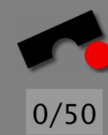


The Scientific Method

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

www.askigor.org

AskIgor – Automated Debugging Service – Mozilla {Build ID: 2002072204}

File Edit View Go Bookmarks Tools Window Help Debug QA

http://www.askigor.org/ Search

AskIgor BETA
AUTOMATED DEBUGGING

Debug a Program · Query Status · About AskIgor

What is your program file? (Linux PC executables only. *More info...*)
File: /home/zeller/pub/cc1 [Browse...]

How do you invoke your program such that it fails? (*More info...*)
Cmd: cc1 –O fail.i

How do you invoke it such the failure does *not* occur? (*More info...*)
Cmd: cc1 –O pass.i

Do the invocations above require additional files? (*More info...*)
File 1: /home/zeller/pub/pass.i [Browse...]
File 2: /home/zeller/pub/fail.i [Browse...]
File 3: [Browse...]
If more than three files are required, please upload them in an archive.

Debug it!

Powered by Delta Debugging · © 2002 Saarland University
FAQ · Usage · Forum · Terms of service

Document: Done (0.664 secs)

Submit buggy program

⇓

Specify invocations

⇓

Push button

⇓

Get diagnosis
by e-mail

Guessing Causes

Some people are very good at *guessing causes:*

- look at code
- point finger at screen
- and tell you: “Did you try X?”.

You try X—and voilà! The failure is gone.

Such *intuition* comes from experience with earlier errors.

Alternatives to Intuition

Intuition is a concept that is hard to grasp.

Sought: a method to find failure causes that

- *does not require* a priori *knowledge*
- *works in a systematic and reproducible fashion*

Enter the *scientific method!*

The Scientific Method

Scientific Method: A general pattern of how to find a theory that explains (and predicts) some aspect of the universe.

1. Observe some aspect of the universe.
2. Invent a tentative description, called a *hypothesis*, that is consistent with what you have observed.
3. Use the hypothesis to make predictions.
4. Test those predictions by experiments or further observations and modify the hypothesis.
5. Repeat steps 3 and 4 until there are no discrepancies between hypothesis and experiment and/or observation.

Eventually, the hypothesis thus becomes a *theory*.

The Scientific Method of Debugging

In the debugging context, the scientific method looks like this:

1. Observe a failure.
2. Invent a hypothesis as to the failure cause that is consistent with the observations and the necessary conditions.
3. Use the hypothesis to make predictions.
4. Test the hypothesis by experiments or further observations and modify the hypothesis.
5. Repeat steps 3 and 4 until you found the actual cause.

But: What is a cause?

And: How do we know a cause is a cause?

Causes and Effects

A *cause* is *an event preceding another event without which the event in question (the effect) would not have occurred.*

A defect causes the failure if the failure *would not have occurred* without the defect.

Debugging = search for a defect that causes the failure;
Debugging = search for *causality*.

Causes and Effects (2)

In natural and social sciences, causality is often hard to establish. Just think about common disputes such as

- Did usage of the butterfly ballot in West Palm Beach cause George W. Bush to be president of the United States?
- Did drugs cause the death of Elvis Presley?
- Does human production of carbon dioxide cause global warming?

To determine whether these are actually causes, formally, we would have to repeat history *without the cause in question.*

Consequence: *speculation, confusion*—and conspiracy theory.

Causes and Effects (3)

Computer science is different from natural science.

We can

- easily repeat program runs over and over,
- change the circumstances of the execution as desired, and
- observe the effects.

Given the right means, the program execution is under total control and totally deterministic.

Debugging is the only scientific discipline which can claim dealing with actual causality.

Establishing Failure Causes


```
a = compute_value();
printf("a = %d\n", a);
```

This code, when executed, prints `a = 0` on the console.

Why does this happen?

Establishing Failure Causes (2)


```
a = compute_value();
printf("a = %d\n", a);
```

We set up a first hypothesis:

`a` *being zero is the cause for* `a = 0` *being printed.*

Establishing Failure Causes (3)


```
a = compute_value();
a = 1;    // New code
printf("a = %d\n", a);
```

In this experiment, `a = 0` is *still* being printed.
The hypothesis is *disproven.*

We set up a new hypothesis:

`a = 0` *is being printed regardless of the value of* `a`.

