What's it all about?

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Overview

We're going to make a *quick tour* through the course material today:

- Understanding Failure Circumstances
- Examining the Run
- Isolating the Defect
- Delta Debugging
- Program Slicing
- Detecting Anomalies

This is just to provide an *overview*—if you don't get it at first, just sit back and relax :-)

A Simple Example

The `sample` program is supposed to sort its arguments:

```
$ sample -7 14 5 -4 1 2 3
Output: -7 -4 1 2 3 5 14
$ _
```

Unfortunately, `sample` has a defect:

```
$ sample -11 14 7 5 4 1 2 3
Output: -11 0 1 2 3 4 5 7
$ _
```

This will be our ongoing example today.

From Defects to Failures

A failure comes to be in three stages:

Defect → Infection → Failure

A defect is an *error in the program* (code).

An infection is an *error in the program state*.

A failure is an observable *error in the program behavior*.

The issue of debugging is to

- relate an observed failure to a defect and
- to remove the defect such that the failure no longer occurs.

From Defects to Failures (2)

Not every defect causes an infection, and not every infection causes a failure.

This is *the curse of testing:*

> *Testing can only show the presence of defects,*
> *but never their absence.* (Dijkstra)

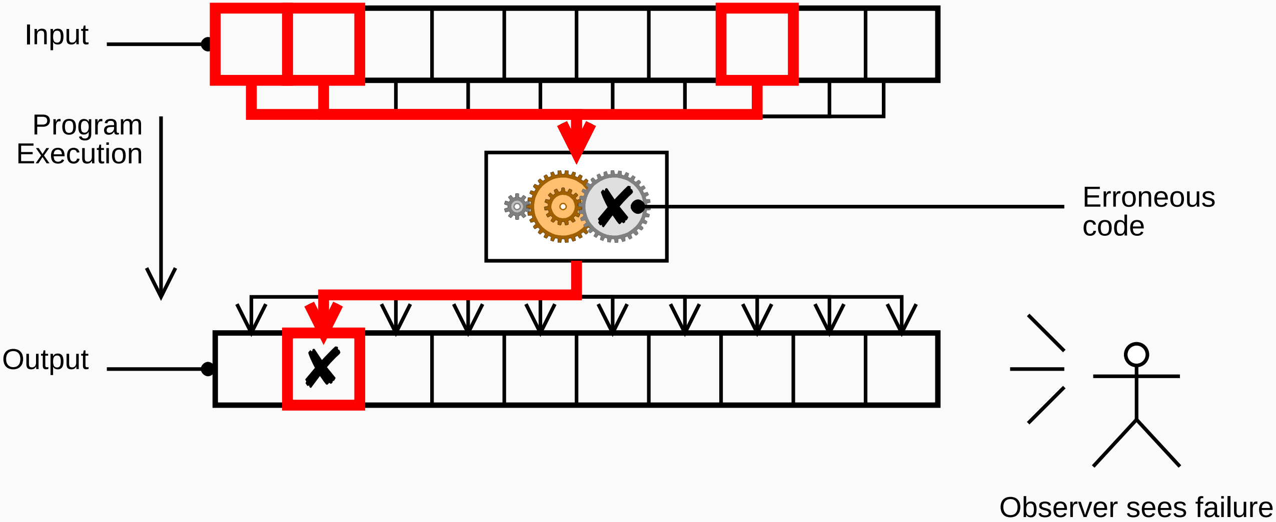
On the other hand, each failure can be traced back to some infection, and each infection can be traced back to a defect.

This is what we shall do with the `sample` program.

Understanding Failure Circumstances

Before you rush to your favorite debugger, first try to understand under *which circumstances* the failure occurs.

