

Tony Hoare: His Path to the ACM Turing Award

KRZYSZTOF R. APT, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands and MIMUW, University of Warsaw, Warsaw, Poland

In 1980 Tony Hoare received the ACM Turing Award “for his fundamental contributions to the definition and design of programming languages”. This article examines the achievements that led to this award.

CCS Concepts: • **Theory of computation** → **Program reasoning**; • **Software and its engineering** → **Distributed programming languages**; **Parallel programming languages**;

Additional Key Words and Phrases: Tony Hoare, period 1961-1977, ACM Turing award

ACM Reference Format:

Krzysztof R. Apt. 2026. Tony Hoare: His Path to the ACM Turing Award. *Form. Asp. Comput.* 38, 2, Article 13 (June 2026), 20 pages. <https://doi.org/10.1145/3816707>

1 Introduction

Each science is driven by its paradigms. They are formed by pioneers, who create new theories or new approaches in the still to be shaped field and who, thanks to their insights, succeed in imposing their research agenda on others. In computer science Tony Hoare was one such pioneer. Tony died on 5th of March 2026 at the age of 92. To better appreciate his influence on computer science, we provide here a chronological outline of his early contributions and assess the impact they have had on the field.

This article covers the crucial years 1961–1977, the period preceding his appointment as Professor of Computation at the University of Oxford. It starts with the publication of his Quicksort algorithm and ends with the revised version of his article titled *Communicating Sequential Processes*. During these 17 years his research was focused on various aspects of imperative and parallel programming and it is for this work that he received the ACM Turing Award in 1980.

Our aim is also to better understand the influences that shaped his work; to do so, we need to reexamine his early contributions in detail, while also taking into account the work and opinions of his peers. It should be noted that in the period under discussion Hoare wrote most of his articles alone and the number of his coauthors can be counted on one hand. His first joint publication, with Niklaus Wirth, dates from 1966.

At that time only mainframe computers existed. Computer programs were prepared on punched cards or paper tape and computer time was scarce. Information about research by others was difficult to obtain and contacts were sparse since e-mail and internet did not exist. Research articles were written on a typewriter and communication was reduced to correspondence by post since international phone calls were expensive. So participation in scientific events was crucial. Hoare

Associate Editor: Cliff Jones

Author’s Contact Information: Krzysztof R. Apt, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands and MIMUW, University of Warsaw, Warsaw, Poland; e-mail: apt@cw.nl.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 1433-299X/2026/06-ART13

<https://doi.org/10.1145/3816707>

had a rare talent to distill the essence from what he learned during these meetings. His works built upon it and showed that often he saw further than his peers. This is remarkable given that he started his research with hardly any knowledge of the budding field of computer science.

2 Quicksort and Find

In 1961, at the age of 27, Hoare published his first article, [46], while working as a programmer at a small company, Elliott Brothers, in London. It described three algorithms, Partition, Quicksort and Find. They were written in ALGOL 60, a language he learned that year during a course, with Peter Naur, Edsger Dijkstra, and Peter Landin as tutors (see [59]). Each algorithm is interesting in its own right. The sorting algorithm Quicksort uses Partition as a subroutine and is so compact that for the sake of further discussion it is worth reproducing it in full.

```

procedure quicksort(A,M,N); value M, N;
          array A; integer M, N;
begin    integer I,J;
          if M < N then begin partition(A,M,N,I,J);
                                quicksort(A,M,J);
                                quicksort (A,I,N)
          end
end      quicksort

Quicksort as presented in [46].

```

So Quicksort invokes the procedure Partition, the body of which is longer, though relatively straightforward. It suffices here to reproduce Hoare's comment about the working of this procedure.

Given an array $A[M..N]$, the call $\text{partition}(A,M,N,I,J)$ modifies the array by means of in place element swaps using a random element X (nowadays called a *pivot*) and the output parameters I and J :

$M \leq J < I \leq N$ provided $M < N$,
 $A[R] \leq X$ for $M \leq R \leq J$,
 $A[R] = X$ for $J < R < I$,
 $A[R] \geq X$ for $I \leq R \leq N$.

Interestingly, Hoare conceived Quicksort in Moscow, where he attended the Moscow State University as a visiting student in the School of Statistics [67]. It became one of the most famous algorithms in computing. It achieves sorting by means of the "divide-and-conquer" paradigm, where the "conquer" part consists of two inner calls operating on disjoint array segments. One of its novelties is the use of recursion, a feature introduced in the ALGOL 60 language almost by accident (see [29, 30]). As stated by Dijkstra in [28] "Tony Hoare's "quicksort" convinced the last doubters that the inclusion of recursion was more than a whim". But Quicksort is also a randomized algorithm since, to avoid the worst-case behaviour, the Partition call includes randomization in the choice of the pivot.

Hoare analysed the algorithm in detail in [47], though, remarkably, he described it there only in plain English. After having carried out a complexity analysis he concluded that the average number of comparisons needed to sort an array with distinct entries is $2N \ln N$, which is about 1.4 off the minimum $N \log_2(N)$. In the article, Hoare also reported that the algorithm compared favourably with the competing Mergesort, invented in 1945 by John von Neumann (see [70, p. 384]). They were both programmed "by Mr. P. Shackleton" and run on the National-Elliott 405 computer using random data produced "by Mr. D. J. Pentecost". This reference to experiments and two researchers

involved in them gives an early indication that Hoare and the Elliot Brothers company were fully aware of the importance of his discovery. Hoare also mentioned a modification of Quicksort made by Pat Shackleton, now called the Multi-key quicksort, that was the first out of many variants studied in the literature.

Starting with [70], where Donald Knuth stated that Hoare's article "contains one of the most comprehensive accounts of a sorting method that has ever been published" (p. 115), its complexity-theoretic analysis became a vibrant research topic. In 1975 Robert Sedgewick defended, under the supervision of Knuth, an award winning 361 pages-long PhD thesis [87] titled *Quicksort*, in which he undertook an extensive study of the algorithm's complexity, also analysing various variants and modifications that appeared in the meantime. These results were published in three journal articles [88–90]. Classic books on algorithms such as [13, 91], contain a chapter-long analysis of Quicksort.

Find, the third algorithm published by Hoare in his [46] article, is also recursive. Its purpose is to find the k th smallest element in an unordered array of elements. The algorithm is relevant in numerous applications in which sorting of the whole array is not needed, for instance in statistical analysis when one just wants to find a median of a dataset, or to analyse, say, the top 10% of data. Like Quicksort, Find is also a randomized algorithm, since it contains a call to Partition, but in contrast to Quicksort the average number of comparisons is linear (see, e.g., [99, p. 84]).

Unknowingly, when publishing his Find algorithm, Hoare just scratched the surface of a respectable research area that originated with a 1883 article by Charles Lutwidge Dodgson. Better known by his pen name Lewis Carroll, he proposed a modification of the knockout elimination tennis tournament to determine the true second- and third-best players, see [70, pp. 209–218]. Somehow Find also assumed a pen name, Quickselect, and as such can be found in various standard books on algorithms (see e.g., in [91, pp. 345–347]).

Both algorithms continue to be studied today (see, e.g., [45, 64]). We shall return to them in Section 4.

3 Extensions of ALGOL 60

In the period that followed his two articles, Hoare continued to work for the Elliot Brothers company, where he oversaw a successful implementation of a subset of ALGOL 60. Subsequently, as a manager of a group of eventually 40 programmers, he faced mostly unsuccessful attempts to deliver new software for the company's new computer, adventures he later described in a colourful way in his Turing Award speech [59]. In 1962 he married Jill Pym, one of the two main implementers of the ALGOL 60 subset, a marriage that was to last his lifetime.

In 1962 he was invited to become a member of a newly formed IFIP Working Group 2.1 on Algorithmic Languages and Calculi that studied maintenance, revision and extensions of the ALGOL 60 language. Several members of the group became leading computer scientists and six of them eventually went on to win the ACM Turing Award. The meetings of the group brought him into regular contact with Edsger Dijkstra and Niklaus Wirth. In the years that followed their research had a profound effect on each other's scientific work.

In this period, Hoare proposed to add to the language the **case** statement and the **record** data structure about which he wrote an extensive report [48], published three years later as [49]. This clearly caught Dijkstra's interest as demonstrated by his 1965 report in which he "experimented" (his formulation) with Hoare's **record** concept [17]. Hoare also advocated the concept of a *null reference* (a value reserved to indicate that a pointer or reference does not refer to a valid object), though later came to regret it. In a 2009 lecture he famously described it as "the billion dollar mistake".¹

¹<https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/>

In 1966 he published, jointly with Wirth, a long article [102] in which these and other additions to the language were proposed.² One of them was the **while** statement, thanks to which loops could be written without the use of the **goto** statement, which was an important contribution toward structured programming. This report formed the basis for the ALGOL W language that in turn formed the basis for the more popular Pascal language, first implemented by Wirth and his colleagues in 1970.

