ensoft
conquer complexity

Learn to Build Automated Software Analysis Tools with Graph Paradigm and Interactive Visual Framework

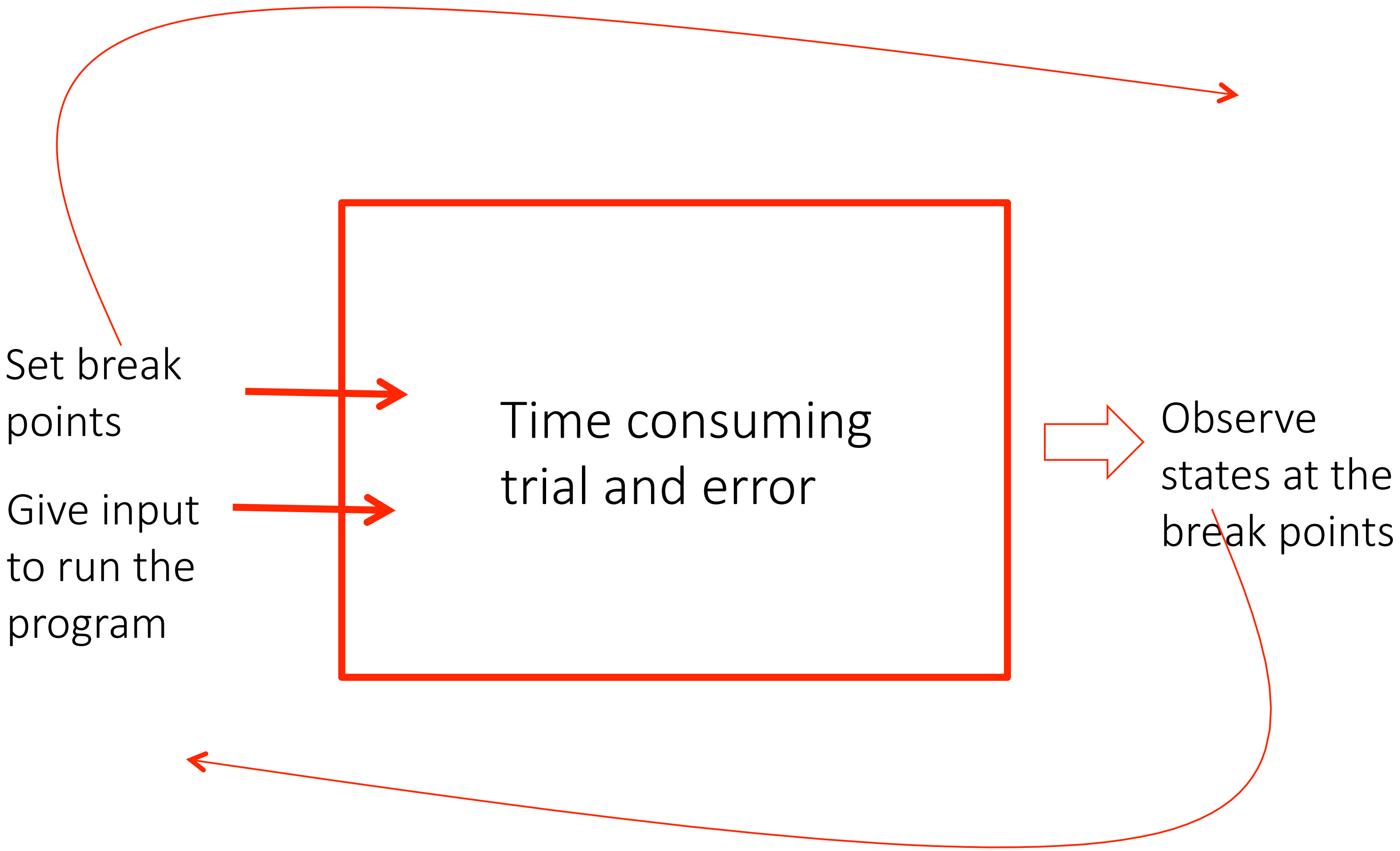
GIAN, September 12-16, 2016

Suresh C. Kothari
Richardson Professor
Department of Electrical and Computer Engineering