The idea is to identify the *relevant circumstances.*



Simplifying sample *input*

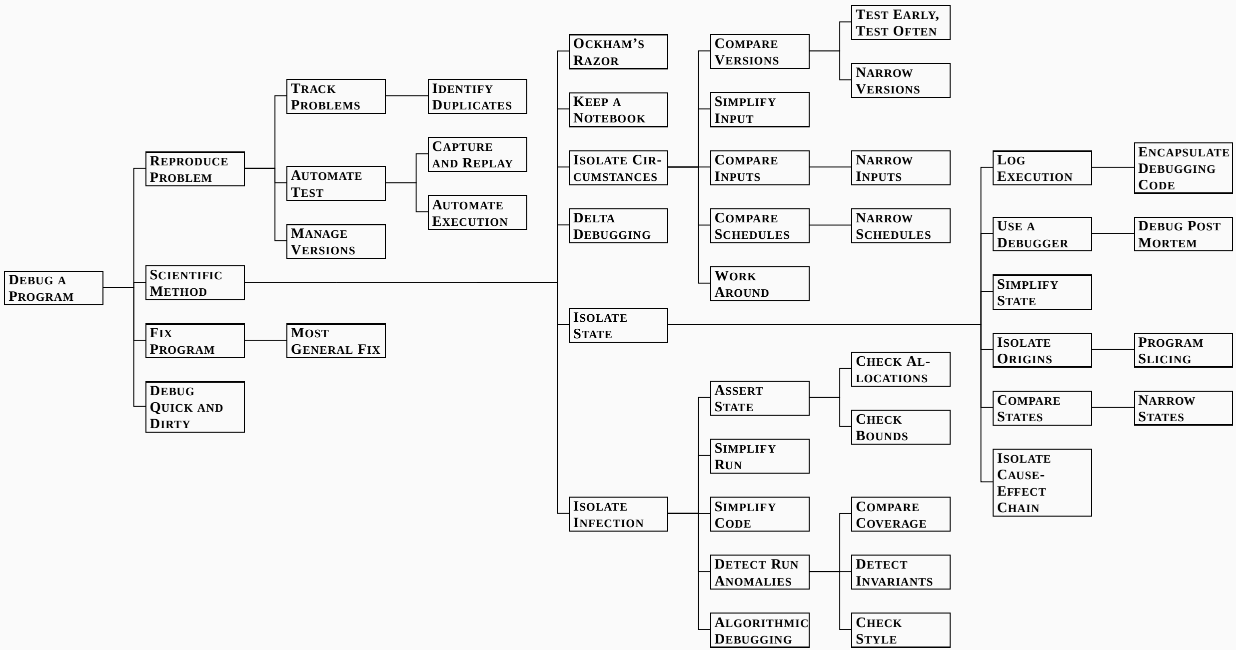
We try to simplify the input

```
$ sample -11 14 7 5 4 1 2 3
Output: -11 0 1 2 3 4 5 7
$ sample -11 14 7 5
Output: -11 0 5 7
$ sample -11 14
Output: -11 0
$ sample -11
Output: -11
$ _
```

This *input simplification* is a general *pattern* which isolates a failure cause (here: the input 14).

Debugging Patterns

Idea: A set of *patterns* where each pattern solves a specific debugging problem.



