assume Line 1 to be faulty (AB(1))

{D=2}

1. ~~R = D / 2;~~

{R=2}

2. A = R * R * PI; **INCONSISTENT!!**

{A=4PI} but {A=PI}

3. C = R * PI;

{C=2PI}

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

{D=2}

1. R = D / 2;

{R=1}

2. ~~A = R * R * PI;~~ **INCONSISTENT!!**

{A=PI}

3. C = R * PI;

{C=PI} but {C=2PI}

## Assume Line 3 to be faulty (AB(3))

```
    {D=2}
1. R = D / 2;
    {R=1}
2. A = R * R * PI;      CONSISTENT
    {A=PI} and {A=PI}
3. C = R * PI;
    {C=2PI}
```

## 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 ⇒ Loop-free representation ⇒ Static single assignment form (SSA form) ⇒ Constraint representation
- For statements add ¬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?

```
6.   int i_1 = 2;
7.   min_1 = input[0];
8.   max_1 = input[0];
     cond = (i < input.length);
9.   if (cond) {
     .....}
     min_n=ϕ(cond,min_i,min_1);
     max_n=ϕ(cond,max_j,max_1);
15.  result_1 =min_n + max_n; } }
```

- 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


- Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa, Model-Based Debugging of Java Programs, Proc. Intl. Workshop on Automated and Algorithmic Debugging (AADEBUG), Munich, Germany, 2000.
- Wolfgang Mayer, Markus Stumptner, Dominik Wieland, and Franz Wotawa, Can AI help to improve debugging substantially? Debugging experiences with value-based models, Proc. European Conference on Artificial Intelligence (ECAI), Lyon, France, 2002.
- Wolfgang Mayer. *Static and Hybrid Analysis in Model-based Debugging*. PhD thesis, School of Computer and Information Science, University of South Australia, Adelaide, Australia, July 2007.
- Wolfgang Mayer and Markus Stumptner. Evaluating Models for Model-Based Debugging. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 128–137, L'Aquila, Italy, September 2008. IEEE Computer Society Press.


# 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”
  - James A. Jones, Mary Jean Harrold, John Stasko, *Visualization of Test Information to Assist Fault Localization*, Proceedings of the 24th International Conference on Software Engineering, 2002.

# Execution traces for each test case

| | A | B | C | D | E | Rank |
|---|---|---|---|---|---|---|
| int i=2; | 1 | 1 | 1 | 1 | 1 | |
| min = input[0]; | 1 | 1 | 1 | 1 | 1 | |
| max = input[0]; | 1 | 1 | 1 | 1 | 1 | |
| while(i<input.length) { | 1 | 1 | 1 | 1 | 1 | |
| if (input[i]<min) { | 0 | 0 | 1 | 0 | 0 | |
| min = input[i]; } | 0 | 0 | 1 | 0 | 0 | |
| if (input[i]>max) { | 0 | 0 | 1 | 1 | 0 | |
| max=input[i]; } | 0 | 0 | 1 | 1 | 0 | |
| i = i + 1; } | 0 | 0 | 1 | 1 | 0 | |
| result = min + max; | 1 | 1 | 1 | 1 | 1 | |
| ERROR VECTOR | 0 | 1 | 0 | 0 | 1 | |

# Computing the rank

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

$$s_0(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) * (a_{11}(j) + a_{10}(j))}}$$

$$a_{pq}(i) = \left|\left\{ i \middle| x_{ij} = p \wedge e_i = q \right\}\right|$$

- R. Abreu, P. Zoeteweij, and A.J. van Gemund, *On the accuracy of spectrum-based fault localization*, Testing: Academia and Industry Conference (TAIC PART), 2007.

# Execution traces with coefficients

| | A | B | C | D | E | $s_0$ |
|---|---|---|---|---|---|---|
| int i=2; | 1 | 1 | 1 | 1 | 1 | 0.632 |
| min = input[0]; | 1 | 1 | 1 | 1 | 1 | 0.632 |
| max = input[0]; | 1 | 1 | 1 | 1 | 1 | 0.632 |
| while(i<input.length) { | 1 | 1 | 1 | 1 | 1 | 0.632 |
| if (input[i]<min) { | 0 | 0 | 1 | 0 | 0 | 0 |
| min = input[i]; } | 0 | 0 | 1 | 0 | 0 | 0 |
| if (input[i]>max) { | 0 | 0 | 1 | 1 | 0 | 0 |
| max=input[i]; } | 0 | 0 | 1 | 1 | 0 | 0 |
| i = i + 1; } | 0 | 0 | 1 | 1 | 0 | 0 |
| result = min + max; | 1 | 1 | 1 | 1 | 1 | 0.632 |
| ERROR VECTOR | 0 | 1 | 0 | 0 | 1 | |

