Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

UNIVERSITÉ DE NAMUR

University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Edsger Wybe Dijkstra

first years in the computing science (1951-1968)

van den Hove, Gauthier

Award date:
2009

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 23. Jun. 2020

Edsger Wybe Dijkstra

First Years in the Computing Science (1951–1968)

Gauthier van den Hove

Edsger Wybe Dijkstra

First Years in the Computing Science (1951–1968)

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Computing Science
by Gauthier van den Hove

Promoter: Prof. Wim Vanhoof
Co-Promoter: Prof. Baudouin Le Charlier

University of Namur
Faculty of Computing Science
Academic Year 2008–2009

See http://www.fibonacci.org/ewd/ for current information about this work.

Copyright © 2009 by Gauthier van den Hove <gauthier@fibonacci.org>

All rights reserved. No part of this work may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the author.

First printing, 26 August 2009

The word which I will speak is not mine.
Plato

Do only what only you can do.
Dijkstra

Abstract

History is often considered as a useless occupation, which conveys nothing other than a knowledge of the past. This is especially true for the history of sciences: a scientific curriculum does usually not comprise a single course on the history of the science concerned. We will try, in this work, to see if an in depth study of the achievements and writings of an important author in a discipline can help us to gain a better understanding of the discipline itself. The chosen discipline is the computing science, and the author is Dijkstra. We restrict our study to the first two decades of his life as a programmer.

Résumé

L'histoire est souvent considérée comme une occupation inutile, qui transmet une simple connaissance du passé. C'est particulièrement vrai pour l'histoire des sciences: les programmes des études scientifiques ne comportent d'habitude pas un seul cours d'histoire de la science en question. Nous essayerons dans ce travail de voir si l'étude approfondie des écrits et des réalisations d'un auteur important dans une discipline peut nous aider à mieux comprendre cette discipline elle-même. La discipline choisie est l'informatique, et l'auteur est Dijkstra. Nous restreignons notre étude aux deux premières décennies de sa vie de programmeur.

Acknowledgments

Before opening this work, I would like to thank my parents and my brother and sisters for their love and support. I would also like to thank Dean M. Boyancé, to which I owe so much I can't even write, Prof. B. Mathonat and Prof. E. Brochier, Prof. J.-L. Chrétien, for his inspiring lectures, Father M. Rougé, Father R. Gallagher, c.ss.r., for correcting my English, and my friends, Robin, Tanguy and Louise, the Balaÿ's family, Geneviève, Paul and Jeanne, Claire, Hélène, Sébastien and Aude-Marie, Nicolas, François, Laurent, my choirmaster, and finally the Monastic Fraternities of Jerusalem, especially Sister Anne and Brother Michel-Marie. Last but not least, I would like to thank Prof. W. Vanhoof, who accepted the direction of this thesis, Prof. P.-A. de Marneffe and Prof. P. Goujon, who accepted to be part of the examination board, and finally Prof. B. Le Charlier, for his continuous support, encouragement and assistance over the last seven years: if he wasn't there, this work would certainly not exist.

TABLE OF CONTENTS

Introduction 1

Biographical Elements 5

Chapter I — **Computer Design (1951–1959)** 7
1. Introduction to Programming 7
2. The X1's Interrupt Handler 10
3. Thoughts on Programming 13

Chapter II — **A Programming Language (1959–1962)** 23
1. The Definition of ALGOL 60 23
2. The "MC" ALGOL 60 Compiler 29
3. Thoughts on Programming 43

Chapter III — **An Operating System (1962–1968)** 55
1. Loosely Synchronizing Concurrent Processes 55
2. The "THE" Multiprogramming System 69
3. Thoughts on Programming 83

Conclusion 95

Bibliography 99

INTRODUCTION

Before embarking on an ambitious project,
try to kill it.
Dijkstra

Dijkstra is often considered as one of the founding fathers of the computing science, along with Turing, von Neumann and Knuth for instance. A characteristic of a founding father is that he gives a new discipline some of its core concepts and methods, and this could well be true for Dijkstra. However, another characteristic of a founding father is that his writings are carefully studied by at least some of the later thinkers in the field, and by this criterion one may argue that Dijkstra really isn't one of them. Of course, most know that he invented an algorithm to compute the shortest path in a graph, that he once wrote a paper entitled "Go To Statement Considered Harmful", that he was a ardent proponent of the so-called "structured programming", and of the use of formal methods. Those who have a more extensive culture in the history of computing science may also cite the inventions of the stack, the semaphores, and the guarded commands. But except for those commonplaces, his writings, which extend over about eight thousand pages, and do probably include more than those few ideas, are for the most part simply ignored: no systematic study of them has ever been undertaken, and no critical edition is planned. This is the gap we will try to begin to fill.

The specific aim of this work is twofold: on the one hand, to give an insight into some of the core concepts of computing science, and on the other hand, to illustrate a way of thinking. In the hope of giving an insight into those core concepts, we will try to precisely identify the origin of the difficulties which lead to their creation. Indeed, one cannot correctly understand the nature of a concept if one does not see how it was born, that is, for what problem it was a solution, what was the difficulty that it allowed to overcome. Those concepts are not really as simple as it might seem to today's computing scientists: if they now look evident, it is because thinkers in the past struggled hard to create them. We will therefore not present raw sequences of facts, or a patchwork of more or less random inspiring quotations, but we will instead try to identify the connections between them: we will show how the problems lead to solutions, how the solutions in turn lead to new problems, and how experience leads to new ideas and new methods. By doing

this we expect to illustrate a way of thinking, a way of approaching new problems. The two goals we pursue may both be inspiring for today's programmers: on the one hand, the problems that the first programmers had to solve are indeed often typical, and we constantly use the concepts they brought to the fore; on the other hand, the problems we are faced with are always new in some of their aspects.

Our aim is thus neither to extol Dijkstra's merits or to write his biography, nor to attribute to him the paternity of this or that concept. The former would be of limited interest, and further it does not belong to the type of work we have to achieve. The latter would imply to compare his writings with those of all other thinkers of the field which were published before or at the same time. This is clearly an impossible task, both because the terminology is extremely fluctuating during the first decades of a new discipline, and because similar ideas are often put forward by many different people. Further, many ideas which seem novel when they appear in a given field of human knowledge did already exist before, in a similar form, in one or more of the other fields of human knowledge. Finally, we believe that the question of the paternity of ideas is relatively unimportant. It is indeed far more enlightening to understand for what reasons someone became aware of this or that idea. We venture the hypothesis that the sequence of problems that any programmer is faced with during his life is, more or less, the same: if this is true, then we may take some benefit from Dijkstra's experience and walk faster on that path, avoiding some blind alleys we would otherwise have been tempted to take.

The inherent limits of this kind of work are numerous. The first and most blatant one is that it does not contribute to the progress of the discipline, as it does not propose or evaluate any new ideas. But this is true for any synthesis effort, and yet they are not devoid of interest. Further, we observe that the claim of novelty attached to many ideas often proves later to be false: at a closer look, they reveal to be old ideas, which have been forgotten because their limited interest had already been experienced. The study of the old ideas may then have an interest to identify truly new ideas. A second inherent limit is that this type of work is partial and subjective, as it is centered around a single author. But it is anyway not possible to attain neutrality and objectivity whenever one wants to go beyond the raw report of facts. In this context, it is then better to study the opinions of a single author in depth than to present a colorless average of conflicting opinions of multiple authors. A third limit is that it will inevitably give an impression of generality, because it does not make use of mathematical formulæ, and because it does not cover all the technical details. But the absence of technical details and mathematical formulæ does not necessarily imply an absence of precision, and we note that, on the contrary, the presence of formulas may often give a false impression of rigor. Finally, a fourth limit, related to the previous one, is that this work, although its subject is clearly delimited, is not exhaustive: many traits will be passed over in silence. But again this is true for any synthesis effort, whose aim is to bring the essentials to light. This can be done only by leaving the secondary points out.

Covering the whole subject in the imposed limits for this work revealed itself to

be impossible; therefore we had to restrict our research to the first two decades of Dijkstra's achievements and writings. The choice of 1968 as the ending limit of our present study is however not entirely arbitrary: it marks the beginning of the "software crisis", identified during the 1968 NATO Software Engineering Conference, which corresponds to a strong shift in Dijkstra's interests. Those seventeen years can be divided in three periods, corresponding to Dijkstra's three successive centers of interest: the first eight years, during which he was involved in the design of four successive computers, and designed an interrupt handler for the last one (Chapter I), the next three years, during which his efforts were mostly centered around ALGOL 60 (Chapter II), and finally the last six years, during which he had the responsibility of building an operating system (Chapter III). To set the scene, we will first present some biographical elements, and then, for each of those periods, we will devote a section to the analysis of his main achievement (§ 2), a section to the analysis of the work preparing it (§ 1), and finally a section to the analysis of his writings (§ 3). We hope, by these means, to achieve our aims as well as possible.

Paris, 26 August 2009

BIOGRAPHICAL ELEMENTS

Dijkstra is born on 11 May 1930 in Rotterdam, The Netherlands. He is the third of four children. His father, D. W. Dijkstra, is a chemist; he is teacher, and then principal, in a secondary school in Rotterdam. His mother, B. C. Kluyver, is a mathematician.

He enters primary school in 1936, and secondary school in 1942, which he finishes in 1948, with the highest possible marks in all scientific disciplines. He wants to study law to represent his country at the United Nations, but his parents and his teachers persuade him to engage in scientific studies. He enters the University of Leyden to study mathematics and theoretical physics. Three and a half years later, in March 1952, while pursuing his studies in Leyden, he is hired by the Computation Department of the Mathematical Center of Amsterdam. He completes his studies in Leyden in 1956. He presents his PhD thesis on 28 October 1959 at the University of Amsterdam.

On 23 April 1957, he marries M. C. Debets. They have three children: Marcus, Femke, and Rutger.

In September 1962, he is appointed Professor of Mathematics at the Technological University of Eindhoven. In August 1973, he joins Burroughs as a Research Fellow, and becomes Professor Extraordinarius at Eindhoven. In 1984, the Computer Science Department of the University of Texas in Austin offer him the Schlumberger Centennial Chair; he leaves Burroughs and the Technological University of Eindhoven, and settles in the United States. He becomes Emeritus in 1999.

In November 2000 he is diagnosed with cancer. He goes back to the Netherlands in April 2002, and dies in Nuenen on 6 August 2002. His cremation takes place four days later.

CHAPTER I

COMPUTER DESIGN (1951–1959)

As a reward for having passed his third year of theoretical physics studies in Leyden, Dijkstra's father offer him to participate in a introduction to programming course on the EDSAC in Cambridge, in September 1951. In connection with a letter of recommendation, he meets A. van Wijngaarden, head of the Computation Department of the Mathematical Centre, who offers to hire him as a programmer on his return. In March 1952, he accepts his proposition; he works part-time in Amsterdam, while pursuing his studies in Leyden. He gives his first courses in 1955, with van Wijngaarden. He soon decides to become a programmer, rather than a theoretical physicist, and completes his studies in theoretical physics as fast as possible, in 1956. Up to 1959, he will be involved in the design of the ARRA, the FERTA, and the ARMAC (§ 1) — and finally of the X1, for which he designs and writes an interrupt handler (§ 2).

§ 1. Introduction to Programming

One of the main objectives of the department van Wijngaarden leads being to design and construct a computer, he had hired, four years earlier, two students in experimental physics: C. S. Scholten and B. J. Loopstra. However, because of their lack of experience, upon Dijkstra's arrival, the machine under construction, the ARRA, still does not work reliably. In November 1952, G. A. Blaauw, a Dutch engineer who just got his PhD at the Computation Laboratory of the Harvard University, is hired; he'll work with them for two and a half years. He convinces the team to replace the electromechanical relays with electronic components, and thirteen months later, in December 1953, a totally new machine, also called ARRA (or sometimes ARRA II, to distinguish it from the original ARRA), runs its first programs.

The small team organizes as follows: after having discussed and decided together on the characteristics of the machine to build, Dijkstra is in charge of writing the programmer's manual, containing a complete functional description of the computer, as well as the notation conventions for the code, which can be considered as a kind of minimal assembly language. Afterward, while the rest of the team builds a machine meeting that description, he writes the basic communication programs, permitting reading and writing

of punched tapes, and the use of the keyboard and the typewriter: this way, the machine and the programs needed to use it are ready at the same time. He's also in charge of writing the numerical subroutines (square and cubic roots, trigonometric functions, calculations on fractional numbers, ...) permitting a more advanced use of the machine.

Technically speaking, the ARRA (1953) and the two machines who follow it, the FERTA (1955) and the ARMAC (1956), are very similar; the FERTA is moreover an improved copy of the ARRA. They are binary machines (that is, not binary-coded decimal machines) working with two accumulators, with a memory of 1024 or 4096 words having a length of 30 or 34 bits. Their thirty or so instructions are for the most strictly arithmetic, and executed at an average speed of forty to thousand instructions per second. The speed increase is mostly due to various improvements (for instance, the presence of two cache memories in the ARMAC, one for the instructions and one for the data); they make the FERTA two times faster than the ARRA, and the ARMAC, ten times faster than the FERTA. Also, the presence of parity bits in the ARMAC make it safer to use than the preceding ones.

They are strictly sequential machines, which means that they only execute a single program, and that its execution is strictly sequential: the individual instructions of that program are performed one after the other. This means particularly that the communication (input and output) operations cannot be completed while other instructions of the program are executed. A certain concurrent execution is however allowed thanks to the following optimization: the execution of a communication instruction only blocks the execution of that same instruction for a certain time, the time usually needed for the corresponding communication operation to complete. The machine can thus execute a few other instructions while the communication operation proceeds, and the program will temporarily be blocked if it asks for the execution of such an instruction during that period of time, until it has elapsed. This optimization is, however, not without defects: after an input operation, one has to take care of the number of instructions to execute, and of their execution time, before one can use the result; after an output operation, one has to take care of the particular cases of characters who, like the end of line character for instance, take longer to print out. Those limitations justify the writing of higher level communication routines taking care of all these details for the programmer.

A typical example of his programming work during those years is the ARMAC's division subroutine. The ARMAC does not have floating point numbers, but it is possible to circumvent that limitation by observing that a word of n bits $b_1 \dots b_n$, with the most significant bit numbered 1, which is usually interpreted as representing the number $b_1 \times 2^{n-1} + \dots + b_n \times 2^0$, can have another interpretation: it can be interpreted as representing the number $b_1 \times 2^{-1} + \dots + b_n \times 2^{-n}$. The multiplication, subtraction and addition operations do not need to be adapted to this fractional or fixed point interpretation, but it is then possible to implement a subroutine to perform the division of two such numbers:

If the machine does not have built-in division, it needs to be programmed. We suppose that the quotient of the division a/b of two fractional numbers is smaller than 1 in absolute value. The iterative processes to calculate the reciprocal b^{-1} are:

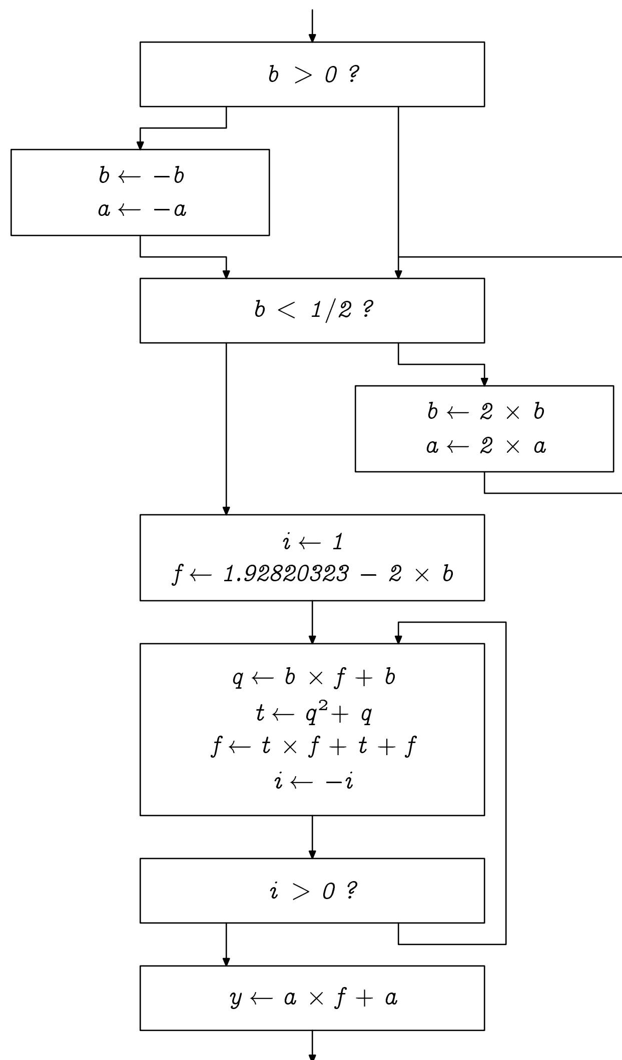
$$c_n \times (2 - b \times c_n) = c_{n+1} \quad c = \lim c_n = b^{-1} \text{ (quadratic convergence)}$$
$$c_n \times (3 - 3 \times (b \times c_n) + (b \times c_n)^2) = c_{n+1} \quad c = \lim c_n = b^{-1} \text{ (cubic convergence)}$$

Initial approximation: $f_0 \leftarrow 1.92820323 - 2 \times b$. Instead of c_n, the machine manipulates $f_n = c_n - 1$.

Iteration scheme:

$$q_n \leftarrow b \times f_n + b - 1$$
$$t_n \leftarrow q_n^2 - q_n$$
$$f_{n+1} \leftarrow t_n \times f_n + t_n + f_n$$
$$f = \lim f_n = b^{-1} - 1$$

Completion: $y \leftarrow a \times f + a$, whereby the operation $y \leftarrow a/b$ is carried out.



The flowchart is executed in such a way that there are always two iterations. The error is then smaller than 0.5×10^{-10}, which is enough for the precision of the ARMAC.[1]

1. Cf. Dijkstra, E. W., *et al.*, *Programmering voor Automatische Rekenmachines*, pp. 56–58

As one sees, the main problems are numerical analysis ones: one the one hand, to find ways to perform elaborate operations with a very restricted instruction set, and on the other hand, to arrange the concrete calculations in order to make sure that the numbers never get out of the bounds $]-1, 1[$. The computer indeed manipulates $f_n = c_n - 1$ instead of c_n since for $0.5 \leqq b < 1$, one would have $1 \leqq b^{-1} < 2$.

Other typical examples of his programming work are the shortest path and the minimum spanning tree algorithms which were both invented, during the year 1956, with a strictly practical aim: the former to demonstrate the power of the ARMAC during its official inauguration, the latter to minimize the use of copper in the wiring of the next computer designed by the team.

§ 2. The X1's Interrupt Handler

No sooner is the ARMAC finished that the team starts the construction of a new machine: the X1. This time however, the conditions are somewhat different: because the mission of the Mathematical Centre is not to manufacture computers, Loopstra and Scholten, together with an insurance company, set up a firm to produce and sell it: Electrologica. But, because the transition cannot happen immediately, the X1 will still be conceived at the Mathematical Centre; it will be completed in 1958.

The X1 is a binary machine with, besides the two accumulator registers, an index register, and a few one-bit condition registers; it has a memory of at most 32768 words (with parity checks) having a length of 27 bits, and it works with about fifty instructions. It is ten times faster than the ARMAC. But it is, technically speaking, revolutionary in two of its aspects: it is the world's first fully transistorized machine, and it is one of the world's first machines having an interrupt system.[2]

The addition of the interrupt mechanism has two goals: efficiency and adaptability. As for efficiency, it aims at circumventing the inherent limitation of the communication apparatus, whose speed cannot be improved as fast as the calculation speed of the computer itself: waiting for the completion of a communication instruction is not tolerable any more on a computer like the X1, calculating more than hundred times faster than the ARRA. Further, one of the design requirements of the X1 being its adaptability, communication apparatus should be easy to add depending on the needs of the user. Because the speed of the different operations of those a priori unidentified devices cannot be known in advance, it is not possible to hard-wire their control entirely in the X1: a part of the control, the determination of the end of their various operations, has to be transferred to the devices themselves. This is where the interrupt system takes place: when a program executes a communication instruction, the control returns to the program without waiting for the actual completion of the communication operation; when this operation finishes, the communication equipment signals its completion to the X1 by means of an interrupt signal. The X1 reacts to that signal by transferring the control to a

2. Cf. LOOPSTRA, B. J., *The X1 Computer*, and DIJKSTRA, E. W., *Communication with an Automatic Computer*, pp. 2–37

"communication program," or interrupt handler, which handles the situation, eventually transferring the control back to the program. If the program does not make use directly of the communiction instructions, but instead performs its communication operations by calling higher level subroutines of the communication program, its communication operations are then spread over the calculation time.

Once again, Dijkstra is in charge of deciding on the notation conventions for the programs, and of writing the basic communication programs. It is the result of this work that constitutes his PhD thesis, written under the direction of van Wijngaarden.

This time, his work is significantly more complicated. Indeed, because of the presence of the interrupt mechanism, the X1, contrary to its predecessors, is not a strictly sequential machine any more. It still executes only one program, but an interrupt, signaling the completion of a communication operation, can take place at any moment, and this makes it a non-deterministic machine. But, as he notes:

> It is clear that this may not imply to make the task heavier for those who the machine primarily use as a tool to obtain results. The conception of the interrupt mechanism gives thus the duty to build up [a communication program] which on the one hand plucks an important part of the possible fruits of the parallel programming, and on the other hand does not burden the user unnecessarily.[3]

The terms "parallel programming" refer to the intertwined execution of the main program and the communication program. It is intertwined because the communication program does more than simply calling the primitive communication instructions to read or write a single word or character: it implements higher level routines to read or write a given number of words in a given format, possibly with a conversion. For instance, if a program asks for the writing of a word as an integer, the communication program determines which is the first character to type out, executes a write instruction for that character, and returns to the main program; when the typewriter has finished typing that character, the X1 is interrupted, and the communication program proceeds with the next character, etc.

> Naturally, after every intermediate interruption the control returns to the main program with complete restoration of the status quo.[4]

This means that the communication program should save and restore, besides its own internal state, the state of the running program, that is, the state of the computer when the program was interrupted: the contents of the various registers, and the instruction counter.

To prevent mangling those records when a program asks for a communication operation while the communication program is still processing a previous request of the same kind, which of course cannot be forbidden, the communication program should be "synchronized" with the communication apparatus, that is, it should automatically

3. Dijkstra, E. W., *Communication with an Automatic Computer*, p. 132

4. Dijkstra, E. W., *Communication with an Automatic Computer*, p. 78

wait for the previous communication operation to complete before proceeding with the next one. This further prevents that the results of those operations be corrupted.

Finally, as many different communication apparatus can be connected to the X1, and operate at the same time, it may be desirable to give them different priorities with respect to each other, to ensure that a more urgent task will be processed before a less urgent one. The X1 has therefore seven interrupt "classes", defining seven priority levels. The class with the lowest priority, which is even lower than the running program, is reserved for the console keyboard; the six other classes, which have a higher priority than the running program, are for the communication devices. At any given moment, interrupts of those classes may be allowed to take place or not, depending on the "interrupt permit" bit of their class; it is the responsibility of the communication program (and of other programs) to use them according to their needs. For the communication program, this means for instance disallowing interrupts of the same class to take place while it is running. Furthermore, an additional global interrupt permit bit makes it possible to prevent all interrupts without altering the individual permit bits of the seven classes.

It is now clear that the requirement of adaptability has been met by this organization, as communication apparatus are easy to add and to manage:

> It is possible to make a communication program for the X1 without paying any attention to the relative timing of the external apparatus on the one hand and the X1 on the other hand.[5]

Indeed, the transfer of a part of the control to the external devices (namely, the determination of the end of their various operations, and consequently the control of the start of the communication program), to solve the problem of the unknown timing of their operations, in turn poses other problems; but these can been solved by the careful writing of a suitable communication program.

The communication program written by Dijkstra for the X1 consists of about a thousand instructions, about fifty reserved words holding variable values, and about twenty constant words.[6] It can deal with four communication apparatus: the console keyboard of the X1, a typewriter, a tape reader and a tape punch. The part of the communication program handling the tape reader includes an assembler, which translates the sequence of five-bit words read on the tape into instructions, numbers or full words, depending on the directives punched on the tapes. The communication program is hard-wired in the X1, but it is designed to be extensible to other apparatus: the control can be transferred at will from its central part to instructions in live (not hard-wired) memory.

