COMPUTING SCIENCE

Turing and Software Verification

Cliff B. Jones

TECHNICAL REPORT SERIES

No. CS-TR-1441 December 2014

TECHNICAL REPORT SERIES

No. CS-TR-1441 **December, 2014**

Turing and Software Verification

Cliff B. Jones

Abstract

Modern society relies heavily on computer programs or 'software'. Nearly everyone is aware that software sometimes malfunctions – it has 'bugs'. In a little known paper from 1949, Alan Turing described a technique for Checking a Large Routine; this paper might have accelerated the development of the subject of reasoning about software by decades, but sadly this gem from Turing had little impact. The current paper sets out the challenge to which Turing was responding, compares his proposals to those that came in the late 1960s, reflects on the missing impact and offers a summary of the state of the art.

© 2014 Newcastle University.
Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

JONES, C.B.

Turing and Software Verification
[By] C. B. Jones

Newcastle upon Tyne: Newcastle University: Computing Science, 2014.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1441)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1441

Abstract

Modern society relies heavily on computer programs or 'software'. Nearly everyone is aware that software sometimes malfunctions – it has 'bugs'. In a little known paper from 1949, Alan Turing described a technique for Checking a Large Routine; this paper might have accelerated the development of the subject of reasoning about software by decades, but sadly this gem from Turing had little impact. The current paper sets out the challenge to which Turing was responding, compares his proposals to those that came in the late 1960s, reflects on the missing impact and offers a summary of the state of the art.

About the authors

Cliff Jones is Professor of Computing Science at Newcastle University. He is best known for his research into "formal methods" for the design and verification of computer systems; under this heading, current topics of research include concurrency, support systems and logics. He is also currently applying research on formal methods to wider issues of dependability. Running up to 2007 his major research involvement was the five university IRC on "Dependability of Computer-Based Systems" of which he was overall Project Director - this was followed by a *Platform Grant* "Trustworthy Ambient Systems" (TrAmS) (Cliff was PI - funding from EPSRC) and is now CI on TrAmS-2. He also coordinates the three work packages on methodology in the DEPLOY project (on which he is CI) and is PI on an EPSRC-funded AI4FM project.

As well as his academic career, Cliff has spent over twenty years in industry (which might explain why "applicability" is an issue in most of his research). His fifteen years in IBM saw, among other things, the creation -with colleagues in the Vienna Lab- of VDM which is one of the better known "formal methods". Under Tony Hoare, Cliff wrote his Oxford doctoral thesis in two years (and enjoyed the family atmosphere of Wolfson College). From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which -among other projects- was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the "mural" (Formal Method) Support Systems theorem proving assistant).

During his time at Manchester, Cliff had a 5-year *Senior Fellowship* from the research council and later spent a sabbatical at Cambridge for the whole of the Newton Institute event on "Semantics" (and there appreciated the hospitality of a Visiting Fellowship at Gonville & Caius College. Much of his research at this time focused on formal (compositional) development methods for concurrent systems.
In 1996 he moved to Harlequin, directing some fifty developers on Information Management projects and finally became overall Technical Director before leaving to re-join academia in 1999.

Suggested keywords

ALAN TURING
PROGRAM VERIFICATION
HISTORY

Turing and Software Verification[1]

CLIFF B. JONES

Modern society relies heavily on computer programs or 'software'. Nearly everyone is aware that software sometimes malfunctions – it has 'bugs'. In a little known paper from 1949[2], Alan Turing described a technique for *Checking a Large Routine*; this paper might have accelerated the development of the subject of reasoning about software by decades, but sadly this gem from Turing had little impact. The current paper sets out the challenge to which Turing was responding, compares his proposals to those that came in the late 1960s, reflects on the missing impact and offers a summary of the state of the art.

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* is unsolvable was to propose an imaginary universal computer and then to prove that there were results that 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. Anyone who has written one knows how frustratingly difficult it can be to perfect a computer program. Several of the founding fathers of computing recognized the difficulty and in early papers set out ideas for reasoning about software – today one would call them 'techniques for proving that a program satisfies its specification'. Turing presented and published a paper in 1949 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 a fascinating insight into how ideas evolve. Comparing this with the current state of the art can give an awareness of a problem that still costs society billions of dollars each year.

