Minds and Machines (2024) 34:S73–S92
https://doi.org/10.1007/s11023-023-09621-5

Three Early Formal Approaches to the Verification of Concurrent Programs

Cliff B. Jones[1]

Received: 14 May 2022 / Accepted: 3 January 2023 / Published online: 23 January 2023

Abstract
This paper traces a relatively linear sequence of early research approaches to the formal verification of concurrent programs. It does so forwards and then backwards in time. After briefly outlining the context, the key insights from three distinct approaches from the 1970s are identified (Ashcroft/Manna, Ashcroft (solo) and Owicki). The main technical material in the paper focuses on a specific program taken from the last published of the three pieces of research (Susan Owicki's): her own verification of her *Findpos* example is outlined followed by attempts at verifying the same example using the earlier approaches. Reconsidering the prior approaches on the basis of Owicki's useful example illuminates similarities and differences between the proposals. Along the way, observations about interactions between researchers (and some "blind spots") are noted.

Keywords Program correctness · Formal methods · History · Concurrency

1 Introduction

Research in the late 1960s provided a firm basis for showing that **sequential** programs satisfied their specifications (key papers by Bob Floyd and Tony Hoare are briefly outlined in Sects. 1.1 and 4). Around 1970, the technical challenges of extending such formal approaches to tackle the verification of **concurrent** software came into focus. Contextually there was pressure from machines becoming large enough to run multiple processes, from the ability to link machines and—crucially—from applications. "Interference" is identified as a key technical challenge to reasoning about concurrency in Sect. 1.2.

The main purpose of the current paper is to review three early approaches to the verification of concurrent programs (joint work by Ed Ashcroft and Zohar Manna is

✉ Cliff B. Jones
cliff.jones@ncl.ac.uk

[1] School of Computing, Newcastle University, 1 Science Square, Newcastle NE4 5TG, UK

described in Sect. 2, a solo publication by Ashcroft in Sect. 3 and Susan Owicki's contribution in Sect. 5). There is no attempt in the current paper to identify "firsts". Extensive other work by, for example, Carl Adam Petri is not even referenced. The three chosen sources contain key insights on concurrency and represent progress in practicality. Readers might find the apparent linearity of these ideas suspicious; this point is addressed in the concluding section.

The objective is to convey—at least in outline—the technical contribution of each of the three approaches. This is attempted by bringing the three approaches to bear on a common example presented in Owicki's PhD thesis. For this reason, the initial descriptions in Sects. 2 and 3 are brief and are revisited in Sects. 6.1 and 6.2.

One issue which dogs all three chosen approaches is "atomicity" and this is discussed in Sect. 7. Conclusions are sketched in Sect. 8.

1.1 Prior Work on Sequential Programs

The reference points for formal verification of **sequential** (non-concurrent) programs are taken here as Floyd (1967) and Hoare (1969).[1] Common to both approaches is the use of **state assertions** that are predicates defining relationships between the values of variables. State assertions can be used as annotations on a program and their consistency can be checked against the meaning of the program. The assertion at the end of the program is a "post condition" describing the overall effect of the program; few programs work for all possible inputs and the assertion at the beginning of the program is its "pre condition".

Although strongly related, there is a crucial distinction between the approaches of Bob Floyd and Tony Hoare to the verification of sequential programs. The starting point for Floyd (1967) is a flowchart of a program to which "state assertions" are added that justify that any execution of the program will be in accord with the assertions.[2] As such, the approach to verification is "*post facto*": a program is written and only after its creation is it subjected to formal reasoning. Hoare (1969) contains a generous acknowledgement to the influence of Floyd (1967)[3] but Hoare's "axiomatic approach" offers a clear path to a development approach that is described in Sect. 4 below and can be labelled "posit and prove" (a designer posits a design decision that is shown to match a given specification; one development step is verified before moving on to the design of any sub-components).

The distinction in the preceding paragraph can also be made by applying the term "bottom up" to the *post facto* approach (in the sense that the assertions construct

[1] Earlier work by John von Neumann and Alan Turing is discussed in various papers including Jones (2017) and Priestley (2020).

[2] Floyd makes the modest comment that "These modes of proof of correctness and termination are not original, they are based on ideas of Perlis and Gorn, and may have made their earliest appearance in an unpublished paper by Gorn". The current author exchanged letters with Floyd after Maurice Wilkes had objected to the description in Morris and Jones (1984) of Turing's contribution in Turing (1949); unfortunately this did not uncover more about Floyd's—possibly overly generous—acknowledgement. Fuller details can be found in Jones (2003).

[3] In fact, Hoare's attempts to find an axiomatic approach can be traced in drafts [discussed in Jones and Misra (2021)] from which one can see the transformative effect of him studying (Floyd, 1967).

an abstraction above the relatively concrete code of the program) and, in contrast, development of a program from an abstract specification can be termed "top down".[4]

1.2 Challenges from Concurrency

There are, in fact, many challenges concerned with the development of concurrent programs. "Liveness" issues such as deadlocks are not addressed by the subject approaches here; their focus is on **interference**.

In a sequential program, an assignment of 42 to a variable x ensures that the next access to x will yield the value 42. If x is a variable that can be affected by a concurrent thread, this assurance no longer pertains. Interference between concurrent threads is a major headache for designers of concurrent software and many key algorithms rely on shared variables; this is often the case to improve performance. Programming language constructs have been conceived that limit and/or make more evident where interference can occur.

A further observation in Ashcroft (1975) about interference and concurrency is worth noting: Ed Ashcroft observes that—because debugging concurrent programs is so inadequate—there is a greater payback for applying formality to concurrent programs than to simpler sequential ones.

2 Ashcroft and Manna (Stanford 1969/1970)

This section reviews an approach to interference that attempts to establish the correctness of a concurrent program by reasoning about an equivalent program with many non-deterministic threads.

Zohar Manna (1939–2018)[5] obtained his PhD at what is now CMU (Pittsburg) supervised by Floyd and Al Perlis. Manna's thesis, Manna (1968), focussed on establishing termination of (sequential) programs but it—and also Manna (1969c)—offered a way of translating a program into logical expressions.[6] Although this represents a distinct idea from Floyd (1967) that required assertions to be added to a flowchart, Manna's approach still fits the *post facto* style and the differences from Floyd's approach do not influence Manna's contribution to concurrency. Manna moved to Stanford University in 1969. Before he began the collaboration with Ed Ashcroft that

[4] Although the distinction is fair of the original publications, it is possible to move closer to a top-down approach using Floyd's ideas. The author of the current paper had the good fortune to use, during its development, the "Effigy" system built by King (1969, 1971). King's PhD research was under the direction of Floyd at CMU and the Effigy system aspired to discharge the proof obligations that were implicit in checking Floyd assertions against code. There was however a way to record the specifications of (PL/I) procedures that could be developed later. This could effectively support a top-down development style.

