BLOG@CACM

Soundness and Completeness: With Precision

By Bertrand Meyer
April 20, 2019
Comments

Over breakfast at your hotel, you read an article berating banks about the fraudulent credit card transactions they let through. You proceed to check out and bang! Your credit card is rejected because (as you find out later) the bank thought [1] it couldn't possibly be you in that exotic place. Ah, those banks! They accept too much. Ah, those banks! They reject too much. Finding the right balance is a case of *soundness* versus *precision*.

Similar notions are essential to the design of tools for program analysis, looking for such suspicious cases as dead code (program parts that will never be executed). An analysis can be sound, or not; it can be complete, or not.

These widely used concepts are sometimes misunderstood. The first answer I get when innocently asking people whether the concepts are clear is yes, of course, everyone knows! Then, as I bring up such examples as credit card rejection or dead code detection, assurance quickly yields to confusion. One sign that things are not going well is when people start throwing in terms like "true positive" and "false negative". By then any prospect of reaching a clear conclusion has vanished. I hope that after reading this article you will never again (in a program analysis context) be tempted to use them.

Now the basic idea is simple. An analysis is *sound* if it reports all errors, and *complete* if it only reports errors. If not complete, it is all the more *precise* that it reports fewer non-errors.

You can stop here and not be too far off [2]. But a more nuanced and precise discussion helps.

1. A relative notion

As an example of common confusion, one often encounters attempts to help through something like Figure 1, which cannot be right since it implies that all sound methods are complete. (We'll have better pictures below.)

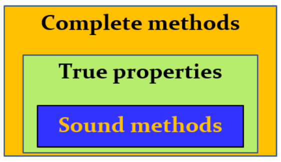

Figure 1: Naïve (and wrong) illustration

Perhaps this example can be dismissed as just a bad use of illustrations [3] but consider the example of looking for dead code. If the analysis wrongly determines that some reachable code is unreachable, is it unsound or incomplete?

With this statement of the question, the only answer is: *it depends!*

It depends on the analyzer's mandate:

> If it is a code checker that alerts programmers to cases of bad programming style, it is incomplete: it reports as an error a case that is not. (Reporting that unreachable code is reachable would cause unsoundness, by missing a case that it should have reported.)
>
> If it is the dead-code-removal algorithm of an optimizing compiler, which will remove unreachable code, it is

> unsound: the compiler will remove code that it should not. (Reporting that unreachable code is reachable would cause incompleteness, by depriving the compiler of an optimization.)

As another example, consider an analyzer that finds out whether a program will terminate. (If you are thinking "*but that can't be done!*", see the section "Appendix: about termination" at the very end of this article.) If it says a program does not terminates when in fact it does, is it unsound or incomplete?

Again, that depends on what the analyzer seeks to establish. If it is about the correctness of a plain input-to-output program (a program that produces results and then is done), we get incompleteness: the analyzer wrongly flags a program that is actually OK. But if it is about verifying that continuously running programs, such as the control system for a factory, will not stop ("liveness"), then the analyzer is unsound.

Examples are not limited to program analysis. A fraud-indentification process that occasionally rejects a legitimate credit card purchase is, from the viewpoint of preserving the bank from fraudulent purchases, incomplete. From the viewpoint of the customer who understands a credit card as an instrument enabling payments as long as you have enough credit, it is unsound.

These examples suffice to show that there cannot be absolute definitions of soundness and precision: the determination depends on which version of a boolean property we consider **desirable**. This decision is human and subjective. Dead code is desirable for the optimizing compiler and undesirable (we will say it is a **violation**) for the style checker. Termination is desirable for input-output programs and a violation for continuously running programs.

Once we have decided which cases are desirable and which are violations, we can define the concepts without any ambiguity: soundness means rejecting all violations, and completeness means accepting all desirables.

While this definition is in line with the unpretentious, informal one in the introduction, it makes two critical aspects explicit:

> *Relativity*. Everything depends on an explicit decision of what is desirable and what is a violation. Do you want customers always to be able to use their credit cards for legitimate purchases, or do you want to detect all frauds attempts?
>
> *Duality*. If you reverse the definitions of desirable and violation (they are the negation of each other), you automatically reverse the concepts of soundness and completeness and the associated properties.

We will now explore the consequences of these observations.

2. Theory and practice

For all sufficiently interesting problems, theoretical limits (known as Rice's theorem) ensure that it is impossible to obtain both soundness and completeness.

