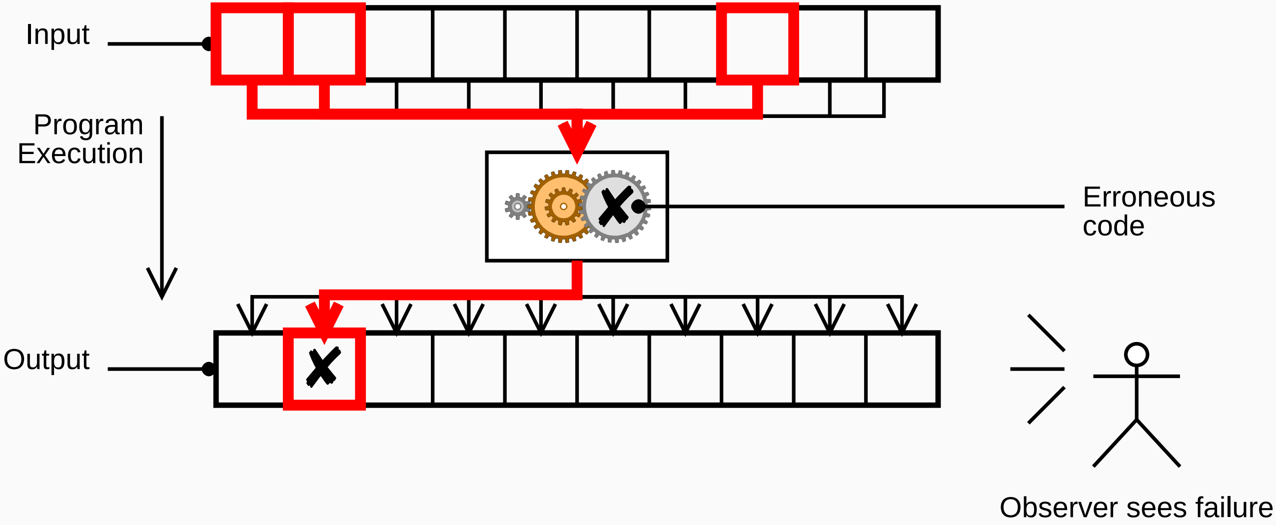
Understanding the Program Run

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

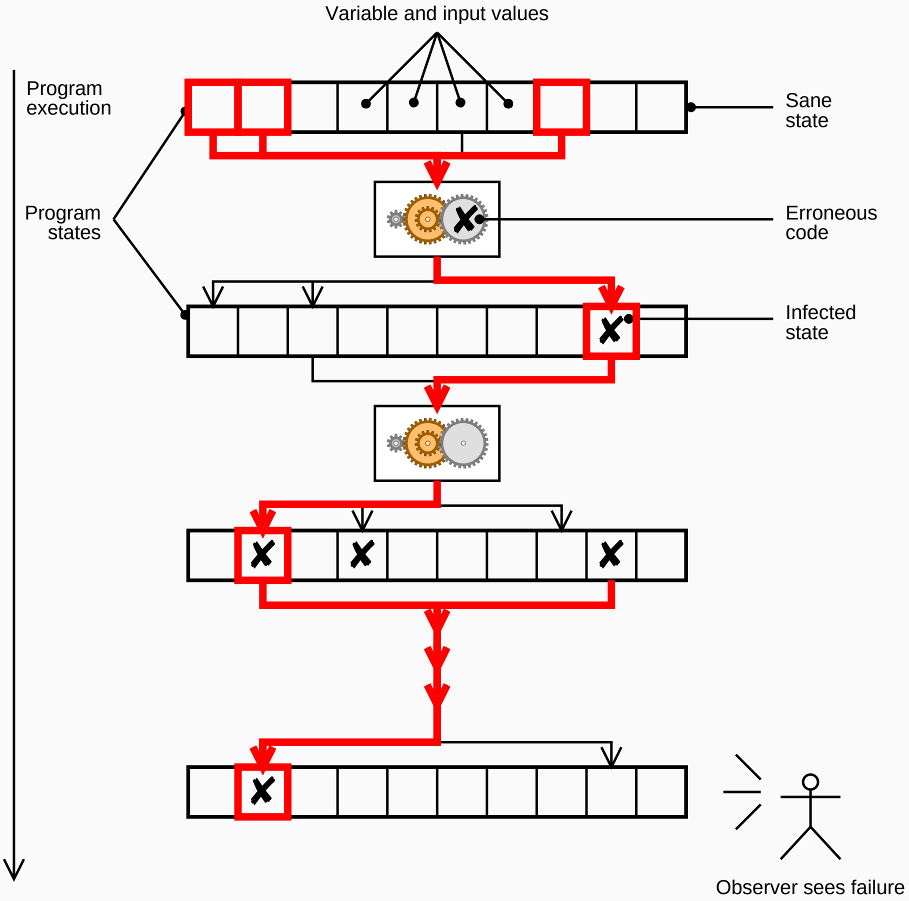
Isolating Failure Causes

So far, we have seen how to isolate causes in the *environment* of the program:



We treated the program as a *black box*, though!

