Minds and Machines
https://doi.org/10.1007/s11023-023-09632-2

From Monitors to Monitors: A Primitive History

Troy K. Astarte[1]

Received: 11 July 2022 / Accepted: 9 March 2023

Abstract
As computers became multi-component systems in the 1950s, handling the speed differentials efficiently was identified as a major challenge. The desire for better understanding and control of 'concurrency' spread into hardware, software, and formalism. This paper examines the way in which the problem emerged and was handled across various computing cultures from 1955 to 1985. In the machinic culture of the late 1950s, system programs called 'monitors' were used for directly managing synchronisation. Attempts to reframe synchronisation in the subsequent algorithmic culture pushed the problem to a higher level of abstraction; Dijkstra's semaphores were a reaction to the algorithms' complexity. Towards the end of the 1960s, the culture of 'structured programming' created a milieu in which Dijkstra, Hoare, and Brinch Hansen (among others) aimed for a concurrency primitive which embodied the new view of programming. Via conditional critical regions and Dijkstra's 'secretaries', the co-produced 'monitor' appeared to provide the desired encapsulation. The construct received embodiment in a few programming languages; this paper ends by considering Modula and Concurrent Pascal.

Keywords History of computing · History of computer science · Programming languages · Concurrency

1 Introduction

Studies on the nature of computer science tend towards the grand and overarching. Eden (2007) and Wegner (1976) discuss paradigms of computer science from philosophical and historical perspectives; Tedre (2014) explores the formation of computer science as a distinct discipline forged from multiple traditions and Mahoney (2002) uses the lens of communities and their agendas. Abbate (2017) finds more specificity through the context of the USA funding landscape and Astarte (2022) examines the role of one particular aspect of theoretical computer science, formal

✉ Troy K. Astarte
t.k.astarte@swansea.ac.uk

[1] Department of Computer Science, Swansea University, Swansea, UK

semantics, in establishing European disciplinary identity. The approach taken in this paper is to develop a particular narrative by exploring the emergence of a specific technical problem across disparate computing cultures.

Concurrency, here loosely defined as the practice of getting a computer to do multiple things at the same time,[1] promises benefits of efficiency and speed, at the cost of significant conceptual and practical complexity. It is one of these challenges, that of mutual exclusion, which is traced through various contexts in this paper. We consider how the problem presents itself, and how the researchers involved attempted to address it. By examining these various approaches, and the descriptions used to illustrate them, we can understand more about the particular computing culture: while computer science, especially theoretical computer science, appears to concern itself with very abstract entities, the use of culturally-laden anthropomorphic metaphor and analogy frequently undermine this supposed neutrality. Starting with early machinic culture, through the emergence of algorithms as a core component in computer science, and finally examining the doctrine of structured programming, we look at the difficulties of apportioning credit in a community that values individual work while promoting co-operation, and finish by considering what this story tells us about the changing nature of computer science.

This paper contains some description of technical concepts but, hopefully, gently. Terms used to describe these concepts will tend to be those used in the relevant historical context since these provide insight into the ways their users viewed the concepts involved.

2 The Dawn of Concurrency

In the machine-focused culture of the 1950s, the effective use of expensive hardware was a significant concern. By this time a standard computer installation comprised a central control unit and a series of peripherals, such as tape readers and printers. As Strachey (1959) noted, faster hardware was more expensive and consequently confined to the central computer. While this could result in cost-effectiveness if continually utilised, peripherals operated considerably slower, and forcing the computer to wait for lengthy delays such as tape reads resulted in unacceptable idle time. Rochester (1955) wrote an early survey on this problem, identifying 'multiprogramming' as the solution: dividing the operations of the central unit between gathering data from input, processing it, and providing output. Part of this could be done in parallel, with a printer being passed some data for output while the computer continued to perform the next stage of calculation. By the end of the decade, it was understood by others, such as Gill (1958), that the problem of managing multiple programs was

[1] Many different terms have been used for this and similar concepts, including 'multiprogramming', 'time-sharing', and 'parallel programming'. Some of those are discussed within this article. 'Concurrency', which is now the most common term, appears to be due to Carl Adam Petri (Brauer & Reisig, 2009) though his graphical approach to modelling concurrency is not discussed here. Exploring the associations inherent in particular terms and the reasons for their use is an interesting area for further research.

conceptually equivalent to 'parallel programming', where the functionality of one program could be spread across multiple processing units to improve the speed of its operations—typically still couched in terms of avoiding waiting for peripherals. The challenge became one of synchronisation, so that time any component spent idly waiting for another could be minimised.

Strachey's solution to this problem, which he referred to as 'time sharing',[2] was to write a 'Director' program running constantly on the computer, which had access to special instructions for controlling interrupts—pausing one program at a signal from another (Strachey, 1959). The Director also allocated separate memory to different program control components operating concurrently to prevent them overwriting information needed by others, something Strachey observed was essential to prevent badly-written programs 'running wild' and consuming the entire system's resources. By framing the Director (with a carefully-allocated capital letter, the only concept in his paper to receive this privilege) as the one in charge, Strachey invoked respect for the gatekeeper of his machine, with its dedicated resources presented like an office manager who needs their own room to operate effectively. This idea of a supervisory program is the germ of large class of programs later referred to as 'operating systems'.

Following a survey of advances in the area of multiprogramming, one such operating system was presented by Codd (1962) from the IBM Data Systems Division. In this system, implemented on IBM's STRETCH machine, programs could run 'concurrently', share data in memory, and manage synchronisation and interference on this data using BLOCK and UNBLOCK instructions. Any arbitration needed between programs was managed by a central supervising program.

A better-known pair of machine-level instructions comes from Conway (1963), who had been working with United States Air Force procurement and was worried by the conflation of process with processor among programmers (Nyman & Laakso, 2016). Conway described a system which could store the states of processes to suspend and resume them effectively using instructions FORK and JOIN. The former branched off a new parallel process while the latter synchronised by waiting until all specified processes had reached a particular ready point, when data could be unified, and extraneous processes could be killed.

FORK and JOIN, though not unique—as acknowledged by Conway—became the standard for such instructions and were implemented in many operating systems. One influential example was in Berkeley's Project Genie, started in 1963, where an operating system component called a 'monitor' managed synchronisation between user programs (Nyman & Laakso, 2016). This name caught on as the common name for such a supervisor (Brinch Hansen, 1996), perhaps because of the success of Project Genie: the resulting computer system was sold as the SDS 940 and the Genie operating system, was bought by a company called Tymshare and marketed as the 'Tymshare Monitor'. Tymshare subsequently became the model for concurrency control in the UNIX kernel (Nyman & Laakso, 2016). Observe that the term

[2] This is not to be confused with the 'time-sharing' of McCarthy and others (Fano & Corbató, 1966), which was a way to allow multiple users to work simultaneously on the same central computer, perhaps through peripheral terminals. As McCarthy (1983) later wrote, Strachey not only hadn't meant this, but didn't even like the idea when he heard it.

'monitor' has less agency than Strachey's 'Director', and perhaps seems less intimidating—but the Tymshare monitor was no less controlling than the Director: user programs could do nothing to control synchronisation beyond removing themselves from contention.