To illustrate that extensibility, after having presented the communication program, Dijkstra solves two additional problems: namely, how to optimize the usage of the computing time by further spreading the communication operations over the calculation time, and how to couple an input and an output device in such a way that they can work together independently of a running program. The first problem is described as follows:

5. Dijkstra, E. W., *Communication with an Automatic Computer*, p. 27

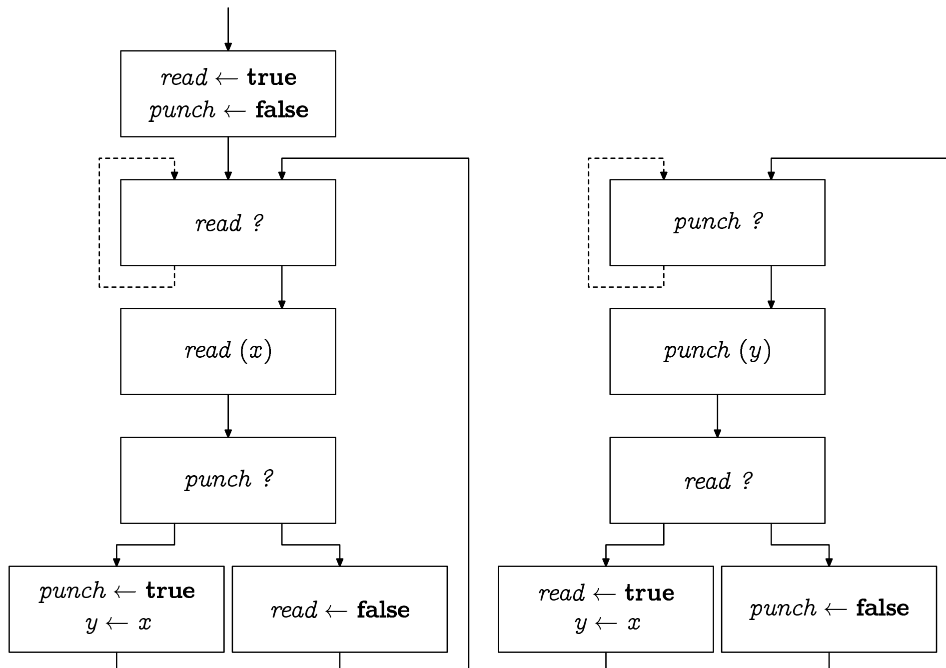
6. Cf. Dijkstra, E. W., *Communication with an Automatic Computer*, pp. 138–166

> The use of [synchronized output operations] can imply a loss of potential computing time if two such calls succeed each other too quickly: when the second call is encountered the control waits [...] until the previous [communication operation] has been completed before actually starting on the new [communication operation]. Only when the latter has been started does the control return to the main program. [...] If the time required by the [output operation] is so much in excess of the total time necessary for the computation that the typewriter nevertheless operates at full speed, this loss in computation time is fictitious. Real loss in total time only occurs when the X1 is unproductive during a "concentrated" number of [output operations], while the [output devices are] not active at a later stage, because the X1 is still busy forming the next result.[7]

It can be solved by the use of a buffer. The solution given has the form of a flowchart, taking the example of a one-line output buffer, but it is directly applicable to larger buffers and it gives the general scheme of the solution. As to the second problem, it is introduced as follows:

> Our next problem is to construct a program that reads a tape and types and/or punches out data depending on what has been read. One of the simplest examples is merely reproducing a tape. Another example is typing out the binary words from a tape in decimal form. In the latter case a number of symbols to be typed out are derived from a group of consecutive [five-bit words] each time. We now demand that it must be possible to execute this program simultaneously with any other program.[8]

The problem is thus to synchronize input and output mutually. The principle of the solution is given as a flowchart using two buffers (whose size is not specified, but depends on the calculations that have to be performed on the input data before sending the result to the output device) and two one-bit variables:[9]



7. Dijkstra, E. W., *Communication with an Automatic Computer*, pp. 95–96

8. Dijkstra, E. W., *Communication with an Automatic Computer*, pp. 113–114

9. Cf. Dijkstra, E. W., *Communication with an Automatic Computer*, p. 114

§ 3. Thoughts on Programming

Dijkstra's thoughts about programming are not yet much developed, but some remarks are worth mentioning. For example, the introduction of the *Functional Description of the ARRA* (1953) begins with:

> The ARRA is an automatic digital machine. In what follows is that machine described, inasmuch as it is relevant for someone who *uses* the machine: it will be described *what* the machine does, not *how* the machine works.[10]

It is noteworthy that, right from the first sentences of the first report he writes, the distinction between functional description (or specification) and implementation (or operational description) seems so important to him that he begins with it; whereas it is not evident, and it will generally be recognized only more than twenty years later that this distinction is a fundamental one. In the remainder of the report, one finds, after a minimal description of the different parts of the machine (the memory, the control unit, the calculating unit and its registers), a very clear and precise description of the twenty-five instructions, and of each of the communication (input and output) units, without any reference to the way they are built.

As for the nature of programming itself, he begins with a kind of general description of his activity, indicating how programming fits within it:

> The specific task of the programmer is a part of the preparation that is needed before the machine can begin to calculate. To be complete, here are the most important stages:
> 1st. mathematical formulation of the problem,
> 2nd. mathematical solving of the problem,
> 3rd. choice or construction of the numerical processes, which [...] to the desired result lead,
> 4th. *programming*: detailed building of those [...] processes, with the elementary operations, by which the machine directly in state is to solve the problem,
> 5th. *coding*: writing of the program in the code of the machine, in such a way that afterward a tape can be directly punched.[11]

One should not interpret those remarks as indicating that he already had a clear understanding of a role of mathematics in programming. They are the consequence of the fact that computers were considered and used above all as calculating machines, as reflected, for that matter, by their instruction set: the problems they have to solve being mostly numerical, their programming requires a mathematical formulation and solving of the problem in question (see p. 8). That formulation and solving, as well as the choice of the numerical processes to realize that solving, aren't part, strictly speaking, of the programming activity. Programming and coding are clearly set apart, the former being a matter of construction and organization, the latter, a nearly mechanical translation of the elementary operations of the program into instructions executable by the machine. In concrete terms, programming consists in writing down a flowchart for the processes,

10. DIJKSTRA, E. W., *Functionele beschrijving van de ARRA*, p. 1

11. DIJKSTRA, E. W., *Functionele beschrijving van de ARRA*, p. 33

coding in translating the components of that flowchart into instructions.

It is clear that the mathematical formulation and resolution of the problem are entirely independent of the machine on which the problem will be coded. On the contrary, the last stage of the process, translating the program into machine code, is of course completely dependent of it. Thus, as one goes along the stages, the dependence of the machine is more and more significant, and the intermediary stage of programming, while not as independent of the machine as the mathematical stages, still permits some general considerations:

> Programming is however, still such a general problem, that things can be said about it, that are not only applicable to the ARRA.[12]

So, how should one proceed to write a program?

> A program breaks down into a few distinct pieces in a natural fashion, for instance, the instructions, a series of constant numbers, a series of separated changing parameters of general numerical data, and a series of so-called *working spaces* (addresses where intermediary results are stored in the meantime). But the instructions can also be divided in groups: "this piece does this" and "this piece does that" etc.
>
> When sketching a program out, one think initially solely in such terms: one first builds the program in broad strokes. (Like: "This table must stay in the memory, different quantities — who must be processed together in different ways — must be interpolated from this table. Interpolation in the table is thus separated into a 'small group of instructions'. We have to calculate a sinus multiple times. This and that number are always calculated by iteration: the piece of program in charge of this, may as well be isolated as a group of instructions that could be considered as an entity, etc.")
>
> Then the programmer makes those separated pieces.[13]

The division criterion that should be used to break a program in parts is the sheet, corresponding to a page in the memory:

> A track contains 32 successive addresses: those are also called a *sheet*: with this last name the accent is differently placed: the track is a concrete unit, that one can indicate on the drum, the sheet is a (paper) unit, by which the instructions are divided in groups by the programmer.[14]

Instructions are thus assembled in groups of thirty-two words; that group forms a sheet, the unit into which programs are divided and written. It is clear then that subroutines are not the concept that gives structure to programming: the programmer does not use them in his programs, but thinks in terms of sheets.

If computers have instructions to call and return from a subroutine, their usage is often limited to the call of the functions of the standard library. The subroutine call and return instructions are simply regarded as specialized jump instructions, which differ from the normal jump instruction in that they store or retrieve an address somewhere in the computer's memory. Subroutines are thus hardly ever used to write programs,

12. DIJKSTRA, E. W., *Functionele beschrijving van de ARRA*, p. 33

13. DIJKSTRA, E. W., *Handboek voor de programmeur — FERTA*, p. 37

14. DIJKSTRA, E. W., *Handboek voor de programmeur — FERTA*, p. 37

the only subroutines are the functions that are part of the library, stored more or less permanently in memory, and written so as to be usable by different programs. Typical subroutines offer the usual numerical and trigonometrical functions (square and cubic roots, logarithms, sine, cosine, . . .), calculations on fractional numbers, input and output operations, conversion of numbers between binary and decimal bases, . . .

> A subroutine is a series of instructions, which perform together a standard operation.
>
> It is indeed the task of the programmer to build up a specific calculation with the instructions. [...] It would be needless much work for him, if he had to build up each program with those little stones, because each calculation can be split into bigger parts, whose function is so general, that the same series of instructions certainly will also appear in another program. [...]
>
> Suppose that we want to type out the content of a memory address [...] as a whole number; and that a series of instructions in a subroutine [...] type out the number contained in the register S as a whole number. One could punch those instructions [...] on a tape and, each time that a whole number must be typed out, arrange the program so that the number to be typed out in S is, then copy the series of instructions on the band, and then continue with the program. [...] However, this would imply, if later on in the program another whole number must be typed out, entering those instructions a second time: an identical series of instructions must then be read in two times, and is in duplicate in the memory, which is a real waste of memory space.
>
> One would readily let the control go through the same series of instructions both times. This means however that, after the type operation, the control should encounter a variable control redirection, according to whether the first or the second number was typed out. This variable control redirection is called a coupling instruction.
>
> In concrete terms, the execution goes now as follows: Somewhere in the memory are a series of instructions, which perform the required operation (in this case the typing out of the content of a register interpreted as a whole number). Each time a whole number must be typed out the control is sent to that series of instructions, with two information: [...] the number that should be typed out, [and the coupling instruction] which sends the control back to the right place in the program after completion.[15]

The main benefit of standard subroutines is that they offer a gain in memory space. They are simply understood at the start (1953) as an alternative to copying a series of identical instructions multiple times in memory, at different places in a program, or in different programs. But that's not all, and their use also has drawbacks.

> The advantages to the use of subroutines are the following ones:
>
> 1st. It limits the possibilities of errors in the program. [...]
> 2nd. It reduces the time needed to build up the program.
> 3rd. It reduces the punch time. [...]
> 4th. It gives the program more clarity. [...]
> 5th. It may, and will most often, give an important saving of time. [...]
> 6th. It may, and will most often, give an important saving of space. [...]
>
> The disadvantages to the use of subroutines are the following ones:
>
> 1st. It increases the number of conventions. [...]
> 2nd. A source of errors may arise because a programmer makes use of a subroutine without precisely and clearly knowing what in that subroutine happens instruction by instruction.

15. Dijkstra, E. W., *Functionele beschrijving van de ARRA*, p. 36

> 3rd. It may, but will not frequently, give a loss of time. [...]
> 4th. It may give a loss of space. [...]
> 6th. The machine executes irrevocably additional operations when taking care of the coupling instruction, and the adaptation to the special conventions of the subroutine.[16]

Anyway, their main benefit is the savings they permit: in calculation time, in memory space, in programming time, and in the time needed to punch and read tapes. The fact that it could be necessary for the programmer who wants to use the subroutine to understand "precisely and clearly" what in that subroutine happens, shows that the distinction between functional and operational description is not evident right away in the practice of programming. But a short while thereafter (1955), his understanding of the nature of subroutines is a little more refined:

> By subroutine, we mean a series of instructions, which together perform a clearly delimited operation. [...] If a subroutine [...] is somewhere in the memory, the calling of the subroutine in the main program acts as an extension to the instruction set.[17]

The series of instructions grouped within a subroutine is henceforth seen as a unity, to the extent that it is considered as a kind of additional instruction.

Besides those reflections on the nature of the programming activity, one finds a few thoughts about the programs themselves. It is desirable indeed for every program that it respects certain general properties, that is, properties which do not depend on some given machine on which it is executed. But they exist only because programs exist, and as such, they are part of the concern of programming.

> The ideals that one pursue in the building of a program are the following:
> 1st. maximal speed [...],
> 2nd. minimal memory usage [...],
> 3rd. maximal safety,
> 4th. maximal accuracy,
> 5th. maximal flexibility,
> 6th. maximal clarity.
>
> 1st and 2nd. The first two ideals are, up to a certain point, conflicting. [...]
>
> 3rd. The pursuit of safety is expressed by the precautions so that the machine gives no wrong answer.. [...]
>
> 4th. The ideal of accuracy means that one should be vigilant that precision is not unnecessarily lost by the use of clumsily chosen methods. [...]
>
> 5th. The pursuit of flexibility is very clearly expressed in the building of subroutines [...] whereby a series of instructions once in the memory is stored, which carry out a complicated operation, that in the course of the program multiple times occur. [...] Flexibility also requires that the programmer the program organizes in such a way that, if an error is found (which is almost always the case), he has not to rewrite the whole program.
>
> 6th. Clarity is an obvious precept.[18]

16. DIJKSTRA, E. W., *Functionele beschrijving van de ARRA*, pp. 38–39

17. DIJKSTRA, E. W., *Handboek voor de programmeur — FERTA*, pp. 40–42

18. DIJKSTRA, E. W., *Functionele beschrijving van de ARRA*, pp. 33–35

The usual opposition between the speed and the memory usage of a program is underscored: on the ARRA for instance, a program calculating the sum of hundred successive numbers will be executed in 2.5 seconds if it is composed of one instruction for each of the terms of the sum, of course occupying about a hundred words in memory, and in 14 seconds if it is written as a loop adding the successive numbers, occupying only about twenty words in memory. The very presence of other requirements than speed and memory usage is uncommon, and it is therefore worth trying to understand them better.

Safety stems from the unreliability of the machines: writing or reading a word in memory sometimes happen incorrectly, without it being detected, because of the lack of error detection mechanisms, which are not systematically used: the ARRA and the FERTA do not have such mechanisms. Safety consists thus in trying to detect the errors of the machine, for instance by doing the calculations on data for which the result is already known, or by performing the same calculations two times in a row and comparing the results. The same kind of errors may arise in the input-output operations, and similar safety checks can be devised.

Accuracy does not consist in insuring that the program gives correct results (a requirement that is probably so evident that it does not need to be explicitly stated), but that those results are as precise as possible: it is well-known, for instance, that in numerical calculations an inappropriate evaluation order can end up in a loss of precision in the final result because of successive rounding offs.

Flexibility consists in a certain adaptability of the programs and of their components. It is desirable, for instance, that subroutines, stored more or less permanently in memory, are written in such a way as to be callable at different places in a program, or even from different programs; it's the very reason of their existence. Further, because programming errors (and not only coding errors) are considered inevitable, programs should be organized so that the correction of an error does not lead to a complete rewrite of the program.

Clarity being considered as "obvious", we cannot develop it more for the time being. This self-evidence perhaps signifies that, contrary to the notions of safety, accuracy and flexibility which receive a specific meaning, it should be understood in its usual sense, namely, the property of being easy to understand. In any case, one can already note that it is a requirement that goes beyond simple usefulness.

This is, of course, not his last word. A few years later, with five years of programming experience, he brings another desirable property to the fore: their restartability. As a sign of its importance, its detailed exposition occupies five pages in the manual *Programming for the ARMAC* (1957), which is about a hundred pages long. What is it about?

> A program is called restartable, if the information entered by the input stays unaltered in the memory during the calculations. [...] With such a program, it is possible to start the calculation again, without reading the tapes anew, only by pressing the start button, for every needed information is still intact in the machine. Restartability is a *requirement*,

> that each program *must* meet.[19]

It is because programs can modify themselves during the execution that this property is not verified right away. The modification of a program by itself is seen as essential to the power of the computers, and, from a practical point of view, the absence of index registers render its usage necessary for writing loops. It is clear that, in general, after those modifications, the program cannot be run as it is, which means that, if the program has to be restarted, one has to load it again in memory by reading the punch tape again. Because it is a lengthy and tedious operation, it would be wise to avoid it, if possible. He does not merely formulate this requirement; he translates it as a concrete, simple and clear, rule:

> The formal consequence of restartability is that each used address should be filled in either during the loading of the program, or by the program, but not by both.[20]

What are the grounds for this requirement? One may, of course, want to restart a program to run it on other data. But in that case reloading it from the tape would be tolerable, as the time needed to read a tape is usually quite low compared to the running time of a program; the reason lies elsewhere. Execution errors (due to memory errors, physical errors in a communication apparatus, ...) are a better justification.

> Stronger again is the argument of the testing of a program. Suppose that one starts a program carelessly, and that, seeing the output — or its absence! — one comes to the conclusion that something is wrong with it. (Inexperienced programmers often tend to draw the nearly always wrong conclusion that the ARMAC a defect has!). In such a case one will load the program again in the ARMAC, to stop the machine after some known intermediary result; if it is already wrong, one starts the machine again, but this time stopping it somewhat earlier, etc. trying to localize the error in this way. Having to reload the program each time again would slow testing excessively; again different is the situation, when the first error has been found and corrected "manually" in the machine: the tapes are then no longer correct, before a correction tape has been made! However one always waits, to punch a correction tape, that a significant number of errors, if not each of them, were found and fixed.[21]

It is thus clear he still thinks that (numerous) programming and coding errors are unavoidable, because those errors are the deepest justification for the requirement of restartability. Errors due to the machine being from now on automatically detected, thanks to the presence of parity checks in the ARMAC, safety is not any more a property that the programmer should worry about, so much so that he can now write:

> In contrast to the past a stage has now been reached, where the weakest link in the process is not the machine any more, but — the programmer![22]

The new features of the X1 result in new thoughts about programming. For example,

19. Dijkstra, E. W., *Programmering voor de ARMAC*, p. 16

20. Dijkstra, E. W., *Programmering voor de ARMAC*, p. 17

21. Dijkstra, E. W., *Programmering voor de ARMAC*, pp. 16–17

22. Dijkstra, E. W., *et al.*, *Programmering voor Automatische Rekenmachines*, p. 79

in comparison to the ARMAC, the subroutine call and return instructions of the X1 have been extended with an index argument m, indicating the place, in an array of successive words in memory, where the return address should be stored and retrieved: this makes it possible to have nested subroutine calls.

> It is convention to choose $m = 0$ for those subroutines that do not call in another subroutine; for subroutines that call in a sub-subroutine with at most $m = 0$, the m is chosen $= 1$, etc. All this applies to subroutines that call in other known subroutines. However, as soon as a subroutine calls in an "arbitrary" subroutine [...] this is no longer possible. In such cases it is always safe to call in the outer subroutine with $m = 0$; the latter starts off by transferring [the return address] to a location in its own working space. The restriction that the index m can "only" take [sixteen] different values can be circumvented by the same technique. (If need be a single value of m, e. g. $m = 0$, would suffice; transferring [the return address] would then be [the] rule and no exception. [...])[23]

This, and the fact that subroutine calls are fast and take little space, has the consequence that it is now possible to use subroutines in ordinary programs. It is then clear that the use of subroutines is not any more restricted to the sole use of the standard library:

> The fact that calling in a subroutine needs only one [instruction] in the main program [...] is also attractive as far as program space is concerned; as a result it is worth considering programming even rather simple operations as subroutines. [...] The fact that [instructions to save the return address] are not necessary on entering the subroutine, facilitates the making of compact, fast and flexible subroutines even further.[24]

It should be noted that this possibility of nested subroutine calls is not used in the standard communication program: it would prevent ordinary programs to use it, as the array of return addresses is not stored and restored by the interrupt mechanism.

This new facility is however not enough to make the subroutines the criterion to compose programs. Regarding that question, there is nonetheless room for improvement. Based on the previous programming experiences, a novelty is included in the tape reading functions of the communication program: the notion of "paragraph".

> The division of the memory into paragraphs has been introduced in order to meet the needs of the programmer. In practice only very simple programs are conceived as a whole and written down order by order from beginning to end. Very soon it is found more convenient to split up the computation into sections, each having a separate function which can be isolated to a greater or lesser extent from that of the other sections. It is very important to choose these sections with care: the more clearly isolated the function of these sections the clearer the arrangement of the program.
>
> Normally one allocates a separate paragraph to each section and supplies each paragraph with its own [identifiers]. These different [identifiers] make the program easier to read, furthermore they make it possible for one to start programming one paragraph while one or more of the others is incomplete.[25]

A "paragraph" is a logical group of sheets (a sheet being, as before, 32 successive words), whose physical place in memory will be determined at the time the program is loaded, by

23. Dijkstra, E. W., *Communication with an Automatic Computer*, pp. 49–50

24. Dijkstra, E. W., *Communication with an Automatic Computer*, pp. 50–51

25. Dijkstra, E. W., *Communication with an Automatic Computer*, p. 65

means of directives punched on the tape. The paragraph abstracts the memory locations in such a way that it is possible to write a program, broken into distinct "sections", without worrying about their physical location in the memory: therefore the main division criterion for programs, while still based on the memory space, is not the sheet any more, and is thus not as much dependent of the physical reality of the programs.

As one sees, flexibility and clarity, which are the ground of the notion of paragraph, are still regarded as desirable qualities of programs. Memory usage and speed are sometimes mentioned, but they don't have the first place any more. No other properties are put forward. Also, while the communication program of the X1 had to be written without any testing, since it is wired into the machine and is necessary to use it, and nevertheless be error-free, the presence of bugs is still seen as inevitable in other programs.

Even more interesting is the fact that a reflection is initiated on the structure of the programs, more or less independently of concrete programs. Flowcharts are still used to present the programs, but they are sometimes used to represent not a particular program, but a general structure common to many programs. For instance, when Dijkstra presents the "counting jump" instruction of the X1 (which sends the control to a given address, depending on the value of a given memory location, and decreases that value by one), he shows the three usual ways for its use: a loop that has to be executed $n \geq 1$ times, a loop that has to be executed $n \geq 0$ times, and a loop that has to be executed $n + \frac{1}{2}$ times (that is, a loop for which one part must be executed n times, and another part $n + 1$ times). Likewise, to explain how the communication program works, he presents a flowchart of the common structure of the individual parts of the program, each dedicated to the management of a distinct communication device, before going into the details of the differences between those individual parts.

Finally, the reflection on the structure of programs is now coupled with the beginning of a reflection on programming languages, in which the programs are written. The programming languages Dijkstra knows of are the instruction sets of the four computers designed in the Mathematical Centre. From his experience with them, he isolates a desirable property for instruction sets: their elegance. Elegance refers to an equilibrium between excessive flexibility and excessive rigidity. An instruction set is excessively flexible if it is redundant, that is, if too many different instructions have an equivalent effect. This causes confusion, as too much different solutions present themselves to the programmer faced with a given problem. Each programmer then uses only a subset of the instruction set, and becomes accustomed to his own methods, which renders the reading of programs harder than necessary. On the contrary, an instruction set is excessively rigid if there is only a single instruction at one's disposal for each possible operation. The most usual excess is obviously in the direction of flexibility.

There is, though, a small gap between the instruction set and the programmer, since he doesn't use the instructions in their binary form when he programs. This small gaps lies in its notation: as the instructions written down on paper are first punched onto a tape,

and then assembled by the communication program, it is possible to have some distance between the way instructions are written down, and their actual representation as a sequence of bits. Indeed, the notation of the instructions for the X1 was carefully designed so that the order in which the name of the instruction and its different parameters appear can logically be read from left to right. For instance, if an instruction should be executed only if one of the condition registers is set, that condition appears on the left of its name; if an instruction should set one of the condition registers, this is written down after all the other parameters.

What preliminary conclusions can be drawn from those first observations? It is manifest that Dijkstra approaches the new discipline with absolutely no preconceived ideas: the principles and the methods he brings to the fore are all derived from his practical experience on programs with a size of a few tens to a few hundreds of instructions, solving well-defined problems. His theoretical education in mathematics and physics seems to have had little influence on the way he works, except perhaps by giving him a sense of precision and clarity. In particular, he views programming as, in most cases, a trial-and-error process. The unusual notion of clarity, which seems to have a certain importance, needs further analysis.

The two aspects of his work during those first eight years find their continuation in his work during the next nine years, on problems with a higher level of complexity: the design of the notation conventions for the instruction sets and the writing of assemblers, in the design and the implementation of a programming language, ALGOL 60 (chapter II); the design and writing of the communication programs, in the design and implementation of an operating system, "THE" (chapter III).

CHAPTER II

A PROGRAMMING LANGUAGE (1959–1962)

Van Wijngaarden is seriously injured in a car accident in 1958. During his convalescence, Dijkstra acts as the manager of the Computation Department. He also takes his place in the international meetings, and participates in the preliminary discussions for the definition of ALGOL. After his recovery, van Wijngaarden's interests shift from the building of computers to the developing of programming languages. Therefore, while Loopstra and Scholten set off on the Electrologica venture, he takes an active part, with the help of Dijkstra, in the definition of ALGOL 60 (§ 1) — and then the latter writes, with J. A. Zonneveld, a compiler for that new language (§ 2), named "MC" for "Mathematisch Centrum".

§ 1. The Definition of ALGOL 60

ALGOL 60 is the result of a three year international effort to produce a universal programming language. The objectives in designing ALGOL were threefold: the language "should be as close as possible to standard mathematical notation", it "should be possible to use it for the description of computing processes", and it "should be mechanically translatable into machine programs." While those two last objectives are compatible, and even go hand in hand with each other, they are to a certain extent conflicting with the first one: on the one hand, standard mathematical notation was not designed to describe computing processes, that is, sequences of operations ordered in time, but rather to denote timeless relations between quantities; on the other hand, it is often ambiguous, which is not a problem for a human reader but renders it improper to a mechanical translation into machine programs. Hence the discussions to try to fill this gap.