What we'd like to see



Today's Topics

Examining Program Execution. How do we know which parts of the program were executed?

Examining Program State. How do we access and examine particular program states?

Isolating a Specific State. Spatial focusing—across the program state.

Isolating the Infection. Temporal focusing—across the program execution.

Examining Program Execution

Basic Question: *What was executed?*

Simplest pattern of all: LOG EXECUTION

Basic idea:

- Insert log statements at specific places in the progras
- As soon as log statement is reached, output is generated
- Examine sum of logs to see
  - what was executed
  - and what was *not* executed.

The No-Op `test` *Program*

Simple program `test.c` is supposed to print the n first primes, with n being the argument:

```
int main(int argc, char *argv[])
{
    int number_of_primes;
    number_of_primes = atoi(argv[1]);
    print_primes(number_of_primes);
}
```

Observation—The program does not print anything:

```
$ test 27
$ _
```

The No-Op test *Program (2)*

Hypothesis: *The* `main` *function was not executed.*

```
int main(int argc, char *argv[])
{
    int number_of_primes;
    printf("main() was called!\n");
    number_of_primes = atoi(argv[1]);
    print_primes(number_of_primes);
}
```

Outcome—`main` was *not* executed (confirmation)

```
$ test 27
$ _
```

`test` invokes the system command, not our program!

Logging Data

While we're logging the location, we might as well log the current state:

```
int main(int argc, char *argv[])
{
    int number_of_primes;
    number_of_primes = atoi(argv[1]);
    printf("main(): number_of_primes = %d\n",
        number_of_primes);
    print_primes(number_of_primes);
    printf("main(): returning\n")
}
```

Logging is the *easiest* and *most common* debugging technique!

Logging in Practice

Use standard formats. This

- applies to *events* (“prefix each line with time”)
- applies to *data* (“output all dates in Y-M-D format”)
- is best achieved by using *dedicated logging functions*.

Make logging optional. For efficiency, logging is typically turned off in production code.

Allow for variable granularity. Depending on the problem you are working on, it may be helpful to focus on specific levels of detail.

Simple Macros for Logging

We use

```
LOG(("number_of_primes = %d", number_of_primes))
```

to get

```
number_of_primes = 3
```

Definition:

```
#define LOG(args) printf args
```

In practice: dedicated *logging function* instead of `printf`

Extra Logging Information

We use

```
LOG(("number_of_primes = %d", number_of_primes))
```

to get

```
main.c:3: number_of_primes = 3
```

Definition:

```
#define LOG(args) \
    printf("%s:%d: ", __FILE__, __LINE__), \
    printf args, \
    printf("\n")
```

This scheme can easily be extended to log date/time, etc.

Optional Logging

We turn logging off at *compile time*
using the NDEBUG (“No Debugging”) macro

```
$ gcc -DNDEBUG -o mytest test.c
```

Definition:

```
#ifndef NDEBUG
#define LOG(args) ⟨as before⟩
#else
#define LOG(args)
#endif
```

If NDEBUG is set, `LOG(args)` compiles to a no-op

Logging Granularity

We turn logging on and off at *runtime*
using a LOG_FILES environment variable:

```
$ LOG_FILES="main.c debug-*.c" mytest
```

Definition:

```
#define LOG(args) \
    do_we_log_this(__FILE__) && \
        (printf("%s:%d: ", __FILE__, __LINE__), \
         printf args, \
         printf("\n"))
```

Complex macro definitions can easily be turned into an appropriate function.

Lots of Logs

Problem: Lots and lots of logging code can easily clutter the “real” program code.

Delete logging code when debugging is finished.

Problem: When do we know that debugging is finished?

Use a debugger instead.

Problem: Have to recreate everything every time.

Encapsulate logging within an *aspect*.

An aspect is a separate syntactical entity that can be interwoven with the program (i.e. it is *optional*).

