CHAPTER 9

Verification and Validation (V&V)

9.1 Two Tasks

The two concepts of verification and validation are central to determining that a software-based system does what it is intended to do. The definitions of these concepts in IEC 61508-3:2010 are as follows. The notion of what it is that software "is intended to do" is informal, and needs must remain partly so. What the software is intended to do is captured as far as possible by engineers in the functional requirements specification. "Functional requirements" are those which say what the software is to do; how devices which execute the software are to behave. Non-functional requirements are those which, for example, specify what higher-level programming language the source code of the software should be written in.

The specification of the requirements may go wrong. The need for certain kinds of behaviour in certain circumstances might have been overlooked, because the particular circumstances were not considered by the team deriving the requirements specification – there is a gap in the formal requirements. Michael Jackson has pointed out that, in moderately complex embedded systems, various parts of the requirements are derived with the help of domain specialists in those parts, and requirements on parts can easily contradict requirements on other parts, so that the overall requirements specification is inconsistent – and that often this is overlooked. Obviously, no system can fulfil an inconsistent requirements specification.

There are, then, two tasks concerned with the requirements specification of a

software system:

- to know that you have the right requirements specification (or one that is “good enough”)
- to know that the software fulfils the requirements specification

These tasks are clearly different. We need two different names for them. Two names which are often used are *verification* and *validation*. Formal verification is a term used for the task of determining using mathematical means that the software fulfils the requirements specification. Validation is a term used by many to check the requirements specification against the informal need. But this is not how these terms are used in the standards for safety-related E/E/PE systems. Here are the definitions.

9.2 The Definitions in Standards

> 3.8.1
> verification
> confirmation by examination and provision of objective evidence that the requirements have been fulfilled
> (ISO 8402, definition 2.17, modified)
>
> NOTE In the context of this standard, verification is the activity of demonstrating for each phase of the relevant safety lifecycle (overall, E/E/PE system and software), by analysis, mathematical reasoning and/or tests, that, for the specific inputs, the outputs meet in all respects the objectives and requirements set for the specific phase.
> EXAMPLE Verification activities include
>
> - reviews on outputs (documents from all phases of the safety lifecycle) to ensure compliance with the objectives and requirements of the phase, taking into account the specific inputs to that phase;
> - design reviews;
> - tests performed on the designed products to ensure that they perform according to their specification;
> - integration tests performed where different parts of a system are put together in a step-by-step manner and by the performance of

environmental tests to ensure that all the parts work together in the specified manner.

3.8.2
validation
confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled
(ISO 8402, definition 2.18, modified)

NOTE 1 In this standard there are three validation phases:

- overall safety validation (see Figure 2 of IEC 61508-1); ?
- E/E/PE system validation (see Figure 3 of IEC 61508-1); ?
- software validation (see Figure 4 of IEC 61508-1).

NOTE 2 Validation is the activity of demonstrating that the safety-related system under consideration, before or after installation, meets in all respects the safety requirements specification for that safety-related system. Therefore, for example, software validation means confirming by examination and provision of objective evidence that the software satisfies the software safety requirements specification.

ISO 8402 is obsolete (I believe it has been formally withdrawn and its content renumbered as part of the ISO 9000 series on quality management). IEC 60050, the International Electrotechical Vocabulary, includes the following two entries in its section on dependability (Part 191).

191-01-17
verification
confirmation, through the provision of objective evidence, that specified requirements have been fulfilled

Note 1 to entry: The term “verified” is used to designate the corresponding status.
Note 2 to entry: Design verification is the application of tests and appraisals to assess conformity of a design to the specified requirement.
Note 3 to entry: In the case of software, verification is conducted at various stages of development, examining the software and its constituents to

determine conformity to the requirements specified at the beginning of that stage.
(SOURCE: ISO 9000:2005, 3.8.4, modified – Note 2 has been modified and Note 3 has been added)

191-01-18
validation
confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled

Note 1 to entry: The term "validated" is used to designate the corresponding status.
Note 2 to entry: The use conditions for validation can be real or simulated.
Note 3 to entry: In design and development, validation concerns the process of examining an item to determine conformity with user needs.
Note 4 to entry: Validation is normally performed during the final stage of development, under defined operating conditions, although it may also be performed in earlier stages.
Note 5 to entry: Multiple validations may be carried out if there are different intended uses.
(SOURCE: ISO 9000:2005, 3.8.5, modified – Notes 3, 4 and 5 have been added)

9.3 What May We Say?

Note that all these definitions speak of requirements simpliciter and not functional requirements. Many non-functional requirements of such a formal nature can be checked using similarly simple criteria. For example, it is obviously a simple matter to check, if there is a specific programming-language requirement, that the required language has been used for source code. So this generalisation seems to be appropriate.

