Turing's 1949 paper in context

Cliff B Jones

School of Computing Science, Newcastle University, United Kingdom

Abstract. Anyone who has written one knows how frustratingly difficult it can be to perfect a computer program. Some of the founding fathers of computing set out ideas for reasoning about software — one would say today 'techniques for proving that a program satisfies its specification'. Alan Turing presented a paper entitled *Checking a Large Routine* that laid out a workable method for reasoning about programs. Sadly his paper had little impact. Understanding the problem faced, Turing's proposal and what followed provides insight into how ideas evolve. Comparing three contributions from the 1940s with the current state of the art clarifies a problem that still costs society a fortune each year.

1 Introduction

In the 1940s, there were at least three published outlines of how it might be possible to reason about computer programs; Herman Goldstine and John von Neumann [GvN47], Alan Turing [Tur49] and Haskell Curry [Cur49]. Frustratingly these early insights appear to have then lain dormant only to be reinvented (and developed enormously) starting two decades later.[1] It is interesting to speculate why the early papers were only studied after the work of Bob Floyd, Tony Hoare and others had been published.

It is Alan Turing's 1949 paper that has the clearest similarity with what followed, for example in Floyd's [Flo67], and Section 3 gives enough detail of Turing's 1949 paper to support comparisons. Here, briefer comments are made on the other two contributions.

The report [GvN47] contains a long account of the process of designing programs that achieve a mathematical task. Looked at today, it is even tempting to say that the account is somewhat rambling but it must be remembered that in 1947 no one had any experience of writing computer programs. A distinction is made between parameters to a program which are likened to *free variables* and those locations whose values change during a computation that are compared to *bound variables*. The notion of constructing a flowchart is, from today's standpoint, handled in a rather pedestrian description. However, crucially, a distinction is made between *operation boxes* (that change the values of bound variables) and *assertion boxes* that can be used to describe logical relations between values.

[1] Connections other than the citations between these three pioneering pieces of work are difficult to trace. The current author has tried to determine whether there is a direct link between von Neumann's assertion boxes and Turing's annotations. It is known that Turing visited the US during the war but there is no evidence that the urgent business of cryptography left any time to discuss program development ideas.

Haskell Curry's [Cur49] cites, and is clearly influenced by, [GvN47]. Curry however puts more emphasis on constructing programs from components or *transformations*. In a subsequent paper from 1950, Curry applies his proposals to constructing a program for *inverse interpolation*. Curry's emphasis on topics such as (homomorphic) transformations is not surprising given his background in combinatory logic.[2]

2 The problem

Users of computers have to endure the situation that most software has bugs; those who purchase software are frustrated by the fact that it comes, not only without guarantees, but with explicit exclusions of any liabilities on the provider for losses incurred by the purchaser. It was claimed in a 2002 NIST report that software 'maintenance' cost US industry over $50Bn per year.

The problem for the programmer is the literal nature of the indefatigable servant called hardware. If one told a human servant to do anything nonsensical, there is at least a chance that the instruction would be queried. If, however, a computer program is written that continues to subtract one from a variable until it reaches zero, it will do just that — and starting the variable with a negative value is unlikely to yield a useful result.

A simplified expression of the question that has occupied many years of research –and which is of vital importance– is 'how can one be sure that a program is correct?'. In fact, this form of the question is imprecise. A better formulation is 'how can we be sure that a program satisfies an agreed specification?'.

Once one is clear that this property must apply to 'all possible inputs', a natural idea is to look at mathematical proof. The position argued in [Jon03] is that the desirability of reasoning about correctness was clear to the early pioneers and a key impediment to adoption has been a search for tractable methods of decomposing the reasoning so as to deal with large programs.

3 The paper *Checking a Large Routine*

Alan Turing made seminal contributions related to software. In fact, it can be argued that 'Turing Machines' provide the first thought-through idea of what constitutes 'software'. Turing's way of showing that the *Entscheidungsproblem* was unsolvable was to propose an imaginary universal computer and then to prove that there were results which no program could compute. The Turing machine language was minimal but just rich enough to describe any step-by-step process that could be envisaged.