Logging with Aspects

Aspects give very elegant ways to handle logging:

```
public aspect Tracer {
    pointcut allMethods():
        call(public * Article.*(..));
    before(): allMethods() {
        System.out.println ("Entering " +
                            thisJoinPoint);
   }
   after(): allMethods() {
        System.out.println ("Leaving "  +
                            thisJoinPoint);
   }
}
```

Even better Logging

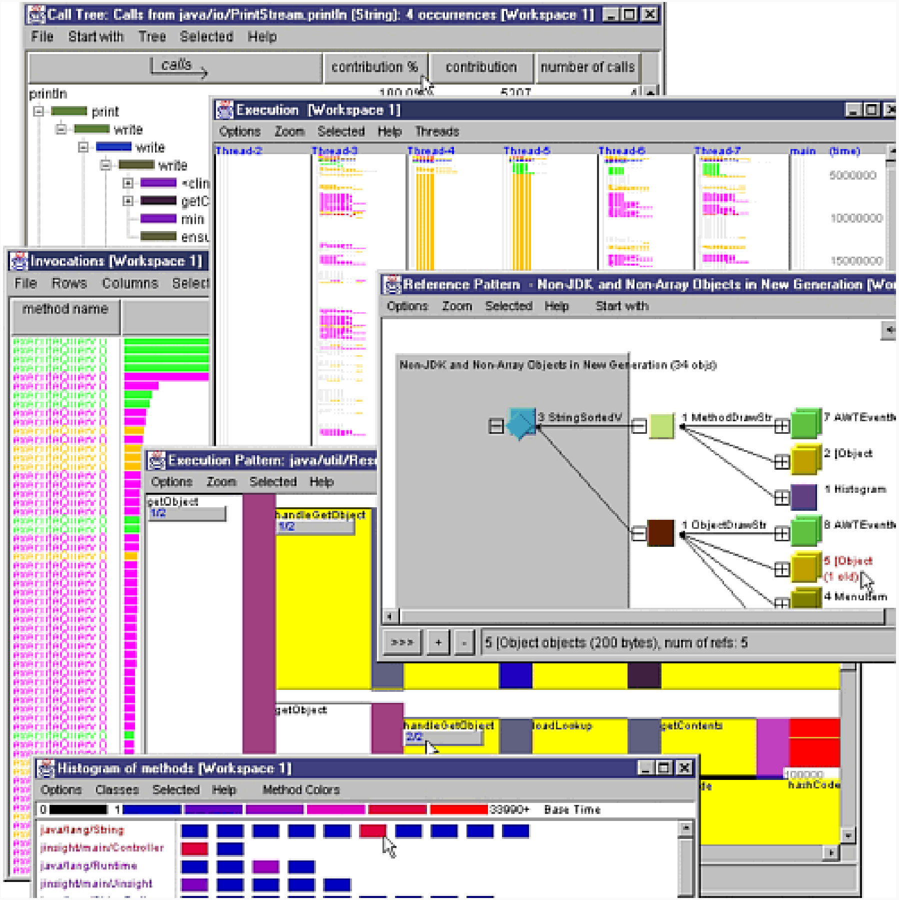
Current trends in logging:

Insert logging code *automatically* (just as with a tracing aspect)

***Visualize* log results** (rather than simply printing them)

Search for *patterns* (such as “this sequence of function calls occurs n times”)—and *deviations*

Tracing with Jinsight



5 [Object objects (200 bytes), num of refs: 5

Persistent vs. Transient Logging

Logging has an advantage and a disadvantage:

- ✔ Logging is compiled within the program
- ✘ Logging is compiled within the program

If I want a more *transient* approach, I use a *debugger* instead.

Basic Debugger Facilities

A *debugger* allows to

- *Start* your program, specifying anything that might affect its behavior.
- Make your program *stop* on specified conditions.
- Examine *what has happened* when your program has stopped.
- *Change* things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Source: gdb(1) manual page

Examining Program Execution

How do we know which parts of the program were executed?

A *breakpoint* makes the program stop as soon as it reaches a specific location.

```
$ gdb sample
(gdb) break main
Breakpoint 1 in main
(gdb) _
```

The program will stop as soon as `main` is reached (formally: the program counter (PC) is `main`)

Breakpoints in Detail

Formally, a breakpoint defines a *predicate* on the program state—the program stops as soon as the predicate holds.