Establishing Failure Causes (4)

It turns out that a is declared as double:

```
double a;
⋮
a = compute_value();
a = 1;
printf("a = %d\n", a);
```

Consequence: a *mismatch* between the format and the type.

New hypothesis:

The format "%d" *is the cause for* a = 0 *being printed.*

Establishing Failure Causes (5)

A proper format for floating-point values in `printf` is "%f".

We alter the format to this value:

```
a = compute_value();
printf("a = %f\n", a);
```

Now, `a = 1.0` is printed. The failure has gone.

Thus, we have a *theory*:

The format "%d" *is the cause for* `a = 0` *being printed.*

Explicit Contrast

Our theory is yet *imprecise*:

> *The format* "%d" *is the cause for* a = 0 *being printed.*

This theory does not state the alternate world.

A more precise statement about the failure cause is thus

> *The format* "%d" *(rather than* "%f"*) is the cause for* a = 0 *being printed.*

This is called *explicit contrast.*

Explicit Contrast (2)

Why is explicit contrast important?

```
x = 0.0;
x = g();
return 1 / x;  /* Division by zero */
```

Failure cause:

`x` *is assigned the* `g()` *return value*

But what's the alternative?

Explicit Contrast (3)

Failure cause: “x is assigned the g() return value”

If we remove the assignment altogether, the program still fails:

```
x = 0.0;
/* x = g(); */
return 1 / x;  /* Division by zero */
```

Precise cause with explicit contrast:

x *is assigned the value of* g() *instead of* f()

or

g() *returns a zero value instead of a non-zero value*

Changes vs. Fixes

Each cause carries a change:

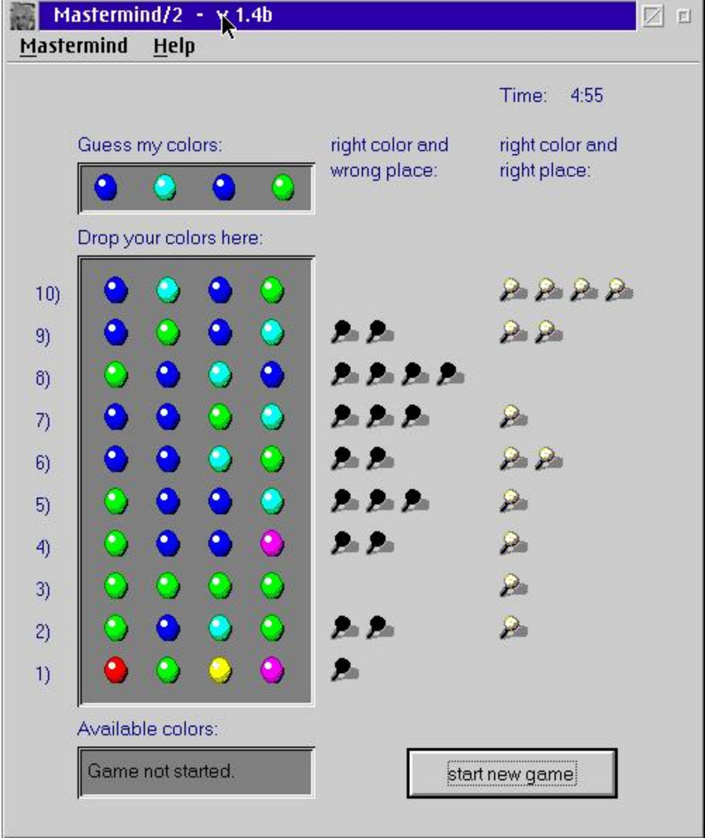
The format "%d" *(rather than* "%f"*) is the cause for* a = 0 *being printed.*

This change (here: from "%d" to "%f") does make the failure go away.

However, this change does not necessarily fix the problem once and for all; its only purpose is to *prove the cause.*

While each fix is a change, not every change is a fix!