[2] Acknowledgements to Curry's influence are present in both the choice of name for the Haskell (functional) programming language and the term *Curry-Howard correspondence* (for proofs as programs).

In 1949 the EDSAC computer in Cambridge became the world's second 'electronic stored program computer' to execute a program.[3] A conference to mark this event was held in Cambridge from 22–25 June 1949. Many people who became famous in the early history of European computing attended the event; among those who gave papers was Alan Turing. His paper [Tur49] is remarkable in several ways. Firstly, it is only three (foolscap) pages long. Secondly, much of the first page is taken up with one of the best motivations ever given for program verification ideas. Most interestingly, Turing presents the germ of an idea that was to lay dormant for almost 20 years; a comparison with Bob Floyd's 1967 paper –which became the seed of an important area of modern computer science research– is made below.

It is worth beginning, as does the 1949 paper, with motivation. The overall task of proving that a program satisfies its specification is, like most mathematical theorems, in need of decomposition. Turing made the point that checking the addition of a long series of numbers is a monolithic task that can be split into separate sub-tasks by recording the carry digits. His paper displays five (four-digit) numbers whose sum is to be computed and he makes the point that checking the four columns can be conducted separately if the carry digits are recorded (and a final addition including carries is another separable task). Thus the overall check can be decomposed into five independent tasks that could even be conducted in parallel.

Turing's insight was that anotating the flow chart of a program with claims that should be true at each point in the execution can also break up the task of recording an argument that the complete program satisfies its specification (a claim written at the exit point of the flowchart).

A program correctness argument is in several respects more difficult than the arithmetic addition: these, and how the 1949 paper tackled them, are considered one by one. Turing's example program computes factorial ($n!$); it was presented as a flowchart in which elementary assignments to variables were written in boxes; the sequential execution of two statements was indicated by linking the boxes with a (directed) line. In the original, test instructions were also written in rectangular boxes (they are enclosed diamond shaped boxes in Fig. 1) with the outgoing lines indicating the results of the tests and thus the dynamic choice of the next instruction.

Suppose a 'decorating claim' is that the values of the variables are such that $r < n$ and $u = (r + 1) * r!$. Consider the effect if the next assignment to be executed changes r to have a value one greater than the previous value of that same variable (in some programming languages this would be written $r := r+1$). Then it is easy to check that after this assignment a valid decoration is $r \leq n$ and $u = r!$. Reasoning about tests is similar. Suppose that the decorating assertion before a test is $s - 1 \leq r < n$ and $u = s * r!$; if the execution of the test indicates that the current values of the variables are such that $(s - 1) \geq r$ then, on the

[3] The world's first embodiment of an 'electronic stored-program computer' to run was the Manchester 'Baby' that executed its first program on midsummer's day 1948.

positive path out of the test, a valid decoration is $r < n$ and $u = (r+1) * r!$ (which expression can be recognised from above).

The flowchart in the 1949 paper represented what today would be written as assignments (e.g. $r := r+1$ from above) by $r' = r+1$. Furthermore, Turing chose to mark where decorating claims applied by a letter in a circle and to record the decorations in a separate table. There is the additional historical complication that his decorations were associated with numerical machine addresses. For these reasons –and those of space– the actual 1949 figures are not given here but exactly the same annotated program is presented in Fig. 1 in a modern form.

In today's terms, it could be said that Turing's programming language was so restricted that the meaning (semantics) of its constructs was obvious. This point is also addressed when subsequent developments are described below.

By decorating flowcharts, Turing –and some subsequent researchers– finessed some delicate issues about loops which are just represented by the layout of the flowchart.[4] In the case of Turing's example, factorial is computed by successive multiplication; and, in fact, the envisaged machine was so limited that even multiplication was not in the instruction set and the actual program has a nested inner loop that computes multiplication by successive addition.

There is a delicate and important issue about loops that Turing addressed formally: the need to argue about their termination. When a computer locks up (and probably has to be restarted) one possible cause is that a program is in a loop that never terminates. The 1949 paper contains a suggestion that loop termination can be justified by showing the reduction of an ordinal number which Turing commented would be a natural argument for a mathematician; he added however that a 'less highbrow' argument could use the fact that the largest number representable on the machine he was considering was 2^{40}.

