How to Gauge Repair Risk?

Francisco Zapata and Vladik Kreinovich

[1] Department of Industrial, Manufacturing, and Systems Engineering
University of Texas at El Paso, El Paso, TX 79968, USA
fazg74@gmail.com
[2] Department of Computer Science
University of Texas at El Paso, El Paso, TX 79968, USA
vladik@utep.edu

Abstract. At present, there exist several automatic tools that, given a software, find locations of possible defects. A general tool does not take into account a specificity of a given program. As a result, while many defects discovered by this tool can be truly harmful, many uncovered alleged defects are, for this particular software, reasonably (or even fully) harmless. A natural reaction is to repair all the alleged defects, but the problem is that every time we correct a program, we risk introducing new faults. From this viewpoint, it is desirable to be able to gauge the repair risk. This will help use decide which part of the repaired code is most likely to fail and thus, needs the most testing, and even whether repairing a probably harmless defect is worth an effort at all – if as a result, we increase the probability of a program malfunction. In this paper, we analyze how repair risk can be gauged.

1 Formulation of the Problem

Traditional approach to software testing. The main objective of software is to compute the desired results for all possible inputs. From this viewpoint, a reasonable way to test the software is:

- to run it on several inputs for which we know the desired answer, and
- to compare the results produced by this software with the desired values.

This was indeed the original approach to software testing.

It turned out that experts can detect some software defects without running the program. Once it turns out that on some inputs, the program is not producing the desired result, the next step is to find – and correct – the defect that leads to the wrong answer.

After going through this procedure many times, programmers started seeing common patterns in the original defect locations. For example, a reasonably typical mistake is forgetting to initiate the value of the variable. In this case, we may get different results depending on what happens to be the initial value stored in the part of the computer memory which is allocated for this variable.

This defect is even more dangerous if the variable is a *pointer*, i.e., crudely speaking, if it stores not the actual value of the corresponding object, but rather the memory address at which the actual value is stored. In this case, if we do not initialize the pointer, not only can we access the wrong value, but we may also end up with a non-existing address or an address outside the memory segment in which your program is allowed to operate – at which point the program stops, since it either does not know what value to pick or is not allowed to pick up the corresponding value (this is known as *segmentation fault*).

In many programming language that do not automatically check the array indices, another typical defect is asking for a value $a[i]$ of an array a for an index i which is outside the array's range. In this case, the compiler obediently finds the corresponding space in the memory, not realizing that it has beyond the place of the original array – this can overwrite important information; this is known as *buffer overrun*.

Static analysis tools. Once programmers realized that there are certain patterns of code typical for software defects, they started to come up with automatic tools for detecting such patterns and thus, warning the user of possible defects of different potential severity.

At present, there are many such tools – Coverity [1], Fortify, Lint, etc. – and most of these tools are efficiently used in practice; see, e.g., [3].

Some "defects" found by static analysis tools do not harm the program's functionality. For the purpose of this paper, it is important to mention that not all "defects" uncovered by a static analysis tool are actually hurting the program.

For example, some programs have *extra variables*, i.e., variables which are never used. This happens if a programmer originally planned to use the variable, started coding with it, then changed her mind but forgot to delete all the occurrences of this variable. Static analysis tools mark it as a possible detect, since in some situations, it is indeed an indication that some important value is never used. However, in many other cases, it may be syntactically clumsy, but does not cause any problem for the program.

Another defect that may not necessarily be harmful is the *logically dead* code, when a branch in a branching code is never visited. For example, if as part of the computations, we compute a square root of some quantity, it makes sense to make sure that this quantity is non-negative. When this quantity appears as result of long computations, it may happen that, due to rounding errors, a small non-negative value becomes small negative. In this case, it makes sense, if the value is negative, to replace this with 0. However, if we write a code this way, but we only use to compute the square root of an input which is always non-negative (e.g., of the weight), then the branch corresponding to a negative value is never used. In some cases, this may be a real defect, indicating that we may have missed something that would lead to the possibility of this condition. However, in cases like described above, this "defect" is mostly harmless.

Yes another example of a possible defect is indentation. In some programming languages like Python, indentation is used to indicate the end of the condition or

the end of the loop. However, in most other programming languages, indentation is ignored by the compiler, it simply helps people better understand each other's code. A static analysis tool will indicate the discrepancy between the indentation and the actual end of the condition or of a loop as a defect – and it indeed may be a defect. However, in many cases, it is just a sloppiness of a programmer that does not affect the program's execution.