The operating system monitor had drawbacks. As a black-box, applications programmers had little control over the concurrency management, making it difficult to predict the system's behaviour, or to use concurrency effectively within programs. By the early 1960s, there was a growing desire for a more sophisticated synchronisation mechanism available to the programmer. This was obvious already to Conway (1963) who suggested that the new 'high-level' programming languages could have built-in concurrency support to make it easier to write parallel programs. It is this trajectory which is explored in the current paper—though operating system management continues to play a key role in concurrency and underlies the implementation of most of the abstractions discussed.

3 The Algorithmic Age

The early 1960s was a period of diversification in computing as various existing cultures adopted and mutated the calculating machine, presented by Haigh and Ceruzzi (2021) as a series of *becomings*. While Project Genie enjoyed a period of commercial success as a business service, the nascent—and still self-reifying—field of computer science latched onto the term 'algorithm'. Indeed, as argued by Ensmenger (2010, Chap. 5), it was the algorithm that computer science instilled as its fundamental object of study. Ensmenger notes that the centring of the algorithm in Knuth's (1968) carefully curated history of computing was a key aspect in cementing a Kuhnian 'normal science' of computers (Kuhn, 1962).

Algorithmic culture was reflected in the publication of first IAL and then ALGOL 60 (Backus, 1960; Backus et al., 1960): while the former was the International *Algebraic* Language, the latter was the *ALGOrithmic* Language. The algorithm represented a new way to arrange problems for machine operation, and suggested inscription at a higher level of abstraction in a way that corresponded more closely with how a programmer thought (De Mol & Bullynck, 2022).[3] In 1960, a new section was created in *Communications of the ACM* (Vol. 3, No. 2) in which algorithms were published. The machine-independent ALGOL 60 was the perfect vehicle: anyone could write and share these proto-programs; any machine could implement them if it had an ALGOL compiler; and problems in computation could be discussed at this higher level of abstraction.

Although ALGOL 60 had no formal support for concurrency, it was nevertheless the notation of choice for algorithms about this. In an algorithm, the intricacies of controlling concurrency could be addressed directly by the programmer. The focus was often on the problem of *mutual exclusion* (what Strachey had called 'locking out'): ensuring

[3] A retrospective on concurrency published by Lamport (2015) following his Turing Award emphasises the importance of an algorithmic view.

that some critical resource, such as a hardware component or a portion of memory, was not accessed by multiple concurrent routines simultaneously, since unpredictable behaviour could occur when one process depended on the value of a variable that had been changed by another.

An early algorithmic solution is owed to T. J. Dekker, working at Mathematisch Centrum in Amsterdam alongside Edsger Dijkstra, who reported the result (Dijkstra, 1962). This algorithm provided mutual exclusion for two processes, $P1$ and $P2$, preventing them accessing a critical resource simultaneously. Each process has a flag variable which it sets to true prior to entering its 'critical section'; $P2$ must check $P1$'s flag is false before continuing. In the case of both flags being true, stalling is prevented by an additional Boolean arbitrator which alternates priority between processes.

As Dijkstra (1962) notes, the algorithm works well for two processes but generalises poorly to n. Each process has to check priority against every other process, introducing new possibilities for interference, and the checking phase gets longer with every new process. A more sophisticated priority system was given by Dijkstra (1968a, Sect. 2.2), and later by the Bakery algorithm of Lamport (1974). This latter invokes the metaphor of a ticketing system at a bakery. Each process takes a ticket upon being ready for its critical section, and the ticket number increases each time, with the lowest-numbered ticket being allowed to proceed. This permits every thread to get a turn eventually and scales well to many processes, but involves a lot of time for each process to check its progression criteria, often to no avail, a phenomenon referred to as 'busy waiting'. As explained by Dijkstra (1971), this means that a process must continually check whether it can proceed with its critical section, taking up computer resources in doing so. An alternative is to put the process to sleep, only waking it up when needed. Lamport's metaphor of a fair and orderly system like polite patrons at a bakery comes with the cachet of simplicity through familiarity, as well as avoiding the assumption of an underlying mutual exclusion mechanism (Lamport, 2015), though the technical details are quite complex.

This complexity, and the problem of busy waiting, hint at the difficulties of algorithmic solutions to concurrency, though plenty of research continued at least into the 1980s (for example Peterson, 1981; Raynal, 1986). Algorithms retain an important role for teaching the concepts of concurrency; Ben-Ari (1990) builds the problem of mutual exclusion using a series of algorithms which reflect the historical development.

4 Primitive Programming

The textual complexity of algorithms to manage synchronisation tended to mirror runtime inefficiency; and while machine and OS operations underlay the implementation of many concurrent systems,[4] a key problem in the 1960s was the development of a concurrency primitive at the programming language level. This could

[4] For example, the 'test and set' operation in IBM's System/360 was present since the mid 1960s (IBM, 1968).

allow a programmer to utilise concurrency without caring about the implementation; and facilitated reasoning about concurrent programs at a higher level (as discussed later, program reasoning was a growing concern in this period). Such primitives could also scale better to larger problems than could algorithms.

The classic primitive is the ‘semaphore’, an early work of Edsger Wybe Dijkstra. His PhD at Mathematisch Centrum concerned the assembly language for the Electrologica X1 computer, which meant grappling with the problems of hardware management and the interrupt (Dijkstra, 1959). By 1962, he had moved to Technische Hogeschoole Eindhoven, where he began working on a multiprogramming system for the Electrologica X8 Dijkstra (1968c). Dijkstra’s attitude was characterised by Daylight (2011) as ‘generalist’, meaning he was interested in moving away from making computing decisions based on the structure of any one machine or system and instead aiming for concepts which were sufficiently abstract as to be portable.

The semaphore’s introductory publication has already been mentioned since it introduced Dekker’s algorithm, in fact as a kind of straw figure against which to pit his superior ideas (Dijkstra, 1962). ‘*Over de sequetialiteit van procesbeschrijvingen* [On the sequentiality of process descriptions]’ was unpublished and privately circulated, as with many of his works.[5] After dismissing some algorithmic solutions to synchronisation, Dijkstra invoked the metaphor of a railway signalling system to name the ‘semaphore’. These wooden posts had movable arms which could be raised or lowered to indicate to train drivers which tracks to use: just like in concurrent programs, collisions could be avoided. Though the semaphores provide organisation, they have no agency, providing instead a structure for simplistic communication.

Technically, a semaphore is a special purpose variable, initially a Boolean value, which indicates the availability of a resource. Dijkstra (1963) reported on the incorporation of semaphores into programs for the X8, which convinced him that a more general solution was to use integers. A semaphore then provides a counter of the available resources up for contention, and in the case of simple mutual exclusion, can range from 0 to 1. The semaphore is accessible only through two operations P and V. These stood for various words: initially *passering* [passing] and *vrijgave* [release], both nouns; later Dijkstra would present them as verbs: *prolaag* [a neologism meaning try-lower] and *verhoog* [raise].[6] Observe that the use of the railway metaphor loses some of its utility as the program semaphore becomes reified and Dijkstra is forced into awkward neologism to properly express his concept.

P checks whether the semaphore is positive; if so, it decrements the semaphore and proceeds, otherwise the process begins waiting. V increases the value of a semaphore by 1 and, to use the term from Dijkstra (1962), notifies a “central alarm clock” which wakes any waiting processes and allows them to try P again.

