The Logic of Correctness in Software Engineering

Mark Priestley

Cavendish School of Computer Science,
University of Westminster,
London, OK
M.Priestley@wmin.ac.uk
http://users.wmin.ac.uk/∼priest

Abstract. This paper uses a framework drawn from work in the philosophy of science to characterize the concepts of program correctness that have been used in software engineering, and the contrasting methodological approaches of formal methods and testing. It is argued that software engineering has neglected performative accounts of software development in favour of those inspired by formal logic.

1 Introduction

The notion of correctness is of great importance to the users of software who, it can be assumed, want software which produces the 'right' results. It has not proved straightforward to make the informal notion of correctness more concrete, however, and a variety of definitions have been used since the 1950s.

The way that correctness is defined has implications for the techniques that are adopted in software development. In a survey of the software engineering literature [22], Stuart Shapiro identified two major approaches to ensuring the correctness of software, one based on the notion of testing completed programs, and the other hoping to guarantee the correctness of software by using formal techniques based on mathematics and logic during the development process.

This paper analyses a number of definitions of software correctness, and examines their different implications for the techniques of proof and testing. The various approaches are characterized using a framework taken from philosophical accounts of scientific methodology.

2 Checking computations

It was not obvious how the results produced by the first electronic computers should be checked. Computers enabled calculations to be carried out that could not be performed by hand, so there was no possibility of a direct manual check of the results produced. At a conference held in Cambridge in 1949 [23], a number of papers explicitly considered the problems involved in checking the calculations performed by automatic computers. J. C. P. Miller [17] identified a number

of possible sources of error in computations: errors made in the mathematical solution to the problem being addressed, errors arising in the coding of the problem, and errors made by the machine when executing the program.

Mathematical sources of errors, although significant, were familiar from the pre-electronic days of computation, and errors introduced by malfunctioning electronic circuitry raised issues of reliability which were relatively easily dealt with by traditional means. Coding, or programming, errors, on the other hand, were a new category of error, and turned out to be more significant than expected. In 1949 Miller wrote that such errors could be "expected, in time, to become infrequent" [17, p. 123], but in 1951, in the first programming textbook to be published, Maurice Wilkes and his colleagues at Cambridge reported that "such mistakes are much more difficult to avoid than might be expected" [24, p. 38].

Programming errors are design errors: unlike hardware errors, they are not caused by mechanical or electronic failure, and so cannot be removed by increasing the reliability of any device. A variety of techniques for preventing such errors were considered, including the inspection of programs to reveal common mistakes, the inclusion of additional code to check the results being obtained, and the automation of the programming process itself. Another technique that was soon adopted was the use of library subroutines: these contain reusable code to perform various common tasks, and because of their frequent use were found more likely to be free from errors than new code.

It is noteworthy that there is no mention at this time of the modern notion of testing, which involves repeatedly executing the program with particular data values for which the expected results are known. Wilkes refers to the amount of machine time that could be lost running erroneous programs; as machine time was in this period a scarce resource, testing was simply too expensive and it was natural to emphasize techniques of error avoidance.

At this time, then, the notion of correctness was applied to the computations carried out by a machine. Correct computations are those which produce correct results, although it may not always be easy to tell which these are: "It cannot therefore be assumed that if a program apparently operates correctly it is giving correct results, and careful numerical checks must always be applied" [24, p. 41]. The coding process emerged as a significant source of errors, and the desirability of reducing the number of coding errors was recognized. Correctness was recognized to be a product of many factors, however, including the algorithm used, the coding of it for a particular machine, any library routines utilized, and the physical machine itself.

3 Correctness and proof

The development of high-level programming languages, and in particular the publication of the Algol 60 report, made explicit a new concept of a computer program: from this point on it became natural to think of programs as expres-

sions in a formal language, rather than as instructions given to a particular machine to perform some computation.