[5] Dershowitz and Waldinger (2019) is an excellent scientific obituary and contains a full publication list.

[6] These first-order predicate calculus (FoPC) formulae contain un-interpreted predicate symbols and mechanisation was aimed at finding instantiations. Satisfaction of one set of formulae determines termination; a different set of equations is satisfiable by assertions about execution which would include the specification of the program. Satisfaction of FoPC is only semi-decidable so the search might not terminate and is anyway exponentially slow.

is the main topic of this section, he had written about non-deterministic programs in Manna (1969a).

Ed Ashcroft did his PhD in London University supervised by John Florentin. The title of Ashcroft's thesis, Ashcroft (1970), is *Mathematical Logic Applied to the Semantics of Computer Programs*. Whilst being in the formal area (and having some extremely elaborate formulae), it contains no hint of the concurrency work of interest in this section. He moved to Stanford around 1969 and this might have occurred before his thesis was finally approved.[7]

The publication of prime interest here is Ashcroft and Manna (1971) [which was pre-printed as a 1970 Stanford report Ashcroft and Manna (1970)]. The key insight is that a concurrent program can be reasoned about by constructing an equivalent *non-deterministic program*. Providing the atomic steps are identified properly, it is not difficult to see that it is possible to generate sequential threads that represent every possible merging of the original concurrent threads. The specific approach is couched in terms of flowcharts which follows the spirit of both Floyd's 1967 paper and Manna's own publications. In Ashcroft and Manna (1970), a parallel program has "and" forks in which both branches must be executed (very much in the way **fork/join** works) whereas non-determinism employs "or" branches where only one branch is executed.

To illustrate this, consider the example in which two concurrent threads must both be executed:

$$[S_1; S_2] \parallel [S_3, S_4]$$

This can be seen to be equivalent to a program that can non-deterministically select between the following set of six threads:

$$\left\{ \begin{array}{l} [S_1; S_2; S_3; S_4] \\ [s_1; S_3; S_2; S_4] \\ [S_1; S_3; S_4; S_2] \\ [S_3; S_4; S_1; S_2] \\ [S_3; S_1; S_4; S_2] \\ [S_3; S_1; S_2; S_4] \end{array} \right\}$$

Manna and Ashcroft concede that the expansion factor is exponential and the programs used as examples in Ashcroft and Manna (1971) are understandably rather limited. Tellingly one of two concurrent threads in their Program 2 (Ashcroft & Manna, 1971, Fig. 4) contains exactly one (atomic) statement: finding all of the places into which this has to be merged in the other thread limits the cardinality of the expanded non-deterministic threads to (roughly) the number of atomic statements in the longer thread of the concurrent original (this point becomes important below). In fact, for just two threads containing respectively m and n atomic assignment statements, the number of merged threads is $(m + n)!/(m! + n!)$; for the trivial example above, this is not too alarming but with only $n = m = 15$ there would be more than 150 million

[7] This conclusion was suggested by Matthew Hennessy (private communication).

merges. Where there are branches or loops[8] in either concurrent thread, the mapping to an equivalent non-deterministic program is more difficult. Referring again to their Program 2 (Ashcroft & Manna, 1971, Fig. 4), the non-trivial thread contains a loop and it would be incorrect to insert the single statement from the other thread into the arc going back to the loop test—the mapping given would, in general, have to arrange that the single statement was only executed in one iteration of the loop.[9] Their Program 3 (Ashcroft & Manna, 1971, Fig. 5) is more elaborate[10] and its expansion occupies two figures (7 and 9). As pointed out in Sect. 6.2 below, they do not offer a general mapping for arbitrary flowcharts.

Having expanded a concurrent program into non-deterministic sequential threads, each of these threads can be verified separately (they are sequential programs). This is accomplished in a style similar to that proposed by Manna's PhD supervisor Floyd. More precisely, the actual approach used in Ashcroft and Manna (1971) reflects Manna's own approach presented in Manna (1969b).

Despite the acknowledged problem of the huge expansion factor in going from a concurrent program to an equivalent non-deterministic collection of sequential threads, the work of Ashcroft and Manna offers a key insight. It is precisely the massive non-determinism that dogs any attempt—formal or informal—to reason about concurrency. Different orders of merging interfering threads can result in changed behaviour. Making the non-determinism explicit by expansion at least identifies the issue. Even achieving correctness for sequential programs is difficult; interference makes the creation of fault-free concurrent programs orders of magnitude more challenging.

There is, however, an additional crucial caveat that the atomic steps of the concurrent threads must be identified and assuming that assignment statements are executed without interference is unrealistic in practice (this topic is explored in more detail in Sect. 7 below).

In his subsequent research, Manna went on to employ "temporal logic"—see Dershowitz and Waldinger (2019) for references. There is also an interesting link between that research and the ideas discussed in Sect. 3.

3 Ashcroft (Waterloo 1971–1975)

The link with the work described in Sect. 2 is obviously Ed Ashcroft himself; he was appointed to Waterloo in January 1971.[11] The starting point for his next step was a recognition of the unacceptable size of the expanded non-deterministic program from

[8] It is important here to remember that the research discussed in this section (and the next) is built upon flowcharts: loops are constructed by backwards branches.

[9] Returning to the number of paths in sequential program, it is clear that a sequential program with n simple tests can have 2^n paths. For example, imagine a program that tests the 32 bits of a word as a long sequence of **if/then/else** constructs. Of course, a shorter and more elegant program might index over the bits in a **while** loop but it could still have 2^{32} potential sequential paths.

[10] There are no specifications given for the examples whose tests and state changes are shown as uninterpreted symbols.

[11] Don Cowan (founding head of Computing) kindly tracked down the date.

the previous joint work. In Ashcroft (1975)[12] he formalises an approach whose insight is to employ *control states* to index the "state assertions" (which latter are the familiar predicates over states). Ashcroft's control states can be pictured by thinking of keeping fingers on all of the statements that could be executed next in the concurrent threads.[13] The need to consider all possible combinations of next moves can however be reduced: where there are transitions in the concurrent program that have no impact on variables that are shared between the threads, the verifier can reduce the number of distinct state assertions that have to be written.

There are still potentially exponentially many transitions between control states but it is much clearer than in the previous joint work how they can be merged. Ashcroft's proposal still faces the non-determinism inherent in the execution of concurrent threads but, rather than starting with a mechanical expansion of a concurrent program into its nondeterministic equivalent, Ashcroft's proposal identifies the grouping of sets of control states which behave in the same way.