A predicate like “the current PC is `main`” is easy to check:

- If the program is stored in RAM, we can replace the instruction at `main` with a *break instruction* (when the breakpoint is reached, the original instruction is restored)
- Many processors have *debugging registers* which interrupt execution as soon as the PC is equal to a registered value

Many debuggers support only simple breakpoints “the PC is x”.

Breakpoints and Watchpoints

Some debuggers provide additional predicates—especially predicates on *data*.

A GDB *watchpoint* will interrupt the program as soon as a specific variable changes its value:

```
(gdb) watch a
Hardware watchpoint 1: a
(gdb) continue
Old value = (int *) 0xbffff518
New value = (int *) 0x8049850
(gdb) _
```

Watchpoints in Detail

Watchpoints can be arbitrarily complex:

```
(gdb) watch f(x) != 42
```

will stop as soon as `f(x)` changes its value

Watchpoints can simulate breakpoints:

```
(gdb) watch $pc != main
```

will stop as soon as the program counter reaches `main`

No support for “is called by”, “within” or other useful predicates from aspect-oriented programming :–(

Watchpoints in Detail (2)

Watchpoints are typically *expensive:*

- Some processors have *debugging data registers* which interrupt execution as soon as the value at the registered address changes its value.

  This is efficient, but works only for simple values (and the program counter).
- If no such registers exist, or if the watched expression must be computed, the debugger must inquire the data *after each single instruction*, reducing speed to 1/1000.

Conditional Breakpoints

Conditional breakpoints allow users to check predicates only at specific locations—i.e. when the PC reaches a certain value.

```
(gdb) break print_primes if n_primes == 2
Breakpoint 1 at print_primes
(gdb) _
```

The program will stop if
the PC is `print_primes` and `n_primes` is 2.

Due to the PC checking, this can again be implemented efficiently.

Conditional Breakpoints (2)

Conditional breakpoints can be used to realize assertions on-the-fly:

Rather than writing

```
int foo() {
    assert (a > 0);
    ...
```

one could set a breakpoint

```
(gdb) break foo if a <= 0
(gdb) _
```

These assertions on-the-fly are *transient*
(not sure whether this is a good thing...)

Breakpoints and Predicates

Overview of breakpoint commands:

Type	GDB Command	Predicate
Breakpoint	`break` *location*	$PC = location$
Watchpoint	`watch` *expr*	*expr* changes
Cond. bp	`break` *location* `if` *expr*	$PC = location \land expr$

The debugger also *automatically* stops the program

- on user interrupts (Ctrl+C)
- if it receives a fatal signal
- if an uncaught exception is thrown

Examining the Stack

Among the first tasks to do when a program stops is to examine the current *backtrace*—the stack of calling functions.

```
(gdb) run
Starting program: sample

Breakpoint 1, shell_sort (a=0x8049850, size=1)
    at sample.c:9
9           int h = 1;
(gdb) where
#0  shell_sort (a=0x8049850, size=1) at sample.c:9
#1  main (argc=1, argv=0xbffff564) at sample.c:35
#2  __libc_start_main () from /lib/libc.so.6
(gdb) _
```

Examining Program Data

Once a program has stopped, we can examine its *data*—in the state where the program stopped.

All debuggers can print single variables:

```
(gdb) print a[0]
$1 = 0
(gdb) _
```

Most debuggers also support *expressions*:

```
(gdb) print a[size - 1]
$2 = 0
(gdb) _
```

Examining Program Data (2)

Some debuggers also support *function calls*:

```
(gdb) print main(argc, argv)
$3 = 0
(gdb) _
```

Method invocations are also possible:

```
(gdb) print c1.operator==(c2)
$4 = false
(gdb) _
```

If execution stops during the evaluation of the expression, interesting things can happen :-)

Examining Program Data (2)

To access the variables of a calling function, one can navigate through the backtrace:

```
(gdb) frame
#0  shell_sort (a=0x8049850, size=4) at sample.c:9
(gdb) info locals
i = 1073834752
j = 1074077312
h = 1961
(gdb) up
#1  0x8048647 in main (argc=4, argv=0xbffff544)
    at sample.c:35
(gdb) info locals
a = (int *) 0x8049850
i = 3
(gdb) _
```

Resuming Execution

After one is done examining the program state, one can *resume execution* (until the next stopping condition is reached):

```
(gdb) continue
Program exited normally.
(gdb) _
```

Oops—obviously, we should have set another breakpoint!