Initially, there was an expectation that the use of high-level languages like FORTRAN and Algol would get rid of programming errors. This turned out to be an illusion: one early programmer recalls that "[t]he only place where we made a mistake ... was believing that when FORTRAN came along we wouldn't make any mistakes in coding", and cites a survey which indicated that FORTRAN programs typically had to be compiled up to 50 times before they were correct [4]. During the 1960s it was widely believed that the failure to be able reliably to produce correct software was causing severe problems for the software industry.

In 1965, Edsgar Dijkstra gave a talk entitled "Programming considered as a human activity" [8]. This talk addressed the issue of the correctness of programs from the perspective of an individual programmer, and asked how programmers can have confidence that the results of a program are the results that were intended, particularly in cases where the results cannot be directly checked.

Dijkstra considered and rejected approaches to correctness based on testing, a position, encapsulated in his famous remark that "[p]rogram testing can be used to show the presence of bugs, but never to show their absence!" [10, p. 6]. The argument hinged on the fact that for all but the most trivial programs, the tests carried out would exercise only a small subset of the program's possible behaviours, and the observation that the successful execution of a set of tests implied nothing about the program's behaviour in untested cases.

Dijkstra's alternative approach to the problem of correctness was to make a connection between the new activity of programming and the established canons of mathematical proof. He is careful to state that mathematical proofs are not guarantees of the truth of the results proved. Nevertheless, the methodology of proof is taken to be the best available model of how the correctness of assertions can be assured, and Dijkstra's plan is to apply the lessons learned from proof to the development of programs in the attempt to increase our epistemological confidence in the results of programs.

But how is this analogy to be exploited? Dijkstra spells out his intended analogy in detail:

> The analogy between proof construction and program construction is, again, striking. In both cases the available starting points are given (axioms and existing theory versus primitives and available library programs); in both cases the goal is given (the theorem to be proved versus the desired performance); in both cases the complexity is tackled by division into parts (lemmas versus subprograms and procedures). [8]

When worked out in detail, the appeal to mathematical standards of rigour in connection with programming proved to be highly attractive. An important aspect of this derives from the understanding, established by the Algol 60 report, that programs could be thought of as expressions in a formal language. For Dijkstra, the process by which a programmer becomes assured of the correctness of a program is text-based: a programmer should examine the source code of a

program, and arrive at a conviction of what the program is doing in much the same way as a mathematician reads a proof and comes to accept the truth of the result that is proved.

As a result of these and similar arguments, correctness became understood not as a property of individual computations, but as a property of source code, of programs understood as expressions in a formal language. Concurrently with this change, the activity of testing was given less importance in methodological discussions.

Refinement: a proof theory for programs

Initial attempts at applying standards of proof to programming soon made it apparent that it was rather difficult to argue mathematically about the correctness of existing programs: the arguments needed to prove the correctness even of very trivial programs proved to be long and tedious.

In 1968, Dijkstra published a paper entitled 'A constructive approach to the problem of program correctness' [9]. Rather than proving the correctness of an existing program, constructive approaches aimed to develop programs in such a way that their correctness would be an automatic consequence of the development method used. The idea of the constructive approach is to develop a program and its proof simultaneously. If this is done, programmers can hope to end up with a program that has been proved to be correct, and thus avoid the need to verify existing code retrospectively.

The constructive approach was brought to wider notice in 1971 by Niklaus Wirth [25] in a paper which introduced the terminology of *stepwise refinement*: 'refinement' refers to the process by which an initial version of a program, written in such as simple way as to be 'obviously' correct, could be transformed into a more concrete version, and the label was chosen to emphasize the iterative, step-by-step nature of the development process that was proposed.