Ashcroft's approach is best illustrated with an example and he tackles a significant one (a simplified airline reservation system) in Ashcroft (1975). For the purposes of this paper, the most convenient source of an example is to reuse Susan Owicki's *Findpos* which is covered in Sect. 5; the application of Ashcroft's approach to that example is presented in Sect. 6.1.

The introduction of control states certainly offers a more succinct way of reasoning about non-determinism than expanding all of the potential threads. It can also be seen as making a more profound change to the way in which reasoning about programs can be conducted: rather than adding assertions to a flowchart, the state assertions indexed by control states offer a way of talking about program execution. This points to the approach pursued with such effect using temporal logic.

Both Ashcroft's solo approach and the joint work described in Sect. 2 discuss linguistic extensions that form blocks in flowcharts which reduce the number of merge points. In both cases, programming languages that offered more structured constructs (than their flowcharts)[14] might have made presentation clearer but this would not have affected the key insights.

In both approaches considered so far, a significant "blind spot" is that they effectively ignore Hoare's 1969 paper (briefly sketched here in Sect. 4 because Hoare (1969) underlies Owicki's approach addressed in Sect. 5). Furthermore, they offer only *post-facto* verification in the sense that they start with (a flowchart representation of) an existing program.

Ashcroft's later research included a collaboration with Bill Wadge that developed the Lucid language, see Ashcroft and Wadge (1976, 1979).

[12] A version is shown as first submitted January 1973. It would be interesting to know what part the referees played in the development of the final text.

[13] There is an interesting echo here of VDL *control trees* that serve to track available next steps in an operational semantic description of a programming language (see Lucas & Walk, 1969). Evidence that Ashcroft could have been aware of the IBM Vienna Lab operational semantics descriptions of PL/I etc. is that his PhD supervisor was John Florentin (with whom the current author collaborated at that time); Florentin certainly knew the VDL research well.

[14] In Ashcroft and Manna (1971), space has to be given to ruling out "invalid" flowcharts.

4 Hoare's "Axiomatic Method" and Its Use in Design

This section picks up from Sect. 1.1 the key background approaches to the verification of sequential programs. Tony Hoare's 1969 paper is briefly reviewed and the fact is emphasised that he followed it two years later with a key transition that showed how his axioms could be used as the basis for using his axiomatic approach in design.

Floyd (1967) gives conditions for state assertions to be consistent with programs. Hoare had been searching for a way to give the semantics to programming languages that made it possible to "leave some things open" and saw that Floyd's "Assigning meanings to programs" provided a key to achieving his objective. [More on the evolution of Hoare's thinking can be found in Jones (2003) and Jones and Misra (2021) and its impact in Apt and Olderog (2019).] Hoare offers—in Hoare (1969)—a generous acknowledgement to Floyd's inspiration and it would be possible to view Hoare's axioms as an alternative presentation of Floyd's conditions for the consistency of assertions and program text. The end of the current section outlines why this would be a serious undervaluation of Hoare's contribution.

A "Hoare-triple" $\{P\}\ \ S\ \ \{Q\}$ asserts that, if the statement S is executed in a state that satisfies the pre condition P and if S terminates, then the resulting state will satisfy the predicate Q. Rules of inference are presented for a simple but powerful set of programming constructs in Hoare (1969). Examples are:

Sequential composition:

$$\textbf{sequence}\ \frac{\{P\}\ \ S1\ \ \{Q\} \quad \{Q\}\ \ S2\ \ \{R\}}{\{P\}\ \ S1;\ \ S2\ \ \{R\}}$$

Iteration:

$$\textbf{while}\ \frac{\{P \wedge b\}\ \ S\ \ \{P\}}{\{P\}\ \ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ \ \{P \wedge \neg b\}}$$

The role of P in the above rule is to provide a loop invariant.

Conditional:

$$\textbf{if}\ \frac{\{P \wedge b\}\ \ S1\ \ \{Q\} \quad \{P \wedge \neg b\}\ \ S2\ \ \{Q\}}{\{P\}\ \ \textbf{if}\ b\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{fi}\ \ \{Q\}}$$

Assignment:
At the end of a development process, assignment statements bring about the changes of state from that in the pre condition to one that satisfies the post condition. The axiom schema for assignment can be given as:

$$\boxed{\leftarrow}\ \frac{}{\{P^e_x\}\ \ x \leftarrow e\ \ \{P\}}$$

where P^e_x is the result of systematically substituting the expression e for every free occurrence of the identifier x throughout P.

Finally a rule facilitates adapting a judgement to stronger pre conditions and weaker post conditions:

$$\textbf{weaken}\ \frac{\begin{array}{c}\{P\}\ \ S\ \ \{Q\}\\ P' \ \Rightarrow\ P\\ Q\ \Rightarrow\ Q'\end{array}}{\{P'\}\ \ S\ \ \{Q'\}}$$

The move from annotated flowcharts to a system of inference rules for "judgements" recorded as Hoare triples opened the door to a crucial development. Furthermore, the history of Hoare (1971) exposes the key transition: an early draft of this paper did attempt to present a *post facto* proof of Hoare's earlier FIND program. At that time, there were no readily available theorem proving assistants and it was thus very hard to achieve confidence in the monolithic proof.[15] Hoare rewrote the paper so as to describe a step-by-step development of the FIND program which was far easier to check. (Unfortunately, he failed to update the first word of the title of "Proof of a program FIND".)

The fact that the inference rules above can be used to decompose a specification into specified sub-components requires reading them from the conclusion below the line to the hypotheses above the line. Once this is done, it is easy to see how the rules provide the basis of a "posit and prove" development method.[16]

Hoare and colleagues worked on inference rules for many programming constructs (see Apt & Olderog, 2019 for details); of particular relevance to the concurrency theme of the current paper is Hoare (1975). Looking at disjoint parallelism where the threads have no shared variables, Hoare proposed an "ideal" rule in which constructs like **cobegin** $\cdots$ **coend** would satisfy the conjunction of the post conditions of the separate

[15] The current author was a referee for both Hoare (1969, 1971); the initial version of the latter proved to be uncheckable.

[16] One reservation about Hoare's original inference rules is the use of post conditions that are predicates of only the final state—a decision followed by many subsequent researchers including Dijkstra's "weakest pre conditions" (Dijkstra, 1976). (It is possible that this actually derived from Floyd's flowchart annotations.) Formal specification techniques from VDM (Jones, 1980), via Event-B (Abrial, 2010) to Alloy (Jackson, 2012) employ relations which are predicates that relate the final to the initial state.

threads (providing the **cobegin** was executed in an environment where the conjunction of the pre conditions was satisfied).

$$\textbf{disjoint-}\| \quad \frac{\begin{array}{c}\{P_1\}\ \ S_1\ \ \{Q_1\} \\ \{P_2\}\ \ S_2\ \ \{Q_2\}\end{array}}{\{P_1 \wedge P_2\}\ \ S_1 \parallel S_2\ \ \{Q_1 \wedge Q_2\}}$$

A version of this rule makes a first appearance in the appendix of Hoare (1972, p. 244) and has clear links with the work described in the next section.