A complication enters, though, in what is meant here by "requirements". Are we speaking of the actual need? If so, when the system fails in a situation which had not been envisaged or adequately accounted for, we may say the requirements were not met. However, if there is a specification of the requirements, that specification will thereby be found wanting. Many people, when they talk about "requirements" mean

the requirements specification if there is one (and for critical systems it is a matter of best practice that there is one). It would be odd to say that the requirements were not met but the requirements specification was fulfilled. But this is apparently what one would have to say.

If by "requirements" we are speaking of the requirements specification, then we have no word for the case in which the system need was not met but the requirements specification was fulfilled.

If there is a case in which an unforeseen circumstance occurs and the system does not behave as we would wish, this unforeseen circumstance could in principle be taken to be a "specific intended use" where the system does not function as intended (and therefore is not validated), but many or most occurrences of unforeseen circumstances are – well, unforeseen, and the phrase "specific intended use" does not fit.

The terms seem geared much more to a use in which a piece of kit, including software or not, has a generic function and then is used in a specific system. For example, Company X might sell a box B to do task M. Company Y is building a system S, for which it needs something to perform task M. Company X would engage in verification: does our box B do task M (under the given constraints?). Company Y comes to Company X and says "We hear you have a box that can do task M". Company Y then checks that box B will actually perform the M task under the conditions pertaining for system S; this fits the definition of validation.

"Verification" speaks to the confirmation that this generic function is fulfilled. A particular application might be much more specific, and "validation" can then be the term which speaks to the fitness of the software for this specific use. "Verification" says that the software does generally what it is supposed to do, and "validation" says that it fits the intended application well.

We are still missing a term for the case in which the software fulfils the requirements specification but acts inappropriately in circumstances which are not covered by the requirements specification.

9.4 Testing

The situation is complicated somewhat further by a statement in a subclause of IEC 61508-3:2010.

> 7.7.2.7 The validation of safety-related software aspects of system safety shall meet the following requirements: a) testing shall be the main validation method for software; analysis, animation and modelling may be used to supplement the validation activities;
> .....

It is astonishing still to find such a requirement for ultra-reliable software. We have already seen in Chapter 1 Table 1.2 that, for continuously-operating software such as feedback control, extensive and indeed impractically much statistical testing is required in order to come to reasonable assurance that the software will function to the reliability levels required for the Safety Integrity Levels (SILs) 2-4 for kit operating in "continuous mode" (the choice of reliability levels in Table 1.2 are those appropriate for the various SILs). This was already pointed out for the different domain of commercial airworthiness certification by Littlewood and Strigini in 1993 [5]. It is hard to see how testing can thereby be the main method of validation of a continuous-mode software-implemented safety function.

Indeed, it was first said 48 years ago that "*Testing shows the presence, not the absence of bugs*" (the computing pioneer and Turing Award winner Edsger W. Dijkstra at the 1969 NATO Rome conference) [1], and repeated a year later as "*Program testing can be used to show the presence of bugs, but never to show their absence!*" [2, Section 3, *On The Reliability of Mechanisms*]. If you give a computer program an input or series of inputs on which it fails, you will see it fail. If you fail to give it an input on which it fails, you will not see it fail. It follows that seeing a program past tests, that is, seeing the program not fail, cannot tell you that it does not fail on some input of interest, for you might not have given it that input. This is well-known to testers, of course, who speak of "coverage" in their test suites. Ideally, if there is an input on which the program fails, there is one in your test suite. Such ideals are not realisable in practical test suites.

Bibliography

[1] Edsger W. Dijkstra, Contribution in J. N. Buxton and Brian Randell (eds.), Software Engineering Techniques, April 1970, Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969. Available from http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF

[2] Edsger W. Dijkstra, *Notes On Structured Programming*, Note EWD249, 1970

[3] B. Littlewood and L. Strigini, Validation of ultra-high-dependability for software-based systems, Communications of the ACM 36(11):69-80, 1993.