The notion on refinement completes the analogy between traditional logic and programming are here rather striking. Classical logic is concerned with truth: the cardinal semantic property of sentences in logic is their truth value, and the most important relationship between sentences in proof theory, namely inference, is explained as being a relation that preserves truth. The notion of truth does not apply straightforwardly to programs, of course, as they are in the imperative rather than the indicative mood. However, Dijkstra and Wirth view the correctness of a program as its most important semantic property, and in developing the constructive approach, refinement emerges as a relationship between programs which preserves correctness. A development using stepwise refinement can therefore be seen as formally the equivalent of a proof where, rather than deriving logical consequences from a set of hopefully self-evident axioms, we derive fully implemented programs from initial versions which are assumed to be, or are 'obviously', correct.

Thus the methodology of stepwise refinement provides a realization of Dijkstra's ideal of bringing mathematical standards of rigour to programming by defining a way in which the activity of programming can itself be viewed as

a proof-like process. As with mathematics, the 'proofs' can be carried out with greater or lesser degrees of formality, but in either case the approach derives from the understanding of programs as linguistic entities to which the basic notion of correctness can be applied.

4 Program specifications

Stepwise refinement places the emphasis in development on program texts, beginning with an 'obviously correct' version of a program. This judgment is dependent on some understanding of what the program is meant to achieve, however, and this insight became formalized in the notion of a program's *specification*. A typical statement of the role of the specification is the following:

> To determine whether a program is correct, we must have some way of specifying what it is intended to do; we cannot speak of the correctness of a program in isolation, but only of its correctness with respect to some specifications. After all, even an incorrect program performs *some* computation correctly, but not the same computation that the programmer had in mind. [16, p. 201]

In this model, the production of a requirements specification is a crucial first step in the process of developing a software system. In 1976, Boehm made the link between specification and correctness explicit: without a "good requirements specification ... testing is impossible, because there is nothing to test against" [5, p. 1227]. According to this view, then, there are two distinct types of entity relevant to correctness, namely specifications and programs. A program is said to be *correct with respect to a specification* if its behaviour is in accordance with the requirements defined in the specification.

Many process have been described for bridging the gap between specification and program, using a wide variety of design notations, both textual and graphical. The relational view of program correctness can therefore be seen as implying a generalization of the refinement relation discussed in Section 3 where the terms related can be expressions in a range of different notations. In principle, however, the different notations used are intertranslatable, or at least can be understood using a uniform semantic model, so the essential feature of refinement as a correctness preserving relation is maintained.

In both cases, software development can be thought of as an analogue of deduction and individual software developments as analogues of theorems. The 'axioms' in the theory are the specification; from the specification other expressions are 'deduced', by correctness preserving refinement steps, culminating in the production of 'conclusions' in the form of executable programs in the target programming language. The relationship between expressions in a software development is not deduction, but the formal structure is identical to a logical theory: this can be expressed by saying that software engineering views software development as being a *quasi-deductive activity*.

Viewing software development as quasi-deductive seems to leave little place for the activity of testing, however. The relationship between these two approaches can best be understood by drawing on accounts of scientific methodology developed in the philosophy of science, which relate the notions of theory and experiment. In the next section these ideas will be applied to software development, using a typology developed by Imre Lakatos.

5 Lakatos's typology

Lakatos [15] models scientific and mathematical theories as deductive systems which relate axioms to 'basic statements': these are the 'final conclusions' drawn by the theory. In scientific theories the basic statements are those which make some testable, empirical assertion. Applying this terminology to the quasi-deductive systems used in software engineering, the basic statements would be the executable programs which are the end points of a software development. These 'confront reality' by being run and tested.

Lakatos identifies two basic types of theory, which he terms 'Euclidean' and 'quasi-empirical'. These are distinguished by the place where truth values are 'injected' into the system. Euclidean theories inject truth at the top, by assuming the existence of true axioms and deducing valid conclusions from them; quasi-empirical theories inject falsity at the bottom, by testing the basic statements. A failed test forces some modifications at 'higher' points in the theory.

