How to Gauge Repair Risk?

Francisco Zapata[1] and Vladik Kreinovich[2]

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

1. 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.

2. 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 defect patterns.
- 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 was stored in the part of the computer memory which is allocated for this variable.

3. Software Defects (cont-d)

- 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;
  - this is known as *segmentation fault*.

4. Software Defects (cont-d)

- Many programming language do not automatically check the array indices.
- Then, a 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.
- However, this space is beyond the place of the original array.
- This can overwrite important information; this is known as *buffer overrun.*

5. Static Analysis Tools

- Programmers realized long time ago that there are certain patterns of code typical for software defects.
- So, they started to come up with automatic tools for detecting such patterns.
- These tools warn the user of possible defects of different potential severity.
- At present, there are many such tools – Coverity, Fortify, Lint, etc.
- Most of these tools are efficiently used in practice.

6. Some "Defects" Found by Static Analysis Tools Do Not Harm the Program's Functionality

- 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.
- Indeed, in some situations, it is indeed an indication that some important value is never used.

7. Some "Defects" Do Not Harm the Program's Functionality (cont-d)

- However, in many other cases:
  - it may be syntactically clumsy,
  - but it 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.

8. Some "Defects" Do Not Harm the Program's Functionality (cont-d)

- 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 it to compute $\sqrt{x}$ of a non-negative x (e.g., of the weight),
  - then the branch corresponding to a negative value is never used.

9. Some "Defects" Do Not Harm the Program's Functionality (cont-d)

- 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 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.

10. Some "Defects" Do Not Harm the Program's Functionality (cont-d)

- Indentation simply helps people better understand each other's code.
- A static analysis tool:
  - will detect the discrepancy between the indentation and the actual end of the condition or of a loop, and
  - indicate it as a defect.
- It indeed may be a serious defect.
- However, in many cases, it is just a sloppiness of a programmer that does not affect the program's execution.

11. 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.

12. Correcting Non-Harmful Defects May Cause Real Problems (cont-d)

- As a result, many of these changes have to be performed without complete testing.
- This introduces many possible points of failures at every place where the code was changed.

13. We Need to Gauge Repair Risk

- To make the repair effort cost-efficient, it would be useful to know:
  - which defect's repair
  - have the highest risk of causing a problem after the fix.
- This way:
  - we can focus our testing effort on these defects, and
  - 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.

14. We Need to Gauge Repair Risk (cont-d)

- This is specially true for legacy software, 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.

15. What We Do in This Talk

- In this talk, 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.
- This makes us confident that this is indeed the correct expression for the repair risk.

16. First Risk 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.

17. Second Risk 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.
- Thus, 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.

18. Second Risk Factor (cont-d)

- 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.
- So, 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.

19. Second Risk Factor (cont-d)

- 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: e.g.,
  - a for-loop for which each of its 1000 iterations is itself a for-loop with 1000 iterations,
  - we get $I = 1000 \times 1000 = 10^6$.
- This often happens with matrix operations.
- We want to be able to gauge the repair risk based on these two parameters: L and I.

20. 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 to a program malfunction; and
  - more frequent *subtle* errors 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.

21. 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 $(1-p_f)^L$.

22. 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.

23. Taking Subtle Errors into Account (cont-d)

- 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 $(1 - p_s)^{I \cdot L}$.

24. Taking Both Errors into Account

- Fatal and subtle errors are reasonably independent: as we have mentioned,
  - discovering a fatal error
  - does not prevent the software from having subtle errors.
- 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 $(1 - p_s)^{I \cdot L}$.

25. Taking Both Errors into Account (cont-d)

- 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

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

26. 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.
- Then, 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.

27. Resulting Criteria for Repair Risk (cont-d)

- 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

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

- Here, we denoted $c \stackrel{\text{def}}{=} \dfrac{\log(1 - p_f)}{\log(1 - p_s)}$.
- So, we arrive at the following conclusion.

28. 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 $L \cdot (I + c)$, for some constant c.
- 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.

29. 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.
- In the first approximation, this assumption may sound reasonable.
- However, 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.

30. Need to Go Beyond the Traditional Probabilistic Approach (cont-d)

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

31. Need to Go Beyond the Traditional Probabilistic Approach (cont-d)

- 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?

32. 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.
- Example: words 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.

33. Fuzzy Approach (cont-d)

- For example, we can ask to what extent the value x is small; we can call the resulting estimate
  - the degree of belief,
  - the degree of confidence,
  - the 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.

34. Fuzzy Approach (cont-d)

- For example, a typical expert rule for driving a car would say something like this:
  - 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.

35. Fuzzy Approach (cont-d)

- How can we find this degree? 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.
- There is no way to ask an expert the resulting millions and billions of questions.
- What Zadeh proposed – and what is one of the main ideas behind what he called *fuzzy logic* 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.

36. Fuzzy Approach (cont-d)

- This algorithm should:
  - 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*.

37. Fuzzy Approach (cont-d)

- 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.

38. 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 of different events,
  - instead of a product,
  - we can use an appropriate t-norm. $f_\&(a, b)$.
- Thus, instead of the product formula, we get a more complex formula

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

39. Let Us Apply Fuzzy Approach (cont-d)

- It is known 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 above formula takes the form
$$P=h^{-1}\left((h(1-p_f))^L\cdot(h(1-p_s))^{I\cdot L}\right).$$

40. Let Us Apply Fuzzy Approach (cont-d)

- Comparing such values is equivalent comparing the values $h(P) = (h(1 - p_f))^L \cdot (h(1 - p_s))^{I \cdot L}$.
- This is equivalent to comparing the values

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

- This, in its turn, is equivalent to comparing the values

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

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

41. Conclusion

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

42. Acknowledgments

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