A Mastermind Game



Keep a Notebook

Robert M. Pirsig on *motorcycle maintenance:*

> Everything gets written down, formally, so that you know at all times where you are, where you've been, where you're going, and where you want to get.
>
> In scientific work and electronics technology this is necessary because otherwise the problems get so complex you get lost in them and confused and forget what you know and what you don't know and have to give up.

Real programs are typically much more complex than motorcycles!

Keep a Notebook (2)

Your notes should include the following points:

1. the *statement* of the problem,
2. *hypotheses* as to the cause of the problem,
3. *experiments* designed to test each hypothesis,
4. *predicted results* of the experiments,
5. *observed results* of the experiments, and
6. *conclusions* from the results of the experiments.

Notebook Example

Problem statement

a = 0 is being printed, but a is not supposed to be zero.

Hypothesis #1 *Variable* a *being zero is the cause for* a = 0 *being printed.*

Experiment Set a to 1.

Predicted result a = 1 is being printed.

Observed result a = 0 is being printed.

Conclusion Hypothesis #1 is *rejected.*

Hypothesis #2 *The format* "%d" *is the cause for* a = 0 *being printed.*

Experiment Alter the format to "%f".

Predicted result a = ⟨*some non-zero value*⟩ is being printed.

Observed result (as predicted)

Conclusion Hypothesis #2 is *confirmed.*

More about Notebooks

- With a well-kept notebook, you can always quit work and resume next morning.
- If you are a lazy writer, set up a *form* with entries like "Predicted result", "Observed result", etc.
- After the problem has been fixed, it may be helpful to *archive* the notes—in case a similar problem occurs.

State the Problem

Often, the mere act of stating a problem explicitly can help to understand it.

> One university center kept a Teddy bear near the help desk. Students with mysterious bugs were required to explain them to the bear before they could speak to a human counselor. To the general surprise, several problems could be fixed simply by stating them explicitly.
>
> (From Kernighan, *The practice of programming*)

tating a problem explicitly is the first step in keeping a notebook!

Debug Quick and Dirty

If you want to fix problems without going into the hassle of some defined process, you can debug *quick and dirty:*

1. Think hard about the problem
2. Fix it.

Don’t think about following any specific process. Just think hard and solve the problem.

But do this for *10 minutes* only—afterwards, go for the scientific method!

Finding “the” Cause

Problem: As each cause implies a specific change, there’s an *infinity* of causes:

- *The format* `"%d"` *(rather than* `"%f"`*) is the cause for* `a = 0` *being printed.*
- *The* `printf` *statement as a whole (rather than the empty statement) is the cause for* `a = 0` *being printed.*
- *The whole program code is a cause, because we can rewrite it from scratch such that it works.*

How do we distinguish “the” cause from “a” cause?

The Closest Possible World

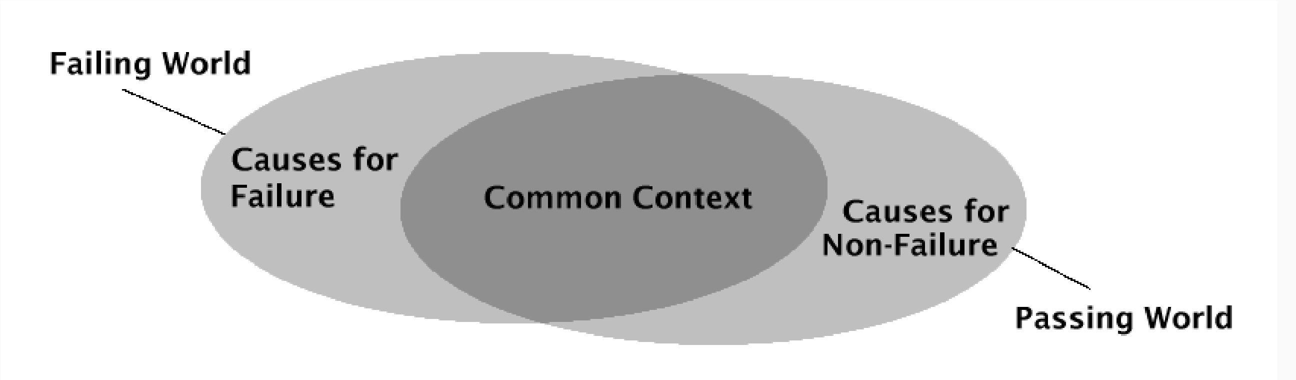
A world is said to be “closer” to the actual world than another if it resembles the actual world more than the other does.