---

[1] This material was prepared as a chapter of a book on Alan Turing but the author lost patience with the way the book was (not) progressing and decided to archive his contribution as a Technical Report.

[2] *Checking a Large Routine*, in *Report of a Conference on High Speed Automatic Calculating Machines*. University Mathematical Laboratory, Cambridge, (June 1949) pages 67–69. A full review of this paper with an indication of necessary typographical corrections is cited in F. L. Morris and C. B. Jones. *An early program proof by Alan Turing*. Annals of the History of Computing, 6(2): 139–143, April 1984

What is the problem?

Users of computers suffer from the fact that most software has bugs; those who purchase commercial 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. In fact, it is reported that software 'maintenance' costs US industry tens of billions of dollars 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 simple version of the question that has occupied many years of research – and which can be of vital importance – is 'how can one be sure that a program is correct?' In fact, this form of the question is not precise enough: 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*. When we understand the most famous theorem attributed to Pythagoras, there is no need to consider example right-angled triangles and check the sums of the squares; the proof covers all possible examples.

When writing historical accounts, an author should beware using concepts that were established only later. However, before we turn to a historical account, one specific reservation about what can be achieved is best explained in modern terms. Programs are inevitably written in a language. The intended meaning of that language (its 'semantics') is an essential assumption for any reasoning about programs. Typically, a programming language is translated into the instructions of a specific type of hardware (*machine code*). Were the translation not to reflect the intended semantics, any effort to reason about programs would be undermined. Fortunately this problem can be seen as decomposing the overall challenge of 'program correctness'. Ways are needed, on the 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. (I return to the idea of going yet further into the correctness of the hardware design in the final section of this paper.)

Fortunately for the discussion of Alan Turing's proposal, the language of his programs is close to that of the hardware available at that time.

Turing’s idea

In 1949 the EDSAC computer in Cambridge became the world’s second[3] ‘electronic stored program computer’ and a conference to mark this event was held in Cambridge that June. Many people who became famous in the early history of European computing attended the event and among those who gave papers was Alan Turing.

Turing’s paper is remarkable from several points of view. 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 importantly, Turing presented the germ of an idea that was to lay dormant for almost 20 years but then became the seed of one of the most important aspects of modern computer science research.

Figure 1

	1	3	7	4
	5	9	0	6
	6	7	1	9
	4	3	3	7
	7	7	6	8
	3	9	7	4
2	2	1	3	
2	6	1	0	4

It is useful to begin, as his 1949 paper does, with the motivation. The overall task of proving that a program satisfies its specification is, like most mathematical theorems, in need of decomposition. Turing made the observation that checking the addition of a long series of, say, four-digit numbers (see Figure 1) is a monolithic task that can be split – by recording the

[3] The world’s first embodiment of an ‘electronic stored-program computer’ was the Manchester ‘Baby’ that executed its first program on midsummer’s day 1948. The Cambridge EDSAC machine achieved the same feat in 1949 but is often listed as the ‘first usable computer’. It is true that the Manchester ‘Baby’ had no input/output devices, but in 1949 it was succeeded by the ‘Mark I’ that was itself a usable machine. Both EDSAC and the Mark I went on to spawn useful industrial products and Manchester produced many subsequent designs that were also converted into industrial computers. For the history of how the UK’s lead in computer hardware was squandered, see John Hendry. *Innovating for Failure*. MIT Press, 1989. An excellent review of early UK computers (including Turing’s design of the ‘Ace’ computer) is contained in Chris Burton, Martin Campbell-Kelly, Roger Johnson, and Simon Lavington. *Alan Turing and His Contemporaries: Building the World’s First Computers*. BCS Learning & Development Limited, 2012.

carry digits – into separate mini-tasks. It is then possible to assign the checking of $4+6+9+7+8 = 10*3+4$ to a completely separate person from the one checking $7 + 0 + 1 + 3 + 6 = 10 * 1 + 7$. Thus there can be five independent tasks (including a final check that $3974 + 22130 = 26104$) that can even be conducted in parallel.