Mutual exclusion is very easy to implement with this binary semaphore: 1 indicates that the resource (which could simply be the opportunity to enter a critical section) is free and 0 that it is not. Any number of processes works; each must wait

[5] While there is no date on this manuscript, Dijkstra’s personal numbering system in which nearly everything he wrote is given an ‘EWD-*x*’ label puts it in 1962 or 1963.

[6] A useful English mnemonic is procure and vacate.

to succeed *P* before continuing to its critical section, then signal with *V* when it has finished, freeing up the semaphore to allow another process to pass *P*. Multiple identical resources can be contended with an integer-valued semaphore, or a single variable surrounded by a binary semaphore.

Clearly a major part of the semaphore's proper operation is the sleeping and waking; a further important detail is the 'atomicity' of *V* and of *P*'s non-waiting aspect—the check and change to the value of the semaphore must happen without any other process interfering while the operation is underway. While Dijkstra's early publications simply assumed an implementation would provide such mechanisms, by 1968 he reported on the programming of semaphores in the THE multiprogramming system using machine code primitives and an interrupt mechanism (Dijkstra, 1968d).

The key publication, and usual citation, on semaphores was less implementation focused. Dijkstra (1968a), first distributed in 1965 and later republished in 2013, was pedagogical in nature. This work grew from the lecture notes for Dijkstra's 1965 programming course and presented many basic concepts in concurrency, including the term 'critical section' used in the present article, and a phenomenon Dijkstra called 'the deadly embrace'. First addressed (in Dutch) a few years earlier, Dijkstra (1964) uses this term to describe a situation in which one process can only continue if another is 'killed'. This is a class of problems now called 'deadlocks'—continuing the fatal theme—in which no process can proceed because each is waiting for another.

This risky possibility is introduced by the conditional waiting in the *P* operation: if the proper conditions to raise the semaphore are never met, every process could be stuck forever waiting on a *P* that will never become valid. The classic illustration of this problem appeared as an exam question on Dijkstra's 1965 programming course, where it was called the 'Dining Quintuple', although its popular formation is as 'Dining Philosophers' (Dijkstra, 1971). In this scenario, five philosophers sit around a central bowl of spaghetti, with one fork between each. A philosopher needs two forks to begin eating and must pick them up one at a time; but if each philosopher takes the fork to their left, no forks remain on the table to be picked up in the right hand. In this situation, no-one can eat; and there is no way to use semaphores to prevent this. Of course, this metaphor seems rather artificial; and as Ben-Ari (1990) notes "it is part of the folklore of computer science to point out that the story would be more believable if the bowl contained rice and the utensils were chopsticks". Perhaps spaghetti and philosophy stick closer together in the minds of European computer scientists thanks to the Enlightenment—the discussion by Hoare (1972b) specifically identifies the philosophers with Benthamism—rather than permitting the more apposite, but less Western, metaphor.

As the Dining Philosophers example illustrates, semaphores can't prevent every problem in concurrency. Indeed, poorly-ordered semaphore operations can even introduce them. With the use and challenge of concurrency only increasing throughout the late 1960s and into the 1970s, motivation arose for a more sophisticated primitive. Once again we turn to Dijkstra to seek its roots. His 1968a treatise originated in the lecture notes for a programming course, but this didactic style was typical of him. The work is full of mantras about good programming, and specifically did not include a “fully worked-out theory”, but instead showed the first developments of a new culture which would soon take over programming.

5 Structuring Programming

From the early 1970s the search was on to find an improved programming construct for concurrency: Dijkstra in particular was very strong in his “constant encouragement in the search for a concept to ‘replace’ the semaphore in a high-level programming language” (Hoare, 1972b). Let us focus on three researchers whose interactions during the first years of the decades shaped the following story: Dijkstra, Tony Hoare, and Per Brinch Hansen. A chronology is provided by Brinch Hansen (1996), which tends to agree with the sequence of events discussed by Hoare (in Jones, 2016). However, single-author retrospectives are notorious for unreliability; and in this case, it becomes quite clear that there is no single-person story here.

The three met at a summer school in Marktoberdorf in 1971 and shared their ideas about managing concurrency, finding they had been thinking along similar lines (Brinch Hansen, 2004). A follow-up seminar was organised by Hoare in Belfast on operating system techniques, which Brinch Hansen attended, though Dijkstra did not, fearing terrorist attack. A further important venue for the discussion, sharing, and improving of these ideas, according to Hoare (1974), was IFIP’s Working Group 2.3. This “elite member’s club” (Haigh, 2019) was formed in 1969 when a group of dissenters (including Dijkstra and Hoare) broke away from Working Group 2.1 in protest at the ALGOL 68 definition. WG 2.3, formed as ‘Programming Methodology’, reflected a new priority in computer science: moving away from programming language definition, by now beginning to seem a little old-hat, and toward application of ideas taken from that older paradigm to the programming level (Astarte, 2022).

We have met Dijkstra and set a little of the context. We meet Brinch Hansen later in this section; now let’s meet Hoare.

Charles Anthony Richard Hoare, known as Tony, had joined the English computer company Elliott Brothers in 1960 following an interest in automatic translation between Russian and English picked up during his national service (Jones, 2016). Hoare’s background in logic stretched to his university days studying ‘Greats’ (Classics) but reading Quine and others for fun—though he admits he did not fully grasp all the logic concepts (Daylight, 2013). At Elliott, Hoare worked on an ALGOL 60 compiler and systems development, but became increasingly interested in the theory of programming, taking a chair at Queen’s University Belfast in the late 1960s to satisfy this desire.[7] To Hoare, this theory included programming language semantics, program proof, and concurrency; the latter two he spoke about at the 1971 meetings.

The first publication of Hoare’s thoughts on concurrency appeared a year later. The paper shows him trying to find a theory for new programming language features

[7] Hoare interviewed at a number of universities, rather speculatively, and was somewhat surprised that he was offered the post (Jones, 2016).

which would follow certain principles: freedom from error, run-time efficiency, conceptual simplicity, and generality of application (Hoare, 1972b). Note the concerns shared with Dijkstra. Hoare observed that programming systems with concurrency made all of these much harder to achieve due to contention over shared resources.

Hoare proposed to isolate these areas of contention, which he called ‘critical regions’, using a simple program construct:

$$\text{with } r \text{ do } C$$

where r is shared resource and C a critical region—a series of program statements.

The syntactic isolation and obvious linking of the program segment with the resource facilitated reasoning in the Hoare-style, which worked well for resources like I/O hardware, but didn’t provide an obvious way to cope with retaining values when the resource was programmatic.

As Strachey had observed over ten years previously, sharing variables across processes caused challenges, but since many key problems required sharing, Hoare suggested a conditional entry mechanism governed by a Boolean condition B alongside the critical region:

$$\text{with } r \text{ when } B \text{ do } C$$

A process would first have to acquire mutual exclusion on r and then test B; only when true could the process proceed, otherwise being placed on a queue to wait. Upon completing C, the process which had been waiting longest would be allowed to contend for the critical region again—now called a ‘conditional critical region’ (CCR). Hoare showed how to implement this system using semaphores, demonstrating both his familiarity with Dijkstra’s concept, and the semaphore’s own penetration into the computer science literature.