From a working document established during the mid-1958 conference in Zürich attended by four Europeans and four Americans (the preliminary report defining the International Algebraic Language which will later be known as ALGOL 58), the language is slowly build up by discussions in committees during meetings and conferences, and by suggestions submitted to examination and approval by the interested parties, in the *Communications of the ACM* for the Americans and in the *ALGOL-Bulletin* for the Europeans.

Among van Wijngaarden and Dijkstra's propositions, one finds for instance the suggestion to denote the definition of a function with a '=' sign rather than with a ':=' sign (neither of those two notations will finally be adopted), or to include other primitive types in the language (besides integers, reals and Booleans) together with their usual operations: complex numbers, vectors, matrices, lists, ... More interesting are their contributions to the creation of the block concept — and to the inclusion of recursion in the language.

The block concept arguably represents the most important and the most innovative contribution of the language: it is the aspect of the language that took the longest discussions to agree on, it will be taken over by virtually any programming language following it, and it will in turn give rise to many subsequent developments in programming languages (closures and objects are the most obvious examples).

Till then, whichever be the programming language, each element (subroutine or variable) of a program is at the same level as any other element: each one is accessible from any point of the program and they all have, during the execution of the program, the same life span as the whole program. This means particularly that there is a single level of subroutines, and that subroutines do not have, except by means of conventions, their own particular variables. In other words, the code of the programs is flat: even if nested subroutine calls are used, the code of the subroutines and the variables are located in the same words in memory from the beginning of the execution of the program to its end.

On the contrary, in ALGOL 58, inspired more by mathematics than by machine languages, a program is a series of declarations of totally independent and self-sufficient processes. There is a strict separation between their inside and their outside, which means that a process can only use, access or modify the elements that it explicitly receives as argument. Like mathematical functions and unlike subroutines, processes in ALGOL 58 therefore do not have any side effects. They form a kind of calculable equivalent of mathematical functions. To emphasize the difference with the concept of subroutine, they are called "procedures".

That mathematical influence can also be felt in the fact that procedure declarations cannot contain other procedure declarations (although the preliminary report is not explicit on this point).

In June 1959, during a conference in Paris, a committee of which Dijkstra is a member (but not van Wijngaarden) observes that it would be convenient to be allowed to refer, from the inside of procedures, to certain entities (functions, which do not modify any variable, but simply return a value calculated with the help of the value of their arguments, and are therefore strictly speaking not processes), without having to supply them as arguments. The committee proposes a notation to declare in the heading of a procedure that certain functions are accessible in its body although they are not received as arguments. The proposition is soon extended to the two other kinds of entities in ALGOL: procedures and variables. Because of this feature, known as "hidden parameters", the final language

will thus have side effects.

Three months later, in September 1959, G. Ehrling observes that a program could be allowed to become a procedure in another program, since it suffices to enclose a program within a procedure heading to turn it into a procedure; consequently, he proposes that procedures could be declared within other procedures.

Besides hidden parameters and nested declarations, a third improvement over standard mathematical notation, regarding the meaning of identifiers, is desirable. Indeed, symbols are often loosely used in mathematical formulæ, where their meaning is supposed to be explicit enough for a human reader. As those formulæ are in particular not written to be executed, there is no risk of conflict between possibly identical symbols used in different formulæ. This problem does not appear in machine languages either, where identifiers are merely used as abbreviations for memory locations. Four independent propositions to solve it were submitted in October 1959, by van Wijngaarden and Dijkstra, K. Samelson, H. Bottenbruch and J. Green *et al.* The first deals with three aspects of the problem with three distinct suggestions, covering respectively the uniqueness of identifiers, redeclarations, and hierarchical declarations; the three others cover only one aspect of the problem.

In the language defined by the preliminary report, nothing prohibits an identifier to be used in a program with multiple meanings, as long it is not used to designate two entities of the same class. The same identifier therefore may have up to five different meanings at a given point of a program text, since it may be used to designate a label, a simple variable, an array variable, a function, and a procedure. As this possibility is probably due to a forgetting, and as it does not contribute towards the clarity of programs, van Wijngaarden and Dijkstra propose and argue that:

> [Identifiers] should not be used for different purposes, e. g. for a variable and a label. It has been shown that unexpected ambiguities may arise under special circumstances, and there does not seem to be any serious need for multiple use of the same name.[1]

A second aspect of the problem lies in the fact that it is common to use the same identifiers in successive formulæ with different meanings. This is not forbidden by the preliminary report, but is neither explicitly allowed, and it does not have a precise signification. They suggest to raise the possible ambiguities by including the following sentences in the final report:

> [Declarations] pertain to that part of the text which follows the declaration and which may be ended by a contradictory declaration. Their effect is not alterable by the running history of the program.[2]

A given identifier would thus be allowed to have successively different meanings, but at any point of the program text it would have only one. It is therefore an offset to the previous suggestion. Samelson's and Bottenbruch's propositions also concern this aspect

1. van Wijngaarden, A., Dijkstra, E. W., *ALGOL-Bulletin* 7.32

2. van Wijngaarden, A., Dijkstra, E. W., *ALGOL-Bulletin* 7.31

of the problem.

Samelson suggests that declarations should always be written in front of statements, and have a meaning only in the statement following them:

> A declaration is a prefix to a statement [...]. It is valid for, and part of, the statement following it: if Δ is a declaration, and Σ a statement, $\Delta.\Sigma$ is a statement and Δ is valid through Σ and Σ alone. Conflicting declarations on different levels of statement are errors.[3]

The novelty is that declarations are given a precise place in program texts, and a precise validity span which does not depend on a contradictory declaration, but the problem of conflicting declarations is not dealt with in detail. Successive conflicting declarations seem to be allowed, but on the same level of statements.

Bottenbruch's proposition, while being less precise, is nevertheless of some importance. He simply suggests to "give the declarations a dynamic meaning." It can be understood from the little example he gives to explain what this means that, to solve the problem of contradictory declarations, he suggests that identifiers could have their meaning changed while the program is running, namely when the control encounters a declaration that is contradictory with another previously encountered one.

Finally, van Wijngaarden and Dijkstra's third proposition explicitly introduces the idea of a hierarchical nomenclature. It goes as follows:

> The level [of nomenclature] declaration
>
> **new** $(I, I, \ldots)$
>
> has the effect that the named entities [with the identifiers I] have no relationship to identically named entities before in the following text, until the level declaration
>
> **old** $(I, I, \ldots)$
>
> which attributes to the entities named herein the meaning that they had before. These level declarations may be nested and form the only way to introduce a new meaning to a name. In particular, in a procedure to be compiled along with the main program, all variables that should have no relationship [to identical variables outside the procedure] should be declared **new** before they have appeared and declared **old** before the end.
>
> These declarations do not only solve the problem of having "old" and "new" variables alongside in a procedure, but are also extremely useful in an ordinary program. It should be noted that after '**new** (x)' the new x is fully independent of the old x and, therefore, type declarations, if necessary, have to be given anew. On the other hand after '**old** (x)' the type declarations of the old x are still valid.[4]

It is understood as a generalization of the hidden parameter feature, by reversing the approach: instead of writing down in the program text which identifiers should be imported from the surrounding environment, identifiers are imported by default, and one should declare which identifiers become local, and when their global meaning is to be restored. Further, the use of this feature to declare local variables in a procedure is now only a particular case: it may be used in any other context.

3. Samelson, K., *ALGOL-Bulletin* 7.22

4. van Wijngaarden, A., Dijkstra, E. W., *ALGOL-Bulletin* 7.33

The problem is not only discussed in Europe; a similar recommendation is made by the Americans Green *et al.*:

> It is desirable to enable procedures to operate on variables which are defined and used outside of the procedure. These variables can be designated in the procedure heading by the following declaration:
>
> **global** $(I, I, \ldots)$
>
> A **global** declaration specifies certain identifiers contained within the procedure to be defined as being identical to the same identifier when used outside of the procedure. The global declaration may appear only in the **procedure** heading. All identifiers in a **procedure** declaration not specified as **global** are considered as having no relation to identical identifiers outside the procedure.
>
> In addition to the **global** declaration it may be convenient to have a **local** declaration, which is the inverse of **global**. A **local** declaration specifies the identifiers within a procedure that have no relationship to identical identifiers outside the procedure:
>
> **local** $(I, I, \ldots)$
>
> If a **local** declaration is used, all identifiers in the procedure not specified in the **local** declaration are considered to be **global**.[5]

As one sees, their proposition also goes further than the hidden parameter feature, by extending it to any identifiers and by suggesting a complementary declaration, but it does not go as far as the '**new**'/'**old**' proposition: it is still limited to the sole procedures, and does not include the idea of a hierarchy.

In November 1959, a conference is held in Paris to discuss the remaining unsolved questions in the language. A committee of which Dijkstra is a member (but again not van Wijngaarden) discusses the problems concerning declarations. No agreement can be reached, but they synthetize the different possibilities for the forthcoming conference. Concerning the problem of the range of declarations, they note:

> The principal problem is considered to be [the] range within which a declaration should be valid. The extreme possibilities are the strict limiting by write-up or alternatively by time succession. A further possibility is that of permitting dynamic declarations only when those two extremes are coincident. Because of this, the [committee] is unable to agree unanimously.[6]

The extreme possibilities mentioned are Samelson's and van Wijngaarden and Dijkstra's propositions to give declarations of a lexical scope, that is, a static meaning in the program text, and Bottenbruch's proposition to give them a dynamic scope, that is, a dynamic meaning while the program is running. Concerning the idea of a hierarchical nomenclature, they remark:

> The [committee] agrees that the notions **new** and **old** [...] are very important, and will deserve a close study. However, since they are intimately connected with the questions of the character of declarations in general, on which no definite decisions can be reached at present, no further step can be taken with regard to them.[7]

5. Green, J., *et al.*, *Recommendations of the SHARE ALGOL Committee*, p. 25

6. Ellis, G. V., *et al.*, *ALGOL-Bulletin* 8.1.2.1

7. Ellis, G. V., *et al.*, *ALGOL-Bulletin* 8.1.2.2

The "questions of the character of declarations" refer to the problems of the uniqueness of identifiers, of their range of validity, and of the dynamic or static meaning of the declarations.

In December 1959, the seven European representatives to the final conference meet in Mainz; van Wijngaarden is one of them, but not Dijkstra. The discussions are almost exclusively concerned with two aspects of the language: declarations and procedures. The block concept emerged as a general, elegant and simple solution to the different aspects of the problems posed by declarations: identifiers have a unique meaning at a given level in a block, re-declarations are permitted in inner blocks, and the blocks are organized hierarchically, inner blocks importing those identifiers declared in outer blocks that they do not re-declare. Lexical scope of identifiers is chosen against dynamic scope.

The final ALGOL 60 conference takes place in Paris in January 1960, and adopts the block concept. The final ALGOL 60 report stands:

> Declarations serve to define certain properties of the identifiers or the program. A declaration for an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes. [...] All identifiers of a program must be declared. No identifier may be declared more than once in any one block head. [...] The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program.[8]

A block is defined as a sequence of declarations followed by a sequence of statements, between a '**begin**' and a '**end**'. This definition is recursive, and it is therefore not needed to mention explicitly that blocks are organized hierarchically, and that inner blocks import identifiers that they do not re-declare. However, as an addition to the original description of the block concept given in Mainz, its precise dynamic meaning is presented:

> Dynamically this implies the following: at the time of an entry into a block (trough [a] **begin** [...]) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning. At the time of an exit from a block (trough **end**, or by a **go to** statement) all identifiers which are declared for the block lose their significance again.[9]

Besides the block concept, another important contribution of ALGOL 60 is that of recursion. ALGOL 58 did not forbid its use, but only through lack of mentioning it. J. McCarthy, who introduced it in LISP a year ago, suggests in August 1959 to introduce it explicitly in the new language, but his proposition does not draw much attention. The problems discussed are indeed very different and seem much more important; recursion, hardly ever used, looks like a minor detail. Eventually, in January 1960, during the discussions to finalize the language, a proposition to include a '**recursive**' declarator is rejected. ALGOL 60 risks to be just as silent on that point.

8. NAUR, P. (ed.), *Report on the Algorithmic Language ALGOL 60*, sections 2.4.3 and 5

9. NAUR, P. (ed.), *Report on the Algorithmic Language ALGOL 60*, section 5

In February 1960, P. Naur, who has the responsibility of the final edition of the report before its publication, receives a phone call from van Wijngaarden and Dijkstra. They point this deficiency out by showing that, if a procedure identifier is allowed to appear in its own body on the left part of an assignment (thus changing the procedure, by giving it a return value, into a function), it can be given no meaning, in the language as it is defined, if it occurs in another place.

They further observe that it would be complicated to set up a rule to prevent the use of recursion. It would for instance not be enough to prohibit that the identifier of a procedure appears in its own body (in another place than in the left part of an assignment statement): this rule could be easily circumvented by the use of mutually recursive procedures. So they propose to add a short sentence to the report, to make it clear that recursion is explicitly allowed:

> Any other [other than in the left part of an assignment statement, where it sets the value of the type procedure used as function designator] occurrence of the procedure identifier within the procedure body denotes activation of the procedure.[10]

Charmed by the simplicity and the clarity of their proposition, Naur takes the risk to follow it, without submission to the other members of the committee which established the final version of the language.[11]

In May 1960, the *Report on the Algorithmic Language ALGOL 60* is published in the *Communications of the ACM*, in *Numerische Mathematik* and in the *Acta Polytechnica Scandinavica*. ALGOL 60 brings the new block concept to the fore, and with it recursion enters into imperative programming languages. It also introduces a new type of variables, the logical or Boolean type (whose domain is the two truth values, denoted by '**true**' and '**false**'), together with their usual operators: negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\supset$), and equivalence ($\equiv$). It is the first language whose syntax is defined formally, with the help of what will later be known as the Backus Naur Form. It is, finally, the first language that was not described in terms of its implementation, but instead specified by the precise semantics of each of the elements that constitute it.

§ 2. The "MC" ALGOL 60 Compiler

ALGOL 60 is far ahead of its time; at least far enough that its very authors do not know how to implement it, and are not even sure that it is possible. Not surprisingly, the two most innovative aspects of the language, namely blocks and recursion, are also the main source of their difficulties. The great majority of the first compilers is, for that matter, limited to a subset of the language, excluding particularly recursion. Dijkstra soon finds a general solution to this problem — then he designs, with Zonneveld, an innovative

10. Naur, P. (ed.), *Report on the Algorithmic Language ALGOL 60*, section 5.4.4

11. Cf. Naur, P., *The European Side of the Last Phase of the Development of ALGOL 60*, p. 30

compiler — and finally they write it.

During the summer of 1959, Dijkstra tries to find a way to implement recursion (blocks aren't invented, and do not yet pose a difficulty). What exactly is the problem he is faced with? Till then, subroutines share the computer's memory by means of conventions: each one uses a fixed and predefined part of the space, and others do not access it. But:

> If every subroutine has its own private fixed working spaces, this has two consequences. In the first place the storage allocations for all the subroutines together will, in general, occupy much more memory space than they ever need *simultaneously*, and the available memory space is therefore used rather uneconomically. Furthermore — and this is a more serious objection — it is then impossible to call in a subroutine while one or more previous activations of the same subroutine have not yet come to an end, without losing the possibility of finishing them off properly later on.[12]

The usual way to code subroutines seems thus fundamentally limited, precisely because it does not support calling a subroutine recursively: a recursive call would use the subroutine's working space previously used by an earlier call, and overwrite its contents, thereby preventing earlier calls to continue their work normally on its return.

It is however possible to avoid this problem by generalizing an already well known method used to evaluate arithmetic and algebraic expressions.[13] For instance, the evaluation of the formula '$43 + (7 - 5) \times (8/4 + 21)$' can be carried out by first transforming it into its postfix notation (also known as reverse Polish notation) '$43\ 7\ 5\ -\ 8\ 4\ /\ 21\ +\ \times\ +$', and by reading this formula from left to right, remembering the successively encountered values, performing an arithmetic operation on the last two remembered values when an arithmetic operator is encountered, forget those last two remembered values, and remember the result. This method can be mechanized by using a stack:

> One uses a stack for storing a sequence of information units that increases and decreases at one end only, i. e. when a unit of information that is no longer of interest is removed from the stack, then this is always the most recently added unit still present in the stack. For example, one can construct a stack as follows: a number of successive storage locations are set aside for the stack and also an administrative quantity, the 'stack pointer', that always points to the first free place in the stack.[14]

The evaluation of the previous formula can thus be performed in the same way, except that "remember" now means "store onto the stack and increase the stack pointer", and "forget" now means "remove from the stack and decrease the stack pointer." If we denote the successive stack locations by s_0, s_1, s_2, $\ldots$, the possible values for the stack pointer being the indices 0, 1, 2, $\ldots$, and "store" by $\leftarrow$, then this evaluation is performed by the following operations: $s_0 \leftarrow 43$, $s_1 \leftarrow 7$, $s_2 \leftarrow 5$, $s_1 \leftarrow s_1 - s_2$, $s_2 \leftarrow 8$, $s_3 \leftarrow 4$, $s_2 \leftarrow s_2/s_3$, $s_3 \leftarrow 21$, $s_2 \leftarrow s_2 + s_3$, $s_1 \leftarrow s_1 \times s_2$, $s_0 \leftarrow s_0 + s_1$. The result of the

12. Dijkstra, E. W., *Recursive Programming*, p. 312

13. Cf. for instance Samelson, K., Bauer, F. L., *Sequential Formula Translation*

14. Dijkstra, E. W., *Recursive Programming*, p. 312

evaluation is then in s_0. The same principle can be used to evaluate an algebraic expression, like ‘$a+(b-5)\times(c/4+d)$’ transformed into ‘$a\ b\ 5 - c\ 4\ /\ d + \times +$’, except of course that, when a symbol is encountered, one should perform the additional operation of looking for its value somewhere in the computer’s memory.

> The above is [...] so elegant that we could not refrain from trying to extend this technique by consistent application of its principles. Let us consider for a moment the operation “$s_2 \leftarrow c$”. This operation can be executed without further claim for memory space, as we assume that the numerical value of c can already be found in the memory. If, instead of c, a compound term had occurred in the expression, e. g. $(p/(q-r+s\times t))$, then we would have used s_2 up to s_5 for the calculation of this sub-expression, but the net result of this piece of program would still be $s_2 \leftarrow p/(q-r+s\times t)$ [...]. In other words, it is immaterial to the “surroundings” in which the value of c is used, whether the value c can be found ready-made in the memory, or whether it is necessary to make temporary use of a number of the next stack locations for its evaluation. When a function occurs instead of c and this function is to be evaluated by means of a subroutine, the above provides a strong argument for arranging the subroutine in such a way that it operates in the first free places of the stack, in just the same way as a compound term written out in full.[15]

The general principle to code recursive subroutines is now set up: instead of dividing the memory by means of conventions, they all use the same space, organized as a stack: each call of a recursive subroutine uses a certain amount of memory on this stack as its working space, and it is freed when the call completes. Since at any moment only the working space needed by the currently activated subroutines is occupied, this organization has moreover the advantage that memory is used economically; it can therefore be used as a general way to share the memory space between subroutines, recursive or not.

This general principle can almost be used as it is for subroutines written in a machine language: when entering a subroutine, the next free places on the stack are used as its working space, and when leaving it, if it has a return value, it is stored in the first previously free place. However, a subroutine is generally not limited to the evaluation of arithmetic or algebraic expressions, and it may make use of certain local variables. Further, some information may also be presented to it when it is called in, that is, it may receive arguments. Finally, some information have to be saved on entering and restored on leaving a subroutine, for instance, the contents of certain registers; likewise, the return address is stored on entering a subroutine and used to leave it (these information are known as the “link”). Thus a certain amount of memory has to be immediately reserved (and partly filled) on entering the subroutine. This can simply be done by increasing the stack pointer by that given amount.

As those information will need to be accessed from within the subroutine, there is a need for a second stack pointer, pointing not to the first free place on the stack, but to the beginning of the portion of the stack that is used by the currently active subroutine, somewhere deeper in the stack. Such a portion of the stack is called a frame, and the second stack pointer is called a frame pointer: all the local quantities are accessed, from

15. Dijkstra, E. W., *Recursive Programming*, p. 313

within the procedure, relatively to it. As its value is derived from the value of the stack pointer at the moment of a call, it will vary from call to call; therefore it needs to be saved and restored along with the contents of the registers and the return address. When the subroutine is completed, the stack pointer takes the value of the frame pointer, thereby freeing the memory space reserved on entering the subroutine, and the frame pointer takes the value of the frame pointer value saved on entering the subroutine.

For programs written in a programming language like ALGOL 60, two additional difficulties arise, caused by the block structure. The first one is that declaring local variables is not limited to procedures, but can generally be done in any block. This problem can be easily settled by extending the previous principle:

> A block that is not a procedure can also be treated as a subroutine, be it that this subroutine is only called in at one point.[16]

A block is thus a procedure that can't be called in from an arbitrary point in the program (in particular, it may not be called in recursively), and that does not receive arguments. Conversely, a procedure can be treated as a block with some additional properties. The two constructions are closely related and may ultimately be considered equivalent: both entering a block and entering a procedure results in allocating a new frame on the stack, both leaving a block and leaving a procedure results in freeing a frame on the stack.

The second difficulty related to ALGOL 60's block structure is that a block or a procedure may refer to non-local variables, that is, variables that are declared outside that block or that procedure declaration, in a lexicographically enclosing block or procedure. With the stack organization, this means that a block or procedure may refer to variables stored in a previously allocated frame corresponding to that enclosing block or procedure. Hence the need for a third stack pointer, or a second frame pointer, pointing again deeper in the stack, to the most recent activation of the first block that lexicographically encloses the current block or procedure:

> When a subroutine is called in, the link contains *two* [frame] pointer values for this purpose. Firstly, the youngest [frame] pointer value corresponding to the block in which the *call* occurs [...], secondly, the value of the [frame] pointer corresponding to the most recent, not yet completed, activation of the first block that lexicographically encloses the *block* of the subroutine called in. [...] The first [frame] pointer value plays a vital role in the return at the end of the subroutine, the second is indispensable in localizing the global variables in the stack. As the second [frame] pointer, by definition, points to a link in the stack, which in its turn contains a second [frame] pointer value corresponding to the next enclosing block, the [calculating] unit can trace this "chain" and, in doing so, will find all [frame] pointer values that may be necessary for localizing any global variable in which it may be interested.[17]

As one sees, this second frame pointer also needs to be saved on entering and restored on leaving a block or procedure. Consequently, a link must contain two frame pointers. One points to the beginning of the previous frame on the stack, that is, to the frame of

16. Dijkstra, E. W., *Recursive Programming*, p. 317

17. Dijkstra, E. W., *Recursive Programming*, pp. 317–318

the dynamic predecessor of the current subroutine; the other one points to the beginning of the frame of the most recent activation of the lexicographically enclosing block, that is, to the frame of the static (or lexical) predecessor of the current subroutine. (If the dynamic scope of identifiers had been chosen against lexical scope for ALGOL 60, a different organization would obviously be needed.) One must naturally know when to stop while traversing the chain of the static predecessors of the current subroutine to find the address of a non-local variable, hence the need for a last information in the link:

> One can assign a so-called *block number* to each block, indicating the number of blocks which enclose it lexicographically: the main program therefore has a block number = 0. If the program refers to a global variable it is obviously necessary to specify the block in which the global variable was declared; the block number serves this purpose, and, under control on this block number, the [calculating] unit can find the [frame] pointer value it now needs.[18]

A link may finally contain the following information: the contents of certain registers that need to be restored when leaving the subroutine, a return address, two frame pointers, and a block number.

Traversing the chain of the static predecessors can be rather time-consuming, and a slight optimization is possible:

> The introduction of the block numbers makes it possible that the [calculating] unit has immediate access to all the [frame] pointer values it may need: they can be stored in order of increasing block number in a so-called “display”.[19]

In other words, to facilitate the access to non-local variables, the frame pointers of the static predecessors can be stored together in an array of successive memory words. It is then straightforward, with the block number of the non-local variable, to find the frame pointer of the frame in which it is stored. This does not however frees one from saving and restoring the frame pointer in the link: they are still needed there, to update the display when leaving a block or procedure.

It is quite clear that the general principle thus specified is an elegant and simple solution to the different difficulties that arise because of the presence of blocks and recursion. However:

> The fact that the proposed methods tend to be rather time consuming on an average present day computer, may give a hint in which future design might go.[20]

This is indeed what will happen: the X1, for which Dijkstra and Zonneveld have to write a compiler, does not include any facilities to work with stacks; twenty years later, virtually any computer will include registers dedicated to the stack pointer, the frame pointer and the return address, which are modified by the subroutine call and return instructions.