5 Owicki (Cornell 1975)

Susan Owicki's Cornell thesis (Owicki, 1975) is most easily approached[17] via a joint paper (Owicki and Gries, 1976b) with her supervisor David Gries (their Owicki & Gries, 1976a) covers broadly the same material but with more emphasis on what were later dubbed liveness properties; it also says more about so-called "auxiliary variables"[18]). The thesis itself has a lot of material but the intention here is to focus on the key approach to the verification of concurrent programs which is commonly referred to as the "Owicki–Gries method". Unlike the ideas in Sects. 2 and 3, Owicki's approach is explicitly based on Hoare's 1969 paper. There is, however, a clear link to the research covered in Sects. 2 and 3: both Owicki (1975) and Owicki and Gries (1976b) cite Ashcroft and Manna (1971) and a 1973 draft of Ashcroft (1975).

The key insight is that the fact that a set of concurrent threads satisfy given pre/post conditions can be established in two stages:

- firstly, construct separate Hoare-style sequential proofs of each thread (as indicated below, these proofs must place assertions after every statement);
- these "proof outlines" are then reviewed to see whether statements in other threads interfere with steps of the sequential proof.

As presented, the approach to verification is still *post facto*; this point is discussed further below.

The approach is best presented with an example and Owicki offers one which is small enough to cover in detail but which contains enough challenge to use in Sect. 6 as a basis for comparison with the earlier approaches. Owicki's $Findpos$ example uses two concurrent threads to search for the lowest array index that points to a positive element.

It is obvious how to write a sequential program that steps through the indexes of array x from 1 to the maximum (M). A decision has to be made about how to handle

[17] Some of the material in this section benefited from a 2021 interview (over Zoom) with Susan Owicki.

[18] I am grateful to Leslie Lamport for reminding me both of this ACM paper and his own (Lamport, 1977) (see Acknowledgements). Lamport's paper outlines an independently invented approach that has similar scope to Owicki's contribution; his paper is most cited for the import discussion of "liveness versus safety". Lamport and the current author differ as to the desirability of employing control state predicates but it is important to record that he considers this to be a distinguishing factor between the Ashcroft and Owicki approaches.

the possibility that there is no positive element in x: this could, of course, be ruled out by a pre condition, or a dummy value could be placed at $x(M+1)$. To simplify comparison with Owicki's text, the working below adheres to her decision to initialise top counters to $M+1$.

It is not difficult to see that the set of index values ($\{1..M\}$) can be partitioned and given to parallel threads that search their subset and record the minimum value (if any) where a positive array value is located. The simplest partition to present is to employ two threads, one searching the even index values (indexing the counter ec up to et) and the other the odd (oc, ot). After execution of both searches the final result t can be set to the minimum of the two minima (et, ot).

Such disjoint parallel threads could actually execute more comparisons than a single sequential loop in cases where a low index satisfies the test in one thread and there is no positive value associated with the index values of the other thread. What makes $Findpos$ interesting is to allow each thread to read the counter of the other thread so that either thread can cease searching beyond a satisfying index found by the other thread. It is this interference that exhibits the key idea in Owicki's approach.

The first step in the Owicki approach is to annotate the separate threads with Hoare-like assertions. Following Owicki (1975, Fig. 3.4), the assertions are shown here in Fig. 1 (coloured blue).[19] The consistency of these assertions with respect to the program can be justified using the Hoare rules of Sect. 4. For what has to be done at the next stage, it is essential that there is an assertion at every point in the two programs. Looking at Fig. 1, this appears to be a substantial commitment but most of the assertions can be generated mechanically using Hoare's rules.

Thus far, interference is ignored: the reasoning about threads is done as though they were sequential programs with no danger of changes to the values of variables other than in the code of the thread.

As made clear in Sect. 1.2 above, the essence of shared-variable concurrency is that threads interfere by changing shared variables. Clearly, changing the value of variables can invalidate proofs (such as those in Fig. 1) conducted on the assumption that there is no interference. Owicki's proof rule for concurrent processes comes disarmingly close to Hoare's **disjoint**− ‖ rule:

$$\textbf{par}\ \frac{\{P_1\}\ \ T_1\ \ \{Q_1\}, \cdots, \{P_n\}\ \ T_n\ \ \{Q_n\} \text{ are interference free}}{\{P_1 \wedge \cdots \wedge P_n\}\ \textbf{cobegin}\ T_1 \parallel \cdots \parallel T_n\ \textbf{coend}\ \ \{Q_1 \wedge \cdots \wedge Q_n\}}$$

The crucial side condition is that the proofs of the threads must be "interference free". Owicki's key proposal is that the impact of the interference of executing the statements in one thread T_i can be judged by seeing whether executable statements in other threads T_j can impact steps in the justification of T_i. Potentially, the number of such checks is proportional to the product of the number of steps in each thread. Fortunately, it is rather easy to cut down on the scale of this task as can be illustrated on the $Findpos$ example.

[19] The overall post condition does not actually prohibit changes to x—see the discussion on relational post conditions in Sect. 4.

Using the following abbreviations for predicates in the even and odd threads:

$$ES = \left\{ \begin{array}{l} even(ec) \wedge \\ et \leq M + 1 \wedge \\ (et \leq M \;\Rightarrow\; x(et) > 0) \wedge \\ \forall n \cdot even(n) \wedge 0 < n < ec \;\Rightarrow\; x(n) \leq 0 \end{array} \right\}$$

$$OS = \left\{ \begin{array}{l} odd(oc) \wedge \\ ot \leq M + 1 \wedge \\ (ot \leq M \;\Rightarrow\; x(ot) > 0) \wedge \\ \forall n \cdot odd(n) \wedge 0 < n < oc \;\Rightarrow\; x(n) \leq 0 \end{array} \right\}$$

The fully annotated proofs of the separate threads are:

$Initialize$: $ec \leftarrow 2$; $et \leftarrow M + 1$;
$\{ES\}$
$Evensearch$: **while** $ec < min(ot, et)$ **do**
$\{ES \wedge ec < et \wedge ec < M + 1\}$
$Eventest$: **if** $x(ec) > 0$
then $\{ES \wedge ec < et \wedge ec < M + 1 \wedge x(ec) > 0\}$
$Evenyes$: $et \leftarrow ec$ $\{ES\}$
else $\{ES \wedge ec < et \wedge x(ec) \leq 0\}$
$Evenno$: $ec \leftarrow ec + 2$ $\{ES\}$
fi
$\{ES\}$
od
$\{ES \wedge ec \geq min(ot, et)\}$