A trend is observable now of these concurrency primitives becoming increasingly abstract, with less concern paid to implementation. While Dijkstra did not have a “fully worked-out theory”, Hoare was already concerned with placing this concurrent programming within his growing theoretical body, and included some first steps towards ‘Hoare Logic’ rules for concurrency.[8]

Per Brinch Hansen was less interested in the theory side. He was an electrical engineering graduate (1963) from the Technical University of Denmark who became engaged in computing during his Master’s degree because he wanted to make fundamental contributions and computing seemed a field ripe with potential novelties (Brinch Hansen, 2004). After interning with IBM Hursley, in England, Brinch Hansen started working at the computer manufacturer Regnecentralen in Copenhagen under Peter Naur following graduation. After developing a COBOL compiler, Brinch Hansen was tasked with creating an operating system for the RC4000 machine. This had been commissioned to help manage a Polish fertiliser plant, and the serious real-time data handling requirements forced Brinch Hansen to confront the challenges of concurrency. Like many others, this systems culture was

[8] For more on this line of the story, see Jones (2023).

his first view of the area (Brinch Hansen, 1970) and led to him writing a textbook on the experience, one of the earliest of its kind (Brinch Hansen, 1973). During the 1971 meetings with Hoare, Dijkstra, and others, Brinch Hansen spoke about operating system principles and the RC4000 experience.

Now focusing specifically on concurrency as a challenge for a programming language, Brinch Hansen (1972a) wrote an analysis of semaphores and CCRs. He looked at a challenge problem, that of ‘readers and writers’ as presented by Courtois et al. (1971). Brinch Hansen compared how semaphores and CCRs managed the task, and proposed his own solution to the problem. He concluded that semaphores are advantageous only when synchronisation is the major goal; data access control benefited from the additional mechanisms of the CCR which made the program text more intuitive—but at the cost of increased busy waiting.

The original authors did not care for Brinch Hansen’s approach and wrote a robust reply (Courtois et al., 1972). Even for this small (but fundamental) problem in concurrency, differing views of ideals like ‘simplicity’, ‘efficiency’, and ‘elegance’ led to different readings of the problem and different writings of solutions. Indeed, Courtois and colleagues objected to some of the apparent aesthetic value Brinch Hansen had brought, writing “It seems to have been the insistence on a symmetric solution to an asymmetric problem which prevented a really satisfactory solution from being found”.

Working towards a better primitive, Brinch Hansen (1972b) identified a problematic situation for CCRs: when many conflicting requests on shared resources occurred simultaneously, they could only be resolved through the intervention of the operating system. Why could the programmer not do this instead? Brinch Hansen suggested that each shared resource within a CCR should have an explicit associated ‘event queue’. If a process failed the condition test, an *await* primitive would be used to add the process to this event queue and take it from the CCR. A paired command, *cause*, would wake up every process in the event queue and allow them all again to try to enter the CCR. This is very similar to Hoare’s approach, with a little more naming of the concepts involved, and the awakening resulting in competition between processes rather than the priority going to the longest waiting.

Brinch Hansen began to demonstrate concern for correctness in this paper, describing concepts as having “simple axiomatic properties” (in the Hoare style) which permit “extensive compile time checking”; as well as growing aesthetic concerns about the structuring of the program text itself. Here we see the convergence of a number of priorities in programming culture at the time embodied in the constructs for concurrency—which Brinch Hansen hoped could implement the OS control features present in his earlier work (Brinch Hansen, 1970) called ‘monitor procedures’.

The pieces are nearly in place to examine the emergence of the monitor as program construct; but first we must consider what Dijkstra had presented at the 1971 meetings. Dijkstra (1971) wrote about the implementation of operating systems through varying “layers” of programs, each preparing a machine to make the programming easier in the next layer. He saw semaphores as a way to address some of the challenges, and also got involved with correctness, writing an (informal) proof demonstrating the solution of mutual exclusion with semaphores. Much of this was material he had already written. But Dijkstra then began to look at more complex problems such as readers and writers which revealed deficiencies in solutions using only semaphores.

Inspired by 1970s management culture, and perhaps also the repeated use of the 'sleep' metaphor for processes, Dijkstra imagines a secretary assisting a group of directors. Although he does not explicitly gender them on first introduction, from the pronouns it is clear the directors are all male and the secretary female.[9] He notes "We have used the metaphor of directors and a common secretary because in the director-secretary relation in real-life organisation its [sic] also unclear who is the master and who is the slave!". Invoking this now-problematised metaphor (Eglash, 2007) as a source of humour justifying another challenging dynamic reveals that even apparently abstract computer science could embody problematic cultural values.[10]

The technical details of how Dijkstra's secretaries managed concurrency are rather brief in the 1971 paper and tend not to be referenced much elsewhere, but the basic idea is as follows. The directors represent processes and the secretary guards their shared critical resources. Entering a critical section is achieved by calling the secretary: if the resource is free, it is granted, and the director proceeds, with the secretary noting which director has the resource. If not, she writes down his request and tells him to go back to sleep until the appropriate resource is free, at which point she calls the director to wake him up.

This idea is not dissimilar to the centralised OS model of concurrency control; the difference is that the programmer can write the routines for waking and sleeping and can control the conditions—unlike Strachey's Director which was strictly out of the programmer's hands. Additionally, secretaries could be nested hierarchically, allowing a secretary to be director to a number of sub-secretaries, for more complex tasks. Dijkstra's idea was in fact also rather similar to the formulation that ended up entering the canon of computer science, just framed differently; but first let us briefly examine a little more of the intellectual context.

6 Monitors Return

Towards the end of the 1960s, there was perceived by some to be a growing "software crisis". Histories argue about the reality of the crisis (see e.g. Astarte, 2022; Haigh, 2019; Peláez Valdez, 1988), but undeniably the narrative was useful for a particular culture of computer scientists to push their own research agendas. Dijkstra and Hoare are clear cases—the term 'software crisis' is all over Dijkstra's Turing Award acceptance (Dijkstra, 1972). Dijkstra was, as already mentioned, something

[9] The paper's title invokes 'hierarchical ordering' which refers to the layers of operating system, but could just as well reflect the cultural hierarchies being reflected in computing.

[10] The literature on history of computing and technology contains plenty of stories about the treatment of feminised labour and its removal from the computing industry as it professionalised (Ensmenger, 2010; Hicks, 2017)—though Misa (2019, 2021) has challenged this narrative as insufficiently rich.

of a ‘generalist’ in his attitude to programming and had argued against the go-to statement in a famous letter (Dijkstra, 1968b). By the end of the 1960s, this had turned into a whole dogma: ‘structured programming’. This attitude emphasised code readability and predictable control flow and opposed abnormal exits and trick-based programming. Perhaps an entire book could be written on the programming cultures of this period, with structured programming and its high-minded ideals and aesthetics playing a central role. So far, however, the topic has received historical treatment only in retrospectives by computer scientists, such as Mills (1986) and Dijkstra (2001).