---

18. DIJKSTRA, E. W., *Recursive Programming*, p. 318
19. DIJKSTRA, E. W., *Recursive Programming*, p. 318
20. DIJKSTRA, E. W., *Recursive Programming*, p. 312

The X1, already briefly described (p. 10), has two accumulator registers, an index register and a few one-bit condition registers. Its fifty or so instructions are for the most part strictly arithmetic and logic. It does not even have a division instruction, and its subroutine call and return instructions are quite rudimentary (see p. 19): they merely store and retrieve the return address in an array of successive words in memory before transferring the control. Further, the X1 can have a memory of at most 32768 27-bit words, but the one installed in the Mathematical Centre only has a memory of 4096 words. Translating ALGOL 60 programs on such a small and rudimentary computer, without limiting the language to one of its subsets, seems at first a near impossible task. As van Wijngaarden and Dijkstra were involved in the creation of ALGOL 60, limiting it to one of its subsets is nevertheless not considered as an acceptable option: the full challenge has to be taken as it stands.

While trying to find a way to get around those limitations, the idea soon comes out that the X1's instruction set is perhaps not very suitable to express ALGOL 60 programs:

> Before one can start making a translator which is fed with an ALGOL program and has to produce the so-called "object program", one has to decide what the structure of the object program will be, because only then the task of the translator becomes well defined. What I call the "object program", has also been described as "an equivalent program in machine language", but I prefer not to use the last description, not being convinced that machine language will be the most appropriate [target] language.[21]

It would not be impossible to use it as a target language, but the gap between the complexity of ALGOL 60's constructions and the X1's very primitive instruction set is so huge that even a simple ALGOL 60 program will be translated into a much longer program in machine language, so that the very scarce memory space will not be used efficiently. Moreover, the translation to be done seems so complicated that writing a complete translator in less than four thousand machine instructions does not look reasonably feasible. This suggests to break down the translation process, and to use some intermediate language as a first target:

> The object program [is] an equivalent description of the process, more adapted to the requirement of the machine which has to do the actual computation, than the source description in ALGOL 60.[22]

The obvious benefit of this idea is that it should break the complexity of the translation in two. However, as it adds the burden of defining that intermediate language, the real gain in simplification is probably somewhat lower. Moreover, it does not by itself address the problem of the insufficient memory space: on the one hand, if the two parts of the translator have to work one after the other, both need to be loaded into memory simultaneously, and on the other hand, the final object program will still be much longer than the original ALGOL 60 program. To sort the memory space problem out, the intermediary object program could be punched on a tape by the first part of the translator, and be

21. DIJKSTRA, E. W., *Making a Translator for ALGOL 60*, p. 1

22. DIJKSTRA, E. W., *Making a Translator for ALGOL 60*, p. 1

loaded and translated into actual machine code by its second part. The actual machine code could again be punched on a tape by the second part of the translator, and would then finally be loaded to be executed. The total available memory space would then be virtually three times larger (or even more, considering that the source and object codes punched on and read from the tapes need not to be stored into memory), but the whole translation process would be terribly slowed down. Further, this still does not give an answer to the fact that the final object code, written in the actual machine language, is much longer than the original ALGOL 60 program.

There is however no other way to artificially augment the memory space than to punch the intermediary object program on a tape, and to load it afterward. On the contrary, it is not necessary that the object program produced by the first part of the translator be translated into actual machine code by the second part of the translator: it could instead be directly executed. Then only the first part is really a translator, the second part being an interpreter.

> But if the object program is not necessarily written in machine language and if, furthermore, certain [...] tasks [...] may be postponed until execution time, one might well raise the question, whether [this does] not reduce the task of the translator to next to nothing.[23]

Indeed, the whole translation process could be done during the actual execution, or interpretation, of the program. Interpreting ALGOL 60 programs directly could solve the problem of the distance between the complexity of ALGOL 60's constructions and the X1's instruction set in a simple way, but the memory space problem remains entire: writing an ALGOL 60 interpreter in less than a thousand machine instructions is no less complicated than writing a complete compiler in less than four thousand instructions.

The problem is thus to define an appropriate intermediary language that will make some benefit out of this intermediate step. To define that language, one should first determine where the translation task should finish, and at the same time where the interpretation task should begin. The whole translation process could naturally be broken down at some arbitrary point: for example, the translator could present the interpreter with a simplified ALGOL 60 program, where the different symbols have been arranged to be read in faster. But this does not offer a significant gain in memory space, and the intermediate language could probably be somewhat more elaborate. The criteria to determine what has to be delegated respectively to the translation stage and to the interpretation stage is simple:

> We have, for instance, the so-called "priority rules." In the statement
>
> $$x := a + b \times c;$$
>
> the execution of the multiplication must precede that of the addition. Another way of specifying this order of execution is
>
> $$x := (a + (b \times c));$$
>
> and we may regard the priority rules as a convenient mechanism for reducing the number

23. Dijkstra, E. W., *Making a Translator for ALGOL 60*, p. 2

> of brackets needed. The translator must be evidently aware of the priority rules and follow all their consequences. But for every ALGOL program this analysis needs only to be done once and is therefore regarded as one of the tasks of the translator.[24]

Anything that needs only be done only once in the translation process of a given ALGOL program, that is, any part of the translation process that will give the same results regardless of the actual computation taking place, can be done during the translation stage. Anything else will be done during the interpretation, or execution, stage. For example, the evaluation order of expressions, defined by the priority rules or by explicit brackets, can be determined regardless of the actual values of the variables. Determining to which **'if'** a given **'then'** or **'else'** corresponds can also be done once and for all. Likewise, assigning a precise meaning to identifiers can be done by the translator:

> If an identifier occurs somewhere in a statement, this identifier has a meaning, but only thanks to the fact that the same identifier has been declared to have this meaning. If one wants to find the relevant declaration, one has to scan the declarations at the beginning of the block in which the statement occurs. Either we find a declaration concerned with the identifier in question, or not. In the first case we have found the declaration we wanted, in the second case we scan the declarations at the beginning of the next lexicographically enclosing block, etc. [...]
>
> It is clear from the above that finding the corresponding declaration may be a rather time consuming process, involving a lot of scanning. However this correspondence is unique: the translator could do a useful job by establishing this correspondence in a more direct way.[25]

On the contrary, the actual evaluation of the expressions, the actual choice of which part of an alternative **'if'** statement should be executed, and the determination of the actual location in memory of a given variable, can only be done during the execution stage.

The general organization principle is thus settled: only one real translation takes place, producing an equivalent description of the process described by the ALGOL 60 program, which is then interpreted. The intermediary object program is punched on a tape, and loaded by the interpreter: this has the advantage of doubling the available memory space artificially, and to use it efficiently, as that intermediary object program will have a length of the same order as that of the original ALGOL 60 program. The fact that the translation process is slowed down by this organization is compensated by the benefit that, once a given ALGOL 60 program is translated, the resulting intermediary object program can be loaded multiple times with different input data by the interpreter, which can execute it very efficiently, as it contains as few ambiguities as possible.

The next task is then to concretely define that intermediary language to be used by the object program:

> The object program is build up from a (limited) number of well chosen operations, each explicitly supplied with the appropriate number of parameters (may be equal to zero).[26]

24. DIJKSTRA, E. W., *Making a Translator for ALGOL 60*, p. 2

25. DIJKSTRA, E. W., *Making a Translator for ALGOL 60*, p. 5

26. DIJKSTRA, E. W., *Making a Translator for ALGOL 60*, p. 1

The "well chosen operations" are the elementary operations that are carried out by ALGOL 60 programs (for instance, the assignation of a value to a variable, or the call and return of a procedure): the intermediary language thus embodies ALGOL 60's primitive operations. One may note that isolating and defining those operations is like writing the functional description of a computer:

> [When choosing] a structure for the object program, [one] chooses a machine and its order code. This we can always do, because if our specific machine does not have the required features built in, we can use it to simulate our chosen machine.[27]

The "order code" of the object program defines an abstract machine tailored for ALGOL 60, and the interpreter, written in the X1's machine language, simulates that abstract machine: this is what will later be known as a "bytcode" and a "virtual machine". The possibility to freely choose and adapt the target order code depending on the source language obviously greatly eases the translation task:

> The making of an ALGOL translator is a relatively simple job if the translator may formulate the object program in operations cut out for the problem.[28]

Indeed, if the order code is actually well chosen, and correctly abstracts the elementary operations of the source language, the correspondence between a construction in a source program and its order in the object program becomes straightforward; on the contrary, if it is not well chosen, that correspondence would be as intricate as with a real machine language.

The abstract machine is of course based on the previously found principle, namely the stack. The translation of arithmetic expressions, for example, is then easy:

> We presume that our object machine performs its arithmetic in what is called a stack. [...] It allows us to write down the computation of
>
> $$a + (b - c) \times d + e$$
>
> in the following form:
>
> | | |
> |---|---|
> | *TAKE a* | $s_0 \leftarrow a$ |
> | *TAKE b* | $s_1 \leftarrow b$ |
> | *TAKE c* | $s_2 \leftarrow c$ |
> | *SUBTRACT* | $s_1 \leftarrow s_1 - s_2$ |
> | *TAKE d* | $s_2 \leftarrow d$ |
> | *MULTIPLY* | $s_1 \leftarrow s_1 \times s_2$ |
> | *ADD* | $s_0 \leftarrow s_0 + s_1$ |
> | *TAKE e* | $s_1 \leftarrow e$ |
> | *ADD* | $s_0 \leftarrow s_0 + s_1$ |
>
> In this description we use two kinds of orders: the order *TAKE* (with the address of a variable) that fills a new [element s_{sp} on the stack with the value of that variable], and the arithmetic operations [*SUBTRACT*, *MULTIPLY* and *ADD*] (without address), that always operate on the two youngest [elements on the stack, s_{sp} and s_{sp-1}], leave the result in the oldest of the two and leave the youngest one free. All these operations work under control of [a so-called "stack pointer" *sp*], which points to the next free [element *s*

27. DIJKSTRA, E. W., *Making a Translator for ALGOL 60*, p. 5

28. DIJKSTRA, E. W., *An ALGOL 60 Translator for the X1*, p. 18

> on the stack]. The operation *TAKE* implies an increase of the stack pointer, the other operations imply a corresponding decrease of the pointer.[29]

If the evaluation of an expression is followed by an assignation of the resulting value to a variable, as in '$x := a + (b - c) \times d + e$', then the above orders are simply enclosed between a '*TAKE ADDRESS x*' and a '*STORE*' order, respectively loading the address of the variable x in the first free element of the stack, and storing the value present in the last element of the stack at the address present in the second last element of the stack. Similar orders are defined for more complicated operations, as those involving arrays for instance. The orders presented in the example above are fairly simple, but the instruction set of the interpreter has of course instructions for more complex operations: it defines about a hundred of such elementary operations. They can roughly be separated in three classes: loading and storing values and addresses, performing arithmetical and logical operations on integer, floating point or Boolean values, and calling and returning from a procedure or function (or, in general, entering and leaving a block). They all work in the same way, making use of the last element, or elements, in the stack. The instructions to enter and leave a block in particular perform very elaborate operations: they handle the arguments if the block is a procedure or a function, they update the pointers and the display, and they allocate the space for the local variables.

As each of those orders performs a precisely defined operation, they can be implemented as subroutines written in the X1's machine language. Each order is then represented by a subroutine call in the intermediary object program. The interpreter is thus simply a set of subroutines, a "complex of subroutines", designed to work together and sharing a common working area organized as a stack. They further share some administrative variables, the X1 having no dedicated registers to store the stack and frame pointers for example.

To represent those orders on the tapes containing the intermediary object programs, one could directly punch the subroutine call instructions, but this has the disadvantage that the addresses of those subroutines should be invariable; otherwise, if the interpreter is modified, any object program previously produced has to be translated anew. It is therefore better to denote the orders by a code number, which the interpreter understands:

> [All the orders] are numbered, and [for each order] the translator only punches the number. The punched tape with the object program has to be read in by a special simple read-program, which is provided beforehand with the data which it has to substitute in place of the numbers.[30]

This has the advantage of a greater flexibility in the coding of the interpreter, and offers the possibility to write different interpreters which could work with the same object programs. For example, it could make sense to write different interpreters working to various degrees of precision on reals, or interpreters offering the possibility to trace the execution of the running program.

29. DIJKSTRA, E. W., *Making a Translator for ALGOL 60*, pp. 5–6

30. DIJKSTRA, E. W., *An ALGOL 60 Translator for the X1*, p. 18

It is again quite clear that this organization is an elegant solution to the different difficulties that arise because of the scarce memory space, the complexity of ALGOL 60 constructions, and the comparatively very primitive X1's instruction set. But it is a general solution as well:

> Our solution is not only valid for the X1: it can be carried through with any good computer.[31]

Any computer, as long as it can perform simple arithmetic and logic operations, call and return from subroutines, and is equipped with a random access memory, is suitable to implement such an abstract machine.

Implementing the interpreter is clearly a relatively easy task. On the contrary, and although the correspondence between ALGOL 60's constructions and the orders of the abstract machine is straightforward, at least for simple programs, when one has to translate them by hand, the task of mechanizing that translation for the complete language is not straightforward at all: ALGOL 60 is very flexible and allows one to write quite intricate programs.

Once again the main constraint in writing the translator is the scarce available memory space: four thousand words are not enough to store, side by side, the translator program, its working space, the source ALGOL 60 program and the corresponding object program. To have the maximum space available for the translator, and to allow the translation of the largest possible ALGOL 60 programs, one would best not store the source program and the object program at all:

> We aim at what I should like to call "immediate translation", i. e. a translation process that reads the ALGOL program from [beginning] to end, simultaneously producing — say, punching out — the corresponding object program. In other words, we do not assume the presence of a memory large enough to store the complete ALGOL 60 program nor the complete object program. In the first case we should be able to do all kinds of scanning of the ALGOL text, in the second case we should have the possibility of making corrections in a piece of object program produced a certain time ago. The [immediate] translation process [...] is much less demanding as regards working space: in fact it only stores information as long as it may be needed during translation.[32]

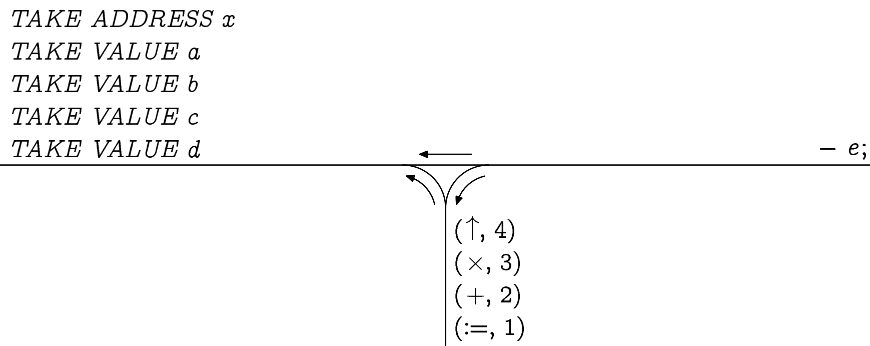
Because the meaning of the different constituents of an ALGOL 60 program often depend on the context in which they appear, storing nothing about the source program while translating it is probably not feasible. But the amount of information stored should, if possible, be kept to a bare minimum.

Not surprisingly, the stack principle, which was identified as a way to run possibly recursive subroutines, can also be used to translate a recursively defined language. For instance, the information about the local variables or the locally declared procedure in a given block may be forgotten when the translation process reaches the end of that block.

31. DIJKSTRA, E. W., *An ALGOL 60 Translator for the X1*, p. 1

32. DIJKSTRA, E. W., *Making a Translator for ALGOL 60*, pp. 6–7

Likewise, when translating an arithmetic expression, the operators need only be stored until an operator of lower priority is encountered. Using a stack to store those information will give the desired result: the necessary information will be stored only while they are needed. To take, as a simple example, the translation of an arithmetic expression:

> The symbols of the ALGOL text come in order, from left to right, [and] the successive orders of the object program are produced. The rule is that incoming identifiers are sent to the output in the form of a *TAKE* order (*TAKE ADDRESS* if the identifier occurs to the left of the ':=' symbol, otherwise *TAKE VALUE*). Incoming operators receive their priority numbers and are then sent to the translator stack, but before the latter happens, operators in the translator stack are transported from it to the output as long as their priority number is greater than or equal to the priority number of the new operator. For instance, [supposing that the symbol ';' has priority 0, ':=' has priority 1, '$+$' and '$-$' have priority 2, '$\times$' and '/' have priority 3, and '$\uparrow$' has priority 4], at a certain stage of translation
>
> $$x := a + b \times c \uparrow d - e;$$
>
> gives the following picture:
>
> 
>
>
> [The orders already produced are on the left, the symbols remaining to read are on the right, the contents of the stack are under the bar.] Identifiers are transported to the object program and operators, with their priority number attached to them, are dumped in the stack. We now consider the [next symbol to be read, the] minus sign with priority number $= 2$. Before this is entered in the stack, $\uparrow$, $\times$ and $+$ are removed from the stack, in this order, giving rise to the orders *TO THE POWER*, *MULTIPLY* and *ADD*. Then follows the order *TAKE VALUE e* and when the ';' with priority number 0 has been read, the two final orders *SUBTRACT* and *STORE* appear [in the object program]. The semicolon, being only a separator, need not be stored in the translator stack.[33]

This is what will later be known as the "shunting yard" algorithm. One might ask if such a simple algorithm is not too simple to be generalized to the whole ALGOL 60 language, which allows very complex programs to be written. But, besides being recursively defined, ALGOL 60 has separators (the symbols ';', ',', ':', ':=', ...) as well as brackets (the symbols '**begin**' and '**end**', '(' and ')', ...), which have a low priority over other symbols. Because of this, however complicated the constructions of a given source program are, it is always possible to determine precisely where a given construction ends: it may be ended either by a separator, or by a closing bracket corresponding to a previous opening bracket. If one uses a stack to store the symbols that have not yet give rise to orders, and if the

33. Cf. DIJKSTRA, E. W., *Making a Translator for ALGOL 60*, pp. 7–8

source program is syntactically correct, a separator or a closing bracket will soon or later remove some symbols previously saved on the stack. When leaving a block, the stack will have exactly the same contents as when that block was entered; only have a few orders been produced in the meantime. And the closing '**end**' symbol of an ALGOL 60 source program will finally remove the last element remaining on the stack: the opening '**begin**' symbol. When translating a "real" ALGOL 60 program, the stack will naturally not only contain operators and other primitive symbols: the declared identifiers have to be remembered until they fall out of scope, since the statements of the program may, and usually do, refer to them. They can also be stored in the stack:

> The name list is organized as a stack. At the beginning of a block its local names are added to the name list, and as soon as the translation of the block is complete these names are again struck off the name list (by suitable lowering of a pointer). Thanks to the complete bracket of the ALGOL language it is moreover not necessary to introduce more than one stack. Algebraic expressions, bracketed conditional expressions and/or statements, bracketed for-statements and procedure declarations can all be translated with the above universal stack.[34]

The "name list" always contains the currently defined identifiers, together with a reference to their current meaning. As one sees, instead of viewing the translator stack as a generalization of the stack used in the translation of algebraic expressions, which only contains primitive symbols and to which identifiers are added, one can also understand it as a generalization of the name stack, which only contains identifiers and to which primitive symbols are added.

However, some aspects of the ALGOL 60 language do not fit perfectly in this model. For instance, ALGOL 60 allows some identifiers, namely labels (which may be referred by '**go to**' statements) and procedure identifiers, to be used before they are declared. When encountering such an identifier, the translator would then be faced with something it does not yet know of. In this case it would be handy to have the whole ALGOL 60 source program in memory to look further ahead for the corresponding declaration. Paying such a high price, in terms of memory space, for those few cases, is however not strictly necessary:

> It turned out that [the translation] pass had to be preceded by a rapid so-called "pre-scan" in which the identifiers of procedures and labels are collected.[35]

The pre-scan simply builds a list of those identifiers, which is usually quite short, and takes up very little space in comparison with the complete source program. A similar problem occurs in the translation of conditional '**if**' statements. If a statement '**if** c **then** s_1 **else** s_2;' is being translated, when the translation of the conditional expression c is completed, one has to insert a conditional jump order to the first order that will later translate the s_2 statement.

> Here we meet the problem of the so-called "future reference." The only thing we can

34. DIJKSTRA, E. W., *An ALGOL 60 Translator for the X1*, p. 19

35. DIJKSTRA, E. W., *Operating Experience with ALGOL 60*, p. 125

> do is to leave the address part of this conditional jump order undefined for the time being. But the translator makes a note of the address, where this undefined jump order in the object program has been produced. This note will be used, when translation has reached [the corresponding '**else**' symbol]. Then a control combination, containing the address of the undefined jump order, can be inserted on the output tape, and for the program that reads in the object program it is an easy matter to fill in the address [bits] of the conditional jump order.
>
> But this note, specifying the address of the incomplete conditional jump order, originates when the symbol '**then**' is encountered and must be kept until the translator has reached the corresponding '**else**'. In this range, however, another conditional statement may occur [...] and this is the reason why such a note is stored in the translator stack.[36]

The stack model is thus general enough to be adaptable to the few irregularities of ALGOL 60 which prevent one from building a true single-pass translator.

The only thing left is to concretely organize the translation process. Each ALGOL 60 primitive symbol has only a limited and well-defined number of uses, and in each context it has only one. For example, the symbol '(' can be used as an arithmetic opening bracket, or as an opening bracket announcing procedure or function parameters, but it is a parameter bracket only if it follows immediately an identifier, and otherwise it is an arithmetic bracket. It is therefore possible, if the current context is precisely recorded in a series of state variables, to give a unique meaning to each ALGOL 60 primitive symbol encountered when reading the source program from left to right; further, each of the ALGOL 60 primitive symbols can be given a precise priority. The identifiers can also be given a precise meaning in each context, since they were either previously declared, or will be declared later but were recorded during the pre-scan pass. Finally, each identifier is enclosed between two ALGOL 60 primitive symbols. Consequently, the translation process can be organized as a loop, which calls a lexical subroutine to read the next symbol, and then calls a syntactic subroutine according to which symbol has been read. The lexical subroutine builds up the symbols by reading the characters one at a time from the source program, stores it in a global variable, and records its meaning in a series of global Boolean variables. A distinct syntactic subroutine is defined for each possible symbol, which has the responsibility to process it or to store it onto the stack, depending on its relative priority. This organization is what will later be known as an operator precedence parser. Technically speaking it is not composed of mutually recursive subroutines, because the X1 is not well-suited for working efficiently with recursive subroutines, but logically speaking it is: the set of syntactic subroutines share a common working area organized as a stack.

The X1's ALGOL 60 translator and interpreter runs its first ALGOL 60 programs in June 1960, and is completed in August 1960, only three months after the publication of the *Report*. Both the translator and the interpreter are about 2500 instructions long, and both are written in the X1's machine language.[37] The system translates and runs the

36. Dijkstra, E. W., *Making a Translator for ALGOL 60*, pp. 10–11

37. The text of a later version of the compiler has been published in Kruseman Aretz, F. E. J., *The Dijkstra-Zonneveld ALGOL 60 compiler for the Electrologica X1*, pp. 148–308

full ALGOL 60 language, with only very few restrictions (the main restriction is that the '**own**' declarator cannot be used in a recursive procedure or for dynamic arrays). The second functional ALGOL 60 compiler, written by a group of four people lead by Naur, is finished a year later. Among its limitations, it does not support recursion at all.

It should be noted that this fast accomplishment has been made possible not because of an extensive knowledge in language translation, or because of a fast and flexible computer, but, on the contrary, because of the strong limitations in those fields:

> We did not have the slightest experience in language translation. We thought that this was a great drawback; it turned out to be one of our greatest advantages. [...] There were no obsolete traditions to get rid of.[38]

Indeed, most of the other groups who tried to write an ALGOL 60 compiler have had a previous experience with some lower-level languages, like Autocodes or FORTRAN. But those languages were conceived as generalizations of machine languages; on the contrary, ALGOL 60 was conceived as a particularization of the mathematical language. The problem had therefore to be tackled differently: instead of trying desperately to translate ALGOL 60 programs into machine languages, the best approach was to concentrate on how to actually execute ALGOL 60 programs.

As soon as the compiler is functional, the Mathematical Centre decides to organize a course, entitled *Programming in ALGOL 60*. It is given in four consecutive days and comprises lectures, demonstrations and exercises. Dijkstra writes the syllabus; it consists in twenty-seven short chapters, progressively introducing all the ALGOL 60 constructions: assignment statements, expressions, conditional statements, **for** statements, arrays, procedures and blocks. That syllabus will be translated and published in 1962, together with the *Report*, under the title *A Primer of ALGOL 60 Programming*; it will often be considered, because of its clarity, as the reference introduction to ALGOL 60.

§ 3. Thoughts on Programming

The design and construction of an ALGOL 60 compiler is obviously a task which is an order of magnitude more difficult than the previous problems Dijkstra had to tackle. Therefore it has an influence on the way he views the programming activity. His thoughts focus on two related subjects: the nature of the programming activity itself — and the nature of programming languages.