This period was also marked by increasingly ominous discussions within Working Group 2.1 on the successor of ALGOL 60. In 1965 a proposal consisting of an implementation of ALGOL W on the IBM 360 computer, combined with the report that became the above mentioned [102], was rejected. Subsequent repeated delays in a delivery of a report on the alternative proposal, eventually called ALGOL 68, finally led to the departure of six scientists, including Dijkstra and Hoare. Their reservations were spelled out in a harshly worded succinct “Minority Report” that criticized both ALGOL 68 and its definition method (see the English fragment of [32]). This eventually led to the creation of IFIP Working Group 2.3 on Programming Methodology of which Dijkstra, Hoare and Wirth became prominent members. Thirteen years later Hoare discussed these frustrating developments in sober terms in his Turing Award speech [59].

4 Hoare Logic

In 1968 Hoare moved to Northern Ireland where he became Professor of Computer Science at the Queen’s University of Belfast. Through his work on extensions of the ALGOL 60 language he got interested in techniques for formal definition of languages. This eventually led to his article [50] in which he introduced what is now Hoare logic. It consisted of the following axiom and proof rules.

D0 Axiom of Assignment
 $\vdash P_0\{x := f\}P$
 where
 x is a variable identifier;
 f is an expression;
 P_0 is obtained from P by substituting f for all occurrences of x .

D1 Rules of Consequence
 If $\vdash P\{Q\}R$ and $S \supset P$ then $\vdash S\{Q\}R$
 If $\vdash P\{Q\}R$ and $R \supset S$ then $\vdash P\{Q\}S$

D2 Rule of Composition
 If $\vdash P\{Q_1\}R_1$ and $\vdash R_1\{Q_2\}R$ then $\vdash P\{(Q_1; Q_2)\}R$.

D3 Rule of Iteration
 If $\vdash P \wedge B\{S\}P$ then $\vdash P\{\mathbf{while} \ B \ \mathbf{do} \ S\}\neg B \wedge P$.

Hoare’s axioms and proof rules, as presented in [50].

The statement $\vdash P\{Q\}R$ is supposed to capture the following intuition: if a program Q is started in a state satisfying P and it terminates then it reaches a state satisfying R . It is now common to write $\{P\} Q \{R\}$ instead of $\vdash P\{Q\}R$. Such a construct is nowadays referred to as *Hoare triple*, while P and R are called *assertions*, where P is the *precondition* and R the *postcondition*.

²It contains the following intriguing comment: “The concept of records has been pioneered in the AED-I language by D. T. Ross.”

Let us discuss each of the axioms and rules in turn. In the Axiom of Assignment one nowadays writes the substitution explicitly, that is, one writes

$$\{P[x := f]\} x := f \{P\},$$

where $P[x := f]$ denotes the result of substitution of f for all free occurrences of x in P .³ Next, the Rules of Consequence are currently combined into one rule and written using the notation more common in logic, so as

$$\frac{P \rightarrow P_1, \{P_1\} Q \{R_1\}, R_1 \rightarrow R}{\{P\} Q \{R\}}.$$

The Rule of Composition then takes the form

$$\frac{\{P\} Q_1 \{R_1\}, \{R_1\} Q_2 \{R\}}{\{P\} Q_1; Q_2 \{R\}}.$$

while the Rule of Iteration becomes

$$\frac{\{P \wedge B\} S \{P\}}{\{P\} \textbf{while } B \textbf{ do } S \{\neg B \wedge P\}}.$$

The Axiom of Assignment should be contrasted with a more complicated version used earlier (in a different setting) by Robert Floyd in [35]

$$\{P\} x := f \{ \exists y : (P[x := y] \wedge x = f[x := y]) \},$$

that Hoare used in the earlier, unpublished, version of his article [62]. The more elegant version he adopted in the journal article apparently appeared first in the PhD thesis of J.C. King [69] (who attributed it to his supervisor Robert Floyd, see [68, page 6]). It provides a direct link between the assignment and substitution (to the extent that we use here the same notation for both). It is achieved by reasoning “backward”, from the post- to the precondition.

Another notable item is the Rule of Iteration. It states that if an assertion P is maintained during each activation of the loop body and is true at the beginning of the loop, then it is also true upon its termination. The assertion P is called a *loop invariant*. The idea of using invariants to reason about programs was already present in earlier works by Alan Turing [95] (see also [80]), Naur [81] and Floyd [35]. However, Hoare’s use of them is much more succinct, since they are only associated with the **while** loops. Incidentally, Hoare learned of Floyd’s article [35] only after having distributed the first draft of his article (see [66, page 27]).

In his 2015 interview with Cliff Jones, Hoare explained that the motivation for his axiomatic approach was to provide a third alternative to two approaches of describing a programming language, none of which appealed to him: denotational semantics and operational semantics (see [65, page 361]). This approach became known as Hoare logic.

5 Hoare Logic for Procedures and Program Development

The reason for the success of Hoare’s article was that it spelled out in simple terms an axiomatic basis for reasoning about programs that was of interest both for researchers analysing programming constructs and for logicians who got interested in standard logical questions concerning the introduced proof systems. The simplicity and elegance of this formalism made it possible to adjust and extend it to practically all types of programming constructs used in imperative programming languages.

Hoare himself took the lead in this research. In the original article he illustrated usefulness of his proof system by presenting an axiomatic correctness proof of a toy example originally used by

³There is no standard notation for writing substitution. In later publications Hoare used P_f^x .

Floyd in [35]. In the subsequent articles on the subject he took upon himself a more ambitious task: to verify the original three algorithms he published in his first article [46].

In [51] he dealt with `Find`, which he rewrote as a nonrecursive program with a triply nested **while** loop. He established its correctness by systematically developing it together with its correctness proof and augmented it by an informal proof of termination (that could not be reasoned about in his logic).

Hoare acknowledged the influence of Dijkstra by citing his “privately circulated” 1970 lecture notes on structured programming [19]. But the influence went both ways. It is quite likely that this article encouraged Dijkstra to study closer systematic development of programs together with their correctness proofs, which eventually led to his 1976 book [25]. However, Dijkstra’s appreciation was not immediate. In 2001, he admitted in [31]:

In the early 70s we studied Tony Hoare’s correctness proof of the routine “`FIND`” and my friends still remember that at the end of that study I declared in disgust that “that meaningless ballet of symbols was not my cup of tea”; only gradually, my resistance to that kind of formula manipulation faded, until eventually, I began to love it.

Hoare’s proof was indeed not easy to follow and he speculated that “In future, it may be possible to enlist the aid of a computer in formulating the lemmas, and perhaps even in checking the proofs.” Several years later Jean-Christophe Filliâtre did exactly that and justified in [34] Hoare’s proof formally using the interactive theorem prover Coq. He also ascertained that Hoare’s reasoning about termination had to be modified.

The `Find` algorithm could be rewritten as a nonrecursive program because it only used so-called left recursion. But reasoning about `Quicksort` required new insights, since here recursion was essential. An appropriate rule for dealing with it was proposed in 1971 by M. Foley and Hoare in [36]. It was modelled after so-called Scott’s induction rule that appeared in a widely circulated, unpublished and handwritten, article by Dana Scott and Jaco de Bakker [86], a rule that in turn had its roots in Gerhard Gentzen’s 1934 natural deduction system. It is not clear whether Foley and Hoare were familiar with these references.

Scott’s induction was concerned with parameterless procedures expressed in relational calculus, while Hoare and Foley dealt with procedures with parameters, written in an imperative language. So to draw conclusions about specific procedure calls, additional rules were added. The article provided a correctness proof of `Quicksort`, but just as in the case of the `Find` program the proof was not easy to read. Several years later, this motivated Krzysztof R. Apt, Frank de Boer and Ernst-Rüdiger Olderog to provide a more modular presentation in their book [1], including the previously missing termination proof.

`Quicksort` can be elegantly programmed in functional and logic programming languages (see, e.g., [94] and [93]). This is occasionally used as an argument for the superiority of these programming styles, since the corresponding correctness proofs are substantially simpler. However, these versions of the program do not perform sorting *in situ*, an important feature of the algorithm already mentioned in [47], so they cannot be considered as faithful counterparts of the original program.