Presented at the first meeting of WG 2.3 (Woodger, 1978), which took place at Brinch Hansen’s workplace Regnecentralen, the structured programming doctrine was laid out in a book by Dahl et al. (1972). Hoare’s interest in structured programming also influenced his work on proof of data representations (Hoare, 1972a). Brinch Hansen, too, extolled the paradigm’s virtues (recall that his readers/writers solution was accused of being unnecessarily symmetric—that being one of the cultural values of structured programming) and framed his own work as ‘structured multiprogramming’ (Brinch Hansen, 1972b). In this same paper Brinch Hansen praised Dijkstra’s use of **cobegin** and **coend** as a “restricted form of concurrency” contrasted with “unstructured *fork* and *join* primitives”. Aside from the fact that Dijkstra actually used **parbegin** and **parend**, the only major difference is that Dijkstra wrote in terms of a high-level programming language and Conway machine code—the description of the terms are otherwise remarkably similar. Nevertheless, the clothes of structured programming were enough to convince Brinch Hansen of Dijkstra’s superiority.

The other important problem leading to the programming monitor was to deal with the spreading of critical regions throughout programs. As Jones (in Hoare & Jones, 1989, p. 171) pointed out, this made the task of comprehending and reasoning about concurrent programs difficult. The insight for dealing with this problem came from another trend in programming: encapsulation, a core concept from the Simula family of simulation programming languages (Dahl & Nygaard, 1966).[11] Note that Dahl was the third author of the book on structured programming; these two ideas are deeply entwined. Encapsulation was achieved by limiting access to data allowing only specified means, such as named procedures. This provided such benefits as the prevention of accidental writing to key variables—a challenge for concurrency since Strachey’s day.

Dahl spoke about Simula and its ‘class’-based approach to structuring data at the first meeting of WG 2.3 as well—with Brinch Hansen, Dijkstra, and Hoare in attendance, this could well have been their introduction to the concept (Woodger, 1978). Brinch Hansen used the ideas in his work on operating systems and was the driving force behind applying it to concurrency (Brinch Hansen, 1996). According to Hoare (in Jones, 2016), Brinch Hansen had “picked up on this idea that the updates to shared data should be all written and understood in a single place rather than

[11] For a retrospective history, see Nygaard and Dahl (1978).

being scattered around, which was the case in my previous proposal for conditional critical regions".

The first publication to use the term 'monitor' to refer to a specific program construct is somewhat tricky to determine. Hoare (1973) uses it without reference; Brinch Hansen (1972b) discussed 'monitor procedures' referring back to an earlier work (Brinch Hansen, 1970), although the term does not actually appear in that publication. It's safest to assume that it emerged as a name for the new concept that the trio were discussing during 1971 and was adopted by both Brinch Hansen and Hoare. According to Brinch Hansen (1996) the use is in reference to the relation with the earlier, monolithic, OS monitor—because the new construct also centralised shared resources.

The key reference on monitors was written by Hoare (1974). This paper frames the monitor as an operating systems concept; a program that controls other (parallel) programs. Hoare was careful to acknowledge the work of others, writing that his approach "develops Brinch Hansen's concept of a monitor as a method of structuring an operating system" and "the development of the monitor concept is due to frequent discussions and communications with E.W. Dijkstra and P. Brinch-Hansen. A monitor corresponds to the 'secretary' described in Dijkstra (1971), and is also described in Brinch Hansen (1972b, 1973)".

Hoare presents a monitor as a gatekeeper for shared data resources bundled together with a collection of allowable operations on each resource. Like in Simula (and later object-oriented programming), defined procedures are the only way to access the data inside the monitor—and these procedures are guaranteed to have mutual exclusion since only one procedure can be active in a monitor at a given time. The significant point is that critical sections are gathered into the procedures, rather than being scattered across the many different processes around the code.

To prevent programs 'running wild' and causing problems, Strachey's Director locked access to the shared memory space. In the same way, monitors protect the system from poorly-formed individual processes, such as those with badly-ordered acquisition and release of mutual exclusion, by refusing to grant their requests to access data. However, despite the responsibility being centralised, more control rests with the programmer who can write the monitor procedures. Written by Hoare in a pseudo-Simula notation, the monitor framework could be implemented in any high-level programming language for further applications beyond operating systems.

Monitors control synchronisation through special operations on a particular kind of variable local to the monitor called a 'condition', which is a queue of processes. Each monitor might have multiple such conditions, potentially tied to multiple resources. Conditions have three associated commands: *wait*, *signal*, and *non-empty*. *Wait* is used to suspend a calling process when the resource is unavailable and puts it at the end of the queue, releasing the mutual exclusion; *signal* wakes the process at the head of the queue; and *non-empty* checks whether processes are waiting. Clearly, there is an obvious relation here to the event queues of Brinch Hansen

(1972b)—although only one process is awakened to reduce busy waiting—and the name comes from Hoare's own CCRs.[12]

Implementing monitors would prove to be a challenge due to the potential inefficiencies of conditions; some ideas are presented by Brinch Hansen (1996). One such, written by Hoare, is intriguing: should it be allowed for a monitor entry to complete some computation postponed by a different monitor entry? This does not break the centralised aspect of the monitor, but did form the germ of Hoare's later work on 'communicating sequential processes' (Hoare, 1978). Multiple conditions in the same monitor increased their utility but made working with them trickier; another extension used priority queueing for more sophisticated scheduling at the risk of starvation for lower priority processes. Queue jumping, where a rogue process gets hold of the monitor mutual exclusion between a *signal* command sleeping one process and the queue leader awakening, is prevented thanks to a suggestion from Dahl that *signal* should come only at the very end of a procedure.

Following his own ideals about bringing correctness to concurrency, Hoare attempted to provide some proof rules for monitors. The mutual exclusion of monitors meant that procedure calls could be considered sequentially and their results could be somewhat predictable. With the non-local data access restrictions from the structured programming paradigm, invariants could be written for monitor local data and conditions, and proof rules could be somewhat symmetrical. Hoare (1973) was able to use this approach to reason about a system for paging, a key OS concept of significant complexity.

However, Hoare's 'ideal' view of proof (as seen in Hoare, 1976) wasn't fully achievable by monitors, and deadlock still couldn't be prevented. Hoare began to move away from the monitor and towards viewing the competition between processes as the key aspect: "the declaration is a better way of dealing with competition than the resource". He began building a theory of communication, leading to a concurrency approach called 'process algebra'. Dijkstra remained interested in programming concepts, working on proof approaches and guarded commands. Brinch Hansen, though, took most interest in monitors and actually implemented them in some programming languages. A consideration of these makes up the final part of this article.

7 Programming Languages

The programming language Pascal began life as a sequel to ALGOL 60 developed by Hoare and a fellow ALGOL committee member, Niklaus Wirth.[13] but evolved through ALGOL-W to being a sole creation of Wirth (1971). Designed to embody the principles of structured programming, it saw frequent use as a teaching

[12] This reuse of names for different concepts is somewhat typical of Hoare; he changed his paper 'Proof of a program: FIND' to be a *construction* following referee reviews, but kept the title the same. He also published multiple different things called CCS.

[13] For a hagiography, see The School of Niklaus Wirth (2000).

language (Wirth, 2007). Brinch Hansen in particular liked Pascal, and took it as a starting point when he began developing an inherently concurrent programming language.