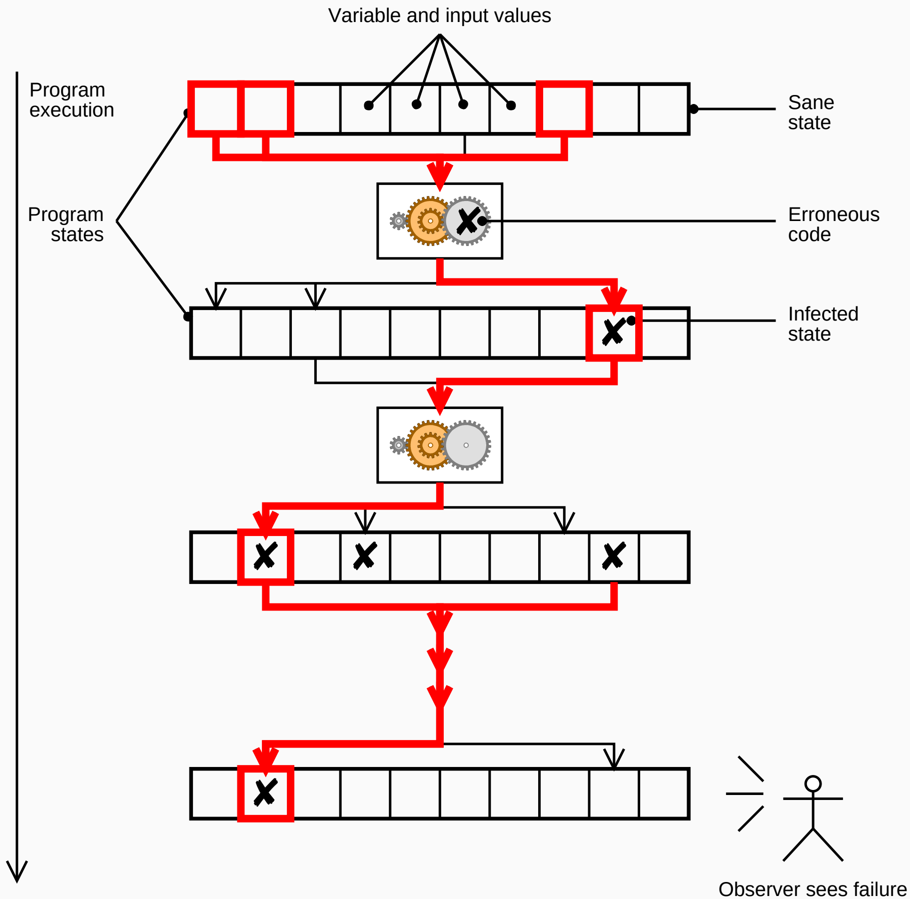
Pattern: The Scientific Method

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.

#	Run with args...	Outcome	Notes
1	8 -7 5 -4 1 2 3	✔	original pass
2	-11 14 7 5 4 1 2 3	✘—output contains 0	original failure
3	-11 14 7 5	✘—output contains 0	simplified
4	-11 14	✘—output contains 0	simplified
5	-11	✔	isolated difference

Conclusion: extra 14 is failure-inducing

Understanding Infection Origins



Debugging as a search problem

Search in space. Each single state is composed of thousands or even millions of variables.

Debugging means to *separate the infected variables from the sane variables*.

Search in time. A program execution consists of thousands, millions or even billions of states.

Debugging means to *isolate the infection*—the transition from a sane state towards an infected state.

Fortunately, debugging is not *that* difficult:
Good programming style limits the information flow between units (= functions, modules, objects. . .)

A divided state is much easier to conquer!

sample.c


```
int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);

    return 0;
}
```

sample.c (2)


```
static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}
```

Isolating the infection

The 0 in the output stems from a[1].

(See printf)

a[1] stems from shell_sort.

(See shell_sort)

At shell_sort invocation, the state is *infected*.

```
a[0] = -11
a[1] = 14
a[2] = 0
size = 3
```

The infection occurs at the invocation of shell_sort:

```
shell_sort(a, argc);
```

How the Failure came to Be

1. The array a[] is allocated and initialized with the correct number of elements (2).
2. shell_sort is invoked such that the size parameter is 3.
3. This causes shell_sort to access a[] beyond the allocated space—namely at a[2].
4. The uninitialized memory at a[2] happens to be zero.
5. During the sort, a[2] is eventually swapped with a[1].
6. Thus the zero value of a[1] is printed, causing the failure.

Fixing the Defect

We replace

```
shell_sort(a, argc);
```

by the correct invocation

```
shell_sort(a, argc - 1);
```

Now we must repeat the test:

```
$ sample -11 14
Output: -11 14
$ _
```

The `sample` program is fixed.

Our Debugging Process

SIMPLIFY INPUT. Reducing the number of `sample` arguments.

ISOLATE ORIGINS. Relating `a[1]` back to `shell_sort`.

LOG EXECUTION. Looking into the values at `shell_sort`.

ISOLATE INFECTION. Finding out that the state was infected at the invocation of `shell_sort`.

MOST GENERAL FIX. We're done!

The General Debugging Process

1. Reproduce the failure, using REPRODUCE PROBLEM.
2. Use the SCIENTIFIC METHOD to
   - isolate failure-inducing circumstances
   - trace back infected values to its origins
   - isolate the moment of infection
3. Fix the program, using MOST GENERAL FIX.

This is a pattern, namely DEBUG A PROGRAM

Isolating Relevant Circumstances

We have

- A means to *simplify* the input (i.e. the arguments)
- A *testing function* that executes the program and tells us whether the failure occurs

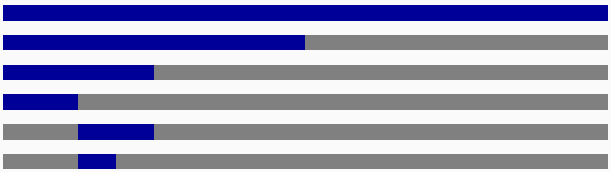
Idea: *Automated simplification*

While we can simplify the input such that the failure still occurs, do it.

This is a pattern: SIMPLIFY INPUT!

Isolating Relevant Circumstances (2)

The pattern DELTA DEBUGGING automates simplifying the input



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⟩	✘

Isolating Relevant Circumstances (3)

Other relevant circumstances that can be isolated include:

Input differences:

The character < in the input causes Mozilla to fail.

Schedule differences:

The failure occurs if the thread switch occurs *here*.

Version differences:

The change in *that* line caused the failure.