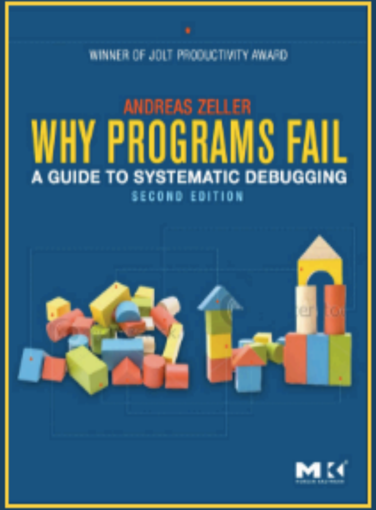
Ben Holland, Iowa State University

Module: Why Programs Fail

Acknowledgement: **Team members at Iowa State University and EnSoft, DARPA contracts FA8750-12-2-0126 & FA8750-15-2-0080**



What can we control to make debugging efficient?



Learn more

- About the Book
- About the Author
- Contents
- Reviews

Buy the book

For readers

- Code and Resources
- Slides and Illustrations
- Instructor's Page
- Errata

Publishers' sites

- Elsevier
- dpunkt.verlag

Paperback · 423 pages
ISBN 978-0-12-374515-6 [US]
ISBN 978-3-89864-620-8 [DE]

WHY PROGRAMS FAIL: A Guide to Systematic Debugging

"Today every computer program written is also debugged, but debugging is not a widely studied or taught skill. Few books, beyond this one, present a systematic approach to finding and fixing programming errors."
—from the foreword by JAMES LARUS, Microsoft Research

WHY PROGRAMS FAIL is a book about bugs in computer programs, how to reproduce them, how to find them, and how to fix them such that they do not occur anymore. This book teaches a number of techniques that allow you to debug any program in a systematic, and sometimes even elegant way. Moreover, the techniques can widely be automated, which allows you to let your computer do most of the debugging.

Learn more about the book, its author, or its contents.

News

2012-09-03: I now offer a free Udacity Online Course on Debugging, which neatly complements the book and vice versa. Enjoy!

2009-12-02: The second edition is out, and the site has been updated.

2006-12-30: A Google preview of the book is now available, containing excerpts of all chapters.

2006-12-03: Arnoud Buzing writes in StickyMinds: "With *Why Programs Fail,* Andreas Zeller has written a most wonderful text on the topic of systematic debugging...This is a classic book that I will place on a shelf near my desk as a reference."

2006-10-23: All code examples of the book can now be downloaded as a single archive.

2006-08-24: John Lam of Dr. Dobbs writes: "This is a practical book where you find excellent discussions, everything from tracking defects to debugging. If you want to write better software, read this book."

2006-03-17: *Why Programs Fail* has won a Software Development Jolt Productivity Award! This is a great honor, and I am deeply grateful to the judges and organizers for this result. My editor, Tim Cox, has been able to attend the ceremony and accept the award.

2005-12-24: Greg Wilson of Dr. Dobbs reviews *Why Programs Fail*: "This well-written, copiously-illustrated book is, in many ways, a status report from the front lines...instead of high-level handwaving, we get a detailed look at what particular tools do, how, and (most importantly) why."

Get the book at Amazon.com · Amazon.de
Comments? Write to Andreas Zeller <zeller@whyprogramsfail.com>.

What the book teaches

- Make debugging efficient:
  - by choosing *minimal* input that produces the bug
  - by using program graphs to set appropriate break points
- Book discusses at length:
  - the advantages using minimal input
  - The extreme difficulty of coming up with the minimal input
  - An algorithm, called DDMIN, to derive *one-minimal* input
- Book introduces the concept the *backward slice* as a program graph to determine the break points

Why *minimal input*? – an illustrative example

- Observed behaviors of a *sorting* routine with different choices of inputs:
  - Input: 3,7,5,11,10; Output: 3,5,7,10,11 – no failure
  - Input: 4,2,8; Output: 4,2,8 – failure
- Observed behaviors by choosing minimal inputs:
  - Input: 2,8; Output: 2,8 – no failure
  - Input: 4,8; Output: 4,8 – no failure
  - Input: 4,2; Output: 4,2 – failure
- How does the minimal input help?