Brinch Hansen had given a talk on his ideas about queueing variable at Caltech in February 1972, and took a faculty position there in the summer of that year. He had the opportunity to develop an operating system for the laboratory's new PDP 11/45 computer, and decided to create a high-level systems programming language for this purpose. Later, Brinch Hansen (1996) wrote that his ambition for the language was to facilitate for operating systems programming what Pascal had done for compilers: reduce the level of effort (compared to writing in machine code) while retaining efficiency.

With Simula still providing inspiration, Brinch Hansen (1976a) made encapsulation a central component of the Solo operating system: "The combination of a data structure and the operations used to access it is called an *abstract data type*. It is abstract because the rest of the system only needs to know what operations one can perform on it but can ignore the details of how they are carried out." Previously, Brinch Hansen had considered monitors as encapsulating *one* instance of a shared resource; but with the inclusion of a class system, he noted a pattern could be defined for *many* instances.

With the addition to Pascal of classes, concurrent processes, and monitors to manage the mutual exclusion, Brinch Hansen created a new language he called Concurrent Pascal (Brinch Hansen, 1976b). Compared to Hoare's monitors there were subtle differences in the queueing mechanism intended to make it easier for the compiler to check for synch bugs; but one-process-per-monitor mutual exclusion was copied, the information perhaps coming from Dahl via Hoare's visit to Caltech in 1974 (Brinch Hansen, 1996). With no recursion and no garbage collection, Concurrent Pascal was deliberately simple, which Brinch Hansen found came at something of a cost.

Brinch Hansen discovered crafting a series of operating systems, starting with single-user and adding more layers of functionality, while also writing a book on concurrent programming, to be more taxing than he had expected. While the results never quite lived up to his hopes, he had managed to demonstrate that an operating system could be written in a high-level programming language with only a small recourse to machine language—and, indeed, that only 4% required the concurrency features of the language.

A little later in the 1970s, another language, Modula-1, developed Pascal using monitors, this time created by Pascal's originator Wirth (1977). A real believer in structured programming, Wirth felt that Pascal needed more features to be a fully useful language. The key concept in Modula was the 'module': "a set of procedures, data types, and variables, where the programmer has precise control over the names that are imported from and exported to the environment". These provided a way to accept that systems programming might necessitate small components which did not fit with nice abstraction principles, such as assembly code peripheral drivers, or concurrency synchronisation. Modules could isolate these parts and carefully manage the interface: encapsulation for synchronisation. This was somewhat similar to the class in Concurrent Pascal, which Wirth mentions in the text of his article, but without providing a citation to Brinch Hansen. For his part, Brinch Hansen talks about Wirth's work in his retrospective only a little; Wirth doesn't mention Brinch Hansen at all (Brinch Hansen, 1996; Wirth, 2007).

By the second version of Modula, Wirth (1980) had removed almost all concurrency due to a new interest in real-time programming, keeping only critical regions and signals, for improved efficiency. Brinch Hansen, too, began to move away from monitors, instead focusing on multicomputers without shared memory, and the remote procedure call mechanism instead (Brinch Hansen, 1996). While a local monitor still provided mutual exclusion, the days of the monitor as a core operating system component were over. These remote procedure calls, though, served as "a source of inspiration in the design of the Ada tasking facilities" (Roubine & Heliard, 1980). While there is no space in the present paper to discuss Ada, it is worth closing with a quotation about the fascinating history of this Department of Defense programming language:

The "facts" of the history, the dates and names, "have nothing to do with what was actually going on. There was a studied and logical progression, but the paper trail was constructed after the fact to please those who needed boxes filled" (Whitaker, 1996).

8 Conclusions

Standard historical narratives of the reification of (theoretical) computer science present it as a story of increasing abstraction: on the one hand, an increasingly thorough exploration of the most basic concepts in computation (Astarte, 2022; Mahoney, 2002); on the other, a deliberate distancing from the practical concerns of industrial programming (Ensmenger, 2010, Chap. 5). As Dick (2015, p. 89) writes, "Theoretical computer science, in a sense, hinged on a disassociation of 'computing' from the computer itself." In some respects, the treatment of concurrency, as explored in part here, fits with this—each successive section discusses a more abstract component.

However, concurrency forced its way into computing at *all* levels of abstraction, from the economic concern of hardware differentials in early machines, to the management of operating systems, and as an intellectual challenge to the ideals of structured programming. Contrary to the desire of many computer scientists to distance themselves from hardware concerns, exploitation of the valuable performance benefits introduced by concurrency required engaging seriously with issues stemming from basic physics.[14]

Studying how the concurrency challenge emerged and was treated in various computing cultures grants insight into the key nature of the problem as well as illuminating those cultures. The monitor was a tool for oversight and control in operating systems but a primitive in the programming paradigm, and became a tool to enforce the culture of structured programming upon the messiness of concurrency.

[14] The difficulties of maintaining synchronisation in ever-larger electronic circuits was also the driver for Petri (1962) to develop his model of concurrency based on small, asynchronous, autonomous agents.

It saw embodiment in programming languages, which persists today: in Java, every object is automatically given its own monitor on creation, activated by the **synchronised** keyword.

Monitors meant a few things to different people; as the middle ground between theoretical concept and practical tool they received criticism from both directions: too hard to implement, or too challenging to write proofs about. As such they add extra nuance to historical narratives about the divergence of 'theoretical' and 'practical' in computing, and serve as a reminder that complex technical phenomena defy easy categorisation as readily as they prevent neat technical solutions. This multifaceted nature of concurrency, and of monitors in particular, caused those involved in their development to move on to other challenges, and Brinch Hansen (1996) remained very mixed in his later assessment. But just as operating systems still manage concurrency, algorithms still are used to teach (e.g. in Ben–Ari, 1990), and semaphores still persist, monitors remain in modern programming languages and taught courses.

The case of monitors shows us an example of collaboration between Brinch Hansen, Dijkstra, and Hoare, assisted by a growing culture of scientific communities and working groups, like IFIP WG 2.3. Trying to apportion individual credit is thus overly simplistic, and often very difficult. Dijkstra was notorious for failing to cite others; his dogmatic book *A Discipline of Programming* contains the line "For the absence of a bibliography I offer neither explanation nor apology" (Dijkstra, 1976). The lack of reciprocal mention in Brinch Hansen and Wirth's concurrent extensions of Pascal—when the two were surely aware of each other—provides us only tantalising hints at the relationship between them. And the final quote about the history of Ada warns historians even from relying on the old standard of archival documents.

Examining the problem of concurrency provides evidence that theoretical computer science is not only concerned with pure abstract thought-stuff. The examples and metaphors used to describe concurrency tell us something about attitudes in computer science cultures and how grounded these remain in broader cultural perspectives. Strachey's Director embodied the centralised view of management into a computing system; and, reacting against that, the desire for decentralisation and programmer control led to the monitor, even as its name evoked surveillance and oversight. The spaghetti eaten by the classic Dining Philosophers reveals the Western-centric attitude to thought experiments, neatly preventing a more appropriate rice-and-chopsticks metaphor. Finally, Dijkstra's male Directors having the nominal power yet sleeping away any time they are not needed, while female Secretaries manage all the complicated work in the background, forefronts 1970s attitudes to office gender roles in an apparently theoretical situation. Dijkstra played this dynamic for laughs by bringing in a racialised comparison to "master" and "slave".