# 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.

```
int i = 2;
min = input[0];
max = input[0];
while (i < input.length) {
    if (input[i] < min) {
        min = input[i];}
    if (input[i] > max) {
        max = input[i]; }
    i = i + 1; }
result =min + max; } }
```

```
int i = 1;
min = input[0];
max = input[0];
while (i < input.length) {
    if (input[i] < min) {
        min = input[i];}
    if (input[i] > max) {
        max = input[i]; }
    i = i + 1; }
result =min + max; } }
```

# Crossover

```
int i = 1;
min = input[0];
max = input[0];
while (i < input.length) {
    if (input[i] < min) {
        min = input[i];}
    if (input[i] > max) {
        max = input[0]; }
    i = i + 1; }
result =min + max; } }
```

```
int i = 2;
min = input[0];
max = input[0];
while (i < input.length) {
    if (input[i] < min) {
        min = input[i];}
    if (input[i] > max) {
        max = input[i]; }
    i = i + 1; }
result =min + max; } }
```

```
int i = 1;
min = input[0];
max = input[0];
while (i < input.length) {
    if (input[i] < min) {
        min = input[i];}
    if (input[i] > max) {
        max = input[i]; }
    i = i + 1; }
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

$$fitness(P) = \left| \left\{ t \middle| t \in NegTC \cup PosTC \wedge pass(P,t) \right\} \right|$$

  - Weighted sum, e.g.

$$fitness(P) = w_{pos} * \left| \left\{ t \middle| t \in PosTC \wedge pass(P,t) \right\} \right| + w_{neg} * \left| \left\{ t \middle| t \in NegTC \wedge pass(P,t) \right\} \right|$$

# Algorithm (sketch)

1. Let M be { $P_{orig}$ }.
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:
  - W. Weimer, T.V. Nguyen, C. Le Goues, S. Forrest, *Automatically finding Patches Using Genetic Programming*, Intl. Conference on Software Engineering (ICSE), 2009.

# COMPARISON

# Summary of methods

| | Slicing | Model-based | Spectrum-based | Mutation-based |
|---|---|---|---|---|
| Granularity | Stmnts | Stmnts/Expr | Stmnts/Module | Stmnts/Expr |
| Single/Multiple Faults | Both | Both | Both | Both |
| Computational costs | Low | High | Low | High |
| Type of fault | | | | |
| #test cases | >=1 | >=1 | >>1 | >>1 |
| Localization/ Repair | Localization | Localization / (Repair) | Localization | Repair |

## Some results

- Taken from W. Mayer and M. Stumptner, *Evaluating Models for Model-Based Debugging*, Automated Software Engineering (ASE), 2008
- 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)

| LoC | Tests | SSlice | DSlice | Exec | VBM | AIM |
|---|---|---|---|---|---|---|
| 55.44 | 17.78 | 0.412 | 0.576 | 0.532 | 0.686 | 0.866 |

## 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:
  - Franz Wotawa, *On the Relationship between Model-based Debugging and Program Slicing*, Artificial Intelligence, 135(1-2), 2002.

# 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:
  Rui Abreu, Peter Zoeteweij and Arjan J.C. van Gemund, *Localizing Software Faults Simultane ously*, 9th International Conference on Quality Software (QSIC), Jeju, Korea, 2009

## Spectrum-based – Mutation-based

- Use information that some statements are more likely (spectrum-based)
- Only these statements are considered for mutation
- To some extend introduced in W. Weimer, T.V. Nguyen, C. Le Goues, S. Forrest, *Automatically finding Patches Using Genetic Programming*, Intl. Conference on Software Engineering (ICSE), 2009

## 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

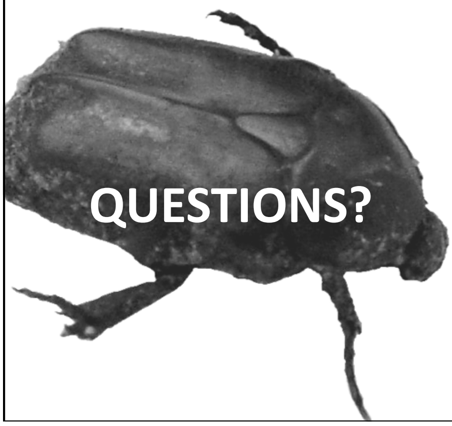
- There are other methods for debugging
  - Tutoring systems
  - Checking (of syntactical rules)
  - Delta Debugging
- More knowledge lead to better results (formal specifications,…)

## 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?