Foley and Hoare’s article was also remarkable for other insights. Given that the formal proof was tedious, the authors speculated about the possibility of having a “mechanical means to generate these lemmas from the text of the program”. The following striking fragment that includes a discussion of the Assignment axiom may raise eyebrows:

For each command type there is the rule of inference permitting the mechanical construction of the “weakest” proposition, which is “provably” true before execution of the command if a certain proposition is true after it.
[...] Call it [the weakest proposition] “Precondition”.

It clearly alludes to the weakest precondition semantics proposed four years later by Dijkstra in [22]. This semantics assigns to a given a program S and postcondition Q the set of states $wp(S, Q)$ from which each execution of S is guaranteed to terminate in a state that satisfies Q . In this notation we have $wp(x := f, P) = P[x := f]$ (where we identify the assertion with the set of states satisfying it). In his article, Dijkstra considered nondeterministic programs that in turn influenced Hoare; more about this later, in Section 8.

Two other articles by Hoare from this period testify to his growing interest in constructing programs together with their correctness proofs. In [52] he developed in this way a program that implements the sieve of Eratosthenes for finding all primes smaller than a given natural number. This program was informally developed earlier by Dijkstra in his 1970 lecture notes on structured programming [19]. Remarkably, Hoare's article does not refer, even in the references, to his proof system, though references to loop invariants abound. He also used the class concept of Ole-Johan Dahl's SIMULA 67 programming language to separate the abstract algorithm from the proof of the concrete representation. Next, in [53] he suggested that the use of data abstraction can be combined with his approach to program verification. To make his point he again used the class concept and showed how correctness of a program that operates on several small sets of integers using the remove, insert and test operations can be established by reasoning first about its abstract version.

Clearly, Hoare became an early advocate of object-oriented programming, in contrast to Dijkstra who never overcame his early aversion to it, exemplified in 1975 by this revealing confession (see [23]):

It all started on Sunday 27th of July 1975, when Tony Hoare explained to me in the garden of Hotel Sepp in Marktoberdorf (Western Germany) upon my request the class-concept of SIMULA (including the so-called inner-concept); at least he explained his version of it. I had always stayed away from it as far as possible, in order to avoid contamination with the extremely operational point of view as practiced by Dahl c.s., and, after some time I could not even (under-) stand their mechanistic descriptions anymore: they just made me shudder.

During this period Dijkstra got interested in turning programming into a systematic activity and published in 1971 his lecture notes *A Short Introduction to the Art of Programming* [21] (the cover mentions the price of 4 guilders) in which he devoted a chapter to Hoare's Partition program rewritten using the **while** statement. Wirth's similar interests led him to publish, in 1971, a paper titled *Program Development by Stepwise Refinement* in which he acknowledged "stimulating influence of many discussions with C.A.R. Hoare and E.W. Dijkstra" [96]. Clearly, the three pursued a similar research agenda in this period, though each of them carried it out by different means.

Dijkstra's aversion to object-oriented programming did not prevent him from contributing to an important milestone of this period, the highly influential book *Structured Programming* [14] that consisted of three long chapters written by Dijkstra, Hoare, and Hoare with Ole-Johan Dahl. The opening chapter by Dijkstra was an expanded version of his already mentioned lecture notes on structured programming. Hoare's chapter titled *Notes on data structuring* discussed the use of various data types, including enumeration type, record, powerset, sequence, and recursive types (with the family tree as an example), together with their representations. The final chapter dealt with the use of the class concept in the SIMULA 67 language and was perhaps the first exposition of object-oriented programming. Hoare later modestly stated that his role was limited to simplifying Dahl's somewhat "dense" presentation (see [92]).

Hoare and Wirth's mutual influence in this period is amply documented. Wirth ended his 1971 article [97] on the presentation of Pascal with this acknowledgement:

The author gratefully acknowledges his indebtedness to C. A. R. Hoare for his many valuable suggestions concerning overall design strategy as well as details, and for his critical scrutiny of this article.

A year later Hoare returned the compliment by ending his above mentioned chapter on data structuring with:

I am also deeply indebted to Professor N. Wirth for many fruitful discussions and suggestions, and for his willingness to test several of the ideas of the paper in his design and implementation of PASCAL, and to Professor E. W. Dijkstra for his perpetual inspiration.”

In 1973, an important book by Wirth appeared; an introductory textbook [98] on systematic programming in Pascal. He introduced in it Hoare’s axioms and proof rules switching to the now common Hoare triple notation, $\{P\} Q \{R\}$. This allowed him to annotate his programs with pre- and postconditions and loop invariants. In 1976, in another book, [99], he included a detailed exposition of Quicksort and Find and also discussed recursive types which he illustrated using Hoare’s family tree example, all rewritten as Pascal programs.

Also in 1973, Hoare kept in mind his original motivation for creation of Hoare logic and published with Wirth a substantial article on the axiomatic definition of a subset of Pascal [61], an approach later adopted in [73] to define the programming language Euclid. Data types were defined by a mixture of logical axioms and comments in English, while program statements were defined using Hoare logic. Hoare and Wirth expressed hope that such a definition may serve as “a reference manual for programmers [and] an axiomatic basis for formal proofs of properties of programs.”

But to justify such expectations the status of the introduced axioms and rules had to be clarified. This undoubtedly motivated Hoare to undertake investigation of his logic from the semantic point of view. In [60], a paper with his first PhD student Peter Lauer, soundness of the original proof system was established with respect to the semantics of programs formulated in relational calculus.⁴

After these seminal publications by Hoare research on Hoare logic took off. We shall discuss these developments in Section 9. Another line of research in Hoare logic, also initiated by Hoare, was related to his study of parallel programs, a subject that we have to discuss first.

6 Parallelism: Conditional Critical Regions

To explain Hoare’s contributions to parallelism, we need to turn our attention briefly to Dijkstra, who single handedly created a theory of parallel processes by publishing his 87-page long report titled *Cooperating Sequential Processes* [16] in 1965. In it he identified the crucial synchronization problem, called by him *the mutual exclusion problem*, and also introduced the first known synchronization primitive, which he termed a *semaphore*. This originated research on high-level synchronization constructs.

In 1971, Dijkstra introduced in [20] the classic synchronization problem, called the Five Dining Philosophers, and provided a solution to it using semaphores. In a 2001 interview he mentioned that he originally used the name of the Dining Quintuple and that the currently used name was conceived by Tony Hoare (see [37]). This problem became a benchmark for presenting new solutions to synchronization. We shall return to it below. In his article, Dijkstra also suggested, without providing syntactic details, a concept of *secretaries*, with an intuition that it is “primarily a bunch of non-reentrant routines with a common state space”.

So Hoare must have closely followed this work by Dijkstra and undoubtedly thanks to him got interested in parallelism. His first article on this subject, [54], appeared in 1972. In it he introduced a simple to manage form of parallelism, which he called *disjoint parallelism*, and the by now

⁴The authors referred to ‘consistency’ but ‘soundness’ is a more appropriate term here.

standard notation

$$\{Q_1 \parallel Q_2 \parallel \dots \parallel Q_n\},$$

indicating that the individual component programs $Q_1, Q_2, \dots, Q_n$, called processes, are executed in parallel. *Disjointness* meant that no variable subject to change in one process was referred to in another one. So the processes could only share “read only” variables. As an example he proposed a natural parallel version of Quicksort in which the inner calls (that operate on disjoint segments of the array) are processes executed in parallel.

From the perspective of time the most important contribution of the article was a suggestion to control interaction between the parallel processes using appropriate high-level features. His idea was that such constructs should prevent time-dependent errors by means of compile-time checking, which was novel at the time. He also proposed the first such feature, a shared *resource* with a *conditional critical region* statement regulating access to it. According to Hoare a resource r is a set of shared variables that can be accessed only by means of conditional statements of the form

with r when B do C ,

where B is an optional Boolean expression and C is a statement that uses the variables of r . It has the following interpretation. When a process attempts to execute such a statement it is delayed until the condition B is true and the resource r is not being used by another process. When both conditions are satisfied the statement C is executed. Upon termination the resource r is available for other processes. When several processes are competing for a particular resource no assumption is made about the order in which they acquire it.

In the article, he illustrated the use of conditional critical regions by a couple of elegant examples, including a simple solution to Dijkstra’s Five Dining Philosophers problem. To clarify the use of conditional critical regions let us present Hoare’s solution.