Critically examining the problematic aspects of gender, race, and embodiment in computing is a key task for scholars, present in, for example, two recent collections of work (Abbate & Dick, 2022; Mullaney et al., 2021). However, the *science* of computing is often absent from these discussions, perhaps due to a perception of its abstract nature. Yet as shown in this article, the task of making computing abstract seems to require creative redescription, which cannot be free from culturally-laden aspects. By studying the use of metaphor and analogy, following the work of Eglash

(2007) and Petrick (2022), we can turn a critical eye to the culture of computer science, too.

Funding The research leading to these results received funding from Leverhulme Trust Grant No. RPG-2019-020. Thanks to attendees at HaPoC 2021, Cliff Jones, and the anonymous reviewers, for providing useful comments. The author has no financial or proprietary interests in any material discussed in this article.

References

Abbate, J. (2017). From handmaiden to "Proper intellectual discipline": Creating a scientific identity for computer science in 1960s America. In T. J. Misa (Ed.), *Communities of computing*. Association for Computing Machinery and Morgan & Claypool.

Abbate, J., & Dick, S. (Eds.). (2022). *Abstractions and embodiments: New histories of computing and society*. Johns Hopkins University Press.

Astarte, T. K. (2022). "Difficult things are difficult to describe": The role of formal semantics in European computer science, 1960–1980. In J. Abbate & S. Dick (Eds.), *Abstractions and embodiments: New histories of computing and society*. Johns Hopkins University Press.

Backus, J. W. (1960). The syntax and semantics of the proposed International Algebraic Language of the Zurich ACM-GAMM conference. In *Information processing: Proceedings of the international conference on information processing* (pp. 125–132). UNESCO.

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., et al. (1960). Report on the algorithmic language ALGOL 60. *Numerische Mathematik, 2*(1), 106–136.

Ben-Ari, M. (1990). *Principles of concurrent and distributed programming*. Prentice Hall.

Böszörményi, L., Gutknecht, J., & Pomberger, G. (Eds.). (2000). *The school of Niklaus Wirth: The art of simplicity*. dpunkt.verlag/Copublication with Morgan-Kaufmann.

Brauer, W., & Reisig, W. (2009). Carl Adam Petri and "Petri Nets''. *Fundamental Concepts in Computer Science, 3*(5), 129–139.

Brinch Hansen, P. (1970). The nucleus of a multiprogramming system. *Communications of the ACM, 13*(4), 238–241.

Brinch Hansen, P. (1972a). A comparison of two synchronizing concepts. *Acta Informatica, 1*(3), 190–199.

Brinch Hansen, P. (1972b). Structured multiprogramming. *Communications of the ACM, 15*(7), 574–578.

Brinch Hansen, P. (1973). *Operating system principles*. Prentice Hall.

Brinch Hansen, P. (1976a). The Solo operating system: A concurrent Pascal program. *Software—Practice and Experience, 6*(2), 141–149.

Brinch Hansen, P. (1976b). The programming language concurrent Pascal. In F. L. Bauer & K. Samelson (Eds.) *Language Hierarchies and Interfaces*, *Part of Lecture Notes in Computer Science*. (Vol. 46, pp. 82–110). Springer.

Brinch Hansen, P. (1996). Monitors and concurrent Pascal: A personal history. In T. J. Bergin & R. G. Gibson (Eds.), *History of Programming Languages—II* (pp. 121–172). ACM Press.

Brinch Hansen, P. (2004). A programmer's story: The life of a computer pioneer. http://www.brinch-hansen.net/

Codd, E. (1962). Multiprogramming. In F. Alt & M. Rubinoff (Eds.), *Advances in computers* (Vol. 3, pp. 77–153). Elsevier.

Conway, M. E. (1963). A multiprocessor system design. In *Proceedings of the November 12–14, 1963, Fall Joint Computer Conference* (pp. 139–146).

Courtois, P.-J., Heymans, F., & Parnas, D. L. (1971). Concurrent control with readers and writers. *Communications of the ACM, 14*(10), 667–668.

Courtois, P.-J., Heymans, F., & Parnas, D. L. (1972). Comments on "A comparison of two synchronizing concepts by PB Hansen''. *Acta Informatica, 1*(4), 375–376.

Dahl, O.-J., Dijkstra, E. W., & Hoare, C. A. R. (Eds.). (1972). *Structured programming*. Academic Press.

Dahl, O.-J., & Nygaard, K. (1966). SIMULA: an ALGOL-based simulation language. *Communications of the ACM, 9*(9), 671–678.

Daylight, E.G. (2011, 03). Dijkstra's Rallying Cry for Generalization: The Advent of the Recursive Procedure, Late 1950s–Early 1960s. The Computer Journal, 54 (11), 1756-1772. https://doi.org/10.1093/comjnl/bxr002

Daylight, E. G. (2013). From mathematical logic to programming-language semantics: A discussion with Tony Hoare. *Journal of Logic and Computation, 25*(4), 1091–1110.

De Mol, L., & Bullynck, M. (2022). What's in a name? Origins, transpositions and transformations of the triptych Algorithm—Code—Program. In J. Abbate & S. Dick (Eds.), *Abstractions and embodiments: New histories of computing and society*. Johns Hopkins University Press.

Dick, S. (2015). Computer science. In G. M. Montgomery & M. A. Largent (Eds.), *A companion to the history of American science* (pp. 79–93). Wileys.

Dijkstra, E. W. (1959). Communication with an automatic computer. Doctoral dissertation. University of Amsterdam.

Dijkstra, E. W. (1962). Over de sequetialiteit van procesbeschrijvingen [on the sequentiality of process descriptions]. Circulated privately, available in Texas Archive. Retrieved from https://www.cs.utexas.edu/users/EWD/translations/EWD35-English.html (EWD35. Date inferred)

Dijkstra, E.W. (1963). Multiprogrammering en de X8 [multiprogramming in the X8]. Circulated privately, available in Texas Archive. Retrieved from http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD51.PDF (EWD51. Date inferred)

Dijkstra, E. W. (1964). Een algorithme ter voorkoming van de dodelijke omarming [An algorithm to prevent the deadly embrace]. Circulated privately, available in Texas Archive. Retrieved from http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF (EWD108. Date inferred)

Dijkstra, E.W. (1965). Cooperating sequential processes. EWD123, Texas Archive. Retrieved from http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF (Published as Dijkstra (1968a))

Dijkstra, E. W. (1968a). Cooperating sequential processes. In F. Genuys (Ed.), *Programming languages* (pp. 43–112). Academic Press.

Dijkstra, E. W. (1968b). Go to statement considered harmful. *Communications of the ACM, 11*(3), 147–148.

Dijkstra, E. W. (1968c). The structure of the "THE''-multiprogramming system. *Communications of the ACM, 11*(5), 341–346.

Dijkstra, E. W. (1968d). The structure of the "THE" multiprogramming system. In P.B. Hansen (Ed.), *The origin of concurrent programming* (pp. 139–152). Springer.