Turing's insight was that decorating the flowchart of a program with claims that should be true at each point in the execution can, as above, 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). There are several respects in which a program correctness argument is more subtle than the arithmetic case: these aspects, 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 Figure 2) with the outgoing lines indicating the results of the tests and thus the dynamic choice of the next instruction. Suppose that 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 that is one greater than the previous value of that same variable - in some programming languages this is 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 the 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 are such that $(s-1) \geq r$, then, on that path out of the test, a valid decoration is

$r < n$ and $u = (r + 1) * n!$

(which expression can be recognized from above).

The flowchart in the 1949 paper represented what today would be written as assignments (such as $r := r + 1$ 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 1949 figures are not given here, but Figure 2 presents, in a more modern form, exactly the same annotated program in the next section.

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. I address this point below where subsequent developments are described.

By decorating flowcharts, Turing – and some subsequent researchers – finessed some delicate issues about loops that are just represented by the layout of the flowchart. 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 mathematical properties that are not in the basic instructions of the language or machine. In the case of Turing's example, factorial is computed by successive multiplication; furthermore, 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 one delicate and important issue with loops that Turing did not duck and this is the need to argue about their termination. When a computer locks up (and probably has to be restarted) a common 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 (as 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 Turing's 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 his 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. There was some subsequent controversy about the generality of his approach (this is discussed in a later section) 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.

The Floyd/Hoare approach

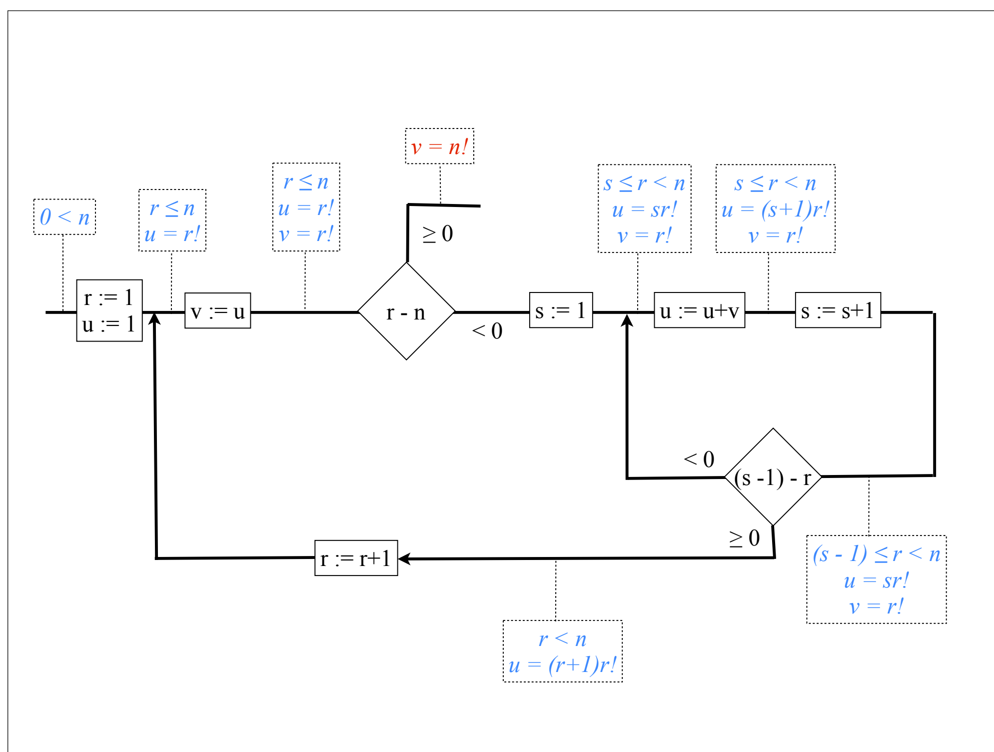
Turing's wonderfully brief and clear 1949 paper identifies the issue of reasoning about programs and also contains a clear way as to how it might be undertaken for simple examples. Given this early recognition of the issue, it is remarkable that the key paper on which so much subsequent research on program reasoning is based did not appear until the late 1960s. One possible explanation for this gap of almost two decades is that developers were so preoccupied with the many other developments that 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 far larger programs can be written. Another possible explanation still plagues the software industry today: to mathematicians like Turing and von Neumann the notion of proof was bread and butter, but many who took up the role of programming were not versed in the certainty to be gained from presenting careful logical arguments.