But it is not good enough to say "we must be ready to renounce either soundness or completeness". After all, it is very easy to obtain soundness if we forsake completeness: *reject every case*. A termination-enforcement analyzer can reject every program as potentially non-terminating. A bank that is concerned with fraud can reject every transaction (this seems to be my bank's approach when I am traveling) as potentially fraudulent. Dually, it is easy to ensure completeness if we just sacrifice soundness: *accept every case*.

These extreme theoretical solutions are useless in practice; here we need to temper the theory with considerations of an engineering nature.

The practical situation is not as symmetric as the concept of duality theoretically suggests. If we have to sacrifice one of the two goals, it is generally better to accept some incompleteness: getting false alarms (spurious reports about cases that turn out to be harmless) is less damaging than missing errors. Soundness, in other words, is essential.

Even on the soundness side, though, practice tempers principle. We have to take into account the engineering reality of how tools get produced. Take a program analyzer. In principle it should cover the entire programming language. In practice, it will be built step by step: initially, it may not handle advanced features such as exceptions, or dynamic mechanisms such as reflection (a particularly hard nut to crack). So we may have to trade soundness for what has been called "*soundiness*" [4], meaning soundness outside of cases that the technology cannot handle yet.

If practical considerations lead us to more tolerance on the soundness side, on the completeness side they drag us (duality strikes again) in the opposite direction. Authors of analysis tools have much less flexibility than the theory would suggest. Actually, close to none. In principle, as noted, false alarms do not cause catastrophes, as missed violations do; but in practice they can be almost as bad. Anyone who has ever worked on or with a static analyzer, going back to the venerable Lint analyzer for C, knows the golden rule: *false alarms kill an analyzer*. When people discover the tool and run it for the first time, they are thrilled to discover how it spots some harmful pattern in their program. What counts is what happens in subsequent runs. If the useful gems among the analyzer's diagnostics are lost in a flood of irrelevant warnings, forget about the tool. People just do not have the patience to sift through the results. In practice any analysis tool has to be darn close to completeness if it has to stand any chance of adoption.

Completeness, the absence of false alarms, is an all-or-nothing property. Since in the general case we cannot achieve it if we also want soundness, the engineering approach suggests using a numerical rather than boolean criterion: *precision*. We may define the precision pr as 1 - im where im is the ***im***precision: the proportion of false alarms.

The theory of classification defines precision differently: as pr = tp / (tp + fp), where tp is the number of false positives and fp the number of true positives. (Then im would be fp / (tp + fp).) We will come back to this definition, which requires some tuning for program analyzers.

From classification theory also comes the notion of *recall*: tp / (tp + fn) where fn is the number of false negatives. In the kind of application that we are looking at, recall corresponds to soundness, taken not as a boolean property ("*is my program sound?*") but a quantitative one ("*how sound is my program?*"). The degree of *unsoundness* un would then be fn / (tp + fn).

3. Rigorous definitions

With the benefit of the preceding definitions, we can illustrate the concepts, correctly this time. Figure 2 shows two different divisions of the set of U of call cases (universe):

Some cases are desirable (D) and others are violations (V).

We would like to know which are which, but we have no way of finding out the exact answer, so instead we run an analysis which passes some cases (P) and rejects some others (R).

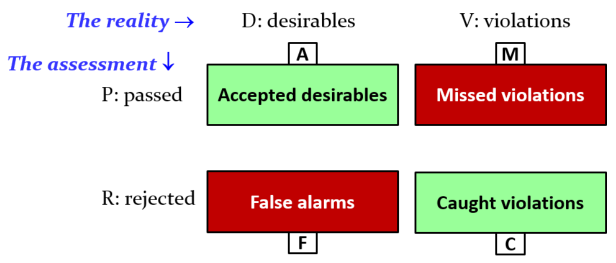

Figure 2: All cases, classified

The first classification, left versus right columns in Figure 2, is how things are (the reality). The second classification, top versus bottom rows, is how we try to assess them. Then we get four possible categories:

In two categories, marked in green, assessment hits reality on the nail: accepted desirables (A), rightly passed, and caught violations (C), rightly rejected.

In the other two, marked in red, the assessment is off the mark: missed violations (M), wrongly passed; and false alarms (F), wrongly accepted.