Stepping through the Program

A common task is to execute the program *until the next statement is reached:*

```
(gdb) run 7 8 9
Breakpoint 1, shell_sort (a=0x8049850, size=4)
  at sample.c:9
9     int h = 1;
(gdb) step
11      h = h * 3 + 1;
(gdb) step
12    } while (h <= size);
(gdb) _
```

Stepping through the Program (2)

Several commands are available to step:

step PC reaches next executed statement, maybe in different function

next PC reaches next executed statement in same function or current function returns

until PC reaches line greater than the current or current function returns

finish current function returns

continue resume execution unconditionally

All these commands are realized using *temporary breakpoints* at the appropriate locations.

Logging Data

Using a debugger, one can also *log values* automatically.

`display` *variable* prints *variable* with each GDB prompt.

```
(gdb) display a
a = 1
(gdb) next
a = 2
(gdb) next
a = 3
(gdb) continue
Breakpoint 1, shell_sort (a=0x8049850, size=4)
    at sample.c:9
a = 4
(gdb) _
```

Logging Data (2)

Alternate idea—*associate breakpoint with commands*

```
(gdb) break 16
Breakpoint 1 at file sample.c, line 16.
(gdb) commands
Type commands for when breakpoint 1 is hit,
one per line.  End with a line saying just "end".
>print i
>cont
>end
(gdb) _
```

Logging Data (3)

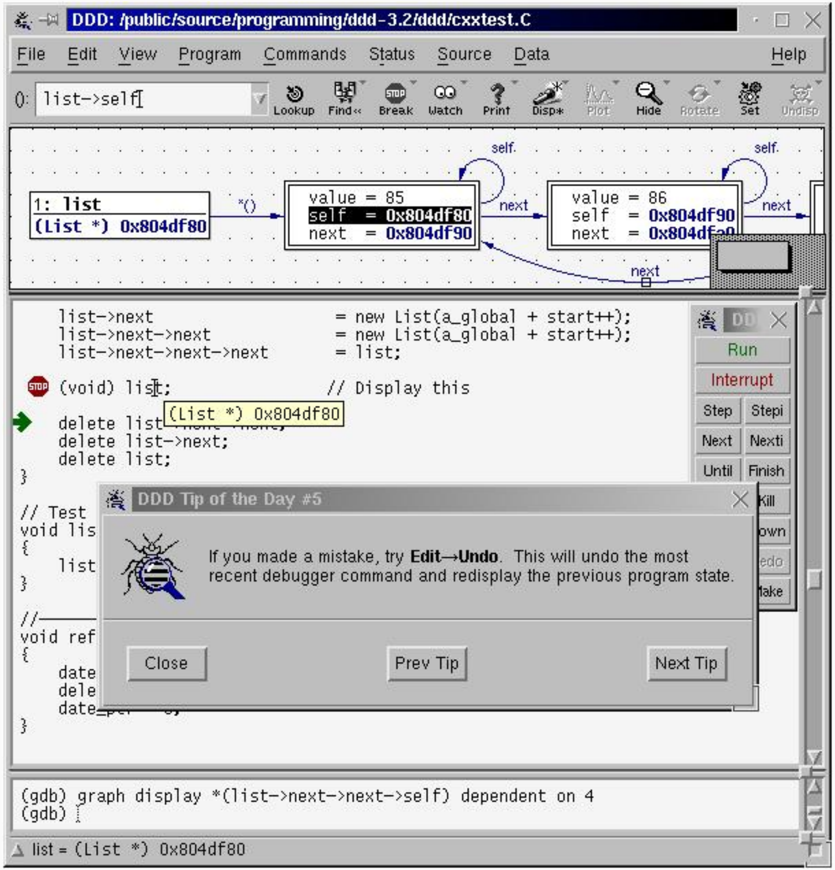
```
(gdb) run
Starting program: sample 7 8 9

Breakpoint 1 at sample.c:17
17                  int v = a[i];
$1 = 1

Breakpoint 1 at sample.c:17
17                  int v = a[i];
$2 = 2

Breakpoint 1 at sample.c:17
17                  int v = a[i];
$3 = 3
...
```

Logging Data (4)

Nicer alternative, using `silent` and `printf`:

```
(gdb) commands 1
Type commands for when breakpoint 1 is hit,
one per line.  End with a line saying just "end".
>silent
>printf "i = %d\n", i
>cont
>end
(gdb) run
Starting program: sample 7 8 9
i = 1
i = 2
i = 3
...
```

