# Locating defects is uncertain

Andreas Zeller
Department of Computer Science
Saarland University, Saarbrücken, Germany
zeller@acm.org

## ABSTRACT

While numerous techniques for detecting the *presence* of defects exist, it is hard to assign the defect to a particular location in the code. In this position paper, I argue that this is necessarily so, and that locating a defect is inseparable from designing a fix—in other words, writing a correct program. This leads to an inherent imprecision, which can be dealt with by ranking locations according to their defect probability.

## 1. ERRORS AND CAUSES

To explain how failures come to be, one needs two central terms: *errors* and *causes.* An *error* is a deviation from what is correct, right, or true. If we see an error in the program outcome (the *failure*), we can trace back this failure to earlier errors in the program state (*faults* or *infections*), until we finally reach the defect—an error in the program code. This defect causes the initial infection, which propagates until the infection becomes visible as a failure.

In this infection chain, a *cause* is an event without which a subsequent event (the *effect*) would not have occurred. Thus, if the program code had been correct, it would not have caused the infection, which again would not have led to the failure. This causality is normally proven by re-testing the program after the defect has been fixed: If the failure no longer occurs, we have proven that the original defect indeed has caused the failure.

While causality is easy to explain (and easy to verify), the term *error* becomes less certain the further one goes back the chain of events. The key issue is: to decide that something is erroneous, one needs a specification of what is correct, right or true. Telling whether a program *outcome* is a failure is the base of testing—and quite straight-forward. Telling whether a program *state* is infected already requires appropriate conditions or representation invariants. Telling whether some program *code* is incorrect, finally, becomes *more and more difficult as granularity increases.*

## 2. DEFECTS AND GRANULARITY

Why does the location of a defect become less certain with increasing granularity? If a computation $P$ fails, we know it must have some defect. Let us assume that $P$ can be separated into two sub-computations $P = P_1 \circ P_2$, each at a different location. Then, we can check the result of $P_1$, and determine whether it is correct (which means that $P_2$ has a defect) or not (then $P_1$ has a defect).

Now assume we can decompose $P$ into $n$ executed procedures, or $P = P_1 \circ \cdots \circ P_n$. By checking the outcome of each $P_i$, we can assign the defect location to a single precise $P_j$. Obviously, this means specifying the postconditions of every single $P_i$, including general obligations such as representation invariants.

Let us now assume we can decompose $P$ into $m$ executed lines, or $P = P_1 \circ \cdots \circ P_m$. To locate the defect, we now need a specification of the correct state at each executed line—for instance, the *correct program.* Thus, to precisely locate the defect, we need a specification that is precise enough to tell us where to correct it.

In practice, such a specification is constructed *on demand:* When programmers search for a defect, they reason about whether this location is the correct way to write the program—and if it does not match, they fix it. Therefore, programmers do not "locate" defects; they *design fixes* along with the implicit specification of what *should* be going on at this location—and the location that is changed is defined as the defect in hindsight.

## 3. DEALING WITH IMPRECISION

As long as a full specification is missing (which we must reasonably assume), it is impossible to locate defects precisely, just as it is impossible to foresee how a problem will be fixed—and whether it will be fixed at all. Therefore, any defect location techniques will always have to live with imprecision. This is not a big deal; we can have our tools make *educated guesses* about where the defect might be located. And we will evaluate our tools by their power to suggest fixes that are as close to some "official" defect as possible.

However, imprecision also must be considered when *evaluating* techniques. For instance, if a technique detects that a function call does not match the function's requirements, the technique cannot decide whether it is better to fix the caller or the callee. Let us now assume that we conduct an evaluation where we have injected a defect in the callee. If the technique now flags the caller as defective, the result is evaluated as being at the wrong location, or even as a false. Nonetheless, the mismatch is helpful for the programmer.

There are entire classes of problems which cannot be located at all. For instance, assume I inject a defect which eliminates the initialization of a variable. Although there are techniques which will detect this situation, they will be unable to tell where the initialization should have taken place. Again, the evaluation will show a mismatch between predicted and expected defect location; nonetheless, the diagnosis will be helpful for the programmer.

How are we going to take this imprecision into account? I suggest to have our tools not only suggest locations, but to actually *rank* the locations by their probability to be related to the defect. The model would be an ideal programmer, starting with the most probable location, and going down the list until the "official" defect is found; obviously, the sooner the "official" defect is found, the better the tool. In a mismatch of caller and callee, both locations would end on top of the list; a missing initialization would result with the function or module containing the declaration being placed at the top. Such *code rankings* would allow us to compare individual defect-locating tools, and eventually establish standards for evaluating them.