$Initialize$: $oc \leftarrow 1$; $ot \leftarrow M + 1$;
$\{OS\}$
$Oddsearch$: **while** $oc < min(ot, et)$ **do**
$\{OS \wedge oc < ot \wedge oc < M + 1\}$
$Oddtest$: **if** $x(oc) > 0$
then $\{OS \wedge oc < ot \wedge oc < M + 1 \wedge x(oc) > 0\}$
$Oddyes$: $ot \leftarrow oc$ $\{OS\}$
else $\{OS \wedge oc < ot \wedge x(oc) \leq 0\}$
$Oddno$: $oc \leftarrow oc + 2$ $\{OS\}$
fi
$\{OS\}$
od
$\{OS \wedge oc \geq min(ot, et)\}$

Fig. 1 Fully decorated sequential proofs of the two threads of $Findpos$

Owicki focusses the "interference freedom" proof (Owicki, 1975, Definition 3.12, p. 39) on any statements in one thread that affect the values shared with the other thread. In $Findpos$, the actual interference is limited: the only dangerous statement in $Oddsearch$ is the assignment $ot \leftarrow oc$ which can interfere with the loop test at

$Findpos$: **begin**
  $Initialize$: $ec \leftarrow 2;\ \ oc \leftarrow 1;\ \ et \leftarrow ot \leftarrow M+1;$
    $\{ec = 2 \wedge oc = 1 \wedge et = ot = M+1\}$
  $Search$: **cobegin**
    $\{ES\}$
    $Evensearch$: $\cdots$
    $\{ES \wedge ec \geq min(ot, et)\}$
  ||
    $\{OS\}$
    $Oddsearch$: $\cdots$
    $\{OS \wedge oc \geq min(ot, et)\}$
  coend
    $\{OS \wedge ES \wedge ec \geq min(ot, et) \wedge oc \geq min(ot, et)\}$
$t \leftarrow min(ot, et)$
    $\{t \leq M+1 \wedge (t \leq M \ \Rightarrow\ x(t) > 0) \wedge$
      $\forall n \cdot 0 < n < t \ \Rightarrow\ x(n) \leq 0\}$
end

Fig. 2 Combining the results from Fig. 1 using Owicki's parallel rule

$Evensearch$. In fact, the positive test of the loop predicate is not affected because:

$$ec < min(ot, et) \ \Rightarrow\ ec < et$$

even if ot changes. The case that needs checking is where the loop terminates because its test is false (see Hoare's **while** rule in Sect. 4) since:

$$ec \geq min(ot, et)$$

can be invalidated by an increase to ot. Fortunately, the assertion at $Oddyes$ has a pre condition of $oc < ot$ on the assignment so $ot \leftarrow oc$ cannot increase the value of ot.[20] The argument about interference in $Oddsearch$ is symmetrical. In conclusion, the two proof outlines of Fig. 1 can be combined in Fig. 2 with Owicki's **par** rule and a concluding use of the assignment rule.

Owicki's approach is clearly *post facto* in the sense that program construction precedes verification. This is aggravated by the two stage verification strategy in which the standard sequential proofs are constructed prior to application of the "interference freedom" test. A failure to clear that final hurdle presents the developer with no choice but to go back to the starting line. Of course, an experienced developer might be able to think ahead and keep the subsequent proof obligation in mind but the fact remains that

[20] This argument is made slightly difficult to follow because of minor slips in the publications. It is therefore worth recording some technical details for future readers of the primary material. In Owicki (1975) (using her identifiers), the essential conjunct $i < eventop$ is omitted in pre condition of $Evenyes$; similarly $j < oddtop$ in the pre condition for $Oddyes$; the choice of the bound identifier (i) in the overall post condition was unhelpful since the program has a variable of this name (l is used in Owicki & Gries, 1976b). In Owicki and Gries (1976b), the loop invariant of $Oddsearch$ has a copy-and-paste error: the i should be j.

there is nothing in Owicki's approach that facilitates recording and reasoning about interference during design.

It is interesting to look at both stimuli from IFIP's WG2.3 ("Programming Methodology") and that group's role in amplifying the influence of the research. The Owicki and Gries (1976b) paper contains a specific acknowledgement to this collection of important players in the world of *Programming Methodology*—both Hoare and Edsger Dijkstra were founder members of WG2.3. Gries first observed[21] at the meeting (Munich) in December 1974 and he was elected a member in September 1975 (Baden bei Wien); Owicki was an observer at the July 1976 meeting (Cazenova). These contacts might have increased the incentive to relate the Owicki–Gries approach to Hoare (1969) [the Owicki–Gries approach could be described in terms of the flowcharts of Floyd (1967)]. Owicki's thesis (Owicki, 1975) provides soundness proofs for the proof obligations and cites Peter Lauer's research with Hoare and Lauer (1974).

Looking in the other direction at the scientific impact of the Owicki–Gries approach, it is again plausible that WG 2.3 helped to amplify and disseminate the message. For example, Dijkstra's EWD554 provides a "personal summary" of the approach.[22] The use of the Hoare framework, Gries' reputation and the fact that Owicki had put foward a good—and well presented—idea are certainly all factors.

6 Attempting Owicki's Example with the Earlier Approaches

As this section illustrates, attempting to verify Owicki's *Findpos* example using the historically earlier approaches described above in Sects. 2 and 3 is revealing. At the risk of further offence to historians, the steps back from Owicki's 1975 example are taken in reverse chronological order with Sect. 6.1 using Ashcroft's solo approach and Sect. 6.2 considering the earlier joint Ashcroft/Manna work.

6.1 Revisiting Ashcroft's (Solo) Approach

The approach published in Ashcroft (1975) works on flowchart presentations of programs but there is no difference in technical details if the starting point represents the arcs of such a chart as **goto** statements. (The idea of placing fingers on the statement to be executed next in each thread still works.) Retaining the (long) identifiers from

[21] IFIP working groups have a process of inviting observers (sometimes several times) before considering people for membership.