Very soon after the completion of the compiler, the lack of a functionality is felt, namely, that of a syntactical checking of the source program:

> Omission of syntactical checking in translation has proved to have been a grave error. Every user finds that his first program contains a number of silly, clerical errors. This number of errors per program decreases very fast as the programmer gets more experience, and it is therefore my impression that it is hardly worth the trouble to let

38. Dijkstra, E. W., *Operating Experience with ALGOL 60*, p. 125

> the translator look for the next error after the first one has been found. The omission of syntactical checking is the more regretful as it could have been incorporated at so little expense.
>
> Furthermore, we find that the program for a particular problem is often processed in a couple of successive versions. Roughly: the first version is just plainly wrong, because it contains some logical errors, neglect of some exceptional cases, etc. The second version works, but the programmer is not satisfied with its performance. In the third version the programmer, who in the meantime understands his problem better, improves his strategy, and in the fourth version he improves on the programming.[39]

Not surprisingly, even with a high level language like ALGOL 60, the programming errors are thus not limited to syntactical errors: logical errors are also very common, and they lead to write programs by trial and error. One may think that the writing of programs by progressive approximation is more a characteristic of the final users of the compiler, that is, of the scientists who use it to perform their calculations, and who are only occasional programmers, than a characteristic of the professional programmers. However this would not be true, since that way of proceeding was also used while building the compiler itself:

> [The X1's ALGOL compiler] is naturally not the result of our first attempt. While the problem was yet new to use we began a few times by treating relatively simple tasks, but every solution we then found turned out later to be inadequate in more complicated cases. When the few times were past us we attacked the whole problem from the other side and subsequently subjected our new approach to, and tested it against, the most difficult situations imaginable. The basic form of this approach has not changed since.[40]

In contrast with the well-defined problems that Dijkstra had to solve before, where both the results and the way to reach them were manifest, the writing of a compiler is undeniably a task that can lead to many different solutions. Its result, namely executing, according to the semantics of ALGOL 60's constructions, any input program that conforms to ALGOL 60's definition rules, is well-defined, but the way to achieve this is at first unapparent. Trying to start with simple cases and adding gradually more complex cases reveals, by experience, to be the wrong way to go. But starting with simple or complicated cases does not change the approach radically: in both situations, programming is indeed driven by tests cases. In this aspect, the task of the programmer closely resembles to the task of the engineers who build computers:

> Concern about the [computers] reliability is as old as the computers themselves. The acceptance test is a well-known phenomenon. But what is the value of such an acceptance test? It is certainly no guarantee that the machine is correct, that the machine acts according to its specifications. It only says that in these specific test programs the machine has worked correctly. [...] The best thing a successful acceptance test can do is to strengthen our belief in the machine's correctness, to increase the plausibility that it will perform any program in accordance with the specifications.[41]

From the successful outcome of an acceptance test one cannot conclude with certainty that a computer respects its specification. The same remark applies to programs: from

39. DIJKSTRA, E. W., *Operating Experience with ALGOL 60*, p. 127

40. DIJKSTRA, E. W., *An ALGOL 60 Translator for the X1*, p. 1

41. DIJKSTRA, E. W., *Some Meditations on Advanced Programming*, p. 7

their correct behavior on a set of test data, even it is very carefully chosen, one cannot conclude that they will, in general, behave correctly. This remark is especially true for programs like a compiler, which have to execute programs written in some high level language exactly like a computer has to execute programs written in its own language. Programs who only solve a small and well-defined problem, for instance, calculating and printing a table of coefficients, are not as general; ensuring that they conform to their specification is of course easier, but by no means straightforward:

> Creating confidence in the correctness of a program is already difficult in the case of a specific program that must produce a finite set of results. [...] The duty of verification becomes much more difficult once the programmer sets himself the task of constructing algorithms with the pretence of general applicability.[42]

Reliability or correctness seems thus a new desirable property of programs to which programmers should pay attention; the means to attain it are however not quite evident. As already noted, one may increase the confidence in the correctness of a program by suitable test cases, but this is never enough if the program is general, that is, if the possible data on which it may operate is infinite. But the comparison with the engineer's task is not the only one; one could also compare the programmer's task to that of a mathematician:

> It is impossible to prove a mathematical theorem completely, because when one thinks that one has done so, one still has the duty to prove that the first proof was flawless, and so on, ad infinitum. [...] One can never guarantee that a proof is correct, the best one can say is "I have not discovered any mistakes." [...] The programmer is exactly in the same position, since it is not possible for him to prove the correctness of his programs. And yet the correctness of the programs is of vital importance: everybody working with an automatic computer knows from sad experience that it is very easy to produce an awful lot of [results], but he also knows that they are worthless if their correctness is subject to doubt.[43]

Seeking at proving the correctness of programs clearly looks like a wrong idea, simply because it is ultimately not possible to do so. It is in no way better than test cases: like a set of test cases, proving the correctness of a program could increase the confidence in its correctness, but, since the proof itself may be wrong, it is not an adequate means to ensure that it is actually correct. But if neither testing nor proving are sufficient to be certain that programs conform to their specification, one may think that the correctness ideal is not reachable, and therefore not worth considering. However this is not the last word on that question:

> The correctness of a [program] can never be founded on successful tests alone, but is ultimately derived from the clean and systematic structure of the [program] and from nothing else. [...] The most difficult aspect of [a programmer's] task is to convince himself — and those others who are really interested — that the program he has written down defines indeed the process he wanted to define.[44]

42. DIJKSTRA, E. W., *On the Design of Machine Independent Programming Languages*, p. 4

43. DIJKSTRA, E. W., *On the Design of Machine Independent Programming Languages*, pp.3–4

44. DIJKSTRA, E. W., *Some Meditations on Advanced Programming*, pp. 8–9

The correctness of a program is thus ultimately a conviction of the programmer, who on the grounds of the "clean and systematic structure" of the program he has written, has the certitude that it does what he wanted it to do. He may of course explain that structure to someone else, thereby allowing him to be convinced in turn that the program is indeed correct.

This last notion seems to be closely related to the precept of clarity yet to be defined (see p. 18 and p. 22). Indeed, while discussing the qualities that a programmer should have, Dijkstra notes:

> Apart from the programs that have been produced the programmers contribution to human knowledge has been fairly useless. They have concocted thousands and thousands of ingenuous tricks but they have given this chaotic contribution without a mechanism to appreciate, to evaluate these tricks, to sort them out. [...] The programmer was judged by his ability to invent and his willingness to apply tricks. And this opinion is still a wide-spread phenomenon: in advertisements asking for programmers and in psychological tests for this job it is often required that the man should be "puzzle-minded", this in strong contrast to the opinion of the slowly growing group of people who think it more valuable that the man should have a clear and systematic mind.[45]

A programmer with a "clear and systematic mind" will write programs with a "clean and systematic structure." Clarity is thus understood as the opposite of chaotic or confused, tricky or mysterious, puzzled or enigmatic. To say it positively, clarity is now explicitly conceived as "easy to understand." If a program has indeed a clean and systematic structure, that is, if it is clear, if it is easy to understand, then one sees how a programmer can be certain that it does what he wanted it to do, without testing or proving its correctness, and how he may communicate that conviction to someone else. This explains what is meant when he writes that:

> The greatest virtues a program can show [are] elegance and beauty.[46]

Elegance and beauty are again other notions to explicit that concept of clarity, which is now the main quality one should seek when writing programs, and from which other qualities, for instance correctness or reliability, and flexibility, derive.

The other qualities which were previously brought to the fore, namely speed, memory usage, safety, accuracy (see p. 17) and restartability (see p. 18) are not considered. Safety is a property that the programmer shouldn't worry about, because computers now have build-in error detection mechanisms. Accuracy is a secondary point: it only concerns programs which deal with numerical data, and many programs do not deal exclusively with numbers any more: it is thus not a property that should be generally sought. Restartability was a concern for programs which modified their own instructions during their execution, but this is regarded as an obsolete technique, and further it is obviously not relevant for ALGOL 60 programs. Memory usage and speed are cited, but only for the record, and with an apparent skepticism:

45. Dijkstra, E. W., *Some Meditations on Advanced Programming*, p. 3

46. Dijkstra, E. W., *Some Meditations on Advanced Programming*, p. 10

> For a large group of people good use of a machine is synonymous with efficient use of a machine. An the only two criteria by which they judge the quality of a program [...] are requirements of "time and space." I have a suspicion, however, that in forming their judgement they restrict themselves to these two criteria, not because they are so much more important than other possible criteria, but because they are so much easier to apply on account on their quantitative nature. [...] The sacrosanctity of these two criteria is a widespread phenomenon. [...] All in all, there is sufficient reason to call for some attention to the more imponderable aspects of the quality of a program.[47]

The most important of those imponderable, or qualitative, aspects of the quality of a program is of course its clarity.

Another aspect regarding the question of the programming activity is the structuring concept to be used when composing programs. Under the influence of ALGOL 60, the notion of "paragraph", consisting in a logical group of memory pages (see p. 20), is abandoned in favor of the notion of subroutine as the main structuring concept. The structure of the compiler, in both its translator and interpreter parts, shows clearly, in comparison with the structure of the X1's communication program, that the notion of subroutine is now indeed the main organizing principle: it is composed of about eighty clearly separated subroutines. However, the corresponding notion of procedure in ALGOL 60 seems to be understood at first in a rather limited sense:

> Now we shall consider an example in which we multiply a matrix A, consisting of 20 columns of 6 elements each, by a column vector B. [...]

```
for i := 1 step 1 until 6 do
begin
  s := 0;
  for j := 1 step 1 until 20 do s := A[i,j] × B[j] + s;
  print (s)
end
```

> [...] This little program controls, among others, the execution of 120 multiplications, and it is, therefore, a striking illustration of the compactness of descriptions in ALGOL. **For** statements are not the only means of shortening program texts: at least as important is the extremely flexible form of abbreviation which is available to us in the form of what is called a 'procedure'.[48]

As it was the case for subroutines (see p. 16), procedures seem to be understood primarily as a way to abbreviate the program text. This is already quite good, as the compactness of a program text may contribute to its clarity. But it is not his last word: the fact that procedures may use local variables, that they may receive parameters by value or by name, and that they may be called recursively, make them a far richer concept and a far more flexible tool:

> Calling a formal parameter by name is a perfectly natural linguistic element, as is shown in the following ALGOL 60 transcription of the normal summation sign. [...] Instead of the double summation

47. Dijkstra, E. W., *On the Design of Machine Independent Programming Languages*, pp. 2–3

48. Dijkstra, E. W., *A Primer of ALGOL 60 Programming*, pp. 39–42

$$\sum_{k=1}^{10}\sum_{h=1}^{20} B[k,h]$$

one may write

$$SIGMA\ (k, 1, 10, SIGMA\ (h, 1, 20, B[k, h]))$$

where the procedure *SIGMA* could be given in the declaration:

```
real procedure SIGMA (i, L, U, ti);
value L, U; integer i, L, U; real ti;
begin
  real s;
  s := 0;
  for i := L step 1 until U do s := s + ti;
  SIGMA := s;
end;
```

> In the procedure *SIGMA*, the last parameter is as a rule an expression depending on the first. This use [...] illustrates the full power of the possibility of calling by name. With this simple example [...] I hope to have shown that calling by name is not so unnatural at all, on the contrary, [and] that [...] it closely corresponds to a well-established mathematical notation, which is known and used all over the world.[49]

This "simple example" shows undeniably that the "full power" of the concept of procedure in ALGOL 60 is not ignored, that it is moreover actually used, and that it contributes, when properly used, to the clarity of programs: it renders programs easier to understand, since it may for instance be used to express well-known, precise and unambiguous mathematical constructions. The "full power" of the concept of procedure is especially related to the possibility to call procedures recursively: in that case the procedure can not be considered as a simple abbreviation any more, since it is then not possible to substitute the body of the procedure in place of the statement calling it.

Dijkstra's participation to the definition of ALGOL 60, his experience in writing a compiler for it, and in teaching it, are at the root of a more developed reflection on the nature and qualities of programming languages, all the more because ALGOL 60's defects rapidly became apparent:

> Through its merits ALGOL 60 has inspired a great number of people to make translators for it, through its defects it has induced a great number of people to think about the aims of a programming language.[50]

Among its defects, ALGOL 60 does not explicitly prescribe an evaluation order for expressions, and ambiguous expressions could be written down, because they may contain function calls which may, as a side effect, modify the value of the variables present in the expression. Also, the writing of really portable programs is not really possible, because it does not define any standard input-output functions. Finally, it only has three variable types, namely integers, reals and Booleans, which limits its applicability to the definition of mostly numerical processes.

49. Dijkstra, E. W., *Defense of ALGOL 60*, pp. 502–503

50. Dijkstra, E. W., *Some Meditations on Advanced Programming*, p. 6

Dijkstra had previously identified a desirable quality for a programming language, namely its elegance (see p. 21), as an equilibrium between excessive flexibility and excessive rigidity of a computer's instruction set. The grounds for this requirement were that a lack of elegance could render the writing and the reading of programs harder than necessary; to say it more precisely, it could lead to a lack of clarity. Elegance is still put forward, but appears to receive another definition:

> We should aim at a programming language consisting of a small number of concepts, the more general the better, the more systematic the better, in short: the more elegant the better.[51]

This appeal for a language to be general and systematic is founded on the observation that existing programming languages often include unnecessary redundancies and restrictions. In ALGOL 60 for example, procedures are an exception amongst declarations: they are declared statically, which means that their definition is bound to an identifier once and for all when they are declared, contrary to variables whose actual value may change during the execution of the program. This is an unnecessary restriction, as this feature could be convenient for procedures under certain circumstances, and it limits the power of expression of the language without real reasons, which thus lacks generality. Likewise, ALGOL 60 requires one to declare the variables one uses. However these information are already present in another form, inside of the statements, where the variables are used, and it may be derived by the compiler from the program text: the language is thus redundant. Unnecessary restrictions may lead to needless complex programs, and unnecessary redundancies may lead to contradictory programs. Those two characteristics correspond to the excessive rigidity and the excessive flexibility, and it is then manifest that the ideal of elegance is in fact unaltered, although it is more precisely defined.

As already noted, this ideal is strongly connected with the ideal of clarity of the programs. If one has to seek for clarity when writing programs, then this clarity will obviously be found, at least partly, in the program texts. The programming language is a tool at the programmer's disposal, and a tool is not without influence on the one who uses it, hence the need to consider it with care. If it is adequately designed, it may help him to attain his goals, and if not, it may hinder him to do so:

> I would require of a programming language that it should facilitate the work of the programmer as much as possible, especially in the most difficult aspects of his task, such as creating confidence in the correctness of his program.[52]

To assist the programmer, the language should naturally be elegant. But this could be considered as a requirement meant more or less to the intellectual satisfaction of language designers, whereas it has a very concrete and practical advantage, which could serve as another justification for its pursuit:

> We must make it as easy as possible for the user to master the language.[53]

51. DIJKSTRA, E. W., *On the Design of Machine Independent Programming Languages*, p. 4

52. DIJKSTRA, E. W., *On the Design of Machine Independent Programming Languages*, p. 4

53. DIJKSTRA, E. W., *On the Design of Machine Independent Programming Languages*, p. 6

Indeed, if a language is general and systematic, if it does not suffer from a number of irregularities, of peculiarities or redundancies, it is easier to master, both for writing programs and for reading them. One should however not take this practical advantage for the goal itself, as it is often the case:

> There is a tendency to design programming languages so that they are easily readable for a semi-professional, semi-interested reader. [...] It looks so attractive: "Everybody can understand it immediately." But giving a plausible semantic interpretation to a text which one assumes to be correct and meaningful, is one thing; writing down such a text in accordance with all the syntactical rules and expressing exactly what one wishes to say, may be quite a different matter! [...]
>
> I do not regard the supposed readability for a general reader as a valid criterion. [...] In human communication the "unpredictability" of those we address plays a fundamental role. If we now apply the norms of human communication to an artificial language, in which we wish to address a computer, then we ignore one of the most essential characteristics of the automatic computer, namely the "predictability" of its behavior.[54]

Programming languages designed with the aim of being easy to learn, to read and to write, for example by making use of words taken from the natural language, like the newly born COBOL which uses English words to express all its operations, take a wrong path. One may however ask why the use of a natural language is a problem, since this seems to only ease the learning of the language, without ignoring the computer's strict predictability. It does indeed not prevent one to write absolutely unambiguous programs, "in accordance with all the syntactical rules and expressing exactly what one wishes to say": any COBOL program, for instance, has only one valid interpretation. The reason lies elsewhere:

> We should be aware of the fact that for the first time in the history of mankind, we have a servant at our disposal who really does what he has been told to do. In man-computer communication there is not only a need to be unusually precise and unambiguous, there is — at last — also a point in being so, at least if we wish to obtain the full benefits of the powerful obedient mechanical servant. Efforts aimed to conceal this new need for preciseness — for the supposed benefit of the user — will in fact be harmful; at the same time they will conceal the equally new possibilities in automatic computing, of having intricate processes under complete control. [...] I am all in favor of clear and convenient algorithmic languages but, please, let them honestly be so — to disguise them in clothes which have been tailored to other purposes can only increase the confusion.[55]

The problem with such a language is thus not that it is imprecise or ambiguous, but that it conceals the need to be precise and unambiguous. Indeed, we all use natural languages, and when we use them, we are given to use their words in a often vague and ambiguous sense. In a programming language which makes an extensive use of those words, they receive a precise and unambiguous sense, but the necessity of precision is less apparent, precisely because they could be interpreted loosely. Instead of helping the programmer to write clear programs, such languages will increase their confusion. Again, this is not

54. DIJKSTRA, E. W., *On the Design of Machine Independent Programming Languages*, pp. 5–7

55. DIJKSTRA, E. W., *Some Comments on the Aims of MIRFAC*

only a matter of intellectual purity: it has the practical consequence that it will make the writing of "intricate processes" harder than needed. Very simple programs will perhaps gain in clarity, at least in the sense that they will be easier to read for a non-programmer, but as soon as the problems become more complicated, the corresponding programs will become needlessly confused. These considerations further reveal that clarity should not be considered as an absolute requirement, but as a relative one. The ideal of clarity should not be understood as a quest for a universal intelligibility, but is to be conceived more accurately as the search to be as easy to understand as possible, for an knowledgeable reader.

It is not enough, however, for a programming language to be elegant to actually make it as easy as possible for programmers to master:

> We can fully master the way in which a computer reacts and this is precisely the reason why addressing an automatic computer presents us with undreamt-of linguistic possibilities. Mastery of the reaction of the computer must not only be a theoretical possibility but a real, practical one, if one is to be able to make full use of those linguistic possibilities. It is therefore mandatory that [the language] be not prohibitively complicated. From this point of view the way in which ALGOL 60 is defined is rather alarming. ALGOL 60 is defined by the official *Report on the Algorithmic Language ALGOL 60*, edited by Peter Naur, but reasonably speaking one cannot expect a user of the language to know this *Report* by heart.[56]

The language has thus also to be defined as clearly as possible. The very fact that the *Report*, which is only twelve pages long, moreover illustrated by many examples, and whose only formalism is the Backus Naur Form, is considered not readable enough, not easy enough to be understandable, that is, not clear enough, reveals that clarity is not to be achieved by taking mathematics as an example, and by making use of a formalism resembling to the one used in mathematics. Indeed, the main differences between the *Report* and the *Primer of ALGOL 60 Programming* is that the latter does not make any use of the Backus Naur Form to present the language, and that ALGOL 60's constructions are introduced by order of difficulty rather than by a logical order. However, since the *Primer of ALGOL 60 Programming* does not pretend to completely replace the *Report*, but only to serve as a readable introduction to the language, it remains to be understood how the definition of a language could be given in a clear way.

In addition to the reflection on the aims and the qualities of programming languages, Dijkstra investigates their relation to computers, from another point of view than that of their translation or execution:

> A machine defines (by its very structure) a language, namely its input language; conversely, the semantic definition of a language specifies a machine that understands it. In other words: machine and language are two faces of one and the same coin.[57]

This is obviously true for hardware computers, whose construction define an order code to which they obey, and conversely for order codes, whose specification define the various

56. DIJKSTRA, E. W., *On the Design of Machine Independent Programming Languages*, p. 8

57. DIJKSTRA, E. W., *An Attempt to Unify the Constituent Concepts of Serial Program Execution*, p. 1

computers which can be build to execute them. Since a hardware computer can also be implemented with a computer program, this is also true for a program whose operational behavior follows the specification of that hardware computer, and for the order code of the hardware computer it implements. But the machine can be an abstract one, and this is also true for a computer program implementing such a machine, like the X1's interpreter, and for its order code. It is then also true in general for any programming language, and for the programs which perform the execution of the programs written in that language. In other words: if a computer is a defined as tool which mechanically interprets a given program text describing a computational process, then a hardware computer is a computer, and any program which interprets such a program text on a hardware computer is also a computer. A compiler is thus an abstract machine, and the programming language it implements is its "order code."

In an attempt to identify and to understand the primitive operations of programming languages which cater for recursiveness, Dijkstra then tries to generalize the design of the X1's ALGOL 60 compiler as much as possible. To that end, he designs a language and an abstract machine which models the primitive operations of those languages in a clear, simple and elegant way. The language is a very simple one: programs are entirely written in a postfix notation. It has a few separators, that is, words which will be interpreted in a special way by the abstract machine: one to ask for evaluation, two to assign a value to a variable, either a simple value or a string (sequence of words) value, one that signals the end of a string, and two which on evaluation will be replaced respectively by the separator asking for evaluation and the separator signaling the end of a string. The corresponding abstract machine reads in the programs word by word, and pushes those words on a stack, called the anonymous stack. If it encounters a separator asking for evaluation, it performs the operation defined by the element on the top of that stack, possibly with the help of a few other elements on the stack: for example, if the first element on the stack is a binary operator, the corresponding operation is performed, and the stack shrinks by two words; if it is a variable, it is replaced by its current value if it is a simple one, or its current value is read in word by word if it is a string. Because the evaluation of a variable may recursively lead to the evaluation of other variables, namely if its value is a string of words, and because the value of new variables must be stored somewhere, a second stack is needed, called the stack of activations. This model lets him conclude for instance that in such languages data and instructions are actually similar objects, since both can be bound to a variable, whose evaluation implies either its replacement by its value, or its recursive evaluation. This language and its abstract machine can be considered as an ancestor of FORTH, developed a few years later by Charles H. Moore: the FORTH language uses a postfix notation, and the FORTH abstract machine uses two stacks, a data stack and a return stack, which correspond respectively to the anonymous stack and the stack of activations.

This rather theoretical reflection has again a practical consequence. We have already noted that a language should be defined as clearly as possible. In other words:

> The semantic definition [of a programming language] should be as rigorous as possible. In short: we need a complete and unambiguous pragmatic definition of the language, stating explicitly how to react to any text.[58]

Languages needs thus to have a precise and unambiguous definition, the semantics of each of their constructions must be specified rigorously. One may think that this characteristic generally goes hand in hand with the requirement of elegance, if it is met, and that this further precision is pointless: it is indeed difficult to imagine a language that is general and systematic without being carefully defined. However, the two requirements of clarity and rigor could be seen as conflicting, and the way in which clarity and rigor should be achieved is subject to interpretation: since the *Report on the Algorithmic Language ALGOL 60* does not seem to meet those qualities, it could be interpreted for instance as an appeal to use a formal language to give the language its semantics, or on the contrary to define the language with as much technical details as possible. Those methods would be both misleading; on the contrary:

> The language [and the abstract machine thus] described [...] may prove to be a suitable means for the formalization of the semantic definition of an algebraic language. The lack of such a rigorous definition is one of the recognized shortcomings of the official *Report on the Algorithmic Language ALGOL 60*.[59]

Indeed, such an abstract machine and its corresponding language embody both the qualities of clarity and rigor: it is as simple as possible, and it gives to each of the constructions of the language strictly sequential semantics. The abstract machine can thus be considered as a defining machine for the language it implements, or, in other words: as an adequate means to define that language.

What conclusions can be drawn from those observations? The concept of clarity, which appeared sporadically in Dijkstra's works during the first eight years, is now seen as central, because on the one hand the correctness of programs depends on their clarity, and on the other hand the elegance of programming languages aims at the clarity of the programs. Its meaning is also refined: it is conceived as the search to be as easy to understand as possible for someone who has the necessary knowledge. The ideal of elegance is more precisely defined too, and the notion of structure makes its appearance without being explicitly examined. Besides, one may note that the need for correctness, which was not explicit till now, is brought to the fore. Another novel aspect of his work is the constant effort towards the abstraction of the problems he is faced with, and the generalization and simplification of the solutions he devises.

As will be seen in chapter III, his work on a compiler for ALGOL 60 is, in many of its aspects, a stepping stone to the design of the "THE" operating system, which is also the continuation of the design and writing of the X1's communication program.

58. DIJKSTRA, E. W., *Some Meditations on Advanced Programming*, p. 10

59. DIJKSTRA, E. W., *An Attempt to Unify the Constituent Concepts of Serial Program Execution*, p. 17

CHAPTER III

AN OPERATING SYSTEM (1962–1968)

In September 1962, Dijkstra is appointed Professor of Mathematics at the Technological University of Eindhoven. Besides a teaching responsibility, he is in charge of developing the existing calculation group of the mathematics department. Upon his arrival, an IBM 1620 computer is under operation, but has become a bit limited to meet the needs: the acquisition of an X8, Electrologica's successor to the X1, is decided. To process the flow of user programs as efficiently as possible, it has to be equipped with an operating system. Dijkstra soon finds a solution to the fundamental problem one is faced with when conceiving an operating system (§ 1) — and then, as the head of a small team, designs and writes an operating system for the X8 based on that principle, named "THE" for "Technische Hogeschool Eindhoven" (§ 2).