Idea: “The” cause or the *actual* cause should be a difference between the actual world where the effect occurs and the *closest possible world* where it would not.

The actual cause should thus be a *minimal difference* between the two worlds.

The Closest Possible World (2)

The cause is a difference between the *passing world* and the *failing world:*



The cause can be some part of the failing world (not present in the passing world), or vice versa

The Closest Possible World (3)

The actual failure cause in ...

- a *program input* is a minimal difference between
  - the actual input (where the failure occurs) and
  - the *closest possible input* that passes
- a *program state* is a minimal difference between
  - the actual program state and
  - the *closest possible state* that passes
- a *program code* is a minimal difference between
  - the actual code and
  - the *closest possible code* that passes.

Ockham's Razor

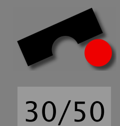
Another consequence of the closest possible world: *Whenever we have the choice between two causes, we can pick the one whose alternate world is closer.*

Consequently, "%d" is "the" defect, but the `printf` statement is not (altering just the format string is a smaller difference than removing the `printf` statement).

This principle is known as *Ockham's Razor:*

> *Whenever you have competing theories for how some effect comes to be, pick the simplest.*

Hanlon's Razor



Hanlon's Razor is a variant of Ockham's Razor:

Never attribute to malice
that which is adequately explained by stupidity.

Useful for slicing away mysterious explanations about failures

operating system defects	*compiler defects*
CPU malfunctions	*cosmic rays*
virus attacks	*secret service bugs*
alien invasions	*malicious professors*

Necessary Conditions

Problem: We may have *multiple* closest possible worlds.

Example:

The format `"%d"` *(rather than* `"%f"`*) is the cause for* `a = 0` *being printed.*

Why not

`a` *being declared as* `double` *(rather than* `int`*) is the cause for* `a = 0` *being printed.*

Necessary Conditions (2)

Solution—impose a *necessary condition:*

a *is* supposed *to be a floating-point value;*
its declaration is correct.

Thus, we must now seek the closest possible code in which

- a is declared as `double` and
- the failure no longer occurs.

In our example: the code where the format string is fixed.

Necessary Conditions (3)

We may also impose the condition that a is *supposed* to be declared as `double`, but printed as an integer value.

This implies a new failure cause:

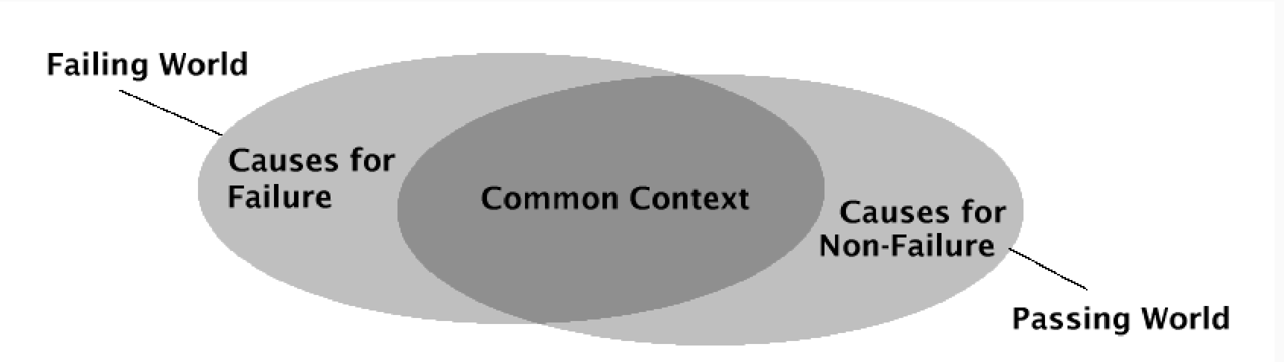
The `printf` *argument being* `a` *(rather than* `(int)a`*) is the cause for* `a = 0` *being printed.*

The original cause (the format string) would violate the imposed condition.