Correcting non-harmful defects may cause real problems. Once a static analysis tool marks a piece of code as containing a possible defect, a natural reaction is to repair this part of the code.

The problem is that, every time you change even a few lines of software, this may introduce additional faults – and this time, serious ones. The only way to avoid this problem is to thoroughly test the changed software. However, an extensive testing – that would, in principle, reveal all new faults – is very expensive. As a result, many of these changes have to be performed without complete testing, thus introducing many possible points of failures at every place where the code was changed.

We need to gauge repair risk. To make the repair effort cost-efficient, it would be useful to know which defect repair have the highest risk of causing a problem after the fix. This way, we can focus our testing effort on these defects, save money by performing only limited testing of low-risk repairs.

And if an alleged defect is usually harmless but its repair may cause trouble, maybe a better strategy would be to keep this alleged defect in place. This is specially true for legacy software, software that was developed before static analysis tools became ubiquitous. If we apply such a tool to this software, we may find lots of alleged defects, but since the program has been running successfully for many years, it is highly probable that most of these alleged defects are actually harmless.

What we do in this paper. In this paper, we describe how repair risk can be gauged.

In our analysis, we use two different approaches: a probabilistic approach and a fuzzy-based approach. Interestingly, both approaches lead to the same expression for the repair risk, which makes us confident that this is indeed the correct expression for the repair risk.

2 Analysis of the Problem: Which Factors Determine the Risk

First factor: how big are the changes. Every time we change a line of code, we increase a risk. The more lines of code we change, the more we increase the risk.

Thus, one of the factors affecting the risk is the number L of lines of code that has been changed.

Second factor: how frequently are the changed lines used. Simple errors, when a piece of code always produced wrong results, are usually mostly filtered out by simple testing.

As a result, a faulty piece of code usually leads to correct results, but sometimes, for some combination of inputs, produces an erroneous value.

If we run this piece of code once, the chances that we accidentally hit the wrong inputs are small, so most probably, this will not lead to any serious problem. However, if this piece of code appears inside a loop, then for each program run, this piece of code runs many times with different inputs. As a result, it becomes more and more possible that in one of these inputs, we will get a wrong result – and thus, that the overall software will fail.

Thus, the second factor that we need to take into account is the number of iterations I that this particular piece of code is repeated in the program.

For example, if this piece of code is inside a for-loop that repeats 1000 times, then $I = 1000$. If this piece of code is inside a double for-loop – i.e., a for-loop for which each of its 1000 iterations is itself a for-loop with 1000 iterations (as often happens with matrix operations), we get $I = 1000 \times 1000 = 10^6$.

What we want. We want to be able to gauge the repair risk based on these two parameters: L and U.

Two types of software errors. As we have mentioned, there are, in effect, two types of software errors:

- rarer *fatal* error that practically always lead to a wrong result or, more generally, to a program malfunction; and
- more frequent *subtle* error which are usually harmless, but can cause trouble for a certain (reasonably rare) combination of inputs.

In our analysis, we need to take into account both types of software errors.

3 How to Gauge Repair Risk: Probabilistic Approach

Taking fatal errors into account. Let p_f denote the probability that a line of code contains a fatal error. Then, the probability that a line of code *does not* contain a fatal error is equal to $1 - p_f$.

Software errors in different lines are reasonably independent. Thus, the probability that an L-line new piece of code does not contain a fatal error can be computed as a product of L probabilities corresponding to each of the lines, i.e., as $(1 - p_f)^L$.

Taking subtle errors into account. Let p_s denote the probability that one run of a line will lead to a fault. So, the probability that a line performs correctly during one run is equal to $1 - p_s$.

Faults on different lines are, as we have mentioned, reasonably independent. Also, inputs corresponding to different iterations are reasonably independent. When we run an L-line piece of new code I times, this means that we perform

a running-of-one-line process $I \cdot L$ times. Thus, the probability that all lines will run correctly on all iterations is equal to the product of $I \cdot L$ individual probabilities, i.e., to the value $(1 - p_s)^{I \cdot L}$.

Taking both errors into account. Fatal and subtle errors are reasonably independent; e.g., as we have mentioned, discovering a fatal error does not prevent the software from having subtle error.

We know that the probability that fatal errors will not affect the result is equal to $(1 - p_f)^L$. We also know that the probability that subtle errors will not affect the result is equal to $(1 - p_s)^{I \cdot L}$. Thus, due to independence, the probability that the new piece of code will perform correctly, i.e., that neither of the two types of errors will surface, is equal to the product of these two probabilities, i.e., to the value

$$P = (1 - p_f)^L \cdot (1 - p_s)^{I \cdot L}. \tag{91}$$

Resulting criteria for repair risk. Ideally, we would like to know the probability of the program's fault. However, this requires that we know two parameters p_f and p_s, which may be difficult to get.