§ 1. Loosely Synchronizing Concurrent Processes

The purpose of an operating system is to share a number of resources amongst a number of processes that are executed concurrently: the processor, the memory, and the communication devices or peripherals. As long as those processes simply perform calculations, no problems appear: one simply has to slice the time, and to allocate a portion of it in turn to each of the running processes. However, as soon as those processes either need to make use of the communication devices, or have to communicate with each other, complications arise: then one has to ensure that they don't use the same communication apparatus at the same time, and that they access the portion of memory they use to communicate without interfering with each other. Those difficulties show some similarity with the problem which had to be solved for the X1, namely, to let the communication devices work concurrently with the execution of a single program. The communication apparatus can indeed be abstracted, and considered as a number of additional processes running concurrently with the program executed on the computer:

> It is not unusual to regard a classical computer as a sequential computer coupled to a number of communication mechanisms for input and output. Such a communication mechanism, however, performs in itself a sequential process. [...] For this reason we can regard a classical machine, its communication mechanism included, as a group of loosely

> connected sequential machines, with interlocks, where necessary, to prevent them to get too much out of phase with one another. The next step is to use the central computer not for only one sequential process but to equip it with the possibility to divide its attention between an arbitrary number of such loosely connected sequential processes.[1]

The only resources to be shared amongst all those processes are then the processor and the memory. If processes have to communicate with each other via a portion of memory, one should ensure that no two processes modify its contents at the same time, and that one of those processes does not read its contents while another one is modifying it: those two operations are mutually exclusive. This implies that a process may have to wait for another process to finish part of its operations, or, in other words, that processes which have to communicate with each other should be synchronized to a certain extent, or "loosely connected." Ensuring that program processes don't use the same communication device at the same time is then a particular case of communication between processes: it amounts to ensuring that program processes don't communicate with the same communication process at the same time. Even the user of the computer can be considered as such a process, with which only one program can communicate at any given moment for instance. Again these are problems of mutual exclusion, which suppose a certain degree of synchronization. The fundamental problem, when a number of processes are executed concurrently, is thus that of mutual exclusion: most of their operations can take place simultaneously, but some cannot, and it is the responsibility of the operating system to ensure that those mutually exclusive operations do indeed take place one after the other.

To devise a mechanism to make sure that mutually exclusive operations do not take place simultaneously, one should first precisely identify what may be supposed, and what may not be supposed, in building up a solution. One could for instance try to measure or to calculate the speed of each of the concurrent processes. It would then be possible to adapt those processes to achieve that synchronization, for example by inserting waiting loops at well-chosen places. But this ad-hoc solution obviously lacks generality: if one of the processes has a speed which depends on its input data, as it is usually the case, those waiting loops will be extremely intricate to write. Likewise, if one of the processes is modified, for instance, if one of the communication apparatus is replaced by a faster one, its speed having changed, all the other processes have to be adapted.

> We have stipulated that the processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes are to be regarded as completely independent of each other. In particular we disallow any assumption about the relative speeds of the different processes.[2]

If assumptions about the speeds of the processes were allowed, this would mean that they are not "completely independent" of each other; in other words, this would mean that they are all implicitly synchronized with another process, for example with an external clock, or

1. DIJKSTRA, E. W., *Some Meditations on Advanced Programming*, p. 5

2. DIJKSTRA, E. W., *Cooperating Sequential Processes*, p. 10

that they implicitly communicate with each other. To explicitly forbid such assumptions, the following is added as an additional requirement that the solution sought for has to satisfy, besides of course that of mutual exclusion:

> Nothing may be assumed about the relative speed of the [processes]; we may not even assume their speeds to be constant in time.[3]

If no assumptions about their speeds are allowed, then synchronization can evidently only be achieved if the processes can communicate with each other, independently of the communication mechanisms they may use to exchange data, to notify each other when they may proceed and when they should wait. This means that they should all have access to a common memory, which is obviously already the case: all processes and all communication devices have reading and writing access to the computer's memory, the latter at least to get the parameters and to write the results of their operations. This memory can be assumed to have some properties which are relevant for the problem at stake:

> In order to effectuate this mutual exclusion the [...] processes have access to a number of common variables. We postulate, that inspecting the present value of such a common variable and assigning a new value to such a common variable are to be regarded as indivisible, non-interfering actions. That is, when the [...] processes assign a new value to the same common variable "simultaneously", then the assignments are to be regarded as done the one after the other, the final value of the variable will be one of the two values assigned, but never a "mixture" of the two. Similarly, when one process inspects the value of a common variable "simultaneously" with the assignment to it by [another] one, then the first process will find either the old or the new value, but never a mixture.[4]

Writing a value in a given memory location, and reading a value from a given memory location, are indivisible or "atomic" operations: only one of the concurrent processes can write in a single memory word at a time, and while one process writes in a given memory word, another process cannot read it, and vice versa. This means that reading and writing single memory words are mutually exclusive operations. This mutual exclusion is realized at the hardware level. If a number of program processes share a single processor, then at any moment only one process will be in the course of its execution, and, since an interrupt can only take place between the execution of two instructions, the processor will never switch to the execution of another process in the middle of the execution of one of its instructions. If a program process and a peripheral process try to access the same memory word simultaneously, then one of them is momentarily blocked until the other one has finished its operation, thanks to a so-called "switch" with which every memory word is equipped; this mechanism was already present in the X1 to avoid possible inconsistencies if the main program and a communication apparatus, or two communication apparatus, ever tried to access the same memory word simultaneously. Finally, if the computer had multiple processors, two program processes executed on two distinct processors would have a mutually exclusive access to the individual memory words, thanks again to the

3. DIJKSTRA, E. W., *Solution of a Problem in Concurrent Programming Control*

4. DIJKSTRA, E. W., *Cooperating Sequential Processes*, p. 11

switch mechanism.

No other assumptions can be made, and it is thus on this low-level mutual exclusion, ensured by the hardware, that the mutual exclusion of some parts of concurrent processes, their so-called "critical sections", has to be build.

Without loss of generality, this problem can be expressed in its most simple form as follows: execute a number of concurrent processes, each one with a single critical section, while ensuring that at any moment at most one of them is in its critical section. A similar problem has already been solved in the writing of an extension to the communication program for the X1, namely in writing a program in which the input and output operations were synchronized mutually (see p. 13). The solution then given, which would be incorrect in a true concurrency context, can be generalized as follows:

```
begin integer turn; turn := 1;
  parbegin
  process 1: begin L1: if turn = 2 then go to L1;
                       critical section 1;
                       turn := 2;
                       remainder of cycle 1;
                       go to L1
             end;
  process 2: begin L2: if turn = 1 then go to L2;
                       critical section 2;
                       turn := 1;
                       remainder of cycle 2;
                       go to L2
             end
  parend
end
```
[5]

This solution appears to be correct: with the help of a variable *turn*, taking alternatively the values 1 and 2, it guarantees that the two critical sections are never executed simultaneously: at most one of the two processes can be in its critical section at any given moment. Moreover, it does not rely on any assumption about the speeds of the two processes. It is, however, unnecessarily restrictive, precisely because the variable *turn* takes alternatively the values 1 and 2. In other words, *process 1* cannot enter its critical section again before *process 2* has entered and completed its own critical section. This solution is only correct for problems like the one solved on the X1, where one of the processes depends on the other one to continue its operations, which implies that when one of the processes is stopped, the other one should not be allowed to continue. It is thus not general enough: the two processes are strictly synchronized instead of being loosely synchronized, they are not independent enough from each other. To clarify this distinction, a second additional requirement to be met is set forth:

> If any of the [processes] is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.[6]

5. DIJKSTRA, E. W., *Cooperating Sequential Processes*, p. 12

6. DIJKSTRA, E. W., *Solution of a Problem in Concurrent Programming Control*

How this generalization should take place is however anything but evident. A first idea would be to attach to each process a Boolean variable indicating whether it is in its critical section, and to set that variable before entering and after exiting the critical section:

```
begin Boolean p1, p2; p1 := p2 := false;
  parbegin
  process 1: begin L1: if p1 then go to L1;
                       p1 := true;
                       critical section 1;
                       p1 := false;
                       remainder of cycle 1;
                       go to L1
             end;
  process 2: begin L2: if p2 then go to L2;
                       p2 := true;
                       critical section 2;
                       p2 := false;
                       remainder of cycle 2;
                       go to L2
             end
  parend
end[7]
```

This solution is simple, but alas wrong: since both processes are executed concurrently, it could happen that they both execute the statements labeled *Li* simultaneously. They will then both find that their respective *pi* variable is set to **false**, they will both perform the statement *pi* := **true**, and they will both enter their critical section. This solution does thus not guarantee the basic requirement of mutual exclusion. To circumvent this difficulty, one could try to invert, at the beginning of the parallel processes, the inspection and the setting of the *pi* variables, and to let each process inspect not his own *pi* but the one of the other process:

```
begin Boolean p1, p2; p1 := p2 := false;
  parbegin
  process 1: begin A1: p1 := true;
                   L1: if p2 then go to L1;
                       critical section 1;
                       p1 := false;
                       remainder of cycle 1;
                       go to A1
             end;
  process 2: begin A2: p2 := true;
                   L2: if p1 then go to L2;
                       critical section 2;
                       p2 := false;
                       remainder of cycle 2;
                       go to A2
             end
  parend
end[8]
```

7. Cf. DIJKSTRA, E. W., *Cooperating Sequential Processes*, p. 14

8. Cf. DIJKSTRA, E. W., *Cooperating Sequential Processes*, p. 15

Mutual exclusion is then indeed guaranteed: if one process enters its critical section, it is because he found the variable *pi* of the other process to be **false**, and at that moment his own variable *pi* is already set to **true**, which prevents the other process to enter its critical section until he sets it again to **false**. But this solution again is too restrictive: if both processes execute the statement labeled *Ai* simultaneously, then both *pi* will be set to **true**, and they will both loop indefinitely in their statement *Li*. In other words, this solution does not protect the processes against an indefinite mutual blocking, known as a "deadlock." This could perhaps be remedied by replacing the statements labeled *Li* with the following ones:

```
L1: if p2 then begin p1 := false; go to A1 end;
L2: if p1 then begin p2 := false; go to A2 end;
```

Even if both processes execute their statements *Ai* and *Li* simultaneously, this will not necessarily lead to a deadlock, because both processes will reset their own *pi* to **false** before trying again to enter their critical section, thereby giving the other process the opportunity to go a little bit faster through its statements *Ai* and *Li*, and to enter its critical section. This will however only work with some luck: it could indeed happen that the speeds of the two processes are equal, in which case they will both be locked in an indefinite loop. To make clear that deadlocks should be prohibited, whatever be the speeds of the different processes, a third and last additional requirement to be verified is stated:

> If more than one [process] is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity.[9]

The final solution, due to T. J. Dekker, to this apparently simple problem, turns out to be fairly complicated, and uses ideas from all the previous tentatives: two Boolean *pi* variables, a loop which resets *pi* to **false** if one finds that the other *pi* has been set to **true** simultaneously, and a *turn* variable to make sure that in this last case only one of the two processes will be allowed to set its *pi* to **true** again. The complete program reads:

```
begin Boolean p1, p2; integer turn; p1 := p2 := false; turn := 1;
  parbegin
  process 1: begin A1: p1 := true;
                   L1: if p2 then
                         begin if turn = 1 then go to L1;
                                 p1 := false;
                            B1: if turn = 2 then go to B1;
                                 go to A1
                         end;
                         critical section 1;
                         turn := 2; p1 := false;
                         remainder of cycle 1;
                         go to A1
             end;
```

9. Dijkstra, E. W., *Solution of a Problem in Concurrent Programming Control*

```
process 2: begin A2: p2 := true;
                 L2: if p1 then
                       begin if turn = 2 then go to L2;
                               p2 := false;
                          B2: if turn = 1 then go to B2;
                               go to A2
                       end;
                       critical section 2;
                       turn := 1; p2 := false;
                       remainder of cycle 2;
                       go to A2
           end
  parend
end
```
[10]

Each one of the requirements which were set forth — mutual exclusion, absence of speed assumptions, process independence, and absence of deadlocks — is met. As in the previous tentative (see p. 59), mutual exclusion is guaranteed: a process can only enter its critical section if the other one is outside its critical section, that is, if the *pi* of the other process is **false**, since to enter its critical section it has to execute the conditional statement *Li*, which will either loop or go back to *Ai* if *pi* is found to be **true**. But, contrary to that previous tentative, no deadlock is possible: if both processes execute their statements labeled *Ai* simultaneously, and enter the statements labeled *Li* simultaneously, then, depending of the value of *turn*, one of them will loop on *Li*, and the other one will proceed and reset its *pi* to **false**, thereby allowing the former to enter its critical section, before looping on *Bi*. The latter will be freed when the former leaves its critical section and resets *turn*. Finally, if one of the processes is stopped outside of its critical section, there is no risk of preventing the other one to enter its critical section, since in that case both *pi*'s are **false**.

This is already a nice result, but it is again not general enough, in this case not because it has a defect, but because it is limited to the mutual exclusion of the critical sections of two processes. It is not straightforward at all how it could be generalized to the case of N processes: adding more *pi*'s and more values to *turn* has the consequence that the statements *Li* and *Bi* cannot be written down as conditionals simply testing for equality. The general solution is discovered by Dijkstra, and reads as follows:

```
begin Boolean array p, t[0:N]; integer turn;
  for turn := 0 step 1 until N do p[turn] := t[turn] := false;
  parbegin
  ...
  process i: begin integer k;
               Ai: p[i] := true;
               Li: if turn ≠ i
                   then begin t[i] := false;
                              if ¬p[turn] then turn := i;
                              go to Li
                        end
```

10. Cf. Dijkstra, E. W., *Cooperating Sequential Processes*, p. 17

```
                    else  begin t[i] := true;
                                for k := 1 step 1 until N do
                                   if k ≠ i ∧ t[k] then go to Li
                          end;
                    critical section i;
                    p[i] := t[i] := false;
                    remainder of cycle i;
                    go to Ai
          end;
  ...
  parend
end
```
[11]

All the necessary requirements are again satisfied. Mutual exclusion is verified and no deadlock is possible. Indeed, on the one hand, it is guaranteed that the value of *turn* will necessarily be fixed after a finite time: if a number of processes execute the statements Ai and Li concurrently, as soon as one of them has set *turn*, $p[turn]$ becomes **true**, and no other process can decide to change the value of *turn* from then on. And, on the other hand, a process i can only enter its critical section after having set $t[i]$:= **true** and checked that all other $t[i]$'s are **false**. If multiple processes perform this check simultaneously, it will succeed for at most one of them. If it succeeds for none of them, then only process *turn* will be allowed to set $t[i]$:= **true** again, and all the other ones will have to set $t[i]$:= **false**, which will necessarily happen in a finite time. Finally, no other process j can enter its critical section once a process i has completed this check: since they will all find either $turn \neq j$ and $p[turn]$ = **true**, or $turn = j$ but $t[i]$ = **true**, they can only loop on Lj, until process i exits from its critical section and performs $p[i]$:= $t[i]$:= **false**. The requirement of independence is obviously also verified: if a process j is stopped outside its critical section, it will not stop other processes from entering their critical section: for that process $p[j]$ = **false** and $t[j]$ = **false** permanently.

This solution, though correct, is indeed extremely complicated. This has the unfortunate consequence that it is hardly adaptable even to only slightly different situations: if, for instance, in four processes A, B, C and D, each one having a critical section, only those of A and B, B and C, C and D, and D and A exclude each other, that is, if the simultaneous execution of the critical sections of A and C or B and D is allowed, the previous algorithm would of course still be correct, but it would be too restrictive, and adapting it to this additional freedom is again not simple at all. It is thus worth the effort to try to find a simpler, and therefore more flexible, solution.

> The origin of the complications, which lead to such intricate solutions [...] is the fact that the indivisible access to common variables are always "one-way information traffic": an individual process can either assign a new value or inspect a new value. Such an inspection, however, leaves no trace for the other processes and the consequence is that, when a process wants to react to the current value of a common variable, its value may be changed by the other processes between the moment of its inspection and the following [execution] of the reaction to it. In other words: the previous set of communication

11. Cf. DIJKSTRA, E. W., *Solution of a Problem in Concurrent Programming Control*

> facilities must be regarded as inadequate for the problem at hand and we should look for better adapted alternatives.[12]

Indeed, if it was possible to have somewhat more elaborate atomic operations than those that are provided by the hardware, which allow for either inspection or modification of an individual memory word, the solution would have been easy. If it was for instance possible to declare **atomic** procedures, then a single common Boolean variable would have been sufficient to solve the problem for N processes:

```
begin Boolean t; t := false;
  atomic Boolean procedure enter (b); Boolean b;
    begin enter := b; if ¬b then b := true end;
  procedure exit (b); Boolean b;
    b := false;
  parbegin
  ...
  process i: begin Li: if enter (t) then go to Li;
                       critical section i;
                       exit (t);
                       remainder of cycle i;
                       go to Li
             end;
  ...
  parend
end
```
[13]

The variable t simply indicates whether one of the concurrent processes is in its critical section. It can only be modified by one process at a time, and it is set to **true** by the process which will enter its critical section. This solution satisfies all the requirements: since the operation *enter* is atomic, it is obvious that mutual exclusion is guaranteed and that no deadlock is possible. It is also obvious that no assumptions are made about the speeds of the processes, and that they are independent.

But even this hypothetical solution has a serious limitation, which it shares with all other previous tentatives: all the processes which are for the moment stopped from entering their critical section wait for this restriction to be removed by repeatedly performing a test and a jump; in other words, they wait in a rather busy way.

> Let us take a period of time during which one of the processes is in its critical section. We all know, that during that period, no other processes can enter their critical section and that, if they want to do so, they have to wait until the current critical section execution has been completed. For the remainder of that period hardly any activity is required from them: they have to wait anyhow, and as far as we are concerned "they could go to sleep."
>
> Our solution does not reflect this at all: we keep the processes busy setting and inspecting common variables all the time, as if no price has to be paid for this activity. But if our implementation [...] is such that "sleeping" is a less expensive activity than this busy form of waiting, then we are fully justified [...] to call our solution misleading.[14]

12. DIJKSTRA, E. W., *Cooperating Sequential Processes*, p. 28

13. Cf. DIJKSTRA, E. W., *Over de sequentialiteit van procesbeschrijvingen*, p. 11

If one has, in addition to the four requirements previously set forth, the requirement that the blocked processes should "sleep" while they cannot enter their critical section, then it reveals to have a great resemblance with the problem and the motivations which led to the creation of the X1's interrupt system (see p. 10). The motivation was to avoid wasting computation time in waiting for the completion of communication operations. The problem was thus to find a way to let the main program continue its execution while the communication program is performing its operations. The solution then found was to let an intermediary process, the communication program (or interrupt handler), suspend its activity and transfer control to the main program until another process, the communication apparatus, has finished its operation. The communication equipment signals this completion by means of an interrupt signal to the computer, which reacts to it by transferring the control back to the communication program, which handles the situation, and eventually transfers control again to the main program. The solution found by Dijkstra for the problem of the mutual exclusion of the critical sections of concurrent processes, the "semaphores", has the same motivation: avoid wasting computation time in waiting operations, and works in the same way: if a process A cannot enter its critical section, instead of waiting in a busy cycle, it suspends its activity until the process B which is currently in its critical section has completed it, and A is then waken up by B.

A semaphore is simply an integer variable, located in a memory shared commonly by the concurrent processes. Each semaphore is accessed by at least two processes, and they can only access it with two primitive atomic operations, called **P** and **V**:

> The semaphores are essentially non-negative integers. [...] The **V**-operation is an operation with one argument, which must be the identification of a semaphore. [...] Its function is to increase the value of its argument semaphore by 1; this increase is to be regarded as an indivisible operation. [...] The **P**-operation is an operation with one argument, which must be the identification of a semaphore. [...] Its function is to decrease the value of its argument semaphore by 1 as soon as the resulting value would be non-negative. The completion of the **P**-operation [...] is to be regarded as an indivisible operation.[15]

A process performing a **P** operation on a semaphore tries to decrease the value of that semaphore by one: if the current value of that semaphore is positive, then it is decreased, and the process proceeds immediately with its next operation; if it is zero, then it goes to sleep until another process performs a **V** operation on that same semaphore. A process performing a **V** operation on a semaphore simply increases the value of that semaphore by one: if it was positive, then this has no other effect; if it was zero, then this can have as consequence that another process waiting to perform a **P** operation can complete that operation and proceed with its next operation. With the help of those two primitives, the solution to the problem of mutual exclusion for N concurrent processes is straightforward:

```
begin semaphore s; s := 1;
  parbegin
  ...
```

14. DIJKSTRA, E. W., *Cooperating Sequential Processes*, pp. 26–27

15. DIJKSTRA, E. W., *Cooperating Sequential Processes*, pp. 28–29

```
    process i: begin Li: P(s);
                         critical section i;
                         V(s);
                         remainder of cycle i;
                         go to Li
               end;
    ...
    parend
end
```
[16]

It is again obvious that each of the requirements is met. Since the **P** operation is atomic, no two processes can perform it simultaneously, and therefore both the mutual exclusion and the absence of deadlocks is guaranteed. Since a process goes to sleep while it can't complete an **P** operation it has initiated, and is waken up when another process performs a **V** operation, no computation time is wasted in waiting cycles. It is also evident that no assumptions are made about the speeds of the processes, and that they are independent.

These primitives have the further advantage that they are extremely flexible: they can then be used without any modification to solve other problems than the sole problem of a number of processes having a single critical section, each of those excluding all the other ones. When they are used to solve that problem, semaphores taking only the two values 0 and 1 are obviously sufficient. The generalization to semaphores taking any natural value is due to Scholten; although logically not necessary to solve other problems than the one of critical sections, this further generalization makes them however more convenient to use in such situations. For example, the problem of two processes exchanging data via a common buffer of limited capacity, one producing a portion of data at a time, and one consuming a portion of data at a time, is also straightforward to solve with the help of semaphores:

```
begin semaphore number of queuing portions, number of empty positions;
    number of queuing portions := 0; number of empty positions := N;
    parbegin
    producer:  begin A1: produce the next portion;
                         P(number of empty positions);
                         add portion to buffer;
                         V(number of queuing portions);
                         go to A1
               end;
    consumer:  begin A2: P(number of queuing portions);
                         take portion from buffer;
                         V(number of empty positions);
                         process portion taken;
                         go to A2
               end
    parend
end
```
[17]

16. Cf. DIJKSTRA, E. W., *Cooperating Sequential Processes*, p. 30

17. Cf. DIJKSTRA, E. W., *Cooperating Sequential Processes*, p. 39

This ensures that the consuming process will never try to use a portion not yet produced by the producing process, and that it will never produce another portion of data if the buffer is already full. In the particular case $N = 1$, the operations of the two processes will take place in a strictly alternating sequence, and the algorithm is then equivalent to the first tentative solution (p. 58), except of course that it does not wait in a busy way.

This flexibility has, however, a drawback: when semaphores are used to solve other problems than the strict mutual exclusion of a number of critical sections, the problem of deadlocks can arise again for subtler reasons. As a first example, semaphores could be used to ensure the concurrent execution of a number of processes whose critical sections exclude each other cyclically, that is, where the critical section of each process only excludes the concurrent execution of the critical section of each of its "neighbors" (see p. 62). This problem is a slight generalization of the previous one, in which every process is a neighbor of every other process. It has been modeled by Dijkstra as the "five dining philosophers" problem: five philosophers sit around a circular table, and they alternatively eat and think; each philosopher has a plate in front of him, filled with a very difficult kind of spaghetti that has to be eaten with two forks, and between each philosopher is one fork. If the philosophers and the forks are numbered from 0 to 4, then one would probably first think of the following solution:

```
begin integer j; semaphore array fork[0:4];
  for j := 0 step 1 until 4 do fork[i] := 1;
  parbegin
  ...
  philosopher i: begin L: think;
                          P(fork[i]); P(fork[(i + 1) mod 5]);
                          eat;
                          V(fork[i]); V(fork[(i + 1) mod 5]);
                          go to L
                 end
  ...
  parend
end[18]
```

It is, however, inadequate: it guarantees that two neighboring philosophers will never eat simultaneously, but if the five philosophers ever get hungry simultaneously, each one will perform a **P** operation on its left hand fork, and the five are then in a deadlock. To solve the problem, it is necessary to take this intermediate "hungry" state into account, and thus to associate a state with each philosopher, which records if he is thinking, hungry or eating: each philosopher will go cyclically through the states *philosopher state*$[i] = 0$, $= 1$, and $= 2$. A philosopher i will go eating only if he is hungry, and if his two neighbors, $(i + 1)$ **mod** 5 and $(i - 1)$ **mod** 5, are not eating. It would be dangerous to let the five philosophers check for this situation and to let them decide to go eating concurrently, since one of them could decide to go eating just after one if his neighbors has seen that he was not eating; in other words, one has to prevent simultaneous consultations and

18. Cf. DIJKSTRA, E. W., *Hierarchical Ordening of Sequential Processes*, p. 21

modifications of the common variables *philosopher state*: to make this sure, an *access semaphore* is used. The complete program reads:

```
begin integer j; semaphore access semaphore;
   integer array philosopher state[0:4];
   semaphore array philosopher semaphore[0:4];
   procedure go eating (k); integer k;
   if philosopher state[(k − 1) mod 5] ≠ 2 ∧ philosopher state[k] = 1
       ∧ philosopher state[(k + 1) mod 5] ≠ 2 then
      begin philosopher state[k] := 2; V(philosopher semaphore[k]) end;
   access semaphore := 1;
   for j := 0 step 1 until 4 do
      philosopher state[j] := philosopher semaphore[j] := 0;
   parbegin
   ...
   philosopher i: begin L: think;
                           P(access semaphore);
                           philosopher state[i] := 1;
                           go eating (i);
                           V(access semaphore);
                           P(philosopher semaphore[i]);
                           eat;
                           P(access semaphore);
                           philosopher state[i] := 0;
                           go eating ((i − 1) mod 5);
                           go eating ((i + 1) mod 5);
                           V(access semaphore);
                           go to L
                  end;
   ...
   parend
end[19]
```

As one sees, a *philosopher semaphore* is associated with each philosopher. If a philosopher is hungry, and if the conditions in *go eating* are verified, that is, if his two neighbors are not eating, then a **V** operation is performed on his semaphore, and on returning the **P** operation on this very same semaphore will immediately succeed. On the contrary, if a philosopher is hungry but if the conditions in *go eating* are not verified, then the **V** operation will not be performed, and on returning the **P** operation will wait until another philosopher, after having finished to eat, goes back to thinking again, and does a **V** operation on his semaphore. It is evident that with a suitable adaptation of the *go eating* procedure, any other type of mutual exclusion can be achieved between any number of processes.