Necessary conditions thus allow us to *focus* the search.

Necessary Conditions (4)

In general, necessary conditions define a *common context:*



This common context must be a *subset* of all examined worlds.

What is necessary?

Actually, a lot:

- The program is executed
- The computer is up and running
- The computer has electrical power
- We have no cosmic ray source close
- Magic does not apply
- All natural laws hold

All this is part of the *common context.*

Short Summary

We know

- how to *organize* the debugging process ("Scientific Method")
- how to define a *cause*
- how to define an *actual* cause ("closest possible world")
- how to set up *necessary conditions.*

But how do we find an actual cause?

Enter *divide and conquer!*

Divide and Conquer

Divide and conquer (from latin “divide et impera”):
The basic principle of reducing a *large* problem into *smaller* subproblems.

General pattern:

- Start with some *initial* difference (= initial cause)
- *Decompose* this difference into smaller differences
- *Test* the resulting hypotheses to see whether the smaller differences are more specific failure causes.

The Initial Difference

Failing World

✘

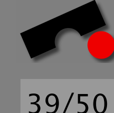
Initial Difference

= Initial Cause

✔

Passing World

Setting up an Alternate World



Failing World ✗

Alternate World

Passing World ✓

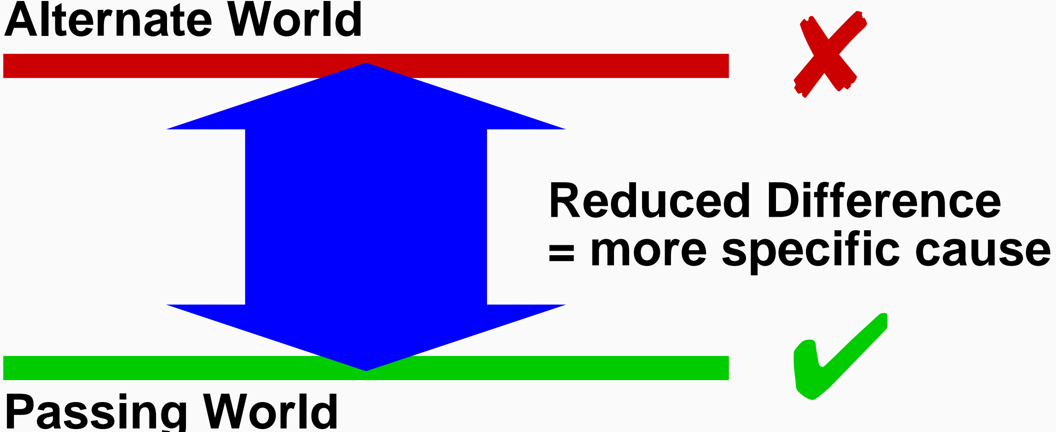
Testing the Alternate World

Failing World

Alternate World

Passing World

Narrowing the Difference

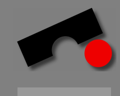


Alternate Test Outcome

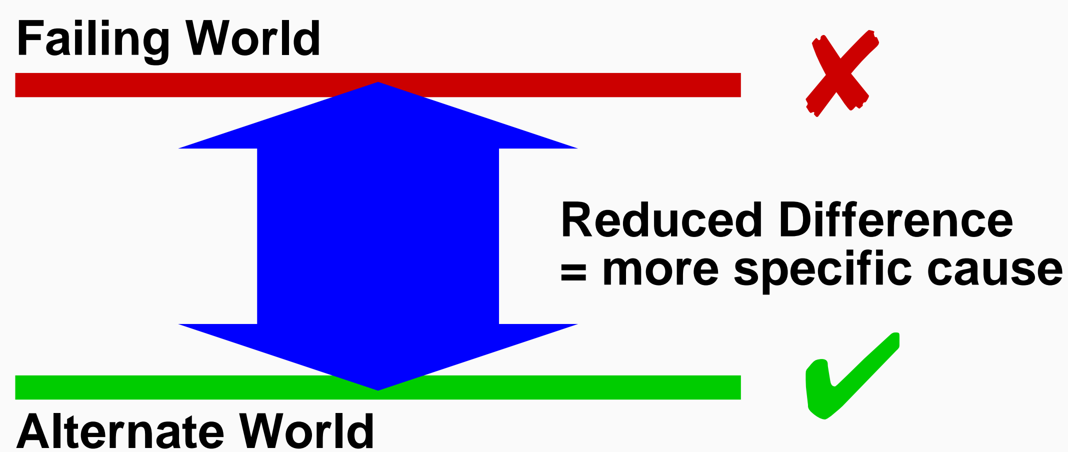
Failing World

Alternate World

Passing World



Narrowing the Difference



The Narrowing Process

Failing World

Initial Difference
= Initial Cause

Passing World

3. Repeat as needed

3. Repeat as needed

1. Set up hypothesis

Alternate World

2a. Hypothesis confirmed

or

2b. Hypothesis rejected

Reduced difference
= more specific cause

Reduced difference
= more specific cause

Simplifying HTML Input

Mozilla bug #24735, reported by *anantk@yahoo.com*:

```
Ok the following operations cause mozilla to crash
consistently on my machine

 -> Start mozilla
 -> Go to bugzilla.mozilla.org
 -> Select search for bug
 -> Print to file setting the bottom and right margins to
    .50 (I use the file /var/tmp/netscape.ps)
 -> Once it's done printing do the exact same thing again on
    the same file (/var/tmp/netscape.ps)
 -> This causes the browser to crash with a segfault
```

What's the cause for this failure?

Simplifying HTML Input (2)

Idea: Apply *Divide and Conquer* to simplify HTML pages

1		⟨896 lines⟩	✘
2		⟨448 lines⟩	✘
3		⟨224 lines⟩	✘
4		⟨112 lines⟩	✔