In the first approximation, it would be sufficient to simply *order* different repaired piece of code by risk – so that, in realistic situations with limited resources, we should concentrate all the testing on the pieces with the highest repair risk – and among probably harmless alleged defects, only repair those whose repair risk is the lowest.

From the viewpoint of such comparison, comparing the probabilities is equivalent to comparing their logarithms

$$\log(P) = L \cdot \log(1 - p_f) + I \cdot L \cdot \log(1 - p_s).$$

This is, in turn, equivalent to comparing the ratios

$$\frac{\log(P)}{\log(1 - p_s)} = I \cdot L + c \cdot L = L \cdot (I + c),$$

where we denoted

$$c \stackrel{\text{def}}{=} \frac{\log(1 - p_f)}{\log(1 - p_s)}.$$

So, we arrive at the following conclusion.

Probabilistic case: conclusion. To gauge the risk of repairing an alleged defect, we need to know:

- the number of lines L changed in the process of this repair, and
- the number of times I that this piece of code is repeated during one run of the software.

The relative repair risk is represented by the product

$$L \cdot (I + c), \tag{2}$$

for some constant c.

Comment. Note that, in contrast to the expression for probability, which required two parameters, this expression requires only one parameter – and one parameter is easier to experimentally determine than two.

4 How to Gauge Repair Risk: Fuzzy Approach

Need to go beyond the traditional probabilistic approach. To follow through with the probabilistic approach, we needed to make an assumption that faults corresponding to different lines and/or different iterations are completely independent. While in the first approximation, this assumption may sound reasonable, it is clear that in reality, this assumption is only approximately true: programmers know that a fault in one line often causes faults in the neighboring lines as well.

This can happen if the same mistake appears in different lines due to the same programmer's misunderstanding, or due to the fact that the second line may be obtained from the first one by editing – and so, an undetected error in the first line is simply copied into the second one.

Ideally, in addition to probabilities of one line being correct, we should also consider:

- a separate probability of two lines being correct – which is, in general, different from the square of the first probability,
- a separate probability that three lines are being correct, etc.

However, as we have mentioned earlier, even obtained two probabilities is difficult. Obtaining many others – corresponding to different numbers of lines and different numbers of iterations – would be practically impossible. What can we do?

Solution: fuzzy approach. Lotfi Zadeh faced a similar problem when he decided to analyze expert knowledge. Expert knowledge contains many imprecise ("fuzzy") rules that uses imprecise words from natural language like "small".

For each such word, and for each value x of the corresponding quantity, we can ask the expert to gauge to what extent the given value satisfies the given property: e.g, to what extent the value x is small. We can call the resulting estimate the degree of belief, the degree of confidence, we can call it a subjective probability – the name does not change anything.

The problem appears if we take into account that the condition of an expert rule contains usually not just one simple statement like "x is small", but an "and"-combination of several such statements. For example, a typical expert rule for driving a car would say something like "if we are going fast *and* the car in front decelerates a little bit, *and* the road is reasonably slippery, then we need to break gently".

To utilize this rule, we need to find the subjective probability (degree of confidence) that for a given velocity v, for a given distance d to the car in front, etc. the corresponding "and"-condition is satisfied.

How can we find this condition? Ideally, we should elicit this subjective probability from the expert for each possible combination of the inputs $(v, d, \ldots)$ However, for a large number of parameters, the number of such combinations becomes astronomical, and there is no way to ask an expertthe resulting millions and billions of questions.

What Zadeh proposed – and what is one of the main ideas behind what he called *fuzzy logic* (see, e.g., [2, 5, 4, 7–9]) is that, since we cannot elicit all degree of belief in "and"-statement $A \,\&\, B$ from the experts, we thus need to come up with an algorithm $f_{\&}(a, b)$ that would:

- given degree of belief a in the statement A and b in the statement B,
- return an estimate $f_{\&}(a, b)$ for the expert's degree of confidence in the "and"-statement $A \,\&\, B$.

This algorithm should satisfy some reasonable properties. For example, since $A \,\&\, B$ means the same as $B \,\&\, A$, it is reasonable to require that $f_{\&}(a, b) = f_{\&}(b, a)$, i.e., in mathematical terms, that the operation $f_{\&}(a, b)$ is *commutative*.