Dijkstra, E. W. (1971). Hierarchical ordering of sequential processes. *Acta Informatica, 1*, 115–138.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM, 15*(10), 859–866.

Dijkstra, E. W. (1976). *A discipline of programming*. Prentice Hall.

Dijkstra, E. W. (2001). What led to "Notes on Structured Programming". Held in the Dijkstra archive online. Retrieved from http://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1308.html (EWD 1308)

Dijkstra, E. W. (2013). Cooperating sequential processes. In P. Brinch Hansen (Ed.), *The origin of concurrent programming: From semaphores to remote procedure calls* (2nd ed., pp. 65–138). Springer.

Eden, A. H. (2007). Three paradigms of computer science. *Minds and Machines, 17*(2), 135–167. https://doi.org/10.1007/s11023-007-9060-8

Eglash, R. (2007). Broken metaphor: The master–slave analogy in technical literature. *Technology and Culture, 48*(2), 360–369.

Ensmenger, N. L. (2010). *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. MIT Press.

Fano, R. M., & Corbató, F. J. (1966). Time-sharing on computers. *Scientific American, 215*(3), 128–143.
Gill, S. (1958). Parallel programming. *The Computer Journal, 1*(1), 2–10.
Haigh, T. (2019). Assembling a prehistory for formal methods: A personal view. *Formal Aspects of Computing, 31*(6), 663–674.
Haigh, T., & Ceruzzi, P. E. (2021). *A new history of modern computing*. MIT Press.
Hicks, M. (2017). *Programmed Inequality: How Britain discarded women technologists and lost its edge in computing*. MIT.
Hoare, C. A. R. (1972a). Proof of correctness of data representations. *Acta Informatica, 1*(4), 271–281. Retrieved from https://doi.org/10.1007/BF00289507
Hoare, C. A. R. (1972b). Towards a theory of parallel programming. In C. A. R. Hoare & R. H. Perrott (Eds.), *Operating system techniques* (pp. 61–71). Academic Press (Proceedings of a Seminar at Queen's University, Belfast, August–September 1971)
Hoare, C. A. R. (1973). A structured paging system. *BCS, Computer Journal, 16*(3), 209–215.
Hoare, C. A. R. (1974). Monitors: An operating system structuring concept. *Communications of the ACM, 17*(10), 549–557.
Hoare, C. A. R. (1976). Parallel programming: An axiomatic approach. In F. L. Bauer & K. Samelson (Eds.), *Language hierarchies and interfaces* (pp. 11–42). Springer.
Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM, 21*(8), 666–677.
Hoare, C. A. R., & Jones, C. B. (1989). *Essays in computing science*. Prentice Hall.
IBM. (1968). *System/360 principles of operation* (8th ed.) (Computer software manual No. A22-6821-7).
Jones, C. (2023). Three Early Formal Approaches to the Verification of Concurrent Programs. *Minds & Machines*. https://doi.org/10.1007/s11023-023-09621-5
Jones, C. B. (2016). Professor Sir Tony Hoare: ACM Turing Award Winner 1980. Online (Interview)
Knuth, D. E. (1968). *Fundamental algorithms* (Vol. I). Addison-Wesley.
Kuhn, T. S. (1962). *The structure of scientific revolutions*. University of Chicago Press.
Lamport, L. (1974). A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM, 17*(8), 453–455. https://doi.org/10.1145/361082.361093
Lamport, L. (2015). Turing lecture. The computer science of concurrency: The early years. *CACM, 58*(6), 71–76. https://doi.org/10.1145/2771951
Mahoney, M. S. (2002). Software as science—science as software. In U. Hashagen, R. Keil-Slawik, & A. Norberg (Eds.), *History of Computing: Software Issues* (pp. 25–48). Springer-Verlag.
McCarthy, J. (1983). Reminiscences on the theory of time-sharing. Retrieved December 6, 2022, from http://jmc.stanford.edu/computingscience/timesharing.html
Mills, H. D. (1986). Structured programming: Retrospect and prospect. *IEEE Software, 3*(06), 58–66.
Misa, T. J. (2019). Gender bias in computing. In W. Aspray (Ed.), *Historical studies in computing, information, and society: Insights from the Flatiron lectures* (pp. 115–136). Springer.
Misa, T. J. (2021). Dynamics of gender bias in computing. *Communications of the ACM, 64*(6), 76–83. https://doi.org/10.1145/3417517
Mullaney, T. S., Peters, B., Hicks, M., & Philip, K. (Eds.). (2021). *Your computer is on fire*. MIT Press.
Nygaard, K., & Dahl, O.-J. (1978). The development of the SIMULA languages. In R. L. Wexelblat (Ed.), *History of programming languages* (pp. 439–480). Association for Computing Machinery. https://doi.org/10.1145/800025.1198392
Nyman, L., & Laakso, M. (2016). Notes on the history of fork and join. *IEEE Annals of the History of Computing, 38*(3), 84–87.
Peláez Valdez, M. E. (1988). A gift from Pandora's box: The software crisis. Doctoral dissertation. University of Edinburgh.
Peterson, G. (1981). Myths about the mutual exclusion problem. *Information Processing Letters, 12*(3), 115–116.
Petri, C. A. (1962). Fundamentals of a theory of asynchronous information flow. In *IFIP Congress* (pp. 386-390). North-Holland.
Petrick, E. (2022). The computer as prosthesis? Embodiment, augmentation, and disability. In J. Abbate & S. Dick (Eds.), *Abstractions and embodiments: New histories of computing and society*. Johns Hopkins University Press.
Raynal, M. (1986). *Algorithms for mutual exclusion*. MIT Press.
Rochester, N. (1955). The computer and its peripheral equipment. In *Proceedings of the Eastern Joint AIEE-IRE computer conference: Computers in business and industrial systems* (pp. 64–69).

Roubine, O., & Heliard, J.-C. (1980). *Parallel processing in Ada. On the construction of programs*. Cambridge University Press.

Strachey, C. (1959). Time sharing in large, fast computers. In *Proceedings of the IFIP congress* (pp. 336–341).

Tedre, M. (2014). *The science of computing: Shaping a discipline*. Chapman and Hall/CRC.

Wegner, P. (1976). Research paradigms in computer science. In *Proceedings of the 2nd international conference on software engineering* (pp. 322–330).

Whitaker, W. A. (1996). ADA—the Project: The DoD High Order Language Working Group. In *History of programming languages—II* (pp. 173–232). Association for Computing Machinery. https://doi.org/10.1145/234286.1057816

Wirth, N. (1971). *The programming language Pascal. Acta informatica, 1*(1), 35–63.

Wirth, N. (1977). Modula: A language for modular multiprogramming. *Software: Practice and Experience, 7*(1), 1–35.

Wirth, N. (1980). *Modula-2* (Vol. 36). Tech. Rep. ETH Institut für Informatik.

Wirth, N. (2007). Modula-2 and Oberon. In *Proceedings of the 3rd ACM SIGPLAN conference on history of programming languages*. https://doi.org/10.1145/1238844.1238847

Woodger, M. (1978). A history of IFIP WG 2.3: Programming methodology. In D. Gries (Ed.), *Programming methodology: A collection of articles by members of IFIP WG 2.3*. Springer.