It is hard to determine the minimal input

- Suppose, an input S with *n* elements produces a failure. In the worst case, 2^n subsets of S must be examined to find the minimal input.
- The book presents DDMIN algorithm as an innovative efficient solution to find the minimal input.
- The DDMIN finds the *one-minimal* input and not *minimal* input.
- What is the distinction?
  - A *minimal input* T_1 implies: (a) T_1 produces the same failure as S, and (b) No subset of T_1 can produce the same failure.
  - A *one-minimal input* T_2 implies: (a) T_2 produces the same failure as S, and (b) No subset of T_2 with one less element can produce the same failure

An abstraction that models debugging

- Inputs as sets
- Let S be the input set which produces the failure
- Debugging runs can be modeled by a binary function F defined on input sets. For an input set X, F(X) = 1 (*fail*) or 0 (*pass*).
  - 1: X produces the failure, and 0: X does not produce the failure.
  - Note that F(S) =1.
- The DDMIN algorithm produces a subset T such that F(T) = 1, and F(X) = 0 for all subsets X of T that have one less element than T.
- Note: DDMIN does not produce an absolute minimal T. So, there could be a subset X of T smaller by two or more elements and F(X) =1.

Motivating the DDMIN

- Starting point: Input is S and F(S) = 1. Let n be the number elements in S.
- A straightforward algorithm would be:
  - Iterate over all subsets X_i of S such that $|X_i| = n$-1
  - If $F(X_i) = 0$ for all I, we are done. S is the one-minimal subset.
  - If $F(X_i) = 1$ for some i, then treat X_i as the new starting point and repeat the process.
- The maximum number of steps would be: n + (n-1) + (n-2) + .. + 2 + 1. Thus, the straight forward algorithm is order n^2.
- The novelty of DDMIN lies in an intelligent strategy for constructing $X_{i.}$

The DDMIN

- Starting point: Input is S and F(S) = 1. Let n be the number elements in S.
- DDMIN steps:
  - Iterate over all subsets X_i of S such that $|X_i|$ = *n/2 (or n/2 - ½)*
  - If $F(X_i) = 0$ for all I, repeat with new choices for subsets X_i of S such that $|X_i|$ = *¾n (or the next integer smaller than ¾n).* This continues until we get $|X_i| = n-1$.
  - If $F(X_i) = 1$ for some i, then treat X_i as the new starting point and repeat the process.
- Thus, the DDMIN is order *n* log(*n*).
- Instead of removing one element at a time to construct a new input, DDMIN removes ½ the size of the original set, then ¼ the size of the original set, and so on until it comes down to removing *one* element.

An example

- Let S = {a, b, c, d}
- Let F be defined as: F(S) = 1, F({b, c}) = 1, and F = 0 on all other subsets of S.
- Note that S is 1-minimal but not the absolute-minimal input.
- The absolute minimal input is {b, c}. DDMIN will not find it.

a b c d

1st iteration – take away ½ input

a b   c d

2nd iteration – take away 1/4th input

a b c   d   b c d   a

Conclusion: a b c d ← 1-minimal

An interesting research idea

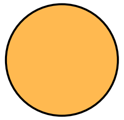
- In important applications such as parsing, if an input T produces a failure then all supersets of T also produce the same failure.
- For such applications, the binary function F() satisfies the property:
  - F(T) = 1 implies F(X) = 1 for all supersets X of T
- Research questions:
  - Is one-minimal subset always also the absolute minimal if F() satisfies the above property?
  - Can we use the assumption to find an algorithm faster than DDMIN?

More research ideas

Script for feeding the input instead of human

Input structure aware counterpart of DDMIN?

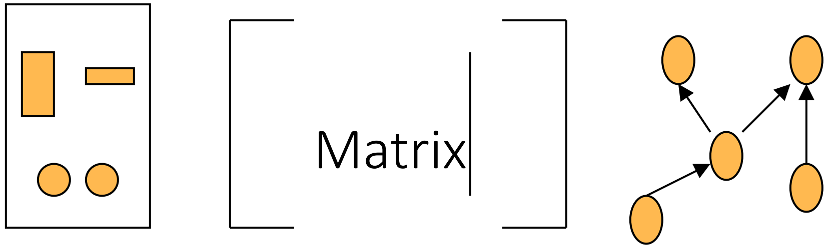
Feed the input



Minimize

?

Smaller input



GUI Clicks

Complex Input

Graph

Automation challenges

1. Smaller?
2. Minimization?
3. Script instead of human?