Software engineering thinks it is Euclidean

It is implicit in much of what has been said in previous sections that the standard view of software engineering is that it is a Euclidean system, in Lakatos' sense. Correctness is injected at the top of the system, in the form of a fixed specification of what the system is to do. The task of software development is understood to be that of systematically developing from this a program which implements the specified functionality, by means of a number of development steps which preserve correctness, in the sense that the consequent of each step is consistent with the given specification. The hope is that the success of this programme would make the activity of testing obsolete:

> I should like to point out that the constructive approach to program correctness sheds some new light on the debugging problem. Personally I cannot refrain from feeling that many debugging aids that are en vogue now are invented as a compensation for the shortcomings of a programming technique that will be denounced as obsolete in the near future. [9, p. 185]

In practice, however, software engineering practice does not follow a rigorous Euclidean methodology, and testing still plays a central role. Even advocates of formal methods do not recommend that formally developed programs be put

into service without a period of operational testing. In part, this stems from a recognition that current Euclidean software technology and techniques are fallible, in that development steps are not guaranteed to preserve correctness.

A pragmatic view of this situation, such as the one taken by Shapiro [22], would be that the use of formal methods and testing on the same project simply represents a belt and braces approach to verification, based on the observation that neither technique on its own is sufficient to guarantee correctness. Proponents of an Euclidean approach, however, have articulated a distinctive view of the role of testing in these methods: its purpose is not to assess directly the correctness of the software, but rather, by checking consistency between the specification and the program, to assess whether the development process has been correctly carried out. The recommended response to a failed test is not to correct the final artefact, but rather to modify and make less fallible the process that led to it [14].

Although this position appears to give a role to testing in software engineering, it is firmly rooted in a belief in the possibility, if not practicability, of Euclidean methods which guarantee the correctness of programs, and failed tests are not given the role that they would have in a quasi-empirical theory.

An inductive view of testing

By contrast with this, the traditional view of testing was that programmers should keep running, testing and modifying a program until it passes all its tests. A passed test represents an injection of *correctness* at the bottom of the system, a confirmation that the program was behaving as required. As Lakatos points out, the belief that correctness can be injected at the bottom of a deductive system is tantamount to a belief in inductive methods, and the comparison between induction and the traditional account of testing has been made in the software engineering literature. The thought is that successful tests are singular statements of a program's correctness; from a set of such statements, we want to be able to infer that the program as a whole will give correct results at all times in the future.

Although this belief underlies much informal, small-scale programming practice, positive statements of an inductive principle are rare in the software engineering literature, no doubt because of the prominence of Dijkstra's early attack on the position, quoted above. One notable attempt was made by Goodenough and Gerhart to characterize the 'logic of testing'; they proved a 'fundamental theorem of testing' which "states that in some cases, a test *is* a proof of correctness" [12, p. 157]. The idea underlying this theorem was to partition the input space of a program in such a way that the successful completion of one test would imply that the program would function correctly for all other inputs from a given partition. This attempt foundered, however, on the impossibility in practice of finding a partition of the testing space with the required formal properties.

Quasi-empirical software development

An alternative view of software engineering would characterize it as a quasi-empirical discipline, in which failed tests represent injections of *incorrectness* at the bottom of the quasi-deductive system. This has suggested to a number of writers an analogy between the testing of programs and the refutation of scientific theories: for example, Fetzer writes that "it might be said that that programs are conjectures, while executions are attempted—and all too frequently successful—refutations (in the spirit of Popper)" [11, p. 1062], and Dasgupta has articulated the thesis that design problem solving, which includes the design of programs, "is a special instance of (and is indistinguishable from) the process of scientific discovery" [7, p. 353].

There is a significant body of work in software engineering research which adopts a broadly quasi-empirical approach. One salient characteristic of this tradition is that it accepts the inherent fallibility of software. In 1971, Bauer wrote in an overview of the then young field of software engineering that the aim of the discipline was "to obtain economically software that is reliable and works efficiently on real machines" [2]. It is noteworthy that Bauer refers not to the correctness of the software, but rather its reliability; a later paper surveying approaches to the study of reliability made the point explicitly: "Our position is that it is neither necessary nor economically feasible to get 100 per cent reliable (totally error-free) software in large, complex systems" [21, p. 105]. Rather than trying to ensure the absolute correctness of software, software engineers who accept the inevitability of errors in software have been concerned with techniques for developing fault-tolerant systems and for statistical characterizations of the reliability of software [20].