The fascinating thing about the 1949 paper was the early recognition of the need for something more mathematical than the execution of test cases to support the claim that a program satisfies its specification. Of course, the example given in a (very short) paper is small but it is clear from the title of the paper that the intention was to apply the proposal to 'large routines'.

Turing gave a workable method for recording such reasoning. Lockwood Morris and the current author had a debate with Maurice Wilkes about the the comparison with Floyd's approach but it should be clear that a major intellectual step had been made beyond the acceptance that 'write then debug' was the only way to achieve correct software.

[4] In fact, some form of looping concept is central to the power of general purpose programs. One could say that they make it possible to compute properties that are not in the basic instructions of the programming language. This point is illustrated with recursive functions in [Pét66, Chap 1] and applied to interesting data structures in the appendix of Rózsa Péter's book. It is interesting to recall Tony Hoare's comment that he couldn't write Quicksort as a program until he learned a language that provided recursive procedures.

4 The Floyd/Hoare approach

Turing's wonderfully brief and clear 1949 paper both identifies the issue of reasoning about programs and contains a clear proposal as to how it might be undertaken for numerical examples. Considering this early recognition of the issue, it is remarkable that the key paper on which so much subsequent research on program reasoning is based did not appear until the late 1960s. The landmark talk by Bob Floyd in 1967 (published as [Flo67]) appears to have been written in complete ignorance of Turing's 1949 paper. Some possible reasons for this are put forward in Section 6. The similarities might indicate the degree to which the idea of separating complex arguments into smaller steps is inevitable; the differences between Turing's and Floyd's approaches are more interesting.

Floyd also annotated flowcharts and Turing's factorial example can be presented in Floyd's style as in Fig. 1 (the two deduction steps traced in the previous section appear in the lower part of this flowchart). For Turing's example, arithmetic relations suffice for the decorating assertions; Floyd explicitly moved to the formal language of first-order predicate calculus for his assertion language. This decision made it possible for Floyd to be precise about what constitutes a valid argument. In fact, Floyd explored what were later called 'healthiness conditions' by Edsger Dijkstra.

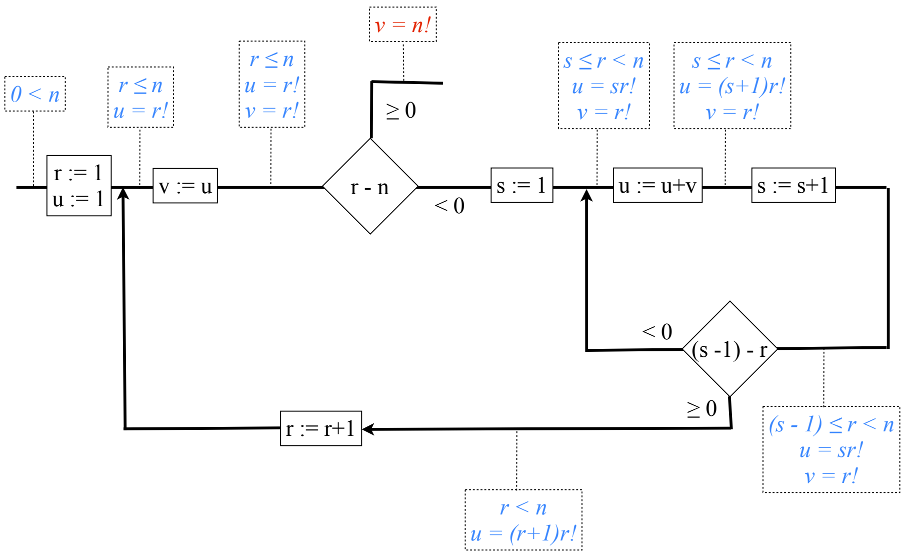

Fig. 1. A presentation of Turing's example in the Floyd style

Like Turing, Floyd offered formal ways of reasoning about termination. Not all later authors achieved this. The idea of showing that a program computes some desirable result *if it terminates* is sometimes misnamed ‘partial correctness’; one can however take the view that, if a program is expected to terminate, this requirement is a part of its specification and needs to be justified.[5]