The following properties hold, where U (Universe) is the set of all cases and ⊕ is disjoint union [5]:

-- Properties applicable in all cases:

U = D ⊕ V
U = P ⊕ R
D = A ⊕ F
V = C ⊕ M
P = A ⊕ M
R = C ⊕ F
U = A ⊕M ⊕ F ⊕ C

We also see how to define the **precision** pr: as the proportion of actual violations to reported violations, that is, the size of C relative to R. With the convention that u is the size of U and so on, then pr = c / r, that is to say:

pr = c / (c + f) -- Precision
im = f / (c + f) -- Imprecision

We can similarly define soundness in its quantitative variant (recall):

so = a / (a + m) -- Soundness (quantitative)
un = m / (a + m) -- Unsoundness

These properties reflect the full duality of soundness and completeness. If we reverse our (subjective) criterion of what makes a case desirable or a violation, everything else gets swapped too, as follows:

D	↔ V	-- and d ↔ v (cardinals of these sets)
P	↔ R	-- and p ↔ r
A	↔ C	-- and a ↔ c
F	↔ M	-- and f ↔ m
pr	↔ so	-- Precision, soundness (quantitative)
im	↔ un	-- Imprecision, unsoundness
desirable	↔ violation	-- Classification of cases
sound	↔ complete	-- See next sections
under-	↔ over-	-- Approximations, see section 4
desirability	↔ violation	-- Kinds of analyses, see section 5

Figure 3: Duality

We will say that properties paired in this way "*dual*" each other [6].

It is just as important (perhaps as a symptom that things are not as obvious as sometimes assumed) to note which properties do *not* dual. The most important examples are the concepts of "true" and "false" as used in "true positive" etc. These expressions are all the more confusing that the concepts of True and False do dual each other in the standard duality of Boolean algebra (where True duals False, Or duals And, and an expression duals its negation). In "true positive" or "false negative", "true" and "false" do not mean True and False: they mean cases in which (see figure 2 again) the assessment respectively *matches* or does not match the reality. Under duality we reverse the criteria in both the reality and the assessment; but matching remains matching! The green areas remain green and the red areas remain red.

The dual of "positive" is "negative", but the dual of true is true and the dual of false is false (in the sense in which those terms are used here: matching or not). So the dual of true positive is true negative, not false negative, and so on. Hereby lies the source of the endless confusions.

The terminology of this article removes these confusions. Desirable duals violation, passed duals rejected, the green areas dual each other and the red areas dual each other.

4. Sound and complete analyses

If we define an ideal world as one in which assessment matches reality [7], then figure 2 would simplify to just two possibilities, the green areas:

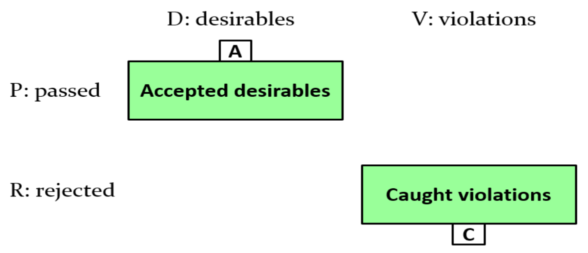

Figure 4: Perfect analysis (sound and complete)

This scheme has the following properties:

```
-- Properties of a perfect (sound and complete) analysis as in Figure 4:
M = ∅        -- No missed violations
F = ∅        -- No false alarms
P = D        -- Identify  desirables exactly
R = V        --Identify violations exactly
```

As we have seen, however, the perfect analysis is usually impossible. We can choose to build a sound solution, potentially incomplete:

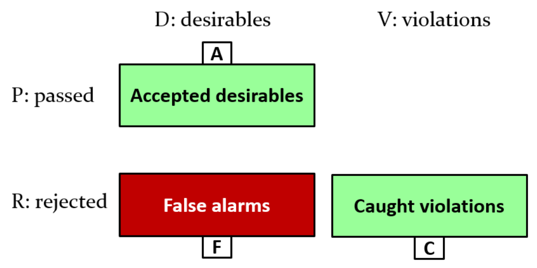

Figure 5: Sound desirability analysis, not complete

In this case:

-- Properties of a sound analysis (not necessarily complete) as in Figure 5:

M = ∅	-- No missed violations
P = A	-- Accept only desirables
V = C	-- Catch all violations
P ⊆ D	-- Under-approximate desirables
R ⊇ V	-- Over-approximate violations

Note the last two properties. In the perfect solution, the properties P = D and R = V mean that the assessment, yielding P and V, exactly matches the reality, D and V. From now on we settle for assessments that *approximate* the sets of interest: under-approximations, where the assessment is guaranteed to compute no more than the reality, and over-approximations, where it computes no less. In all cases the assessed sets are either subsets or supersets of their counterparts. (Non-strict, i.e. ⊆ and ⊇ rather than ⊂ and ⊃; "approximation" means *possible* approximation. We may on occasion be lucky and capture reality exactly.)