Whatever the reason, the landmark 1967 talk by Bob Floyd in the USA appears – as discussed in the next section – to have been written in complete ignorance of Turing's 1949 paper. 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 extremely interesting.

Floyd also annotates flowcharts and Turing's factorial example can be presented in Floyd's style as in Figure 2 (the two deduction steps traced in the previous section appear in the lower part of this flowchart):

Figure 2



For Turing's example, arithmetic expressions suffice for the decorating arguments that decompose the overall correctness argument; Floyd explicitly moved to a formal language used by logicians (known as *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' for inference rules.

Like Turing, Floyd offered formal ways of reasoning about termination. Not all later authors achieved this. 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 check 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'. 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 argued.

There was in fact another paper published in the same year as Floyd's: in a paper in the journal BIT, the Danish computer scientist Peter Naur proposed '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.

A crucial further step – still in ignorance of Turing's 1949 paper – was made by the British researcher Tony Hoare who proposed an 'axiomatic' view of program semantics. In his 1969 paper, he freed program reasoning from flowcharts by treating what are now 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 characterize states. Valid triples record that, for all states satisfying the pre condition, the execution of the program text results in states satisfying the post condition. This change of viewpoint moves the programmer away from thinking in terms of program tracing and prompts thinking of programs and assertions as combined terms in an extended logical system.

A further advantage became clear in a second paper by Hoare. Its title, *Proof of a Program: FIND*, betrays the fact that Hoare initially wrote a paper to prove the correctness of a non-trivial sorting program of his own invention. A first version of this proof was sent to referees. In the absence of the sort of mechanical theorem provers that are in use today, it was difficult to have confidence in the detailed proof. Hoare realized this and rewrote the paper (but not the title) to embody a step-wise 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[4] did not formalize termination arguments. He was clearly aware of the importance of showing that programs always terminated in any state satisfying the pre condition but it was not included in the initial set of rules. Hoare reasoned about termination

---

[4] As well as the key papers by Hoare, specific material relating to his axiomatic approach can be found in C. A. R. Hoare and C. B. Jones, (eds), *Essays in Computing Science*. Prentice Hall International, 1989.

(and the relationship to the initial states) separately. Were a full history being given, the issue of termination would be picked up under the heading of 'weakest pre conditions' due to the Dutchman Edsger Dijkstra.[5]

Because of its importance in the next section, it is worth recording that Hoare was generous in his credits to earlier researchers, specifically mentioning Floyd, Naur and van Wijngaarden (whose role is explored in the next section).

Why was Turing's paper not taken up?

It is interesting to try to analyze the impact (or lack thereof) of Turing's 1949 paper on what has become one of the most important research areas in computing science. 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 tempting to speculate what might have happened had Turing's paper been more widely read and understood at an earlier point in time.

First, it is worth recording that Turing was not careful in the production of written materials, especially for what he might have seen as a rather passing contribution. The reproduced version of the 1949 paper is a nightmare to interpret. 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 to write a factorial as a box around *n* rather than *n!*; a perfectly acceptable notation, but in the printed paper it is missing in several places, presumably because it should have been added by hand but wasn't.