A crucial further step –still in ignorance of Turing’s 1949 paper– was made by Tony Hoare who proposed an ‘axiomatic’ view of program semantics. In [Hoa69], he separated program reasoning from the flowchart view by proposing –what have come to be called– ‘Hoare triples’ as terms in an extended logic containing pre conditions, program constructs and post conditions. Both pre and post conditions are logical expressions that characterise sets of states. Valid triples record that, for all states satisfying the pre condition, the execution of the program text will result in states satisfying the post condition. What is important about this change of viewpoint is that it moves the programmer away from thinking in terms of program tracing and prompts viewing programs and assertions as combined terms in an extended logical system. A further advantage became clear in a second paper [Hoa71] by Hoare. The title of *Proof of a Program: FIND* betrays the fact that Hoare initially wrote a paper to provide a *post facto* proof of the correctness of a non-trivial sorting program (of his own invention). A first version of this proof was sent to referees one of whom was the current author. In the absence of the sort of mechanical theorem provers that are in use today, it was difficult to build confidence in the long and detailed proof. Hoare realised this and rewrote the paper (but did not change the title) to embody a stepwise development of the program with layers of abstraction that were much easier to grasp and reason about. Hoare’s axiomatic approach became the foundation of a huge field of research on the formal design of software.

Hoare’s original paper did not formalise termination arguments; of course, he was aware of the importance of showing that programs always terminated in any state satisfying the pre condition but the topic was not included in the initial set of rules. Hoare reasoned about termination (and in fact the relationship to the initial states) separately and informally.

Because of its importance below, it is worth recording that Hoare was generous in his credits to earlier researchers mentioning Floyd, Naur[6] and van Wijngaarden (whose role is explored in Section 6).

[5] John McCarthy –who did much to promote formal methods– used the following imaginary scenario to highlight the need for termination arguments ‘an algorithm that might appear to ensure that someone becomes a millionaire: the person should walk along the street picking up any piece of paper — if it is a cheque made out to that person for one million dollars, take it to the bank; if not, discard the piece of paper and resume the process’.

[6] Peter Naur proposed in a paper in the journal *BIT* ‘general snapshots’ as a way of annotating a program text to reason about its correctness. Naur’s system was based on comments in a program and was less formal than Floyd’s but it is another indication that the idea of decomposing an argument about correctness had reached its moment in time.

5 Current situation

Research on reasoning about programs (and designs of complex hardware) has not only been a major topic for academic research, it is now an essential approach used by industry in cases where life or business is at risk.

Jim King was one of Floyd's students and built Effigy [Kin71] which was an early system in which programs could be annotated with assertions (using first-order predicate calculus as in Floyd's paper). Today, powerful theorem proving assistant software such as Isabelle, PVS and Coq use heuristics from research on Artificial Intelligence and greatly reduce the human effort in creating completely formal proofs. Several notations have been developed for specifying both overall systems and components that arise during design. Integrated systems that generate 'proof obligations' from design steps link to theorem proving assistants.

Perhaps the development that would cause the founding figures most surprise is today's emphasis on using abstract forms of data in specification and design. With the limited store sizes of the early machines, vectors –or perhaps multi-dimensional arrays– constituted the extent of data abstraction. Now computer software manipulates huge and interlinked data structures whose representation is itself a major design challenge. Fortunately, the ideas of data abstraction and reification are also handled in many modern design support systems.

The use of 'formal methods' for hardware design accelerated after Intel reportedly took a $475M loss because of an undetected design flaw in the Pentium chip. Many researchers emphasise a 'stack' of verified components from programs, through compilers to the hardware designs themselves. As mentioned, the earliest uses of formal methods were for 'safety critical' software in which errors could put lives at risk. In some ways, the more interesting development is that some organisations use formal methods even where there is no requirement to do so, the argument being that they provide a cost-effective way of creating predictable and maintainable software. A useful review of the current state of deployment of formal methods is [WLBF09] (which is currently being updated).