A further characteristic that we might expect to find in quasi-empirical software engineering is 'bold hypotheses, followed by dramatic refutations', as in Popperian rhetoric about fallible science. Much current practice can in fact be interpreted in this way. For example, many writers have commented on the fact that software products are full of errors, and rapidly evolve with subsequent versions correcting some of the faults in earlier ones and also adding new functionality, a situation familiar to users of many commercial software packages. Traditional software engineering views this as a problem, feeling that a mature engineering profession ought to be able to do better. It is precisely what would be expected, however, if software engineering was in fact a quasi-empirical discipline. [1]

[1] In passing, it is worth noting that a recent school in software engineering has characterized itself as 'empirical', and is based on the belief that "the most important thing to understand is the relationship between various process characteristics and product characteristics" [1]. In terms of the characterization proposed in this paper, this school would seem to fall squarely in the Euclidean tradition, but to emphasize the external, managerial aspects of the development process rather than the internal properties of software-related artefacts. What is being proposed appears to be an empirical study of a Euclidean process, not an empirical approach to development itself.

6 Correctness and the user

In Euclidean approaches, failed tests must be interpreted as indicating a problem in the development process, on the assumption that correctness has been injection at the start of the process in the form of the specification. However, the very possibility of such a correctness-injection can also be questioned, as it was for example by Douglas Ross in 1968:

> The most deadly thing in software is the concept, which almost universally seems to be followed, that you are going to specify what you are going to do, and then do it. And that is where most of our troubles come from. The projects that are called successful, have met their specifications. But those specifications were based upon the designers' ignorance before they started the job. [18, p. 32]

Implicit in this is a distinct definition of correctness: it is not adequate to view correctness as a relationship between a program and its specification, but rather it takes the form of a relationship between a program and its users.

These two notions of correctness are widely discussed in the software engineering literature, and terminology has been introduced to describe the process of checking the correctness of a program in both senses. The process of checking that a program meets its specification is known as *verification* whereas the process of checking that a software engineering artefact—either specification or program—meets the actual requirements of its users is known as *validation*.

In a quasi-empirical approach, failed tests can be taken as evidence of errors in process, but more fundamentally they can indicate problems with the specification. Quasi-empirical software development would be a process which involved the future users of a program, as the response to a failed test might routinely involve revisions to the specification established with regard to the users' requirements.

Such approaches have been consistently recommended in the history of software engineering, in a stream of work running parallel to the main, Euclidean tradition. Examples include approaches involving ideas such as 'evolutionary prototyping' and more recently 'extreme programming' [3], which emphasize the ongoing involvement of users throughout a development.

Recent work in the philosophy of science has described a model in which scientific knowledge is not articulated as part of a deductive structure, but rather emerges in the course of an unpredictable process in which scientists explore the resistances provided by non-human actors. Examples of this approach include the actor network [6] and Pickering's notion of the 'mangle' [19]. It appears that this work can be directly applied to characterize certain approaches to software engineering.

7 Conclusions

This paper has argued that software engineering methodologies can be located and understood using ideas developed in the philosophy of science. In the broad-

est terms, philosophers have drawn a distinction between science as representation and science as performance [13], and argued that the performative aspect of science deserves much more attention that it has been given. There are strong echoes of this distinction in the two approaches to software development described above. Euclidean approaches emphasize the specification, a representation of the users' requirements, as a starting point, whereas quasi-empirical approaches emphasize the role of experimentation and prototyping as a way of uncovering the users' needs.