The problem involves five philosophers sitting around the table. In the middle there is a bowl of spaghetti. On the table five forks are placed, one to the left and one to the right of each philosopher. In order to eat spaghetti, each philosopher needs two forks. Their lives alternate between eating and thinking. The problem is to synchronize their actions so that each philosopher eventually can eat. In particular one should prevent the situation in which each philosopher takes just one fork.

Hoare’s program uses as a shared resource an integer array *possforks*[0 : 4], where *possforks*[i] indicates the number of forks available to philosopher i . He can eat spaghetti when both of his neighbours are not eating, which is modelled by the condition *possforks*[i] = 2. Taking (respectively, releasing) forks by philosopher i is modelled by decreasing (respectively, increasing) the number of forks available to his neighbours, $i \ominus 1$ and $i \oplus 1$, by 1, where $\ominus$ and $\oplus$ indicate counting modulo 5. The program for philosopher i is given below.

```

with possforks when possforks[i]= 2 do
    begin possforks[i  $\ominus$  1]:= possforks[i  $\ominus$  1] - 1;
        possforks[i  $\oplus$  1]:= possforks[i  $\oplus$  1] - 1
    end;
eat spaghetti;
with possforks do
    begin possforks[i  $\ominus$  1]:= possforks[i  $\ominus$  1] + 1;
        possforks[i  $\oplus$  1]:= possforks[i  $\oplus$  1] + 1
    end;
think

```

Hoare’s solution to the Dining Philosophers problem presented in [54].

One of the first researchers who recognized the importance of Hoare's proposal was Per Brinch Hansen who soon after published an article [39] on comparison of semaphores and conditional critical regions in which he wrote about a solution to a resource allocation problem:

A fairly long argument is needed to justify the solution which uses semaphores. In contrast, the solution based on conditional critical regions is so obvious that a proof is hardly needed.

Thirty years later he published an anthology of, in his view, the 19 most important articles on concurrent programming [43] in which he included this article by Hoare. In the 60-page introduction he stated (*italics in the original*):

This is *the first notable attempt to extend programming languages with abstract features for parallel programming.*

and reproduced Hoare's opinion that the search for such features is "one of the major challenges to the invention, imagination and intellect of Computer Scientists of the present day."

Hoare himself was much more modest. Even though, remarkably, he did not cite any references, in the acknowledgements he attributed the concept of critical region to Dijkstra, perhaps by viewing it as a realization of Dijkstra's secretary concept. He also wrote:

Less obvious but equally invaluable has been his [Dijkstra's] constant encouragement in the search for a concept to "replace" the semaphore in a high-level programming language, and my ambition of meeting his high standards of rigour and programming style.

Hoare pursued this study in [57], the first version of which appeared as a Stanford University report in 1973. In it he identified three ways of extending disjoint processes, to quote:

- (1) Competing processes, which require exclusive use of some global resource during certain phases of their execution.
- (2) Co-operating processes, each of which makes its contribution to the construction of some final desired result.
- (3) Communicating processes, which transmit information to each other at intermediate stages in their progress.

Communicating processes soon became a central focus of Hoare's research. Admittedly, the initial ideas presented in this article were still quite preliminary. In particular, while the article proposed a two-way communication between processes, it did not yet consider simultaneous communication—a concept that would become crucial in his later work that we shall discuss in Section 8.

7 Parallelism: Monitors

Hoare's proposal of critical regions was very simple and elegant, but at the cost of efficiency. In particular, each delayed process would have to keep reevaluating the condition that caused its suspension and no scheduling of access to resources was possible. Further, resources and operations associated with them could not be grouped into one module. This led to a search for more sophisticated constructs that could remedy such deficiencies, in particular by allowing the processes to have some influence on the scheduling of delayed processes.

This eventually led to the invention of a *monitor*. It is a collection of procedures, variables, and data structures that are grouped together in a class. Processes can synchronize their actions exclusively by calling the monitor procedures. However, for each monitor at all times at most one process can enter a monitor procedure. To achieve synchronization within the monitor procedures *condition variables* are used with two operations on them, *wait* and *signal*. The *wait* operation results

in suspending the process, while the *signal* operation results in resuming one of the suspended processes. This is the essence of the monitor concept as described and illustrated in Hoare's 1973–74 articles [55] and [56].

Who exactly invented monitors is not obvious at first glance. Hoare's first article did not use the word "monitor", while he began the second article with the following sentence: "This paper develops Brinch Hansen's concept of a monitor as a method of structuring an operating system". In the article, he cited Brinch Hansen's 1972 article [40] and 1973 book [41]. In the acknowledgements he wrote

The development of the monitor concept is due to frequent discussions and communications with E.W. Dijkstra and P. Brinch Hansen.

which is consistent with what Brinch Hansen wrote 20 years later in [42]:

The idea of *monitors* evolved through discussions and communications between E.W. Dijkstra, C.A.R. Hoare, and me during the summer and fall of 1971.

Brinch Hansen's memoirs [44, page 92] make clear that he referred to the Marktoberdorf Summer School that was followed by a Seminar on Operating Systems Techniques held at the Queen's University in Belfast. About the latter he wrote: "The threat of terrorist bombings had discouraged several contributors, including Dijkstra, from attending the seminar. At the time, bombings were frequent in downtown Belfast". This may explain why certain aspects of Hoare's monitor proposal were unfamiliar to Dijkstra, as reflected in his analysis of it in [24], in which—one should note—he did not mention his idea of secretaries.

In 2002 Brinch Hansen traced, in the introduction to his book [43], the history of monitors and wrote about Hoare's [56] article (*italics in the original*):

This influential paper deserves a place in history of concurrent programming as *the first monitor tutorial*

subsequently explaining the differences between Hoare's proposal and his earlier proposals.

Hoare's comments in his 2015 interview make clear that they both invented the concept of a monitor (see [65, page 373]). In the literature the idea is attributed both to Brinch Hansen and Hoare. It became undoubtedly the most popular synchronisation concept and its proposal was followed by intense research on languages for concurrent programming. Already in 1989 a survey [6] appeared that mentioned nearly 100 programming languages. The widely used Java and C# languages use monitors for synchronization of parallel processes. Hoare's idea of shared resources with the corresponding conditional critical region statement was the first stepping stone in this research. Also, as shown in [85], it turned out to be a natural abstraction for implementing monitors.

8 Communicating Sequential Processes

To explain the next seminal contribution of Hoare, his 1978 (but submitted and revised in 1977) article [58] titled *Communicating Sequential Processes*, we need first to explain some background. The title is a clear reference to Dijkstra's ground breaking *Cooperating Sequential Processes* article [16]. The proposed language, called in short CSP, is an outcome of three ideas, two due to Hoare and one due to Dijkstra. These are: disjoint parallelism, guarded commands and synchronous communication.

The first one has already been discussed. The second one was introduced by Dijkstra in [22]. In this article he introduced a small programming language, called *guarded commands* and provided for it the already mentioned weakest precondition semantics. The language contained two non-deterministic constructs, the *alternative construct* and the *repetitive construct*, written in the CSP syntax respectively as

$$[B_1 \rightarrow S_1 \square \dots \square B_n \rightarrow S_n],$$

and

$$*[B_1 \rightarrow S_1 \square \dots \square B_n \rightarrow S_n],$$

where $B_1, \dots, B_n$ are Boolean expressions called *guards* and $S_1, \dots, S_n$ are program statements.

In the alternative construct the statement S_i can be chosen only if its guard B_i evaluates to true; then S_i remains to be executed. Crucially, if more than one guard B_i evaluates to true then *any* of the corresponding statements S_i may be executed next. If all guards evaluate to false, the alternative construct terminates in failure. The repetitive construct is the loop around the corresponding alternative construct with the important difference that when all guards evaluate to false, the repetitive construct properly terminates. Guarded commands lead to what is now called *demonic nondeterminism*. Informally, it states that *all* computations of a nondeterministic program have to deliver the desired result.

The third crucial idea in CSP consisted of *synchronous communication*, according to which the sender of a message can deliver it only when the receiver is ready to accept it at the same moment. Syntactically, it is achieved by means of *input* and *output commands* that specify communication between two concurrent processes.

The input command, written as $P?x$, where P is the name of a process and x is a variable, expresses the request to receive from the process P a value that upon reception is assigned to x . The output command $Q!t$ expresses the readiness to send to the process Q the value of the expression t . When in such a pair of input and output commands the processes address each other and the types of x and t match, the input and output commands are said to *correspond*. They are then executed simultaneously and their combined effect is that of the assignment $x := t$. Each input/output command is delayed until a corresponding command is reached.

In CSP the processes are simply guarded command statements in which the input and output commands are allowed as atomic statements. Additionally, variable declarations and, more crucially, input commands can appear in guards of the alternative and repetitive commands. A CSP program is a collection of named disjoint processes.