[22] Much could be said about this note [original manuscript dated 14th of March 1976 available from the Austin archive and reprinted as Dijkstra (1982)]. To give a flavour it begins: "This is a very personal summary of the theory developed by Susan Speer Owicki under the supervision of David Gries. I had a flu, and on its first day I just slept and shivered; later I passed the time in bed with trying to reconstruct what I had learned from reading Susan Owicki's doctoral thesis. If the following fails to do justice to their work—someone has borrowed my copy of her thesis!—-I am the only one to blame." (Perhaps the loan of the key document accounts for the distance between what is identified here as Owicki's key contribution and Dijkstra's description of what he insists on referring to as the "Gries–Owicki Theory".) Interestingly, the text of EWD554 reflects the fact that programs achieve relations but uses predicates of single states in the formulae.

the program in Sect. 5, the flowchart equivalent would be:

$$
\begin{array}{l}
Findpos: \ ec \leftarrow 2; \ oc \leftarrow 1; \ et \leftarrow ot \leftarrow M+1; \\
Search: \textbf{fork go to}(Evensearch, Oddsearch); \\
Evensearch: \ \textbf{if } ec < min(ot, et) \textbf{ then go to } Eventest \textbf{ else go to } J; \\
\quad Eventest: \textbf{if } x(ec) > 0 \textbf{ then go to } Evenyes \textbf{ else go to } Evenno; \\
\quad\quad Evenyes: et \leftarrow ec; \ \textbf{go to } Evensearch; \\
\quad\quad Evenno: ec \leftarrow ec+2; \ \textbf{go to } Evensearch; \\
\quad Oddsearch: \textbf{if } oc < min(ot, et) \textbf{ then go to } Oddtest \textbf{ else go to } J; \\
\quad\quad Oddtest: \textbf{if } x(oc) > 0 \textbf{ then go to } Oddyes \textbf{ else go to } Oddno; \\
\quad\quad\quad Oddyes: ot \leftarrow oc; \ \textbf{go to } Oddsearch; \\
\quad\quad\quad Oddno: oc \leftarrow oc+2; \ \textbf{go to } Oddsearch; \\
\quad J: t \leftarrow min(ot, et); \\
\quad F: \textbf{HALT}
\end{array}
$$

Recall that Ashcroft indexes state assertions (that relate the values of variables) by "control states" ($\mathcal{C}$) that record where execution is in the threads. The initial step to label $Search$ (from $Findpos$) is sequential so the control state $\mathcal{C}$ is a unit set and the corresponding state assertion is:

$$\mathcal{C} = \{Search\} : ec = 2 \wedge oc = 1 \wedge et = ot = M+1$$

As Ashcroft observes, singleton control states yield to standard Floyd reasoning so this step is trivial.

Once the concurrency is initiated, $\mathcal{C}$ has pointers into both threads so the indexing is expressed as $p \in \mathcal{C}$. Using the same definitions of ES as in Fig. 1, it is sufficient to consider groups of control states for the even thread as follows (with their attached state assertions):

$$
\begin{array}{l}
Evensearch \in \mathcal{C} : ES \\
Eventest \in \mathcal{C} : ES \wedge ec < et \wedge ec < M+1 \\
Evenyes \in \mathcal{C} : ES \wedge ec < et \wedge ec < M+1 \wedge x(ec) > 0 \\
Evenno \in \mathcal{C} : ES \wedge ec < et \wedge x(ec) \leq 0
\end{array}
$$

Each of these steps can also follow by standard Floyd-like reasoning because there is no interference that could change the truth of the state assertions.

The only difficult step in the even thread is when it completes. Consider the situation when the even finger is on $Evensearch$ with $ec \geq min(ot, et)$: this could be because ot has the lower value; the next move of the even finger will replace $Evensearch$ by J in the control state. The state assertion at J still requires that $ec \geq min(ot, et)$ which begs the question whether ot could increase. Fortunately, the only change to ot can come about when the odd finger is on $Oddyes$ at which point the relevant control-state-indexed assertion establishes that $oc < ot$.

The reasoning for the Odd thread is completely symmetric. The two results ensure that:

$$J \in \mathcal{C} : ES \wedge OS \wedge ec \geq min(ot, et) \wedge oc \geq min(ot, et)$$

The final step from J to F is again sequential reasoning (singleton control states):

$$\mathcal{C} = \{F\} : t \leq M + 1 \wedge (t \leq M \;\Rightarrow\; x(t) > 0) \wedge \\ \forall n \cdot 0 < n < t \;\Rightarrow\; x(n) \leq 0$$

Tackling Owicki's $Findpos$ example using Ashcroft's approach shows stronger similarities than superficial examination of their rather different texts might suggest. It is not surprising that, in both cases, the key interference step is the same: that comes from the algorithm chosen. (And the similarities are emphasised here by employing Owicki's ES/OS predicates in this section.) But the way in which Ashcroft's approach groups control states provides a surprisingly accurate prediction of where Owicki's approach has to identify potential proof interference.

The distinction that remains between Ashcroft's and Owicki's approaches is both more interesting and, in a sense, historically surprising. Owicki's method is presented in the style of Hoare's way of annotating programs (although the complete sets of assertions are in fact Floyd-like). Ashcroft's indexing by control states separates out the state assertions from the program text to record facts about the program. It could be argued that this is a hint of later verification approaches using temporal logics (a research area to which Manna made significant contributions).

6.2 Revisiting the Ashcroft/Manna Approach

Attempting to verify $Findpos$ using the approach in Ashcroft and Manna (1971) (see Sect. 2 above) is challenging. Although the authors exhibit a moderately complicated example (Ashcroft& Manna, 1971, Fig. 5) of how to expand a concurrent program into an equivalent non-deterministic program, they do not provide a general mapping algorithm. In particular, they do not describe how to handle the mapping of two concurrent threads that both contain loops. This is, of course, exactly the structure of $Findpos$.

Looking at the version of the $Findpos$ program in Sect. 6.1, there are four control points in each thread and a straightforward mapping would yield 16 non-deterministic threads. When preparing the talk from which this paper derives, the current author baulked at trying to construct such a large flowchart. The decision to mechanically generate the non-deterministic equivalent to a concurrent program—before even considering its content—is certainly the most obvious shortcoming of the Ashcroft/Manna approach.

Subsequent to giving the talk at HaPoC, an intriguing ahistorical idea has come to the rescue. The current author noticed that Dijkstra's "guarded commands" offer a way of describing the selection between the large number of non-deterministic threads required by the Ashcroft/Manna approach. By a happy coincidence, communicating this thought to Krzysztof Apt uncovered the fact that, not only had he and Ernst–

Rüdiger Olderog explored such a use of guarded commands in Apt and Olderog (1991, Chap. 10)—see also Flon and Suzuki (1978) where idea of using guarded commands to express non-deterministic programs is explored, but that—in their current writing (Apt & Hoare, 2022, Chap. 8)—they tackle exactly the same program and have a fairly compact mapping to guarded commands.