Similarly, since $A \,\&\, (B \,\&\, C)$ means the same as $(A \,\&\, B) \,\&\, C$, it is reasonable to require that $f_{\&}(a, f_{\&}(b, c)) = f_{\&}(f_{\&}(a, b), c)$, i.e., that the operation $f_{\&}(a, b)$ is *associative*. An "and"-operation $f_{\&}(a, b)$ that satisfies these and other similar properties is known as a *t-norm.*

There are many possible t-norms. One of them is the product $f_{\&}(a, b) = a \cdot b$, that corresponds to the case when all the events are independent. However, there are many other t-norms – that correspond to possible dependence.

Let us apply this approach to our problem. In this approach, we no longer assume independence. To compute the subjective probability (degree of confidence) in an "and"–combination o different events, instead of a product, we can use an appropriate t-norm.$f_{\&}(a, b)$. Thus, instead of the formula (1), we get a more complex formula

$$P = f_{\&}(1 - p_f, \ldots, 1 - p_f \,(L \text{ times}), 1 - p_s, \ldots, 1 - p_s \,(I \cdot L \text{ times})). \qquad (3)$$

It is known (see, e.g., [6])) that every t-norm can be approximated, with arbitrary accuracy, by t-norms of the type $f_{\&}(a, b) = h^{-1}(h(a) \cdot h(b))$, for some strictly increasing function $h(x)$, where $h^{-1}(x)$ denotes an inverse function, for which $h^{-1}(h(x)) = x$. So, for all practical purposes, we can safely assume that our t-norm is exactly of this type.

For such t-norms, $f_{\&}(a, b, \ldots, c) = h^{-1}(h(a) \cdot h(b) \cdot \ldots h(c))$. Thus, the formula (3) takes the form

$$P = h^{-1}\left((h(1 - p_f))^L \cdot (h(1 - p_s))^{I \cdot L}\right). \qquad (4)$$

Comparing such values is equivalent comparing the values

$$h(P) = (h(1 - p_f))^L \cdot (h(1 - p_s))^{I \cdot L},$$

or, equivalently, the value

$$\log(h(P)) = L \cdot \log(h(1 - p_f)) + I \cdot L \cdot \log(h(1 - p_s)),$$

or the value

$$\frac{\log(h(P))}{\log(h(1-p_s))} = I \cdot L + c \cdot L = L \cdot (I + c),$$

where

$$c \stackrel{\text{def}}{=} \frac{\log(h(1-p_f))}{\log(h(1-p_s))}.$$

Conclusion. The fact that in this more general not-necessarily-independent case, we get the same expression $L \cdot (I + c)$ for repair risk makes us confident that this is indeed the correct expression.

Acknowledgments

This work was supported in part by the US National Science Foundation grant HRD-1242122.

References

1. A. Almossawi, K. Lim, and T. Sinha, *Analysis Tool Evaluation: Coverity Prevent. Final Report*, Carnegie Mellon University, Pittsubrgh, Pannsylvania, 2006, http://www.cs.cmu.edu/ aldrich/courses/654-sp09/tools/cure-coverity-06.pdf
2. R. Belohlavek, J. W. Dauben, and G. J. Klir, *Fuzzy Logic and Mathematics: A Historical Perspective*, Oxford University Press, New York, 2017.
3. P. Emanuelsson and U. Nilsson, "A Comparative Study of Industrial Static Analysis Tools", *Electronic Notes in Theoretical Computer Science*, 2008, Vol. 217, pp. 5–21.
4. G. Klir and B. Yuan, *Fuzzy Sets and Fuzzy Logic*, Prentice Hall, Upper Saddle River, New Jersey, 1995.
5. J. M. Mendel, *Uncertain Rule-Based Fuzzy Systems: Introduction and New Directions*, Springer, Cham, Switzerland, 2017.
6. H. T. Nguyen, V. Kreinovich, and P. Wojciechowski, "Strict Archimedean t-norms and t-conorms as universal approximators", *International Journal of Approximate Reasoning*, 1998, Vol. 18, Nos. 3–4, pp. 239–249.
7. H. T. Nguyen and E. A. Walker, *A First Course in Fuzzy Logic*, Chapman and Hall/CRC, Boca Raton, Florida, 2006.
8. V. Novák, I. Perfilieva, and J. Močkoř, *Mathematical Principles of Fuzzy Logic*, Kluwer, Boston, Dordrecht, 1999.
9. L. A. Zadeh, "Fuzzy sets", *Information and Control*, 1965, Vol. 8, pp. 338–353.