5		⟨112 lines⟩	✘
6		⟨56 lines⟩	✔
⋮			
57	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨40 characters⟩	✘
58	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
59	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
60	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨30 characters⟩	✔
61	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✘
62	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨10 characters⟩	✘
⋮			
75	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
76	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
77	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
⋮			
90	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✘

Simplified bug report: `Printing <SELECT> crashes.`

Divide and Conquer in Action

We shall see how to use Divide and Conquer

- For *simplifying* test cases (as shown)
- For isolating failure-inducing *input*
- For isolating failure-inducing *code changes*
- For isolating failure-inducing *thread schedules*
- For isolating failure-inducing *program states*
- For isolating *infection sites*

—and we'll try to *automate this* as much as possible.

Concepts

➠ The SCIENTIFIC METHOD, as applied to debugging, consists of five steps:

1. Observe a failure.
2. Invent a hypothesis as to the failure cause that is consistent with the observations and the necessary conditions.
3. Use the hypothesis to make predictions.
4. Test the hypothesis by experiments or further observations and modify the hypothesis in the light of your results.
5. Repeat steps 3 and 4 until you found the actual cause.

➠ It is useful to KEEP A NOTEBOOK to make the individual steps explicit.

Concepts (2)

➠ A cause is an event preceding another event without which the event in question (the *effect*) would not have occurred.

A cause is thus described as a difference between the world where the effect occurs and a *possible world* where the effect does not occur.

➠ Only experimentation can prove causality—that is, one must set up the possible world and show that the effect in question does not occur. Reasoning alone does not suffice to establish causality.

➠ An *actual cause* is a *minimal* difference between the two worlds—that is, a difference between the world where the effect occurs and the *closest possible world* where the effect does not occur.

Concepts (3)

- *Necessary conditions* set up a common context between the actual and possible worlds and thus focus the search for an actual cause.
- DIVIDE AND CONQUER is useful to invent and modify hypotheses about failure causes. Its main idea is to break some initial (trivial) cause into smaller parts and to narrow down the actual cause along the scientific method.