Of course, guarded commands were invented several years after Ashcroft and Manna (1971) was written[23] but their ability to express succinctly the massive non-determinism in the Ashcroft/Manna expansions is impressive. Interested readers are invited to study the guarded command version of $Findpos$ in Apt and Hoare (2022, Sect. 8.6) and note the similarity between the guards on control points and Ashcroft's "keeping fingers on the potential next statements". (They do not go on to provide proofs.)

7 Atomicity

An issue that presents difficulties for many approaches to reasoning about concurrent programs is that of the atomicity of statement execution. All three approaches described above assume that assignment statements are atomic in the sense that the state does not change during their execution. In general, this is unrealistic: executing $x \leftarrow x + 1$ concurrently with $x \leftarrow x + 2$ does not necessarily increase the value of x by 3. The problem is, of course, that a compiler has to map assignments into sequences of atomic machine instructions. Assuming that single (word length) variables are accessed and changed atomically, the above pair of concurrent statements might amount to:

$$\{t1 \leftarrow x;\ t1 \leftarrow t1 + 1;\ x \leftarrow t1\} \parallel \{t2 \leftarrow x;\ t2 \leftarrow t2 + 2;\ x \leftarrow t2\}$$

Merging the steps of these two sequences could increase the value of x by either one or two.

It has been observed that if each assignment statement only refers to one shared variable, then it is safe to regard assignment statements as being executed "atomically".[24] Unfortunately, as the above example shows, this only exposes—rather than solves—the problem.

Although the $Findpos$ example has no offending assignments, it does suffer from the closely related issue of assuming that expressions in tests are executed "atomically".

[23] The dates of drafts of the guarded command paper are interesting: the published paper (Dijkstra, 1975) was printed in August 1975 but had been submitted in July 1974; The CACM paper shows a revision date of January 1975 which is the same month as Dijkstra's privately circulated EWD472. It would appear that in this case his "EWD" did not represent his first thoughts on the topic.

[24] This is sometimes referred to as "Reynolds rule". But John Reynolds claimed to the current author that he had nothing to do with its invention.

8 Conclusions

Identifying contributions made by researchers 50 years ago runs the risk of them being classed as "obvious" to a modern reader. Notwithstanding the benefit of hindsight, the following insights are important:

- Ashcroft/Manna build their approach around non-deterministic programs that are expanded equivalents of concurrent programs; the (many) threads of the expansion can then be verified using techniques for sequential programs.
- Ashcroft retains the idea of reasoning explicitly about the non-deterministic behaviour of concurrency but uses control states to keep track of the potential transitions in the original concurrent program.
- Owicki splits verification of concurrent programs into two phases: her approach first considers each thread as a sequential program in isolation from its environment; this is followed by a check that no interference can invalidate the separate proofs.

All three approaches are *post facto* (or "bottom up") in the sense that the starting point in each case is described as having a program. It is, of course, true that developers might learn to watch—during initial design—for symptoms that they would have learned can be problematic in the verification phase. But the fact remains that there is no way offered to record and reason about interference.[25]

The first two approaches are clearly Floyd-like (and barely mention Hoare's 1969 paper) in that programs are presented as graphs or flowcharts. Although Owicki uses Hoare-like rules in the first phase of her verification strategy, the use of "proof outlines" which require complete annotations could have been conducted using Floyd reasoning. In fact, a more "top-down" flavour could have been added to the first phase of Owicki's approach by following Hoare (1971). A stronger reflection of Hoare's approach can be detected in the way Owicki's interference freedom rule for concurrency is presented.

Another consideration might be the feasibility of mechanising the three approaches. In principle, the Ashcroft–Manna approach could be programmed and combined with Manna's verification proposal[26] but for non-trivial programs the expansion factor in getting to the non-deterministic equivalent of a concurrent program would make this unworkable. Ashcroft's solo approach shows how human judgement can reduce the expansion but it is difficult to see how this required step could be mechanised. Owicki's approach is clearly mechanisable: there are tools that could assist the separate Hoare-style proofs of the individual threads and the interference freedom test could be readily programmed.

Turning to the issue of the apparent "linearity" of the three approaches. First it must be clear that their strong links were the reason for selecting these specific contributions. There was certainly other research being conducted on concurrency during the 1970s. It must also be remembered that the number of researchers involved in concurrency 50 years ago was far less than today.

[25] This deficiency was the insight behind later work on "Rely-Guarantee" approaches (see Jones, 1981; Jones & Hayes, 2016; Hayes & Jones, 2018).

[26] See Footnote 6.

Much work on concurrency in that decade was on devising programming language constructs that constrained the interference and made it safe to synchronise threads: Chap. 7 of Hoare (1985) gives a useful technical summary; Troy Astarte's paper in this volume enlarges on the contributions of Per Brinch-Hansen. Much the most developed formal approach to concurrency was the work of Carl Adam Petri and this topic is discussed in Wolfgang Reisig's books (Reisig, 1985, 2013). The end of the decade saw the use of Temporal Logics and a move to communication-based concurrency as in CCS and CSP. Each of these topics would justify separate historical papers.

Acknowledgements This paper is a development of the author's talk at the HaPoC-21 meeting in ETH Zurich; the author is grateful to the organisers of that event. Susan Owicki kindly gave her time for a "discussion" (perhaps a more accurate description than "interview") over the Internet. Don Cowen shared recollections of Ed Ashcroft and this contact was established by Dan Berry at Waterloo. Matthew Hennessy also helped with background information on Ed Ashcroft and Bill Wadge (an interesting additional historical snippet is that Ashcroft was Hennessy's PhD supervisor). Troy Astarte commented on an early draft of the paper. Leslie Lamport reviewed a full draft and made insightful comments of his own which led to an interesting email exchange. In noting that he and the current author did not agree on all points, it is hoped that this might stimulate Lamport to publish his own historical reflections. (Lamport's annotated bibliography at http://lamport.azurewebsites.net/pubs/pubs.html is a useful resource.) The author is also grateful for the input from the anonymous referees selected by the editors. The author's research is supported by Leverhulme Grant RPG-2019-020.

References

Abrial, J. R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

Apt, K. R., & Hoare, T. (Eds.). (2022). *Edsger Wybe Dijkstra: His life, work and legacy*. ACM.

Apt, K. R., & Olderog, E. R. (1991). *Verification of sequential and concurrent programs*. Springer.

Apt, K. R., & Olderog, E. R. (2019). Fifty years of Hoare's logic. *Formal Aspects of Computing, 31*(6), 751–807.

Ashcroft, E. A. (1970). Mathematical logic applied to the semantics of computer programs. PhD thesis, University of London.

Ashcroft, E. A. (1975). Proving assertions about parallel programs. *Journal of Computer and System Sciences, 10*(1), 110–135.