Even where completely formal (machine checked) proofs are not considered necessary, an approach sometimes called 'formal methods light' offers an engineering approach founded on mathematics.

One area of research that is of strong interest today concerns 'concurrency'. This is important because hardware designers are putting ever more processors on a single chip in order to continue to provide the exponential speed increases that have revolutionised computing. Of particular interest here is the development of special logics such as Temporal Logic or Separation Logic to reason about concurrency.

6 Why did Turing's paper not have immediate impact?

Reasoning about programs is today seen as the only way to ensure the correctness of so-called 'safety critical' software and as a cost-effective way of achieving high quality software in a predictable process. It is, therefore, interesting to try to understand the lack of impact of Turing's 1949 gem on what has become such an important area of computing science.

First, it is worth noting that Turing was not careful in the production of printed materials especially for what he might have seen as a rather minor contribution. The reproduced version of the 1949 paper is a nightmare to interpret.[7] The identifiers used in the program could hardly have been more badly chosen for someone with unclear handwriting and little time for proofreading: u, v, n and r are mistyped 10 times. This is compounded by the fact that Turing chose –rather than $n!$– to write factorial as a box around n; a perfectly acceptable notation but in the printed paper this symbol is missing in several places presumably because it should have been added by hand but was not.[8]

The 'typos' were, however, certainly not the main reason for the lack of impact. Neither Floyd nor Hoare were old enough to have attended the 1949 conference. It would also appear that few people who were at that meeting were inclined to follow Turing's rigorous approach to reasoning about software. There was one tantalising attendee: Aad van Wijngaarden is listed as a participant in the Cambridge meeting. He went on to publish an important paper [vW66] entitled *Numerical Analysis as an Independent Science*. The contribution of this paper was to provide axioms of finite computer arithmetic in which, for example, adding one to the largest number representable in a single word might yield a negative result. As mentioned above, Hoare acknowledges this contribution because it is one aspect of his 1969 'axiomatic' approach. In some sense then, van Wijngaarden had in his hands all of the pieces of Hoare's approach but failed to put them together.[9] The current author had the good fortune to know many of the main players (but not Turing). In particular, Aad van Wijngaarden was always a charming and interesting dinner companion; it is most unfortunate that Turing's 1949 paper only came to hand after van Wijngaarden's death. Neither first hand questions, nor recollections from closer colleagues have exposed any record of what van Wijngaarden thought of Turing's paper.[10]

Another consideration when looking for paths to impact is the language in which Turing's programs were written. Programs are inevitably written in a language. The intended meaning of that formal language (its 'semantics') is an essential assumption for any reasoning about programs. Typically, a programming language is translated into the machine code of a specific type of hardware. Were the translation not to reflect the intended semantics, any effort to reason about programs would be undermined. Fortunately this can be seen as decomposing the overall challenge of 'program correctness': ways are needed, on the

[7] A full review of Turing's 1949 paper with indications of the necessary typographical corrections is contained in [MJ84].

[8] For example: 'with n in line 29 we shall find a quantity in line 31 when the machine stops which is n'.

[9] More detail on the history of reasoning about programs can be found in [Jon03].

[10] Gerard Alberts has however made two important points (private communication March 2017): it is important to remember that van Wijngaarden's background was numerical analysis rather than logic; furthermore, Alberts has evidence that van Wijngaarden 'had little affinity' with Turing and his work.

one hand, for reasoning about programs written in a specific 'high level' language and, on the other hand, for showing that the translation of that language into machine code is correct. One suggestion for the lack of progress over almost two decades (1949–67) is that programmers were so preoccupied with the many other developments their attention was diverted from the crucial issue of whether programs satisfied their specifications. Those 18 years also saw the development from machine code to ('high level') programming languages in which large programs can be written.

Another possible explanation is one that still plagues the software industry: to mathematicians like Turing and von Neumann the notion of proof was bread and butter, but many who took up the role of programmer were not conscious of the certainty to be gained from presenting careful logical arguments.