To illustrate the elegance and simplicity of the resulting language let us consider just one simple example from Hoare's article. The problem is to write a process X that copies characters output by process *west* to process *east*, where every pair of consecutive asterisks “**” should be replaced by an upward arrow “↑”. It is assumed that the final character input is not an asterisk. The solution is given below.

```
X:: *[c : character; west?c →
    [c ≠ * → east!c
    □ c = * → west?c;
    [c ≠ * → east!*; east!c
    □ c = * → east! ↑
    ]
  ]
]
```

Hoare's CSP solution to a copy/squash problem as presented in [58].

Synchronous communication is sometimes called *handshake communication*. It had an important impact on distributed programming, which is concerned with parallel processes sharing information exclusively by means of message passing. It is a bold abstraction from *asynchronous communication*, according to which the sender can always send its message. This assumes the existence of a buffer where messages are kept until the receiver collects them.

The CSP proposal had substantial impact on subsequent developments in distributed computing. On the theoretical side it led to a number of interesting research problems, in particular how to implement output commands in guards (see, e.g., [7]) and how to ensure global termination when all processes terminate communication but cannot exit their repetitive command loops on their own. This convention, identified by Nissim Francez in [38] and called *distributed termination*, occasionally results in simpler CSP programs.

On the practical side CSP led to the development of the programming language occam (see [74] and especially [75] for a historical retrospective). It was specifically designed for the computer architecture developed at the INMOS company, based on transputers, chips that contained a processor, cache memory and communications hardware. In occam communication is synchronous and takes place through point-to-point channels that need to be declared.

The idea of synchronous communication through channels also caught the attention of Robin Milner. In his 1978 article [78] on processes and synchronous communication he cited Hoare's 1976 original technical report on CSP and Milne and Milner's 1977 technical report on a mathematical model of such processes; the latter was eventually published as [77].

In 2010, in a article [9] co-authored with Mikkel Bundgaard for a volume celebrating Tony Hoare's 75th birthday, Robin Milner wrote:

Tony Hoare and I have exchanged ideas on concurrent processes for three decades. To a sufficiently distant observer it has seemed that we were doing the same thing, and that therefore we should not have made it look different. A closer view shows this to be false.

Complementary things, yes; and both have been enriched by the cross-fertilisation.

and clarified that (*italics in the original*) “he was keen to find a mathematical concept of *process* that could stand in analogy with the familiar notion of (single valued) *function*.”

This notion of a process led to his Calculus of Communicating Systems (CCS) [79]. In contrast, Hoare's original motivation was to design an elegant programming language for distributed computing. Hoare's and Milner's interests started to converge later, when Hoare embarked on his research on an algebraic version of CSP, initially put forward in [8]. The account of these connections falls outside of scope of this article.

9 Hoare Logic: Further Developments

By its very nature Hoare logic developed into an open-ended subject. We will now briefly mention the main developments that followed Hoare's initial contributions.

One line of research focussed on incorporating reasoning about termination into his logic. It culminated in 1976 in an elegant rule by Susan Owicki and David Gries [82] in which simply two new premises were added to the original Rule of Iteration. They state that an integer expression, called a *bound function*, decreases with each loop iteration and that it remains nonnegative, both under the assumption that the precondition $P \wedge B$ holds. The rule takes the following form

$$\frac{\begin{array}{l} \{P \wedge B\} S \{P\}, \\ \{P \wedge B \wedge t = z\} S \{t < z\}, \\ P \wedge B \rightarrow t \geq 0 \end{array}}{\{P\} \textbf{while } B \textbf{ do } S \textbf{ od } \{\neg B \wedge P\}}$$

where t is an integer expression and z is a fresh integer variable.

Another line of research was concerned with the logical status of Hoare logic. It was initiated by the already mentioned article [60] of Hoare and Lauer in which the soundness of Hoare's original proof system was established. In 1978 Stephen Cook introduced in [12] an appropriate notion of relative completeness, now called *Cook's completeness*, and established this form of completeness

for Hoare logic for a programming language that contained blocks and non-recursive procedures. Several articles on the subject followed, culminating in Edmund Clarke's seminal article [11] in which a combination of programming language features was identified for which it is impossible to obtain a Hoare logic that is sound and Cook complete.

The third line of research concerned extension of Hoare logic to parallel programs and was, again, initiated by Hoare. In his article [54] on disjoint parallelism and conditional critical regions he introduced appropriate proof rules in his logic that aimed at establishing correctness of the corresponding programs. He pursued this study in [54], where he admitted that his initial proof rule concerned with critical regions was "quite inadequate to prove cooperation in achieving any goal".

The appropriate solution required fundamentally new insights, in particular to deal with arbitrary parallel programs. The breakthrough was achieved in 1976 by Owicki and Gries who in [82] introduced the crucial concept of *interference freedom* between correctness proofs of the component programs. The key observation was that proofs in Hoare logic can be presented as *proof outlines*, which are—informally—programs appropriately interleaved with assertions. Interference freedom formalizes the informal requirement that the proof outlines *remain* valid proofs in the presence of other components.

A similar approach was introduced independently by Leslie Lamport in [71] in the context of parallel programs described by means of flowcharts, but the Owicki and Gries method was better amenable to reason about structured programs. This allowed them to show in [83] how their approach can be used to reason about parallel programs with Hoare's critical regions. To illustrate the usefulness of their approach they provided a correctness proof of Hoare's [54] solution to the Dining Philosophers problem written using shared resource and conditional critical regions. Their article thus extends Hoare logic to prove correctness of a program he himself developed.

An appropriate approach to reason about correctness of CSP programs was proposed independently by Krzysztof R. Apt, Nissim Francez and Willem-Paul de Roever in [2] and by Gary Levin and David Gries in [72], where the concept of *cooperation* between proof outlines was introduced. The cooperation test formalizes in the context of CSP programs, the requirement that the proof outlines *become* valid proofs in presence of other components.

Hoare logic also influenced the way programs were discussed and presented. As already mentioned, Wirth's 1973 book [98] discussed Hoare logic. It helped to spread the idea that programs should be presented together with appropriate assertions, notably with loop invariants. This idea was developed further by Bertrand Meyer who introduced in [76] the concept of *design by contract* for his object-oriented programming language Eiffel. A contract is a specification in the form of assertions (class invariants and pre- and postconditions for each method) and serves as an agreement before an implementation is developed.

Starting with [15] several books were published that discussed the theory behind Hoare logic or had a chapter on it. In 1991 a book exclusively devoted to the subject appeared [4]. It covered, among others, Hoare logics for parallel programs and CSP programs. Its third edition, [1], also dealt with recursive procedures and object-oriented programs. The state of the art in this area was discussed in a number of surveys. The last one, from 2019, aptly titled *Fifty years of Hoare's logic* [5], has more than 200 references. Since then articles related to Hoare logic continue to appear. Just to mention two recent references from the past two years: in [63] an extension of Hoare logic was developed to verify quantum error correction programs and in [10] it was extended to model and verify security, real-time, and mobile features of Internet of Things systems.

But there is more to report. With the advent of automated theorem provers it became possible to mechanically verify correctness proofs based on Hoare logic, a development foreseen by Hoare. The most prominent effort was undertaken by researchers of the Why3 platform built for deductive

program verification (see, e.g., [33]). The accompanying website <https://toccata.gitlabpages.inria.fr/toccata/gallery/why3.en.html> lists more than 200 programs mechanically verified using Hoare logic. Among them are Hoare's original three algorithms. So it is fair to say that Hoare's original article set a fruitful research agenda for more than 50 years.

10 Summary

We discussed here Tony Hoare's scientific contributions in the period 1961–1977. They show his remarkable talent for developing innovative algorithms, combined with a sustained effort to find appropriate ways to reason about them, and—separately—with a search for apposite abstractions to deal with the complexities of parallel programming.

The above account makes clear that Hoare's research agenda and opinions in the period under discussion were greatly influenced by his colleagues and friends, Niklaus Wirth and Edsger Dijkstra, and to smaller extent by Robert Floyd, Ole-Johan Dahl, Per Brinch Hansen and Robin Milner, all but one eventual recipients of the ACM Turing Award. This influence was often mutual.

Hoare's interaction with Niklaus Wirth led to two substantial articles and their close scientific interaction had a visible effect on the development of Pascal. Wirth clearly appreciated Hoare's quest for elegance and simplicity. In his account of the history of Pascal [101] he defended Pascal's omissions with respect to ALGOL 60 by citing Hoare's remark:

A language is characterized not only by what it permits programmers to specify, but even more so by what it does not allow.