The Euclidean model gained prominence as a response to the perceived 'software crisis' in the 1960s, but it has not resolved all the problems with software development. In contrast, much software seems to evolve through multiple versions as it is 'mangled' by a combination of users' needs and developers' ambitions, and while not error-free, is widely and successfully used. Perhaps it is time for software engineering to pay more attention to the performative idiom, as providing a more realistic account of the process of software development.

References

1. Basili, V.: The role of experimentation in software engineering: past, present and future. *Proc. ICSE* (1996).
2. Bauer, F.L.: Software Engineering. In *Proceedings of the IFIP Congress 71*, ed. Friedman, C.V., North Holland (1972), 530–538.
3. Beck, K.: *Extreme Programming Explained*. Addison-Wesley (2000).
4. Bemer, R.W.: Computing prior to FORTRAN. Annals of the History of Computing **6(1)** 16–18 (1984).
5. Boehm, B.: Software Engineering. IEEE Trans. on Computers **C-25(12)** (1976), 1226–1241.
6. Callon, M.: Society in the Making: The Study of Technology as a Tool for Sociological Analysis. In Bijker, W.E., Hughes, T.P., Pinch, T.J., *The Social Construction of Technological Systems*, MIT Press (1987), 83–103.
7. Dasgupta, S.: *Design Theory and Computer Science*. Cambridge University Press (1991).
8. Dijkstra, E.W.: Programming considered as a human actvity. in *Proceedings of the 1965 IFIP Congress*, North-Holland Publishing Co. (1965) 213–217.
9. Dijkstra, E.W.: A constructive approach to the problem of program correctness. BIT **8** (1968) 174–186.
10. Dijkstra, E.W.: Notes on structured programming. In *Structured Programming*, Academic Press (1972).
11. Fetzer, J.H.: Program verification: the very idea. Comm. ACM **31(9)** (1988), 1048–1063.
12. Goodenough, J.B., Gerhart, S.L.: Toward a theory of test data selection. IEEE Trans. on Soft. Eng. **SE-1(2)** (1975), 156–173.
13. Hacking, I.: *Representing and intervening*. Cambridge University Press (1983).
14. Hoare, C.A.R.: How did software get so reliable without proof? In Proc. FME'96, Springer-Verlag **LNCS 1051**, 576–580.
15. Lakatos, I.: A renaissance of empiricism in the recent philosophy of mathematics? In *Philosophical Papers Vol. II: Mathematics, science and epistemology* (eds. Worrall, J. and Currie, G.), Cambridge University Press (1978).

16. Manna, Z., Waldinger, R.: The logic of computer programming. IEEE Trans. on Soft. Eng. **SE-4(3)** (1978) 199–229.
17. Miller, J.C.P.: Remarks on checking. in [23], 123–124 (1949).
18. Naur, R., Randell, B.: *Software Engineering: Report on a conference sponsored by the NATO Science Committee. Garmisch, Germany, 7th to 11th October 1968.* Scientific Affairs Division, NATO (1969).
19. Pickering, A.: *The Mangle of Practice.* The University of Chicago Press (1995).
20. Randell, B.: Facing up to faults. The Computer Journal **43(2)** (2000), 95–106.
21. Schick, G.J., Wolverton, R.W.: An analysis of competing software reliability models. IEEE Trans. on Soft. Eng. **SE-4(2)** (1978), 104–120.
22. Shapiro, S.: Splitting the difference: The historical necessity of synthesis in software engineering. IEEE Ann. Hist. Comp. **19(1)** (1997), 20–54.
23. *Report of a Conference on High Speed Automatic Calculating Machines, 22-25 June 1949.* University Mathematical Laboratory, Cambridge (1950).
24. Wilkes, M.V., Wheeler, D.J., Gill, S.: *The Preparation of Programs for an Electronic Digital Computer.* Addison-Wesley (1951).
25. Wirth, N.: Program development by stepwise refinement. Comm. ACM **14(4)** (1968) 221–227.