We can go dual and reach for completeness at the price of possible unsoundness:

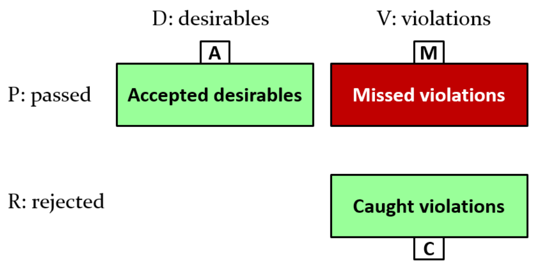

Figure 6: Complete desirability analysis, not sound

The properties are dualled too:

-- Properties of a complete analysis (not necessarily sound), as in Figure 6:

F = ∅	-- No false alarms
R = C	-- Reject only violations
D = A	-- Accept all desirables
P ⊇ D	-- Over-approximate desirables
R ⊆ V	-- Under-approximate violations

5. Desirability analysis versus violation analysis

We saw above why the terms "true positives", "false negatives" etc., which do not cause any qualms in classification theory, are deceptive when applied to the kind of pass/fail analysis (desirables versus violations) of interest here. The definition of precision provides further evidence of the damage. Figure 7 takes us back to the general case of Figure 2 (for analysis that is guaranteed neither sound nor complete) but adds these terms to the respective categories.

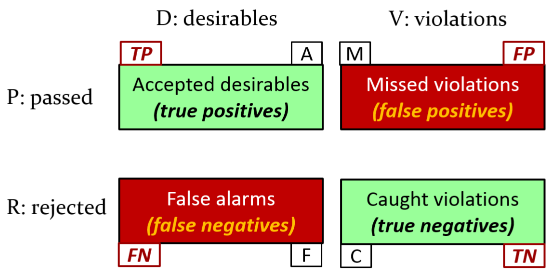
D: desirables
V: violations
TP
A
M
FP
P: passed
Accepted desirables (true positives)
Missed violations (false positives)
R: rejected
False alarms (false negatives)
Caught violations (true negatives)
FN
F
C
TN

Figure 7: Desirability analysis (same as fig. 2 with added labeling)

The analyzer checks for a certain desirable property, so if it wrongly reports a violation (F) that is a false negative, and if it misses a violation (M) it is a false positive. In the definition from classification theory (section 2, with abbreviations standing for True/False Positives/Negatives): TP = A, FP = M, FN = F, TN = C, and similarly for the set sizes: tp = a, fp = m, fn = f, tn = c.

The definition of precision from classification theory was pr = tp / (tp + fp), which here gives a / (a + m). This cannot be right! Precision has to do with how close the analysis is to completeness, that is to day, catching all violations.

Is classification theory wrong? Of course not. It is simply that, just as Alice stepped on the wrong side of the mirror, we stepped on the wrong side of duality. Figures 2 and 7 describe *desirability analysis*: checking that a tool does something good. We assess non-fraud from the bank's viewpoint, not the stranded customer's; termination of input-to-output programs, not continuously running ones; code reachability for a static checker, not an optimizing compiler. Then, as seen in section 3, a / (a + m) describes not precision but soundness (in its quantitative interpretation, the parameter called "so" above).

To restore the link with classification theory , we simply have to go dual and take the viewpoint of *violation* *analysis*. If we are looking for possible violations, the picture looks like this:

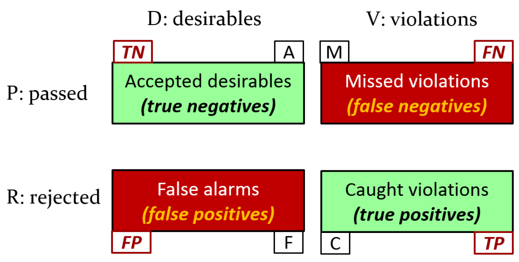
D: desirables
V: violations
P: passed
R: rejected
TN
A
Accepted desirables (true negatives)
M
FN
Missed violations (false negatives)
False alarms (false positives)
FP
F
Caught violations (true positives)
C
TP

Figure 8: Violation analysis (same as fig. 7 with different positive/negative labeling)

Then everything falls into place: tp = c, fp = f, fn = m, tn = a, and the classical definition of precision as pr = tp / (tp + fp) yields c / (c + f) as we are entitled to expect.