DDD—A Graphical User Interface



```
    list->next                    = new List(a_global + start++);
    list->next->next              = new List(a_global + start++);
    list->next->next->next        = list;

    (void) list;                  // Display this

    delete list
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//—
void ref
{
    date
    dele
    date
}
```

(List *) 0x804df80

If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

list = (List *) 0x804df80

Logging vs. Debugger

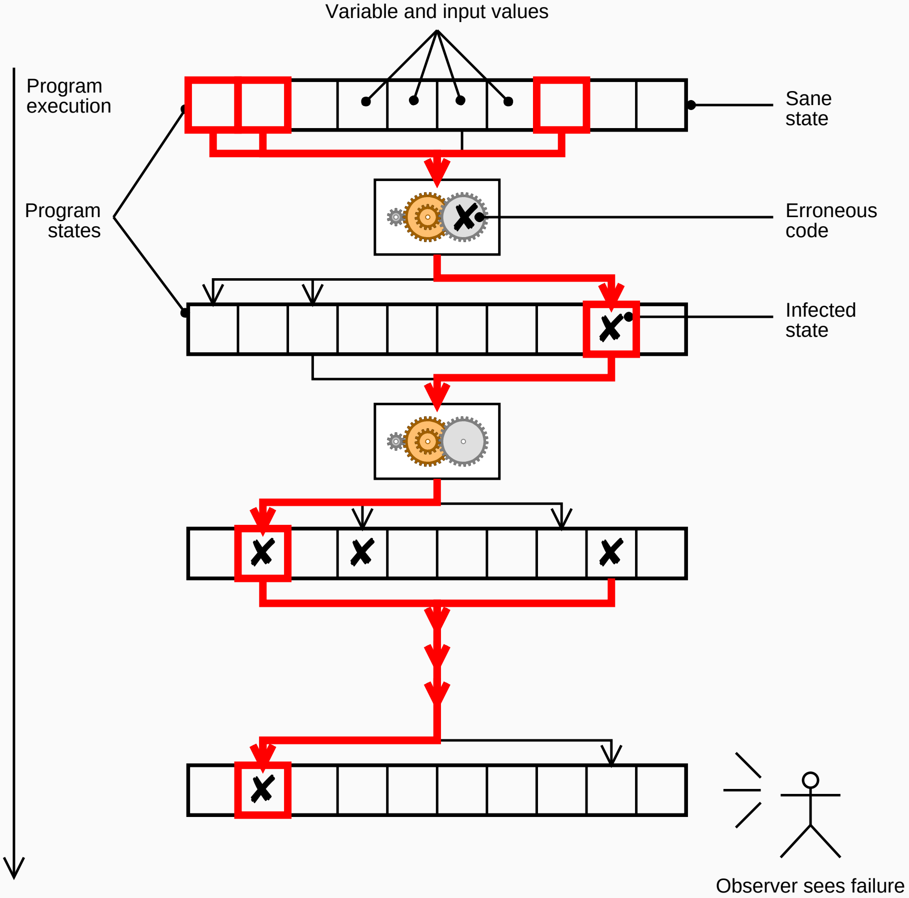
Examining Program Execution.

- Logging: Close to the code, persistent
- Debugger: Tedious, interactive, but versatile

Examining Program State.

- Logging: Close to the code, persistent
- Debugger: Tedious, interactive, very versatile

Spatial and Temporal Focusing



Spatial focusing

Basic idea: Separate *sane state* (= as intended) from *infected state* (= not as intended)

- Use logging (or a debugger) to access state
- Use *assertions* (or likewise debugger techniques) to separate sane from infected state

Temporal focusing

Basic idea: identify the *moment in time* where the state becomes infected

- Use logging (or a debugger) to access execution
- Use binary search to find out the moment in time where the state first became infected
- Trace back possible *origins* of the infection

To be addressed in remainder of the course!

Concepts

- *Logging* is a simple technique to understand
  - what was executed
  - what states the program was in
- Programmers use or define dedicated *logging* facilities
- Aspects allow encapsulating logging in own syntactical entities

Concepts (2)

- *Debuggers* allow a *versatile* and *transient* access to execution and data
- The program can be stopped as soon as a specific predicate holds (typically $PC = location$)
- In a stopped program, we can examine arbitrary data
- Assertions and logging can be added on the fly

Concepts (3)

- *Spatial focusing* means to separate the state into *sane* (= as intended) and *infected*
- *Temporal focusing* means to isolate the moment in time where the infection occurs
- *All this must be (and can be) automated!*