Hoare's archive <https://www.cs.ox.ac.uk/people/publications/date/Tony.Hoare.html> contains Wirth's 1976 four-page long letter [100] in which he discussed in detail Hoare's CSP proposal. In particular, he wrote:

It is both fascinating and aggravating. The former one when one recognises the appealing simplicity of many concepts that hitherto seemed complicated, the latter when one is forced to give up ideas that one had just been getting familiar and comfortable with.

But, as should be clear from this account, Hoare's relationship with Dijkstra stands out. Dijkstra quickly developed a great deal of respect for Hoare. Already in 1970 in a letter, in which he reported that he lost his copy of Hoare's article on the Find program, he wrote to him: "Yours are fine shoulders to stand upon!". The following year, as already mentioned, he devoted one of the chapters of his lecture notes [21] to Hoare's Partition program. A search for "Hoare" in the E.W.D. Dijkstra Archive <https://www.cs.utexas.edu/~EWD/> yields the astonishing number of 704 hits. Also, eight of Dijkstra's EWD reports bear Hoare's name in their title. The interaction in the other direction is more difficult to trace though fortunately, Hoare was always detailed in his acknowledgements. Yet, despite interacting regularly for 40 years, they never wrote a joint article.

Hoare, like many others, learned from Dijkstra the importance of concurrent programs and the elegance of guarded commands, but like nobody else he was able to build on these ideas to make new, lasting contributions.

11 Postscript

Let me finish this article on a personal note. In my research in computer science I have been mostly influenced by Tony's works, though I have to admit that I never discussed them with him. It could have been because of our age difference (15 years), but also because of our different background, education and career paths.

In 2021 I undertook with Tony editing of a book dedicated to Edsger Dijkstra. It took us a year to complete it. It is more than 500 pages long and contains 31 contributions by leading computer scientists, including a moving chapter by Tony titled *Forty Years with Edsger*, and four other

contributions by Turing award winners (see [3]). Tony was then 88. While cooperating with him I realized how gentle a person he was and how important Edsger and his works were for him. This article is in turn my attempt at conveying my thanks to Tony for his contributions to our field. I hope to have made it clear that their impact continues to this day.

Acknowledgements

Tony Hoare's life and scientific career are well known thanks to a number of interviews he gave and to his remarkably candid Turing Award speech. This made my work substantially easier. Still, this article would not be possible without the help and encouragement of Cliff Jones, who in particular made me aware of the website <https://www.cs.ox.ac.uk/people/publications/date/Tony.Hoare.html> with all Tony's publications. Also, two books on Tony Hoare that he coedited, [84] and [67], were an invaluable source of information. The final version benefited from careful readings by two referees and from useful comments by Rocco De Nicola and Leslie Ann Goldberg. Investigation of Hoare's influence on Dijkstra's research was greatly helped by the E.W. Dijkstra Archive <https://www.cs.utexas.edu/~EWD/> conceived in 1989 by Ham Richards. Joanne Allison kindly proofread this article. She also suggested the title.

I learned of Hoare's Quicksort and Find algorithms in 1976 from Niklaus Wirth's book [99]. I was introduced to Hoare logic in 1975 by Jaco de Bakker and to CSP in 1980 by Nissim Francez and Willem-Paul de Roever. My understanding of Hoare logic greatly profited from the longstanding cooperation with Ernst-Rüdiger Olderog.

References

- [1] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. 2009. *Verification of Sequential and Concurrent Programs* (3rd ed.). Springer. DOI: <https://doi.org/10.1007/978-1-84882-745-5>
- [2] Krzysztof R. Apt, Nissim Francez, and Willem P. de Roever. 1980. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems* 2, 3 (1980), 359–385. DOI: <https://doi.org/10.1145/357103.357110>
- [3] Krzysztof R. Apt and Tony Hoare (Eds.). 2022. *Edsger Wybe Dijkstra: His Life, Work, and Legacy*. ACM Books, Vol. 45. ACM / Morgan & Claypool. DOI: <https://doi.org/10.1145/3544585>
- [4] Krzysztof R. Apt and Ernst-Rüdiger Olderog. 1991. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, New York. Retrieved from <https://link.springer.com/book/10.1007/978-1-4757-4376-0>
- [5] Krzysztof R. Apt and Ernst-Rüdiger Olderog. 2019. Fifty years of Hoare's logic. *Formal Aspects of Computing* 31, 6 (2019), 751–807. DOI: <https://doi.org/10.1007/s00165-019-00501-3>
- [6] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. 1989. Programming languages for distributed computing systems. *ACM Computing Surveys* 21, 3 (1989), 261–322. DOI: <https://doi.org/10.1145/72551.72552>
- [7] Arthur J. Bernstein. 1980. Output guards and nondeterminism in “Communicating sequential processes”. *ACM Transactions on Programming Languages and Systems* 2, 2 (1980), 234–238. DOI: <https://doi.org/10.1145/357094.357101>
- [8] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. 1984. A theory of communicating sequential processes. *Journal of the ACM* 31, 3 (1984), 560–599. DOI: <https://doi.org/10.1145/828.833>
- [9] Mikkel Bundgaard and Robin Milner. 2010. Unfolding CSP. In *Reflections on the Work of C. A. R. Hoare, A. W. Roscoe, Clifford B. Jones, and Kenneth R. Wood* (Eds.). Springer, 213–228. DOI: https://doi.org/10.1007/978-1-84882-912-1_10
- [10] Ningning Chen and Huibiao Zhu. 2026. A proof system for the SMrCaIT calculus. *ACM Transactions on Embedded Computing Systems* 25, 2 (2026), Pages 1–43. DOI: <https://doi.org/10.1145/3701729>
- [11] Edmund M. Clarke. 1979. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *Journal of the ACM* 26, 1 (1979), 129–147. DOI: <https://doi.org/10.1145/322108.322121>
- [12] Stephen A. Cook. 1978. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing* 7, 1 (1978), 70–90. DOI: <https://doi.org/10.1137/0207005>
- [13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2012. *Introduction to Algorithms* (4th ed.). MIT. Retrieved from <http://mitpress.mit.edu/books/introduction-algorithms>
- [14] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. 1972. *Structured Programming*. A.P.I.C. Studies in Data Processing, Vol. 8. Academic Press. Retrieved from <https://dblp.org/rec/books/daglib/0070495.bib>
- [15] Jacobus W. de Bakker. 1980. *Mathematical Theory of Program Correctness*. Prentice Hall. Retrieved from <https://dblp.org/rec/books/daglib/0002432.bib>