As a second example, semaphores could also be used to allocate limited resources, for instance a limited number of tape punches, among a number of processes. If each process is not allowed to make use of more than one punch at any given moment, then it is sufficient to initialize the value of a semaphore *punches* to the number of available tape

19. Cf. DIJKSTRA, E. W., *Hierarchical Ordening of Sequential Processes*, p. 22

punches: when a process needs a tape punch, it performs a **P**(*punches*), and when it has finished using it, it performs a **V**(*punches*). If, however, a process is allowed to make use concurrently of more than one punch, then a deadlock could take place: if, for instance, the number of available punches is 8, and if the two processes currently running need at most 6 punches to complete their operations, and if 2 punches have already be allocated to each of the two processes, then the two processes could finish, since 4 more punches are still available and could be allocated to one of the processes. But if one more punch is allocated to each of the two processes, that is, if both now make use of 3 punches, then there are only 2 remaining punches, and if both processes happen to actually need 3 more punches, neither of them will be able to finish, unless of course one of them is aborted. The most easy way to solve this problem is to execute the processes one after the other, but is not optimal regarding the use of the available resources: it would mean that each process virtually uses its maximum demand of punches during the complete time of its execution. To solve this problem without being too restrictive, and nevertheless without leaving the possibility of deadlocks, Dijkstra invents an algorithm, known as the "banker's algorithm." This situation is indeed comparable with that of a banker, which has a certain finite *capital* in a certain unit of money and a number N of customers to which he temporarily borrows some money, which he knows they will pay back sooner or later. Each customer has a given *maximum need* of money, and a *current loan*. It is obviously unsafe for the banker to accept a new customer if its *maximum need* exceeds his *capital*, but he can accept any customer whose *maximum need* is lower than his *capital*. When one of his customers asks for one more unit of money, the banker should check whether it is safe or not for him to lent it. If it is not, that is, if he would run the risk of being later short of money when another customer wants to borrow money, he may ask him to wait, but the customer is guaranteed to get his money later on. To decide if it is safe for him to lend one more unit of money to a customer c, he can use the following algorithm:

```
Boolean procedure safe (c); integer c;
begin
   integer cash, i; integer array maximum claim[1:N];
   Boolean array finish doubtful[1:N];
   current loan[c] := current loan[c] + 1;
   cash := capital;
   for i := 1 step 1 until N do
      cash := cash − current loan[i];
   for i := 1 step 1 until N do
      maximum claim[i] := maximum need[i] − current loan[i];
   for i := 1 step 1 until N do finish doubtful[i] := true;
L: for i := 1 step 1 until N do
   begin if finish doubtful[i] ∧ maximum claim[i] ≤ cash then
         begin finish doubtful[i] := false;
               cash := cash + current loan[i];
               go to L
         end
   end;
```

```
    if cash = capital then safe := true else safe := false;
    current loan[c] := current loan[c] − 1
  end
```
[20]

In order to decide whether a requested unit of money can be borrowed by a customer c, the banker inspects the situation that would arise if he had lend it. This situation is safe if at least one of the customers has a *maximum claim* not exceeding the banker's current *cash*, and if, supposing that this customer has completely returned its *current loan*, the situation would still be safe for all the other customers; in other words, if the banker is assured to finally get all his money back, the whole situation is safe. This algorithm can then be used to solve the problem of the concurrent use of a limited number of punches by a number of processes, which is a second example of a specialized type of mutual exclusion, following more subtle rules than those of a strict mutual exclusion. Before performing a **P**(*punches*), a process should check if it can do this in a safe way: this can be achieved by using an organization similar to that of the five dining philosophers solution.

Those last examples show that the semaphores are indeed undeniably an extremely flexible tool. But they may look as a way to merely postpone the difficulty: the possibility of their implementation has been taken for granted till now, but how the primitives **P** and **V** could be concretely implemented as atomic operations, how the former could be implemented without a busy waiting, and how the latter can wake up one of the processes which is waiting on the increase of its semaphore, is anything but manifest. This will be explained in the next section; it will be shown at the same time how they are, among others, at the basis of the whole construction of the operating system designed and written by Dijkstra and his team for the X8.

§ 2. The "THE" Multiprogramming System

When the acquisition of the X8 is decided, in December 1963, it is still in its conception stage: it is due to be delivered only in the course of 1965. It is a binary machine, almost backwards compatible with the X1; it is planned to be eight times faster, hence its name. It has the same two accumulator registers, an index register and one-bit condition registers, plus an additional two-word accumulator register dedicated to the built-in floating point arithmetic operations. It has a memory of at most 2^{18} words having a length of 27 bits, of which $2^{15} = 32768$ are directly accessible. It has the same instruction set as the X1, extended with instruction variants using the index register for indirect addressing, and a new subroutine call instruction storing its link in the memory address indicated by the index register. Those additional instructions, allowing to use a stack in an efficient way if the index register serves as a stack pointer, make the X8 evidently more suitable than the X1 to execute ALGOL programs.

It can be extended with at most forty peripheral equipments: drum or disk secondary

20. Cf. Dijkstra, E. W., *Cooperating Sequential Processes*, p. 77

storage, command teleprinters, line printers, tape or card readers and punches, plotters, and magnetic tape units. It can also be equipped with a time and interval clock. Those peripherals are controlled by an independent processor, CHARON, which takes care independently, but under control of the main processor, of the details of the data transfers between the peripherals and the memory. The X8 installed at the University of Eindhoven will have a memory of 32768 words, a drum of about 2^{19} words, a clock, three tape readers, three tape punches, a command teleprinter, a plotter, and a line printer. Its memory will be later extended by 16384 words, and three magnetic tape unit, as well as a second command teleprinter, will be added.

The X8 is again revolutionary in one of its aspects, namely its interrupt system, which improves the X1's interrupt organization in two ways. On the one hand, on the X1, the interrupt signals sent by the peripherals were grouped under seven interrupt classes (see p. 12), which raised seven distinct interrupts, and it was possible, by means of the setting of an interrupt permit bit for each class, to disable the interrupts of one or more classes. This makes it possible to react with some flexibility to the interrupt signals, but the grouping of the communication apparatus under seven classes is to a certain degree arbitrary: it could be desirable for instance to disable the interrupt signals of certain apparatus in a class, but not the others. Of course, each apparatus still has its own interrupt signal, but the notion of interrupt class is abandoned on the X8, which means that only a single interrupt can take place, and instead of having an interrupt permit bit for each class, it has one for each apparatus. As it was the case on the X1, an additional global interrupt permit bit makes it possible to prevent all interrupts without altering the individual permit bits of the peripherals.

On the other hand, on the X1, the interrupt signals sent by the peripherals are basically a single bit of information, signaling that the apparatus has completed the operation it was asked to perform. This is of course enough if, as it was the case for the X1's peripherals, they can only be asked for a single operation at a time. For efficiency reasons, it may be desirable to allow the computer to ask for more than one operation at a time: if it has, for instance, to read a number of tracks which follow each other on a drum, and if it asks for the reading of one track at a time, then even if it quickly reacts to the interrupt following the completion of the previous reading operation by sending the next reading command, the reading head of the drum may already be further than the beginning of the next track, and the drum will have to wait during a complete rotation before actually beginning the next reading operation. This is the main motivation behind the addition of CHARON: instead of single commands, it accepts queues of commands, which it transmits one at a time to the peripherals. Since its operations are extremely simple, they are not coded as programs, but hard-wired: it can thus react instantaneously to the completion of an operation by sending the next pending command, if there is one in the queue. Further, since the operations of CHARON are entirely transparent for the programmer, one may totally make abstraction of its existence.

With such an organization, signaling the completion of the communication operations

with a single bit interrupt signal is obviously still possible: it is sufficient to let the interrupt handler know that the execution of one or more commands sent to the apparatus corresponding to the interrupt signal have been completed. But it is then entirely the responsibility of the computer to determine how much and which ones of the commands it has sent have been completed; when the number of apparatus and their peculiarities augment, this will become more and more difficult. Observing that the commands are sent one at a time by a program to a peripheral, and that the peripheral executes them one at a time leads naturally to consider the relation between the computer and a peripheral as a relation between a producer and a consumer: the computer produces communication commands which are consumed one at a time by the peripheral. Since the easiest way to control the relation between two such processes is to use two semaphores (see p. 65), one that is raised by the producer and lowered by the consumer, and the other one the other way around, the X8's interrupt system will be entirely organized around semaphores: each peripheral will be equipped with two so-called "hardware semaphores", called *action* and *interruption*, the former being increased by the computer and lowered by the peripheral, the latter being increased by the peripheral and lowered by the computer. If we make abstraction of the details, the cyclical program process running on the computer and the cyclical process of a peripheral with which it communicates can be represented as follows:

```
begin semaphore action, interruption;
  action := 0; interruption := 1;
  parbegin
  program:    begin A: ...
                       prepare communication command;
                       P(interruption);
                       send communication command;
                       V(action);
                       ...
                       go to A
              end;
  peripheral: begin B: P(action)
                       execute communication command;
                       V(interruption);
                       go to B
              end
  parend
end[21]
```

Raising the semaphore *action* has the effect that the peripheral is waked up, and that it executes the command just send by the program; raising the semaphore *interruption* has the effect that the program is waked up and can continue by sending the next command. This simplified example obviously does not make use of the queuing facility formerly described: only a single command will be sent to the peripheral at a time. But it is of course possible to have a program with a more complex organization, performing multiple

21. Cf. DIJKSTRA, E. W., *Multiprogrammering en de X8*, p. 4 and DIJKSTRA, E. W., *Documentatie over de communi` catieapparatuur aan de X8*, pp. 6–7

V(*action*) operations before waiting on a **P**(*interruption*) operation: the semaphore *action* will then indicate the length of the queue, and the semaphore *interruption* the number of unacknowledged completions. All this would be possible without the use of semaphores, but it is manifest that they greatly simplify the overall organization. As one sees, the two semaphores are used in a perfectly symmetric way: **P**(*action*) suspends the peripheral, **V**(*action*) wakes it up, **P**(*interruption*) suspends the program, and **V**(*interruption*) wakes it up. Their names are given from the point of view of the program, but could have been given the other way around, from the point of view of the peripheral: its cycle is interrupted on the semaphore *action* and by raising the semaphore *interruption* it provokes an action of the computer.

It is for that system, whose design will be finalized only in June 1966, that Dijkstra and his team of five half-time people, C. Bron, A. N. Habermann, F. J. A. Hendriks, C. Ligtmans, and P. A. Voorhoeve, will design and write the THE operating system.

An operating system can be, and often is, implemented as a set of subroutines sharing a set of variables, organized for instance in tables or lists, whose function is to keep a record of the state of the running processes, namely the running programs and the peripherals. Those subroutines are entered either when a program needs to communicate with a peripheral, or conversely when a peripheral needs to signal the completion of an operation to one of the programs by means of an interrupt. Those subroutines usually ensure that there is never more than one program communicating with each peripheral. Further, they usually give the programmers easier ways to communicate with the peripherals, for instance by hiding the details of the communication orders behind higher level primitives. Roughly speaking, this is for example how the X1's communication program, which is a kind of minimal operating system, is organized (see pp. 11–12).

For reasons which will become apparent later on, this is not the arrangement chosen for the THE. Its organization is instead based on the previously found principle, that is, the semaphore:

> We have arranged the whole system as a society of sequential processes, progressing with undefined speed ratios. To each user program [...] corresponds a sequential process, to each input peripheral corresponds a sequential process (buffering input streams in synchronism with the execution of the input commands), to each output peripheral corresponds a sequential process (unbuffering output streams in synchronism with the execution of the output commands); furthermore, we have the "segment controller" associated with the drum and the "message interpreter" associated with the console keyboard. This enabled us to design the whole system in terms of these abstract "sequential processes." Their harmonious cooperation is regulated by means of explicit mutual synchronization statements.[22]

The complete system is thus composed of twenty-five cyclical processes: ten "concrete" processes, namely, the peripherals, and fifteen "abstract" processes, namely, the programmed processes. To each peripheral device corresponds one programmed process,

22. Dijkstra, E. W., *The Structure of the "THE" Multiprogramming System*, p. 343

and five additional programmed processes act as slots into which user programs can be inserted. All those processes cooperate to the global functioning of the system, they are all synchronized with each other exclusively with semaphores, and a twenty-sixth process, the operator, controls the whole system. The building of the concrete processes is obviously Electrologica's task, and the writing of the operating system thus amounts to write the fifteen abstract processes.

In comparison with the more conventional undifferentiated arrangement, this division into ten abstract processes representing and controlling the ten concrete peripheral processes and five abstract processes for the user programs evidently contributes to a simplification of the task. Breaking the tasks of the operating system into separate processes, each one having a clearly defined task, is indeed somehow like breaking the operations of a program into procedures: it will probably significantly lower the complexity of the resulting system. But it is possible to go a step further, and consequently to put more structure in the system, by observing that those fifteen processes are logically not equivalent. The user program processes for example are independent of each other, and will never have to communicate with each other. During the execution of a user program, they will however communicate with the peripheral processes, but the converse is not true: a peripheral process will never take the initiative to communicate with a user program. Likewise, the processes representing the input and output peripherals will make use of the process representing the drum for their buffering activities, but the latter has no reason to make use of the formers. It is thus possible to arrange those fifteen processes hierarchically:

> The total system admits a strict hierarchical structure. At level 0 we find the responsibility for processor allocation. [...] At level 1 we find the ["segment controller" associated with the drum]. [...] At level 2 we find the "message interpreter" taking care of the allocation of the console keyboard via which conversations between the operator and any of the higher level processes can be carried out. [...] At level 3 we find the sequential processes associated with buffering of input streams and unbuffering of output streams. [...] At level 4 we find the independent user programs, and at level 5 the operator (not implemented by us).[23]

Of those five levels, levels 0 et 1 are of particular interest, since they are the foundation of the whole system. The nine processes in levels 2 and 3 control communication devices having different characteristics, but function according to the same principles. Finally, level 4 comprises five identical processes.

Level 0 is the so-called "coordinator". It has the "responsibility for processor allocation," which means that it allocates the processor in turn to one of the abstract processes. A first idea would be to slice the time, and to allocate a portion of it in turn to each of the processes. If a process performs a **P** operation that cannot be completed because the current value of the semaphore equals zero, it stops and signals to the coordinator that the rest of the slice can be allocated to the next process. When the processor is allocated to a process which stopped on an uncompleted **P** operation, it can try anew

23. Dijkstra, E. W., *The Structure of the "THE" Multiprogramming System*, p. 343

to complete it, and the rest of the slice is again allocated to the next process if it still cannot be completed. This is, however, not very efficient: the computer could well spend a significant part of its time in trying and failing again to finish previously uncompleted **P** operations.

If a process is waiting on the completion of one of its **P** operations on a given semaphore, it is obviously pointless to allocate the processor to it, until another process has performed a **V** operation on this very same semaphore. It is therefore better to let the coordinator keep a list of the processes, separated in two, the runnable and the blocked ones, and to allocate the processor in turn only to the runnable ones. This list will possibly be updated only when a process performs a **P** or a **V** operation: the best solution is then to implement those operations in the coordinator, in the form of two subroutines.

The coordinator is thus entered whenever a process performs an operation on a semaphore. When a process performs a **P** operation, this can have an effect on that very process, but cannot have an effect on the other processes: either the current value of the semaphore is positive, and the operation immediately succeeds, or it equals zero, and the process is then moved from the list of the runnable processes to the list of blocked processes. Conversely, when a process performs a **V** operation, this has no effect on itself, but can have an effect on the other processes: either the current value of the semaphore is positive, and the operation immediately succeeds, since no other processes are waiting on this semaphore, or it equals zero, which implies that a number of processes may be waiting on it; in the latter case, if there is at least one process waiting on it, one of them is moved from the list of the blocked processes to the list of the runnable ones, and the semaphore is not increased, since this move means that the selected process has completed its **P** operation. Finally, at the end of the **P** and **V** subroutines, the coordinator allocates the processor to one of the processes in the list of the runnable ones. How the **P** operation can be implemented without a busy waiting is now manifest: it simply means that the coordinator does not allocate the processor to the blocked processes. And how a **V** operation can wake up one of the processes which is waiting on the increase of its semaphore is now also evident: it simply means that the coordinator moves one of the processes waiting on this semaphore from the list of the blocked ones to the list of the runnable ones. But the **P** and **V** operations have a third fundamental property, namely, their atomicity, and if they are implemented in the coordinator as subroutines, then they are not atomic at all, for a subroutine is by definition a series of instructions. It is however possible to achieve this requirement of atomicity simply by making use of the global interrupt permit bit of the X8 (see p. 70), a feature that was already present in the X1 (see p. 12): before calling the **P** or **V** subroutine, the program sets this bit, and at their end it is reset. Since setting this bit and calling a subroutine are two operations which will often follow each other, the X8 even has a specialized subroutine call instruction, which also sets this bit: it will thus be used to call the **P** and **V** subroutines, instead of the ordinary subroutine call instruction.

The general working principle of the coordinator is now sketched, but a few problems

are still open. The first one is that of the time slicing. Indeed, with a coordinator implementing only the **P** and **V** operations and keeping a list of the runnable and blocked processes, it is guaranteed that no computation time will be lost in trying to run blocked processes, but a process could monopolize the processor for a long time, since switching from one process to another only happens at the end of one of the synchronizing primitives. To prevent this situation, an interrupting clock, that is, a clock sending an interruption signal to the computer at regular intervals, is all that is necessary; the X8 is equipped with such a clock. The computer reacts to its signal by setting the global interrupt permit bit and by transferring control to a certain address. A second function of the coordinator is thus to react to the clock interrupts: it simply saves the state of the currently running process, selects another process in the list of the runnable ones, restores it state, and gives it the processor.

A second problem is that a **V** operation on a semaphore may have as side-effect that a process which was waiting on that semaphore is moved from the list of blocked processes to the list of the runnable ones. This simply supposes that the coordinator keeps, in a way or another, a list of the processes waiting on each semaphore. This is relatively easy to achieve: when a process performs a **P** operation on a given semaphore, if it does not succeed, besides being moved from the list of the runnable processes to the list of the blocked ones, it is added to the list of the processes waiting on that semaphore. The **V** operation can then easily select one of those waiting processes and move it back to the list of the runnable processes. These lists will necessarily be very small: each process may of course wait on at most 1 semaphore, and, in a system with n processes, a semaphore may block at most $n-1$ processes.

A third problem, which we did not mention till now, although it was one of the limitations of Dijkstra's general busy waiting solution to the problem of the strict mutual exclusion of a number of critical sections (p. 61), is the requirement of fairness. It basically signifies that each runnable process should be assured that the coordinator will give it the processor in a short period of time. It is obvious that, the coordinator having the full control on both the lists of the runnable and blocked processes, and the lists of processes waiting on each semaphore, it can implement any desired scheduling policy. The most simple policy is to allocate the processor to each process in turn in a given order, and to remove the processes from the semaphore waiting lists in a first in, first out way. But it is also possible, for instance, to order the processes by increasing order of running time, or to give the processes of a lower level priority over the processes of a higher level.

A fourth and last problem is that of the hardware semaphores. Like normal semaphores, hardware semaphores are simply single memory words, be it that they are increased or decreased by both concrete processes and abstract processes. The scheme presented fits perfectly for a number of concurrent abstract processes or programs with a number of normal semaphores, but if they have to cooperate with a number of concurrent concrete processes or peripherals by means of hardware semaphores (see p. 71), since those peripherals cannot call the **P** and **V** synchronizing subroutines, the waiting lists

kept by the coordinator will rapidly become wrong. One may however observe that it is not necessary to put those concrete processes in the waiting lists of the semaphores, since they always work, by construction, with the same two semaphores. One may then note that, since the concrete processes do not need to be stored in the waiting lists, when a concrete process performs a **P** operation on a hardware semaphore, the only effect will be that the semaphores in question will be decreased; in other words, the waiting lists remain untouched. On the contrary, if a concrete process performs a **V** operation on a hardware semaphore whose current value is zero, then this possibly means that an abstract processes waiting on that semaphore should be moved to the list of runnable processes. Since this is only possible by entering the coordinator, the **V** operation performed by a peripheral on a hardware semaphore, that is, the **V**(*interruption*) operation, is implemented with a side-effect: if the resulting value of the semaphore is positive, the peripheral sets its interrupt signal. The X8 is then interrupted, the global interrupt permit bit is set, and control is transferred to the coordinator. By inspecting the interrupt signals, it can determine which peripheral is the cause of the interruption, and eventually move a process waiting on its *interruption* semaphore to the list of the runnable processes. This solution makes use of both the interruption mechanism for the **V** operation, and the atomicity of the access to individual memory words (see p. 58) for the **P** operation. The last function of the coordinator is thus to handle the interruptions, which are raised when the resulting value of a **V** operation on a hardware semaphore by a peripheral is positive.

It is now possible to give the semaphores a complete, precise and operational definition:

> A process [...] that performs the operation "**P**(*sem*)" decreases the value of the semaphore called "*sem*" by 1. If the resulting value of the semaphore concerned is nonnegative, [that] process can continue with the execution of its next statement; if, however, the resulting value is negative, [that] process is stopped and booked on a waiting list associated with the semaphore concerned. Until [...] a **V**-operation [is performed] on this very same semaphore, dynamic progress of [that] process is not logically permissible and no processor will be allocated to it. [...]
>
> A process [...] that performs the operation "**V**(*sem*)" increases the value of the semaphore called "*sem*" by 1. If the resulting value of the semaphore concerned is positive, the **V**-operation in question has no further effect; if, however, the resulting value of the semaphore concerned is nonpositive, one of the processes booked on its waiting list is removed from this waiting list, i. e. its dynamic progress is again logically permissible and in due time a processor will be allocated to it.[24]

This definition is also more general than the previous one (p. 64), in that it allows semaphores to have negative values: a semaphore with a negative value records the number of processes booked on its waiting list. The previous paragraphs can easily be adapted to that new definition.

The level 0, the coordinator, has thus three main functions to accomplish the processor allocation: it implements the **P** and **V** operations, it handles the periodical clock interrupts, and it handles the interrupts raised by the peripherals. By keeping a list of the

24. DIJKSTRA, E. W., *The Structure of the "THE" Multiprogramming System*, p. 345

processes separated in two, the runnable and the blocked ones, it can allocate the processor in turn to each of the runnable processes. Finally, since it is exclusively entered by subroutine calls and interrupts, it is not a process: it is thus an interrupt handler. This implies that above level 0, all the interrupts have disappeared: the fifteen processes in levels 1 to 4 are synchronized with each other and with the ten peripherals exclusively by means of the synchronizing **P** and **V** operations.

Level 1 is the so-called “segment controller”, which has the responsibility to manage the available memory for the benefit of the other processes. When a computer executes a single process, this process has the whole memory at its disposal, and it has a complete freedom to arrange it according to its needs. A part of it is usually be reserved for the program instructions, and the rest is used to store the data. The programmer has the duty to organize this part of the memory, but this will rapidly become an extremely laborious task, since the location if he has to do so exclusively with simple load and store instructions, for the sizes and life times of the various data structures will usually vary from one execution to another, depending for instance on the input data. It is therefore desirable to place some higher-level primitives at his disposal to simplify this task. This is evidently all the more true when multiple processes are executed on a computer, since they have then to share the memory for both their instructions and data, in such a way that they don’t interfere with each other. The most simple way to do this would be to allocate a fixed portion of the memory to each of the running processes, and to let each of them manage its own portion. This solution has, however, two serious drawbacks. On the one hand, it is then necessary to allocate to each running process its maximum need of memory at the beginning of its execution. Even if one supposes that it is possible to determine that maximum need, it means that each process effectively occupies its maximum need of memory during the complete time of its execution, which implies that the memory will not be used efficiently most of the time. On the other hand, this will not by itself address the problem of the mutual protection of the processes: it supposes that each process behaves correctly, since nothing prohibits it to access and modify the memory reserved for another process. Hence the need for a more general and flexible solution.