Program Slicing

While debugging `sample`, we traced back `a[1]` to its origins.

This can also be partially automated, using PROGRAM SLICING.

Basic idea of slicing: isolate *dependencies* of variable values by *analyzing the program code.*

Any value of `a[i]` at the program end, for instance, is dependent on

- `a[i] = atoi(argv[i + 1]);`
- `a[j] = v;`
- `a[j] = a[j - h];`

`a[i]` can get its value *only* from these locations!

Dynamic Slicing

Dynamic slicing tracks the dependencies for a *concrete program run.*

Using a dynamic slicer, we can determine that `a[1]` was last assigned in

```
a[j] = v;    // j = 1, v = 0
```

The previous assignment to `v` was at

```
v = a[i];    // i = 2
```

We have traced back that `a[1]`'s bad value came from `a[2]`.

But `i` is 2 because of `size`, so `a[1]` is also dependent on `size`.

Cause-Effect Chains

Basic idea: COMPARE STATES of a *passing run* and a *failing run*; differences indicate failure causes.

Line	Run	argc	argv[0]	argv[1]	argv[2]	a[0]	a[1]	a[2]	size
30	✔ ✘	2 3	"sample"	"-11"	NULL "14"	n/a	n/a	n/a	n/a
8	✔ ✘	2 3	"sample"	"-11"	NULL "14"	-11	256 14	512 0	2 3
37	✔ ✘	2 3	"sample"	"-11"	NULL "14"	-11	256 0	512 14	n/a

Which one of these differences is relevant for the failure?

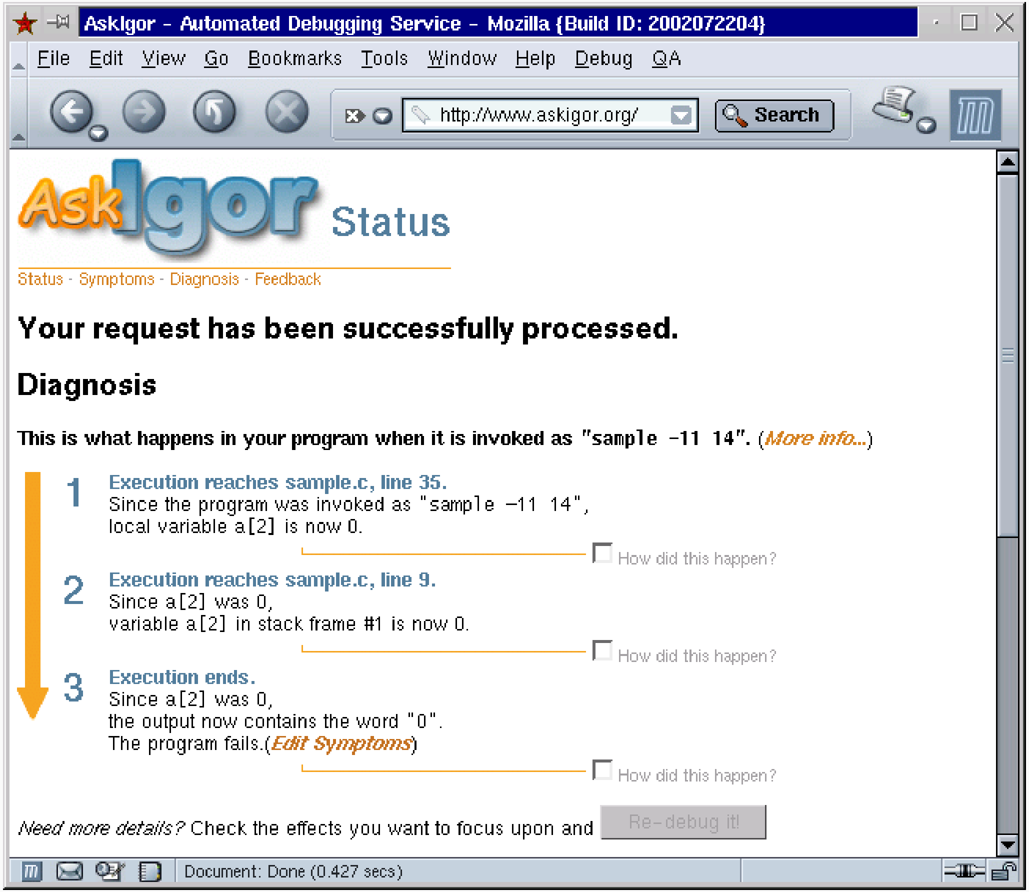
Cause-Effect Chains (2)

Basic idea: We *one difference at a time* and check the outcome.