Ashcroft, E. A., & Manna, Z. (1970). Formalization of properties of parallel programs. Tech. Rep. AIM-110, Stanford Artificial Intelligence Project. https://apps.dtic.mil/sti/citations/AD0708740. Published as Ashcroft and Manna (1971)

Ashcroft, E. A., & Manna, Z. (1971). Formalization of properties of parallel programs. In B. Meltzer & D. Michie (Eds.), *Machine intelligence* (Vol. 6, pp. 17–41). Edinburgh University Press.

Ashcroft, E. A., & Wadge, W. W. (1976). Lucid—a formal system for writing and proving programs. *SIAM Journal on Computing, 5*(3), 336–354.

Ashcroft, E. A., & Wadge, W. W. (1979). R for semantics. Tech. Rep. CS-79-37, Faculty of Mathematics, University of Waterloo, Canada

Dershowitz, N., & Waldinger, R. (2019). Zohar Manna (1939–2018). *Formal Aspects of Computing, 31*(6), 643–660.

Dijkstra, E. (1975). Guarded commands, non-determinacy, and formal languages. *Communications of the ACM, 18*(8), 453–457.
Dijkstra, E. W. (1976). *A discipline of programming*. Prentice Hall.
Dijkstra, E. W. (1982). A personal summary of the Gries–Owicki theory. In E. W. Dijkstra (Ed.), *Selected writings on computing: A personal perspective*. Texts and monographs in computer science. Springer, EWD554.
Flon, L., & Suzuki, N. (1978). Consistent and complete proof rules for the total correctness of parallel programs. Tech. Rep. CSl-78-6, Xerox, Palo Alto.
Floyd, R. W. (1967). Assigning meanings to programs. In J. Schwartz (Ed.), *Mathematical aspects of computer science, Proceedings of symposia in applied mathematics* (Vol. 19, pp. 19–32). American Mathematical Society.
Hayes, I. J., & Jones, C. B. (2018). A guide to rely/guarantee thinking. Lecture notes in computer scienceIn J. Bowen, Z. Liu, & Z. Zhan (Eds.), *Engineering trustworthy software systems—Third International School, SETSS 2017* (Vol. 11174, pp. 1–38). Springer.
Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM, 12*(10), 576–580.
Hoare, C. A. R. (1971). Proof of a program: FIND. *Communications of the ACM, 14*(1), 39–45. https://doi.org/10.1145/362452.362489
Hoare, C. A. R. (1972). Towards a theory of parallel programming. In C. A. R. Hoare & R. Perrott (Eds.), *Operating System Techniques* (pp. 61–71). Academic Press.
Hoare, C. A. R. (1975). Parallel programming: An axiomatic approach. *Computer Languages, 1*(2), 151–160. Also have hard copy.
Hoare, C. A. R. (1985). *Communicating sequential processes*. Prentice Hall.
Hoare, C. A. R., & Lauer, P. E. (1974). Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica, 3*(2), 135–153. https://doi.org/10.1007/BF00264034
Jackson, D. N. (2012). *Software abstractions: Logic, language, and analysis*. MIT.
Jones, C. B. (1980). *Software development: A rigorous approach*. Prentice Hall International. http://portal.acm.org/citation.cfm?id=539771
Jones, C. B. (1981). Development methods for computer programs including a notion of interference. PhD thesis, Oxford University. Printed as: Programming Research Group, Technical Monograph 25.
Jones, C. B. (2003). The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing, 25*(2), 26–49.
Jones, C. B. (2017). Turing's 1949 paper in context. Lecture notes in computer scienceIn J. Kari, F. Manea, & I. Petre (Eds.), *Computability in Europe 2017* (Vol. 10307, pp. 21–41). Springer.
Jones, C. B., & Hayes, I. J. (2016). Possible values: Exploring a concept for concurrency. *Journal of Logical and Algebraic Methods in Programming, 85*(5), 972–984. https://doi.org/10.1016/j.jlamp.2016.01.002
Jones, C. B., & Misra, J. (2021). Finding effective abstractions. In C. B. Jones & J. Misra (Eds.), *Theories of programming: The life and works of Tony Hoare* (pp. 23–40). ACM. https://doi.org/10.1145/3477355
King, J. C. (1969). A program verifier. PhD thesis, Department of Computer Science, Carnegie-Mellon University.
King, J. C. (1971). A program verifier. In C. V. Freiman, J. E. Griffith, & J. L. Rosenfeld (Eds.), *Information processing, proceedings of IFIP congress 1971* (pp. 234–249). North-Holland.
Lamport, L. (1977). Proving the correctness of mutiprocess programs. *IEEE Transactions on Software Engineering, 3*, 125–143. https://doi.org/10.1109/TSE.1977.229904
Lucas, P., & Walk, K. (1969). On the formal description of PL/I. *Annual Review in Automatic Programming, 6*, 105–182.
Manna, Z. (1968). Termination of algorithms. PhD thesis, Carnegie-Mellon University. https://apps.dtic.mil/dtic/tr/fulltext/u2/670558.pdf
Manna, Z. (1969a). The correctness of non-deterministic programs. Memo AI-95, Department of Computer Science, Stanford University.
Manna, Z. (1969b). The correctness of programs. *Journal of Computer and System Sciences, 3*(2), 119–127. https://doi.org/10.1016/S0022-0000(69)80009-7
Manna, Z. (1969c). Properties of programs and the first-order predicate calculus. *Journal of the ACM, 16*(2), 244–255. https://doi.org/10.1145/321510.321516
Morris, F. L., & Jones, C. B. (1984). An early program proof by Alan Turing. *Annals of the History of Computing, 6*(2), 139–143. https://doi.org/10.1109/MAHC.1984.10017

Owicki, S., & Gries, D. (1976). Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM, 19*(5), 279–285.
Owicki, S. S. (1975). Axiomatic proof techniques for parallel programs. PhD thesis, Department of Computer Science, Cornell University, published as technical report 75-251.
Owicki, S. S., & Gries, D. (1976). An axiomatic proof technique for parallel programs I. *Acta Informatica, 6*, 319–340.
Priestley, M. (2020). Flow diagrams, assertions, and formal methods. In T. K. Astarte (Ed.), *HFM 2019—history of formal methods workshop*, Porto, Portugal, 7–11 October 2019, Revised selected papers, Part II, No. 12233. Lecture notes in computer science (pp. 15–34). Springer. https://doi.org/10.1007/978-3-030-54997-8
Reisig, W. (1985). *Petri Nets: An introduction*. Monographs in theoretical computer science (Vol. 4). Springer.
Reisig, W. (2013). *Understanding Petri Nets: Modeling techniques, analysis methods, case studies*. Springer.
Turing, A. M. (1949). Checking a large routine. *Report of a conference on high speed automatic calculating machines* (pp. 67–69). University Mathematical Laboratory.