The 'typos' were however not the main reason for the paper's lack of impact. Neither Hoare nor Floyd was old enough to have attended the 1949 conference in Cambridge. It also appears that few people who were at that meeting were inclined to follow Turing's rigorous approach to reasoning about software. There was, however, one fascinating exception: the Dutch mathematician Aad van Wijngaarden is listed as a participant in the 1949 event at Cambridge. He went on to publish an important paper in 1966 entitled *Numerical Analysis as an Independent Science*. Its contribution was to provide axioms of finite computer arithmetic in which, for example, adding one to the largest number representable in a single value might yield a negative number: as mentioned above, Hoare acknowledges this contribution because

---

[5] More detail on the history of reasoning about programs can be found in 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

it is one component of his 1969 'axiomatic' approach. In some sense then, van Wijngaarden had all of the pieces of Hoare's approach in his hands but failed to put them together.[6]

This fact ought to give pause to anyone who might suggest that the subject of program reasoning would have been automatically advanced by two decades had Turing's paper been more widely known. Who knows when someone of Tony Hoare's disposition and ability would have come along? It is interesting to note that Hoare was actually looking for ways to record the meaning of programming languages when he proposed his axiomatic method; so there is the additional 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 issues 'under determined'.

A slightly more useful speculation is to wonder what influence an awareness of Turing's 1949 paper might have had on Floyd. Floyd's paper, like Hoare's, is also generous in its acknowledgement of previous work. In fact, there is almost excessive modesty in his 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*.

Together with Lockwood Morris, I corrected the typographical errors in Turing's 1949 paper, linked it to Floyd's contribution, and had it reprinted in the Annals of Computer History.[7] The renowned Professor Wilkes took exception to this and wrote to Tony Hoare, who then held a chair at the Programming Research Group of Oxford University where the Morris/Jones paper was written. Hoare of course showed Wilkes' letter to the offending authors: it included '*I would not like the idea to get around that Floyd's great idea had been anticipated by Turing*' and the rather unusual argument about priority '*His [Turing's] approach was, what a few years later, would have been described as the conventional one*'. Morris effectively put a stop

---

[6] I had the good fortune to know many of the main players (but not Turing). In particular, Aad van Wijngaarden was a charming and interesting dinner companion; it is unfortunate that Turing's 1949 paper came to hand only after van Wijngaarden's passing. Neither first-hand questions, nor recollections from even closer colleagues, have left any record of what van Wijngaarden thought of Turing's 1949 paper. Fortunately, Wilkes's objections to the Morris/Jones paper were in time for me to check that Bob Floyd did not feel that we were slighting one of his significant contributions.

[7] See footnote 2.

to what might have become an unseemly dispute when he replied: *'[Turing's paper]…is grippingly interesting and gracefully written ... the paper's light is rather hidden under a bushel by the unfortunate number of typographical errors. We have no intention of making or marring any reputations (not that Turing's or Floyd's could be in any danger from us).*'

The facts are simple: Turing's 1949 paper is another example of a paper from a brilliant mind; the paper was ahead of its time; sadly, this particular contribution appears to have gone unrecognized until after Floyd's independent invention of a scheme that went beyond Turing's had been published and taken up by other researchers.

My belief is that ideas have their time and that independent inventions are not unusual; part of the question of timing is that receptiveness can depend on people having struggled with preparatory problems, a suspicion that for many years Turing's contributions were somewhat undervalued (a state of affairs that has recently been handsomely redressed), and a reservation about over-emphasizing single contributions.

Where are we today?

The topic of reasoning about programs (and designs of complex hardware) has become both a major one for academic research and an essential approach used by industry in cases where life or business is at risk. Turing would probably have appreciated the developments.

Jim King was one of Floyd's students and built 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 PVS, Isabelle 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) were only just possible. 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 design support systems.

The use of 'formal methods' for hardware design was accelerated after Intel reportedly took a $475M loss because of a design flaw in the Pentium chip. This led many researchers to emphasize a 'stack' of verified components from programs, through compilers to the hardware designs themselves. As has been 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 organizations 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.

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 more processors on a single chip in order to continue to offer the exponential speed increases that have revolutionized computing since Turing’s time.