It is tempting to speculate what might have happened had Turing's paper been more widely read and understood at an earlier point in time. The story about van Wijngaarden ought give pause to anyone who suggests that, had Turing's paper been more widely known, the subject of program reasoning would have been automatically advanced by two decades. Who knows when someone of Tony Hoare's disposition and ability would have come along to make the crucial step to an axiomatic presentation? It is interesting to note that Hoare was actually looking for ways to record the meaning of programming languages when he proposed his axiomatic approach; so there is the initial requirement that the sought after person might have needed to have struggled with the specific thorny question of how to describe the semantics of a language precisely but to leave some aspects 'under determined'.

A more compelling avenue for speculation is to wonder what influence awareness of Turing's paper might have had on Bob Floyd. Floyd's paper, like Hoare's, is generous in its acknowledgement of previous work. In fact, Don Knuth's obituary note suggests excessive modesty in Floyd's statement that 'These modes of proof of correctness and termination are not original; they are based on ideas of Perlis and Gorn, and may have had their earliest appearance in an unpublished paper by Gorn.'[11]

The facts are simple: Turing's [Tur49] is another example of a paper from a brilliant mind; the paper was ahead of its time; sadly, this contribution appears to have gone unrecognised until after Floyd's independent invention of a scheme that went beyond Turing's had been published and taken up by other researchers.

Ideas have their time and independent inventions are not unusual; one part of the question of timing is that receptiveness can depend on people having struggled with preparatory problems;[12] for many years Turing's contributions were somewhat undervalued (a state of affairs that has recently been handsomely redressed); however his *Checking a large routine* had regrettably little impact. It is also important not to go too far in explaining scientific progress as a sequence of landmark papers.

[11] Repeated attempts to track down this elusive paper have so far yielded nothing: [Gor61] does not appear to be the missing document and [Gor68] is too late.

[12] A similar conclusion is drawn in Priestly's interesting book [Pri11, p.302].

Acknowledgements

I should like to dedicate this paper to Lockwood Morris (1943–2014) who was a great but under-appreciated scientist. We worked together on [MJ84] when we were both in Oxford and he subsequently spent his sabbatical with me in Manchester. As well as many happy memories, Lockwood did me a great personal favour in December 2013 which was sadly the last time we met.

I am extremely grateful to Liesbeth de Mol for bringing Curry's work to my attention and to Gerard Alberts for interesting input on van Wijngaarden. Comments from anonymous referees have also helped improve the paper although some of their suggestions will not fit in the page ration. My funding for this research comes from the EPSRC *Strata* Platform Grant.

References

[Cur49] Haskell B Curry. On the composition of programs for automatic computing. *Naval Ordnance Laboratory Memorandum*, 9806(52):19–8, 1949.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[Gor61] Saul Gorn. Specification languages for mechanical languages and their processors a baker's dozen: a set of examples presented to asa x3. 4 subcommittee. *Communications of the ACM*, 4(12):532–542, 1961.

[Gor68] Saul Gorn. The identification of the computer and information sciences: their fundamental semiotic concepts and relationships. *Foundations of language*, pages 339–372, 1968.

[GvN47] Herman H. Goldstine and John von Neuman. Planning and coding of problems for an electronic computing instrument. Technical report, Institute of Advanced Studies, Princeton, 1947.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[Hoa71] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14:39–45, January 1971.

[Jon03] Cliff B. Jones. The early search for tractable ways of reasoning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.

[Kin71] J. C. King. A program verifier. In C. V. Freiman, editor, *Information Processing 71*, pages 234–249. North-Holland, 1971. Proceedings of IFIP'71.

[MJ84] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, 1984.

[Pét66] Rózsa Péter. *Recursive Functions.* Academic Press, 1966.

[Pri11] Mark Priestley. *A science of operations: machines, logic and the invention of programming.* Springer Science & Business Media, 2011.

[Tur49] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, Cambridge, June 1949.

[vW66] A. van Wijngaarden. Numerical analysis as an independent science. *BIT*, 6:66–81, 1966.

[WLBF09] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), Oct 2009.