- [16] Edsger W. Dijkstra. 1965. Cooperating Sequential Processes. (1965). Retrieved April 19, 2026 from <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>. Published as [18].
- [17] Edsger W. Dijkstra. 1965. An Experiment with the “Record Class” as Suggested by C.A.R. Hoare. (1965). Retrieved April 19, 2026 from <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD132.PDF>. Circulated privately.
- [18] Edsger W. Dijkstra. 1968. Cooperating sequential processes. In *Proceedings of the Programming Languages: NATO Advanced Study Institute*. F. Genuys (Ed.), Academic, 43–112. Retrieved from <https://research.tue.nl/en/publications/co-operating-sequential-processes/>
- [19] Edsger W. Dijkstra. 1970. Notes on Structured Programming. (April 1970). Retrieved April 19, 2026 from <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>. Circulated privately.
- [20] Edsger W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Informatica* 1 (1971), 115–138. DOI : <https://doi.org/10.1007/BF00289519>
- [21] Edsger W. Dijkstra. 1971. A Short Introduction to the Art of Programming. (Aug. 1971). Retrieved April 19, 2026 from <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD316.PDF>. Circulated privately.
- [22] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18, 8 (1975), 453–457. DOI : <https://doi.org/10.1145/360933.360975>
- [23] Edsger W. Dijkstra. 1975. A Synthesis Emerging? (Aug. 1975). Retrieved April 19, 2026 from <http://www.cs.utexas.edu/users/EWD/ewd05xx/EWD508.PDF>. Published as [26].
- [24] Edsger W. Dijkstra. 1975. Variations on a theme: An open letter to C.A.R. Hoare. (July 1975). Retrieved April 19, 2026 from <http://www.cs.utexas.edu/users/EWD/ewd05xx/EWD501.PDF>. Published as [27].
- [25] Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall. Retrieved from <https://www.worldcat.org/oclc/01958445>
- [26] Edsger W. Dijkstra. 1982. *A Synthesis Emerging?* Springer New York, New York, NY, 147–160. DOI : https://doi.org/10.1007/978-1-4612-5695-3_26
- [27] Edsger W. Dijkstra. 1982. Variations on a theme: An open letter to C.A.R. Hoare. In *Proceedings of the Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 132–140. Retrieved from https://link.springer.com/chapter/10.1007/978-1-4612-5695-3_23
- [28] Edsger W. Dijkstra. 1988. A New Science, From Birth to Maturity. (Oct. 1988). Retrieved April 19, 2026 from <http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1024.PDF>
- [29] Edsger W. Dijkstra. 1999. Computing Science: Achievements and Challenges. (1999). Retrieved April 19, 2026 from <http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1284.PDF>. Published as [30].
- [30] Edsger W. Dijkstra. 1999. Computing science: Achievements and challenges. *ACM SIGAPP Applied Computing Review* 7, 2 (1999), 2–9. DOI : <https://doi.org/10.1145/335527.335528>
- [31] Edsger W. Dijkstra. 2001. Under the spell of Leibniz’s dream. *Information Processing Letters* 77, 2-4 (2001), 53–61. DOI : [https://doi.org/10.1016/S0020-0190\(00\)00204-0](https://doi.org/10.1016/S0020-0190(00)00204-0)
- [32] Edsger W. Dijkstra. n.d.. Mijn laatste verslag van een bijeenkomst van W.G.2.1. (n.d.). Retrieved April 19, 2026 from <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD252.PDF>. Circulated privately.
- [33] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where programs meet provers. In *Proceedings of the European Symposium on Programming*. Matthias Felleisen and Philippa Gardner (Eds.), Lecture Notes in Computer Science, Springer, 125–128. DOI : https://doi.org/10.1007/978-3-642-37036-6_8
- [34] Jean-Christophe Filliâtre. 2007. Formal proof of a program: Find. *Science of Computer Programming* 64, 3 (2007), 332–340. DOI : <https://doi.org/doi.org/10.1016/j.scico.2006.10.002>
- [35] R. Floyd. 1967. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Mathematics 19, Mathematical Aspects of Computer Science*. J. T. Schwartz (Ed.), American Mathematical Society, New York, 19–32. Retrieved from <https://pubs.ams.org/ebooks/psapm/019/>
- [36] M. Foley and C. A. R. Hoare. 1971. Proof of a recursive program: Quicksort. *The Computer Journal* 14, 4 (1971), 391–395. DOI : <https://doi.org/10.1093/comjnl/14.4.391>
- [37] Philip L. Frana and Thomas J. Misa. 2010. An interview with Edsger W. Dijkstra. *Communications of the ACM* 53, 8 (2010), 41–47. DOI : <https://doi.org/10.1145/1787234.1787249>
- [38] Nissim Francez. 1980. Distributed termination. *ACM Transactions on Programming Languages and Systems* 2, 1 (1980), 42–55. DOI : <https://doi.org/10.1145/357084.357087>
- [39] Per Brinch Hansen. 1972. A comparison of two synchronizing concepts. *Acta Informatica* 1 (1972), 190–199. DOI : <https://doi.org/10.1007/BF00288684>
- [40] Per Brinch Hansen. 1972. Structured multiprogramming. *Communications of the ACM* 15, 7 (1972), 574–578. DOI : <https://doi.org/10.1145/361454.361473>
- [41] Per Brinch Hansen. 1973. *Operating Systems Principles*. Prentice-Hall, Englewood Cliffs, N.J. Retrieved from <https://dl.acm.org/doi/10.5555/540365>

- [42] Per Brinch Hansen. 1993. Monitors and Concurrent Pascal: A personal history. In *Proceedings of the History of Programming Languages Conference*. John A. N. Lee and Jean E. Sammet (Eds.), ACM, 1–35. DOI : <https://doi.org/10.1145/154766.155361>
- [43] Per Brinch Hansen (Ed.). 2002. *The Origin of Concurrent Programming*. Springer. Retrieved from <https://dl.acm.org/doi/10.5555/540365>
- [44] Per Brinch Hansen. 2004. A Programmer’s Story: The Life of a Computer Pioneer. (2004). Retrieved April 19, 2026 from <http://brinch-hansen.net/>
- [45] Gabriel Hévr, Ivan Šimeček, and Ondřej Voronecký. 2026. PPQSort: Pattern-defeating parallel quicksort. *Concurrency and Computation: Practice and Experience* 38, 7 (2026), e70673. DOI : <https://doi.org/10.1002/cpe.70673>
- [46] C. A. R. Hoare. 1961. Algorithm 63, partition; algorithm 64, quicksort; algorithm 65, find. *Communications of the ACM* 4, 7 (1961), 321–322. Retrieved from <https://dl.acm.org/doi/epdf/10.1145/366622.366642>
- [47] C. A. R. Hoare. 1962. Quicksort. *BCS, Computer Journal* 5, 1 (1962), 10–16. DOI : <https://doi.org/10.1093/comjnl/5.1.10>
- [48] C. A. R. Hoare. 1965. Record handling. *Algol Bulletin* 21 (1965), 39–69. DOI : <https://doi.org/doi/10.5555/1061032.1061041>
- [49] C. A. R. Hoare. 1968. Record handling. In *Proceedings of the Programming Languages*. F. Genuys (Ed.), Academic Press, 291–347. Retrieved from <https://ora.ox.ac.uk/objects/uuid%3Ad759ca7a-6fe4-4362-ab5f-34a78958e55c>
- [50] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Communications of the ACM* 12, 10 (1969), 576–580. DOI : <https://doi.org/10.1145/363235.363259>
- [51] C. A. R. Hoare. 1971. Proof of a program: Find. *Communications of the ACM* 14, 1 (1971), 39–45. DOI : <http://doi.acm.org/10.1145/362452.362489>
- [52] C. A. R. Hoare. 1972. Proof of a structured program: ‘The sieve of Eratosthenes. *Computer Journal* 15, 4 (1972), 321–325. <https://doi.org/10.1093/comjnl/15.4.321>
- [53] C. A. R. Hoare. 1972. Proof of correctness of data representations. *Acta Informatica* 1, 4 (1972), 271–281. DOI : <https://doi.org/10.1007/BF00289507>
- [54] C. A. R. Hoare. 1972. Towards a theory of parallel programming. In *Proceedings of the Operating System Techniques*. C.A.R. Hoare and R. Perrot (Eds.), Academic Press, 61–71. Retrieved from <https://ora.ox.ac.uk/objects/uuid%3Aa2f19c4d-8e82-42b7-9787-32a0d11df323>
- [55] C. A. R. Hoare. 1973. A structured paging system. *BCS, Computer Journal* 16, 3 (1973), 209–215. DOI : <https://doi.org/doi.org/10.1093/comjnl/16.3.209>
- [56] C. A. R. Hoare. 1974. Monitors: An operating system structuring concept. *Communications of the ACM* 17, 10 (1974), 549–557. Retrieved from <https://www.classes.cs.uchicago.edu/archive/2020/winter/33100-1/papers/hoare-monitors.pdf>
- [57] C. A. R. Hoare. 1975. Parallel programming: An axiomatic approach. *Computer Languages* 1, 2 (1975), 151–160. DOI : [https://doi.org/10.1016/0096-0551\(75\)90014-4](https://doi.org/10.1016/0096-0551(75)90014-4)
- [58] C. A. R. Hoare. 1978. Communicating sequential processes. *Communications of the ACM* 21, 8 (1978), 666–677. DOI : <https://doi.org/10.1145/359576.359585>
- [59] C. A. R. Hoare. 1981. The emperor’s old clothes: The ACM Turing award lecture. *Communications of the ACM* 24, 2 (1981), 75–83. Retrieved from <https://www.cs.fsu.edu/~engelen/courses/COP4610/hoare.pdf>
- [60] C. A. R. Hoare and P. E. Lauer. 1974. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica* 3, 2 (1974), 135–153. DOI : <https://doi.org/10.1007/BF00264034>
- [61] C. A. R. Hoare and N. Wirth. 1973. An axiomatic definition of the programming language Pascal. *Acta Informatica* 2 (1973), 335–355. DOI : <https://doi.org/10.1007/BF00289504>
- [62] C. A. R. Hoare. 1968. The Axiomatic Method. (1968). Retrieved April 19, 2026 from <https://www.cs.ox.ac.uk/files/6079/H68%20-%20Axiomatic.pdf>
- [63] Qifan Huang, Li Zhou, Wang Fang, Mengyu Zhao, and Mingsheng Ying. 2025. Efficient formal verification of quantum error correcting programs. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1068–1093. DOI : <https://doi.org/10.1145/3729293>
- [64] Jasper Ischebeck and Ralph Neininger. 2024. On fluctuations of complexity measures for the FIND algorithm. In *Proceedings of the 35th International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:15. DOI : <https://doi.org/10.4230/LIPICS.AOFA.2024.9>
- [65] Cliff Jones. 2021. ACM Interview. In *Theories of Programming: The Life and Works of Tony Hoare*, Cliff B. Jones and Jayadev Misra (Eds.). ACM Books, Vol. 39. ACM / Morgan & Claypool, 359–386. DOI : <https://doi.org/10.1145/3477355.3477372>
- [66] Cliff Jones and Jayadev Misra. 2021. Finding effective abstractions. In *Theories of Programming: The Life and Works of Tony Hoare*, Cliff B. Jones and Jayadev Misra (Eds.). ACM Books, Vol. 39. ACM / Morgan & Claypool, 23–40. DOI : <https://doi.org/10.1145/3477355.3477358>
- [67] Cliff B. Jones and Jayadev Misra (Eds.). 2021. *Theories of Programming: The Life and Works of Tony Hoare*. ACM Books, Vol. 39. ACM / Morgan & Claypool. DOI : <https://doi.org/10.1145/3477355>