In truth there should have been no confusion since we always have the same picture, going back to Figure 2, which accurately covers all cases and supports both interpretations: desirability analysis and violation analysis. The confusion, as noted, comes from using the duality-resistant "true"/"false" opposition.

To avoid such needless confusion, we should use the four categories of the present discussion: accepted desirables, false alarms, caught violations and missed violations [8]. Figure 2 and its variants clearly show the duality, given explicitly in Figure 3, and sustains interpretations both for desirability analysis and for violation analysis. Soundness and completeness are simply special cases of the general framework, obtained by ruling out one of the cases of incorrect analysis in each of Figures 4 and 5. The set-theoretical properties listed after Figure 2 express the key concepts and remain applicable in all variants. Precision c / (c + f) and quantitative soundness a / (a + m) have unambiguous definitions matching intuition.

The discussion is, I hope, sound. I have tried to make it complete. Well, at least it is precise.

Notes and references

[1] Actually it's not your bank that "thinks" so but its wonderful new "Artificial Intelligence" program.

[2] For a discussion of these concepts as used in testing see Mauro Pezzè and Michal Young, *Software Testing and Analysis: Process, Principles and Techniques*, Wiley, 2008.

[3] Edward E. Tufte: *The Visual Display of Quantitative Information*, 2nd edition, Graphics Press, 2001.

[4] Michael Hicks,*What is soundness (in static analysis)?*, blog article available here, October 2017.

[5] The disjoint union property X = Y ⊕ Z means that Y ∩ Z = ∅ (Y and Z are disjoint) and X = Y ∪ Z (together, they yield X).

[6] I thought this article would mark the introduction into the English language of "*dual*" as a verb, but no, it already exists in the sense of turning a road from one-lane to two-lane (dual).

[7] As immortalized in a toast from the cult movie *The Prisoner of the Caucasus*: "*My great-grandfather says: I have the desire to buy a house, but I do not have the possibility. I have the possibility to buy a goat, but I do not have the desire. So let us drink to the matching of our desires with our possibilities*." See 6:52 in the version with English subtitles.

[8] To be fully consistent we should replace the term "false alarm" by *rejected desirable*. I have retained it because it is so well established and, with the rest of the terminology as presented, does not cause confusion.

[9] Byron Cook, Andreas Podelski, Andrey Rybalchenko: *Proving Program Termination*, in *Communications of the ACM*, May 2011, Vol. 54 No. 5, Pages 88-98.

Background and acknowledgments

This reflection arose from ongoing work on static analysis of OO structures, when I needed to write formal proofs of soundness and completeness and found that the definitions of these concepts are more subtle than commonly assumed. I almost renounced writing the present article when I saw Michael Hicks's contribution [4]; it is illuminating, but I felt there was still something to add. For example, Hicks's set-based illustration is correct but still in my opinion too complex; I believe that the simple 2 x 2 pictures used above convey the ideas more clearly. On substance, his presentation and others that I have seen do not explicitly mention duality, which in my view is the key concept at work here.

I am grateful to Carlo Ghezzi for enlightening discussions, and benefitted from comments by Alexandr Naumchev and others from the Software Engineering Laboratory at Innopolis University.

Appendix: about termination

With apologies to readers who have known all of the following from kindergarten: a statement such as (section 1): "*consider an analyzer that finds out whether a program will terminate*" can elicit no particular reaction (the enviable bliss of ignorance) or the shocked rejoinder that such an analyzer is impossible because termination (the "halting" problem) is undecidable. This reaction is just as incorrect as the first. The undecidability result for the halting problem says that it is impossible to write a general termination analyzer that will always provide the right answer, in the sense of both soundness and completeness, for any program in a realistic programming language. But that does not preclude writing termination analyzers that answer the question correctly, in finite time, for given programs. After all it is not hard to write an analyzer that will tell us that the program **from** do_nothing **until True loop** do_nothing **end** will terminate and that the program **from** do_nothing **until False loop** do_nothing **end** will not terminate. In the practice of software verification today, analyzers can give such sound answers for very large classes of programs, particularly with some help from programmers who can obligingly provide *variants* (loop variants, recursion variants). For a look into the state of the art on termination, see the beautiful survey by Cook, Podelski and Rybalchenko [9].

Bertrand Meyer *is a professor of software engineering (emeritus) at ETH Zurich (Switzerland), chief technology officer of Eiffel Software (Goleta, CA), professor at Politecnico di Milano (Italy), and head of the software engineering lab at Innopolis University (Russia).*

No entries found