1. If we apply no difference at all, a[1] is eventually 256; the output is -11. The program passes.
2. If we run the program until Line 8, set argc from 2 to 3 and resume execution, a[1] is still 256; the output is -11 256.
3. If we repeat the experiment and also set argv[2] from NULL to "14", a[1] and output are unchanged.
4. If we also set a[1] from 256 to 14, variable a[1] remains 14; the output is -11 14.
5. If we also set a[2] from 512 to 0, variable a[1] remains 14, the output is still -11 14.
6. If we also set size from 2 to 3, variable a[1] becomes 0; the output is -11 0.

   Only with this last step did the failure occur—a[1] became zero.

Cause-Effect Chains (3)



Isolating Infections

Issue: Find the moment in time when the state changes from *sane* to *infected.*

Automation requires means to *check the sanity of the state.*

Isolating Infections (2)

Assertions.

```
assert(is_sorted(a, size));
assert(is_permutation_of(a, original_a, size));
```

Checking the heap.

```
$ MALLOC_CHECK_=2 sample -11 14
Output: -11 14
$ _
```

Checking array boundaries.

```
$ gcc -g -o sample-with-efence sample.c -lefence
$ sample-with-efence -11 14
Electric Fence 2.1
Segmentation fault (core dumped)
$ _
```

Searching for Anomalies

Another place to search infections at are *anomalies*.

An *anomaly* in a program run is a property which deviates from the (non-failing or *normal*) standard.

We can look for anomalies

- *in the code*—CHECK STYLE
- *in the execution*—COMPARE COVERAGE
- *in the data*—DETECT INVARIANTS

Anomaly in the Execution

We compare the *coverage* of the passing and the failing run:

Coverage	Code
✔ ✘	`static void shell_sort(int a[], int size)`
✔ ✘	`{`
✔ ✘	`int i, j;`
✔ ✘	`int h = 1;`
✔ ✘	`do {`
✔ ✘	`h = h * 3 + 1;`
✔ ✘	`} while (h <= size);`
✔ ✘	`do {`
✔ ✘	`h /= 3;`
✔ ✘	`for (i = h; i < size; i++)`
✔ ✘	`{`
✔ ✘	`int v = a[i];`
✔ ✘	`for (j = i; j >= h && a[j - h] > v; j -= h)`
✔ ✘	**`a[j] = a[j - h];`**
✔ ✘	`if (i != j)`
✔ ✘	**`a[j] = v;`**
✔ ✘	`}`
✔ ✘	`} while (h != 1);`
✔ ✘	`}`

Anomaly in the Data

Another area to search anomalies for is *data*.

DETECT INVARIANTS—see how the program data differs from *inferred invariants*:

At start of shell_sort

```
size == size(a[])
a[] one of  [-11, 14], [7, -1, 25, 9], [14, 11]
size one of  2, 4
```

At exit of shell_sort

```
orig(size) == orig(size(a[]))
a[] one of  [-11, 14], [-1, 7, 9, 25], [11, 14]
```

Invariants in the Data (2)

At start of `main`

```
argc == size(argv[])-1
argc one of  3, 5
size(argv[]) one of  4, 6
argv[argc..] == [null]
argv[argc..] elements == null
argv[argc+1..] == []
```

At exit of `main`

```
argv[] == orig(argv[])
return == 0
argv[orig(argc)..] == [null]
argv[orig(argc)..] elements == null
argv[orig(argc)+1..] == []
```

Debugging Details

More material to be covered:

- *Reproducing* the Problem (faithfully and automatically)
- *Fixing* the Program (in the best possible way)
- *Preventing* Failures

. . . all in the remaining course.

Concepts

➨ In general, a failure comes to be in three stages:

1. The programmer creates a *defect* in the program code (also known as *bug* or *fault*).
2. The defect causes an *infection* in the program state.
3. The infection causes a *failure*—an externally observable error.

➨ Not every defect results in an infection, and not every infection results in a failure.

Yet, every failure can be traced back to some infection, which again can be traced back to a defect.

Concepts (2)

➠ *Debugging a program* consists of three activities:

1. *Reproducing* the failure,
2. *Relating* the failure to a defect in the program, and
3. *Fixing* the defect.

Of these three activities, the second is by far the most time-consuming.

➠ *Debugging patterns* encapsulate solutions to specific debugging problems.

➠ A variety of systematic and automated approaches is available that help in debugging.

See remaining course!