- [68] C. B. Jones and A. W. Roscoe. 2010. Insight, inspiration and collaboration. In *Reflections on the Work of C. A. R. Hoare*, A. W. Roscoe, Clifford B. Jones, and Kenneth R. Wood (Eds.). Springer, 1–32. DOI : https://doi.org/10.1007/978-1-84882-912-1_1
- [69] J. C. King. 1969. *A Program Verifier*. Ph. D. Dissertation. Carnegie-Mellon University. Retrieved from <https://books.google.nl/books?id=kJWszAEACAAJ>
- [70] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley. Retrieved from <https://dblp.org/rec/books/aw/Knuth73.bib>
- [71] Leslie Lamport. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.* 3, 2 (1977), 125–143. DOI : <https://doi.org/10.1109/TSE.1977.229904>
- [72] Gary Levin and David Gries. 1981. A proof technique for communicating sequential processes. *Acta Informatica* 15 (1981), 281–302. DOI : <https://doi.org/10.1007/BF00289266>
- [73] Ralph L. London, John V. Guttag, James J. Horning, Butler W. Lampson, James G. Mitchell, and Gerald J. Popek. 1978. Proof rules for the programming language euclid. *Acta Informatica* 10 (1978), 1–26. <https://doi.org/10.1007/BF00260921>
- [74] David May. 1983. OCCAM. *ACM SIGPLAN Notices* 18, 4 (1983), 69–79. DOI : <https://doi.org/10.1145/948176.948183>
- [75] David May. 2021. CSP, Occam, and Inmos. In *Theories of Programming: The Life and Works of Tony Hoare*, Cliff B. Jones and Jayadev Misra (Eds.). ACM Books, Vol. 39. ACM / Morgan & Claypool, 271–284. DOI : <https://doi.org/10.1145/3477355.3477368>
- [76] B. Meyer. 1997. *Object-Oriented Software Construction* (2nd. ed.). Prentice Hall. Retrieved from <http://www.eiffel.com/doc/oosc/page.html>
- [77] George J. Milne and Robin Milner. 1979. Concurrent processes and their syntax. *Journal of the ACM* 26, 2 (1979), 302–321. DOI : <https://doi.org/10.1145/322123.322134>
- [78] Robin Milner. 1978. Synthesis of communicating behaviour. In *Proceedings of the 7th International Symposium on Mathematical Foundations of Computer Science*. Józef Winkowski (Ed.), Lecture Notes in Computer Science, Springer, 71–83. DOI : https://doi.org/10.1007/3-540-08921-7_57
- [79] Robin Milner. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92, Springer. DOI : <https://doi.org/10.1007/3-540-10235-3>
- [80] F. L. Morris and C. B. Jones. 1984. An early program proof by Alan Turing. *Annals of the History of Computing* 6, 2 (1984), 139–143. Retrieved from <https://www.computer.org/csdl/magazine/an/1984/02/man1984020139/13rRUy3gn2R>
- [81] Peter Naur. 1966. Proof of algorithms by general snapshots. *BIT Numerical Mathematics* 6, 4 (1966), 310–316. Retrieved from <https://link.springer.com/article/10.1007/BF01966091>
- [82] Susan S. Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Informatica* 6 (1976), 319–340. DOI : <https://doi.org/10.1007/BF00268134>
- [83] Susan S. Owicki and David Gries. 1976. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM* 19, 5 (1976), 279–285. DOI : <https://doi.org/10.1145/360051.360224>
- [84] A. W. Roscoe, Clifford B. Jones, and Kenneth R. Wood (Eds.). 2010. *Reflections on the Work of C. A. R. Hoare*. Springer. DOI : <https://doi.org/10.1007/978-1-84882-912-1>
- [85] Hans Albrecht Schmid. 1976. On the efficient implementation of conditional critical regions and the construction of monitors. *Acta Informatica* 6 (1976), 227–249. DOI : <https://doi.org/10.1007/BF00288656>
- [86] D. S. Scott and J. W. de Bakker. 1969. A Theory of Programs. Retrieved April 19, 2026 from <https://eudml.org/doc/92628>. Notes of an IBM Vienna Seminar.
- [87] Robert Sedgewick. 1975. *Quicksort*. Ph. D. Dissertation. Stanford University, USA. Retrieved from <https://searchworks.stanford.edu/view/870273>
- [88] Robert Sedgewick. 1977. The analysis of quicksort programs. *Acta Informatica* 7 (1977), 327–355. DOI : <https://doi.org/10.1007/BF00289467>
- [89] Robert Sedgewick. 1977. Quicksort with equal keys. *SIAM J. Comput.* 6, 2 (1977), 240–268. DOI : <https://doi.org/10.1137/0206018>
- [90] Robert Sedgewick. 1978. Implementing quicksort programs. *Communications of the ACM* 21, 10 (1978), 847–857. DOI : <https://doi.org/10.1145/359619.359631>
- [91] Robert Sedgewick and Kevin Wayne. 2016. *Algorithms* (4th ed.). Addison-Wesley.
- [92] Len Shustek. 2009. Interview - An interview with C.A.R. Hoare. *Communications of the ACM* 52, 3 (2009), 38–41. DOI : <https://doi.org/10.1145/1467247.1467261>
- [93] Leon Sterling and Ehud Shapiro. 1986. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press. Retrieved from <https://mitpress.mit.edu/9780262193382/the-art-of-prolog/>
- [94] Simon J. Thompson. 2011. *Haskell - The Craft of Functional Programming* (3rd. ed.). Addison-Wesley. Retrieved from <https://dblp.org/rec/books/daglib/0033100.bib>
- [95] Dr. A. Turing. 1949. On checking a large routine. *Report of a Conference on High Speed Automatic Calculating Machines* (1949), 67–69. Retrieved from <https://turingarchive.kings.cam.ac.uk/checking-large-routine>. Univ. Math. Laboratory, Cambridge.

- [96] Niklaus Wirth. 1971. Program development by stepwise refinement. *Communications of the ACM* 14, 4 (1971), 221–227. DOI : <https://doi.org/10.1145/362575.362577>
- [97] Niklaus Wirth. 1971. The programming language Pascal. *Acta Informatica* 1 (1971), 35–63. DOI : <https://doi.org/10.1007/BF00264291>
- [98] Niklaus Wirth. 1973. *Systematic Programming: An Introduction*. Prentice-Hall, Englewood Cliffs, N.J. Retrieved from <https://search.worldcat.org/title/223094>
- [99] Niklaus Wirth. 1976. *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood Cliffs, N.J. Retrieved from <https://search.worldcat.org/title/2005288>
- [100] Niklaus Wirth. 1976. Typed letter to C.A.R. Hoare. (1976). Retrieved from <https://www.cs.ox.ac.uk/files/6155/Wirth76%20-%20Cooperating.pdf>.
- [101] Niklaus Wirth. 2021. 50 years of Pascal. *Communications of the ACM* 64, 3 (2021), 39–41. DOI : <https://doi.org/10.1145/3447525>
- [102] Niklaus Wirth and C. A. R. Hoare. 1966. A contribution to the development of ALGOL. *Communications of the ACM* 9, 6 (1966), 413–432. DOI : <https://doi.org/10.1145/365696.365702>

Received 20 April 2026; revised 12 May 2026; accepted 13 May 2026