The segment controller solves these problems by abstracting the memory space in so-called “segments”: it is divided into pages of 512 words, and each of those pages is occupied by zero or one segment. When a process needs more memory space, it asks for one more segment to the segment controller. If there are remaining free pages, one of them is chosen, its occupation and location is recorded, and the segment controller returns a so-called “segment variable” identifying that segment to the process. When a process needs to read or write a given word in a segment, it transmits that segment variable, together with the word index, to the segment controller, which translates these information into a physical memory address, and transmits it to the process. Finally, when a process does not need a given segment any more, it may ask to the segment controller to release that segment, and the segment controller simply records that the corresponding memory

page is now free. The segment controller thus implements a software virtual memory, by means of three simple primitives: one to request a new segment, one to release a segment, and one to access the data stored in a segment. This arrangement has the advantage of flexibility, since at every moment each running process will only occupy the memory space it actually needs. It has also the advantage of protecting the memory spaces of the processes from each other: if they do not try to directly access the memory directly, that is, if they only access memory words with addresses resulting from the translation done by the segment controller, then they will never access the memory space of another process. Finally, it has the advantage that a process can share data with another process simply by transmitting a segment variable. Likewise, if a process needs to send data to another process, it simply has to transmit a segment variable, and remove it from its own segment table.

But this organization has a further benefit: it allows one to easily implement a so-called "two-level store". The X8 installed at the University of Eindhoven has indeed a drum, which, although its contents cannot be accessed directly by the instructions, can be used as a secondary storage medium. It is much slower, but also much larger, than the core memory, and its contents can also be divided into pages of 512 words:

> We made a strict distinction between memory units (we called them "pages" and had "core pages" and "drum pages") and corresponding information units (for lack of a better word we called them "segments"), a segment just fitting in a page. For segments we created a completely independent identification mechanism in which the number of possible segment identifiers is much larger than the total number of pages in primary and secondary store. The segment identifier gives fast access to a so-called "segment variable" in core whose value denotes whether the segment is still empty or not, and if not empty, in which page (or pages) it can be found.
>
> As a consequence of this approach, if a segment of information, residing in a core page, has to be dumped onto the drum in order to make the core page available for other use, there is no need to return the segment to the same drum page from which it originally came. [...]
>
> A next consequence is the total absence of a drum allocation problem: there is not the slightest reason why, say, a program should occupy consecutive drum pages.[25]

A segment can thus be located in the core, or on the drum: besides a table of the core pages, the segment controller then obviously needs to store a table of the drum pages. The segment requests and segment releases function in the same way, but the segment access primitive needs to be adapted to this new situation. When a program needs to access a word in a given segment, it asks to the segment controller to translate the segment variable and the index into a physical address. If the segment happens to be currently located on the drum, the process is temporarily suspended, the segment controller moves that segment from the drum to the core memory, wakes the process up, and transmits it the physical address of the segment. Since the segments are always accessed through a call to the segment controller, it can keep a record of the access frequency to each of the segments located in core. When it needs a free core page to store a segment, it can then

25. DIJKSTRA, E. W., *The Structure of the "THE" Multiprogramming System*, p. 342

choose the least recently used one to move it the other way around, that is, from the core memory to the drum.

The level 1, the segment controller, is synchronized with a peripheral, namely the drum, and therefore it cannot be implemented simply as a few subroutines like the coordinator: it needs to be a process. To the processes of the higher levels, it presents a large and uniform virtual memory, that is, the distinction between the fast and slow memory is completely hidden behind the scenes: the transports between them take place automatically when they are needed. This implies that above level 1, the actual core and drum pages have disappeared: the fourteen processes in levels 2 to 4 access the memory exclusively by means of segment variables.

Level 2 is the so-called "message interpreter" process, which controls the command teleprinter, and level 3 are the eight processes controlling the eight communication peripherals: the three tape readers, three tape punches, plotter, and line printer. Each one of those processes is synchronized with one peripheral, and each of those peripherals has particular properties, but nevertheless those nine processes all function according to a few common principles.

The message interpreter has the responsibility to ensure that at any moment at most one conversation takes place between the operator and all the other processes. Since the operator has the possibility, to postpone an answer at any moment in a conversation and to switch to another conversation, it also has the responsibility to keep a record of the currently suspended conversations. The eight peripheral processes have the responsibility of buffering the input streams and unbuffering the output streams. When multiple processes are executed on a computer, it is indeed necessary to buffer the input and output streams, but the reason for this necessity is the inverse of the one which led to it on a computer executing a single process: in the latter case, the main motivation for buffering it that the speed of the peripherals is much slower than the speed of the program, in the former case, it is because the peripherals are scarce resources, which have to be liberated as fast as possible. Without buffering facilities, a process may monopolize a peripheral, the single plotter for instance, for a too long time, thereby hindering the other processes who need to use that plotter to proceed. But the peripheral processes have to achieve this without using too much of the available memory space.

Those refined constraints, namely "one conversation at a time," or "without using too much of the memory space," can be implemented with the notion of "private semaphore" and the notion of states which are the basis of the solution to the five dining philosophers problem (see p. 67). The principle of the solution is to introduce at least three states for each process regarding its use of a given resource: it does not use it, it needs to use it, it uses it; if need be, one may of course distinguish multiple states instead of those three. When a process requests a resource, for instance, the attention of the operator, or an additional segment, instead of directly trying to get it, it first records in its *process state* that it needs that resource, and then tries to get it. If a number of *further conditions*

are satisfied, the resource is allocated to that process, and a **V** operation is performed on its *private semaphore*. On returning from the *try to get resource* procedure, the **P** operation on that very same *private semaphore* immediately succeeds. Conversely, when a process releases a resource, instead of simply returning it, it first records in its *process state* that it does not use it any more, and then gives a chance to the other processes to get that resource by calling the *try to get resource* procedure for each of them. If one of those processes now fulfills the *further conditions*, the resource will be allocated to it, and a **V** operation will be performed on its *private semaphore*, which will allow it to proceed. If we make abstraction of the details, the general scheme of the solution reads:

```
begin integer j; semaphore access semaphore;
   integer array process state[0:N];
   semaphore array private semaphore[0:N];
   access semaphore := 1;
   for j := 0 step 1 until N do
      begin process state[j] := do not use resource;
            private semaphore[j] := 0 end;
   procedure try to get resource (k); integer k;
   if process state[k] = need resource ∧ further conditions
   then begin process state[k] := use resource;
              get resource;
              V(private semaphore[k])
        end;
   procedure request resource (k); integer k;
   begin P(access semaphore);
         process state[k] := need resource;
         try to get resource (k);
         V(access semaphore);
         P(private semaphore[k])
   end;
   procedure release resource (k); integer k;
   begin P(access semaphore);
         process state[k] := do not use resource;
         return resource;
         for j := 0 step 1 until N do try to get resource(j);
         V(access semaphore)
   end;
   parbegin
   ...
   process i: begin ...
                    request resource (i);
                    ...
                    release resource (i);
                    ...
              end;
   ...
   parend
end[26]
```

As one sees, a global *access semaphore* ensures that the common *process state* variables

26. Cf. Dijkstra, E. W., *Cooperating Sequential Processes*, pp. 65–66

are not accessed and modified by more than one process at a time. A private semaphore is thus a semaphore that on which other processes will only perform **V** operations.

By the introduction of suitable states, and appropriate programming of the *further conditions*, any resource allocation strategy can be implemented. The "without using too much of the memory space" constraint for the buffers of the eight communication processes for instance is implemented as a series of inequalities, which ensure that among the total number of available segments, 256 are always reserved for those buffers, and that at least 64 of them are reserved for the output buffers. When a process needs one more segment, it records this in its state, and checks if those inequalities are satisfied. If so, then it actually asks for a new segment; if not, it is suspended, and one of the other processes will sooner or later free a segment, and check if other processes are currently waiting for a new segment. Likewise, the "one conversation at a time" constraint for the communication between the processes and the command teleprinter is implemented as a series of states: at any moment each process is out of a conversation, or wants to enter a conversation, or is at a certain stage in a conversation and waiting for an answer from the operator.

The levels 2 and 3, namely the nine processes synchronized with the communication peripherals, present to the processes of level 4 a set of virtual peripherals, which means that above level 3, the actual peripherals have disappeared: they operate behind the scenes. When a process at level 4 asks for the reading of the next character on a tape for instance, the actual tape might already have been entirely read and stored in a buffer, and the tape readers may be busy reading other tapes. Likewise, when a process at level 4 punch data on a tape, or make use of the plotter, usually this does not actually happen while the program is running: the data for the tape or the instructions for the plotter are accumulated in a buffer, and when the tape or the picture is completed, the actual output can take place at full speed.

Level 4 is a set of the five identical processes, which act as slots into which user programs can be inserted. The operator inserts a tape containing an ALGOL 60 program in one of the tape readers, and signals this to one of the cyclical processes at level 4, which reads the tape, and calls the integrated load-and-go ALGOL 60 compiler. When the execution of the program is completed, the program process waits again for a program tape. Each program has a complete virtual machine at its disposal, with a virtual processor and a virtual memory, one virtual command teleprinter, two virtual tape readers, two virtual tape punches, one virtual plotter and one virtual line printer.

At this level a last resource allocation strategy is implemented, again by means of private semaphores. When a virtual communication device is used by a program, it will at some moment and during a certain time correspond to an actual communication device; this is automatically achieved by the underlying program process. Since the memory space is limited, a program process may be forced to begin to use an actual communication device before the output tape or the plotter picture has been completed by the program,

in order to free some memory space. In other words, it may happen that there is not enough free space to store a complete output document before actually starting its output at full speed. If this happens with a document to be output on the line printer, it is not really a problem, since in such a situation a number of pages may be typed out by a program in between the pages of another program. But if this happens with a tape or a picture, then a tape punch or a plotter has to be tied by a program process, and it will be free again only when the program has finished its output. Contrary to a line printer, a tape punch and a plotter thus have the property that, when they are used by a program, they cannot be liberated quickly for the benefit of another program in an urgent situation. This may lead to deadlocks (see p. 67), but it has already been shown that the banker's algorithm can be used to avoid them. The *further conditions* in the private semaphore scheme (see p. 80) is simply that allocating the output peripheral to the program process is safe according to the banker's algorithm. The algorithm given (p. 68) only works with a single type of resource, but has been extended by Habermann to multiple "currencies" or resources in his PhD thesis.[27]

To the user, the complete system presents a set of five identical ALGOL 60 virtual machines, with which he can communicate with a command teleprinter. ALGOL 60 programs are feed in by means of punched tapes, which have to begin with the symbol '**algol**', followed by the maximum number punches and plotters that the program may use concurrently. A few other commands are implemented by the system, to help the operator edit punched tapes for instance.

The design of the THE operating system started in 1963. It was made hand in hand with the design of the X8, which will be finalized only in June 1966: a committee led by van Wijngaarden, in which each one of the interested parties was represented, met about twice a month during those three years. An X8 is delivered to the University of Eindhoven in August 1966. The implementation of the THE begins immediately, and is completed in June 1967. The system comprises, besides the five layers, an ALGOL 60 compiler written by F. E. J. Kruseman Aretz at the Mathematical Center, producing an object code for an abstract machine, a number of subroutines extending the X8's instruction set to support the execution of that object code, and a common library of reentrant procedures at the disposal of the ALGOL 60 programs. The system and the compiler are entirely written in assembly language; the system occupies about ten thousand words, and the compiler about four thousand words.

The purpose of the THE operating system is thus not merely to share a number of resources amongst a number of processes that are executed concurrently, but to abstract the machine and its peripherals, in order to simplify the task of the programmer as much as possible. Its design is, moreover, extremely flexible: adding a new peripheral to the system simply amounts to adding an additional process to the system, synchronized with

27. Cf. Habermann, A. N., *On the Harmonious Co-Operation of Abstract Machines*, pp. 103–106

the peripheral by its semaphore. Likewise, adding a new processor, adding memory, or adding a new drum simply requires an adaptation of, respectively, the layers 0 and 1.

§ 3. Thoughts on Programming

The design and writing of an operating system is obviously a task an order of magnitude more difficult than the writing of an ALGOL 60 compiler. The design of a compiler is indeed a task which is at least partly well-defined: its result, namely executing, according to the semantics of ALGOL 60's constructions, any input program that conforms to ALGOL 60's syntactic rules, is well-defined, it is only the way to achieve it that is not. On the contrary, in designing an operating system, neither the result nor the way to achieve it are manifest: it aims at processing a flow of user programs as efficiently as possible, but this vague requirement leaves room for a large variety of interpretations, and each of those possible interpretations can probably in turn give rise to a variety of possible implementations. This new experience therefore has a strong influence on the way Dijkstra views the programming activity. He more or less abandons his reflections on the nature of programming languages, and his thoughts now focus on the nature of programs — and the nature of the programming activity.

Regarding the nature of programs, a new structuring concept makes its appearance, under the influence of multiprogramming: the notion of a sequential process. This does not mean, of course, that the notion of procedure as a structuring concept to be used when composing individual programs is abandoned in favor of a more general concept, but it means that the notion of a complete program, which was hitherto implicit, has now its own importance and needs a careful analysis. In a multiprogramming system, a number of programs may indeed be designed to cooperate for the solving of a problem, very much in the same way in which, in a program, a number of procedures cooperate for the solving of a problem, but with the difference that the individual procedures are executed by a program in sequence, whereas in a multiprogramming system the individual programs are executed simultaneously. In order to make an effective use of the concurrent execution of a number of programs, one first has to understand their nature as clearly as possible. A sequential process is first defined by comparison with a non-sequential process, for instance an electronic circuit which, fed with two series of bits representing two numbers, calculates their sum:

> Our problem field proper is the cooperation between two or more sequential processes. Before we can enter this field, however, we have to know quite clearly what we call "a sequential process". [...]
>
> [In a non-sequential process the operations are executed simultaneously. In a sequential process the operations have to be executed] in *sequence*, the one after the other. The existence of such an order relation is the distinctive feature of a "sequential process". [...]
>
> [A sequential process can operate on any number of input data], whereas [a] non-sequential [process can] only be build for a previously well-defined number of [input data]. [The operations of a sequential process are mapped] in time instead of in space,

> and if we wish to compare the two methods, it is as if the sequential [process] "extends itself" in terms of [a non-sequential process] as the need arises.[28]

An electronic circuit performing an addition is indeed typically composed of a number of n smaller circuits which perform an addition on a single bit, which will all operate in parallel, in a single step, but its input data is strictly limited, in the sense that it cannot add numbers whose representation need more than n bits. On the contrary, a sequential process would perform this addition in sequence, in n successive steps, and the sum of its possible executions will cover all the possible paths taken by the bits in the electronic circuit, but it can then also accept any number whose representation needs more than n bits.

As one sees, a process is essentially defined by the fact that it executes its operations in sequence. This implies that the notion of a sequential process is not the same as the notion of a program: a program is fundamentally a static reality, while a process represents the dynamic execution of such a program. Simply trying to get a better understanding of the nature of programs would have been inadequate, since the problem at stake is, strictly speaking, not to let a number of programs cooperate, but to let their dynamic execution cooperate. It is thus less the nature of programs that has to be better understood, than the nature of their execution.

With the help of this distinction between a program and a process, that is, between a static program text and a dynamic program execution, another fundamental property of processes, besides being sequences of operations, can be identified:

> It is customary to regard a sequential process as a time succession of actions, each of which has a well defined effect. The total effect of the process is then the cumulative effect of the successive actions. [...] Disregarding input and output we can describe the effect of each action as a "change in the current state of the process", the current state at the end of the action — and thereby the effect — being a unique function of the state at the beginning of the action. [...] As long as we regard a sequential process as a line of actions, leading from one state to the other, then the current state is only defined in between those actions. During an action the current state is undefined, or better perhaps: the whole concept of "current state" is inapplicable.[29]

The very reason of the existence of a process is indeed to perform a global operation, that is, to perform a series of actions in order to reach, starting from a certain initial state, a certain final state. Since a sequential process performs those actions "in sequence, the one after the other," they do not overlap in time, and between each one of them, it is thus in a certain intermediate state somewhere in the middle of those two extreme states. Since the notion of action or operation is not necessarily restricted to the primitive operations of the computer, a certain operation can in turn be composed of a number of operations, and the notion of an intermediate state is therefore a flexible one: if one only considers the global operation of the process, then there are no intermediate states, and if one analyzes that global operation into a number of sub-operations, then this gives rise to a

28. DIJKSTRA, E. W., *Cooperating Sequential Processes*, pp. 1, 4, 8

29. DIJKSTRA, E. W., *An Effort Towards Structuring of Programmed Processes*, pp. 1–2

finite number of intermediate states. This notion of an intermediate state between two operations can be used both to compose programs, and to reason about them.

It can be used to compose programs when they are meant to be executed concurrently, that is, when multiple programs are meant to be executed simultaneously on a computer with a single processor, or with multiple processors, or on multiple computers. Indeed, when a process runs independently of any other, the intermediate state does not change between the end of an operation and the beginning of the next one. But this is not true any more when a number of processes are executed concurrently and share a number of common variables, that is, if they share a part of their current state: even if one considers the elementary operations of a given process, its state may have changed between the end of one of its operations and the beginning of the next one. It is thus, logically speaking, not possible that a number of concurrent processes with unknown relative speeds communicate with each other with those common variables if each of them is continuously performing its actions immediately the one after the other: it is necessary to devise a mechanism by which a process can decide to wait for another one if certain conditions are not met, and conversely a mechanism by which a process can signal to another one that it can now proceed when certain conditions are met; this mechanism will necessarily make use of the shared parts of the states, since otherwise a process would have no means to determine if certain conditions are met or not, and conversely a process would have no means to determine which one of the other processes is waiting for those conditions to proceed. In other words, it is necessary to devise a mechanism by which the processes can synchronize with each other. For this mechanism to be effective, two processes should not be allowed to decide to wait or not for another one at the same time, and two processes should not be allowed to decide to signal at the same time to other processes that they can proceed, for this could obviously lead to situations where multiple processes proceed where only one of them should have. This implies that processes can only take such decisions between two such decisions of all the other processes, that is, that these decisions need to be elementary at the level of all of those concurrent processes. The **P** and **V** operations are two such operations. A **P** operation performed by a process delays the execution of its next elementary operation if a condition is not met, and the process delegates to the other processes the responsibility to give it a signal by means of a **V** operation when this condition is met. More precisely, a **P** operation performed by a process either delays, after inspection of the shared parts of the states, the execution of its next elementary operation, and if so, it records in the shared part of the states that it is waiting for a signal of another process to proceed, or it records in the shared parts of the states that it has not delayed its next operation. The **V** operation performed by a process either signals, after inspection of the shared parts of the states, to one of the other processes that it may proceed, or it records in the shared parts of the states that the next process performing a **P** operation should not delay its next operation.

A similar use of these intermediate states to compose programs is that of the private semaphore scheme (see p. 80), in which a process records in the shared parts of the states

that it has delayed the execution of its next operation because it needs a certain resource, which it cannot get because a number of conditions are not met. When those conditions are met, the global state of the processes is in a certain sense unstable, since there is both a process needing a resource under certain conditions and a resource available under those conditions, and another process stabilizes the global state by allocating that resource to that process.

The notion of an intermediate state between two operations can also be used to reason about the programs:

> I have the feeling that for the human mind it is just terribly hard to think in terms of processing evolving in time and that our greatest aid in controlling them is by attaching meanings to the values of identified quantities. For instance, in the program section
>
> $i := 10$;
> L: $x := sqrt\ (x)$; $i := i - 1$;
> **if** $i > 0$ **then go to** L
>
> we conclude that the operation "$x := sqrt\ (x)$" is repeated ten times, but I have the impression that we can do so by attaching to "i" the meaning of "the number of times that the operation "$x := sqrt\ (x)$" still has to be repeated." [...] But we should be aware of the fact that such a timeless meaning (a statement of fact or relation) is no permanently correct: immediately after the execution of "$x := sqrt\ (x)$" but before that of the subsequent "$i := i - 1$" the value of "i" is "one more than the number of times that the operation "$x := sqrt\ (x)$" still has to be repeated." In other words: we have to specify at what stages of the process such a meaning is applicable. [...]
>
> In purely sequential programming [...] the regions of applicability of such meanings are usually closely connected with places in the program text. [...] In multiprogramming [...] it is a worth-while effort to create such regions of applicability of meaning very consciously.[30]

It is well-known that it is "terribly hard for the human mind" to understand or to reason about something which is in motion. The human mind is indeed discursive, which means that it cannot grasp something instantaneously, but that it needs a certain time of reflection to understand it. Therefore something in continuous motion has already changed as soon as it has been understood, and the only way to get out of this circle is then to abstract static properties out of this motion. Fortunately, a computing process is not a continuously moving reality, since its operations are separated by static states. The difficulty thus lies elsewhere, in the fact that it is extremely difficult to think in terms of sequences of actions, and easier to think in terms of sequences of states.

If a process is defined by a static text, and if to each of the operations of the text corresponds one and only one operation of the process, then those intermediate states are defined both during the dynamic execution of the process and in the static text. The number of intermediate states between the operations of such a process is thus strictly finite, and it is then relatively easy to determine their properties by inspection of its static text. On the contrary, a program text is usually more general than the processes it defines, that is, it represents a possibly infinite number of processes: to each of the operations

30. Dijkstra, E. W., *Cooperating Sequential Processes*, pp. 82–83

defined in the program text corresponds an indeterminate number of operations in a process that executes that program. In other words, because of the generality of the program texts, there is not a one-to-one mapping between the operations of a program text and the operations of the processes that it defines, and it is thus not easy at all to determine the properties of the intermediate states between the operations of those processes by inspection of the static text of the program:

> It is my impression that one of the most deep-rooted difficulties in programming is to bridge in one's imagination the gap between the (static) program text and its (dynamic) behavior, between the algorithm and the process described by it.[31]

One of the possible ways to bridge this gap is to "attach timeless meanings" to certain variables of the process at certain of its stages. In other words, one tries to determine, by inspection of the static program text, certain properties of a number of intermediate states of the dynamic process. In a single program, the "regions of applicability" of those timeless meanings can be clearly delimited in the program text; when one tries to give such timeless meanings to variables common to a number of concurrent processes, it is necessary to do so with the help of critical sections: outside of a critical section protecting its inspection and its modification by other processes, the value of a common variable, and consequently its meaning, is in general never guaranteed to remain constant, even between two elementary operations. It is thus necessary to modify them in critical sections to ensure that they have coherent values.

During the design phase of the THE operating system, which lasted three years, because the solution sought for is not well-defined, many alternative possibilities are envisioned, for instance, the use of parallel **P** and **V** operations, different orderings of the components of the system, or different ways to organize the available memory, and decisions are taken. The experience of designing such a large system is not without influence on the way Dijkstra views the programming activity.

The central notion thus far is that of clarity of programs: the correctness of programs depends on their clarity, and the elegance of programming languages aims at the writing of clear programs. It is understood as the search to be as easy to understand as possible. Clarity is still a central requirement: for instance, the busy waiting solution to the problem of the strict mutual exclusion of a number of critical sections in concurrent processes (see p. 61) is mainly criticized because of its lack of clarity:

> The solution given [...] gives rise to a rather cumbersome description of the individual processes, in which it is all but transparent that the overall behavior is in accordance with the conceptually so simple requirement of the mutual exclusion. In other words, in some way or another this solution is a tremendous mystification.[32]

It lacks clarity not because it extremely complicated, but because it is much more complicated than the "conceptually simple requirement of the mutual exclusion," and

31. DIJKSTRA, E. W., *An Effort Towards Structuring of Programmed Processes*, p. 0

32. DIJKSTRA, E. W., *Cooperating Sequential Processes*, p. 26

this is a sign that it is probably not as simple as possible, that it is a "mystification." The consequence of this needless complication is that it is hardly adaptable even to only slightly different situations, like the one of cyclical mutual exclusion for instance. Likewise, clarity is also still linked with the correctness of programs:

> The convincing power of the results [of a program] is greatly dependent on the clarity of the program, on the degree in which it reflects the structure of the process to be performed.[33]

A program is indeed "convincing" insofar as it is clear, since a programmer can only convince himself that the program he has written does what he wanted it to do if it is clearly written. As one sees, the notion of structure, which previously appeared sporadically without being explicitly defined (see p. 45 and p. 51), is now directly linked with the notion of clarity. It is again not much more analyzed, which probably means that the notion of structure is understood in its usual sense, namely, the arrangement of the individual parts of the whole program or process with respect to each other. The main difference between the notions of clarity and structure is thus that clarity is a property that concerns the whole program or some of its parts, whereas the notion of structure puts the accent on the relations between the different parts of the program. It then gives a clue as to how to proceed to write a program that is clear: a program composed out of clear individual parts is not necessarily clear itself, for this would mean that in a certain way every program is naturally clear, since they are all composed out of a number of clear elementary operations. In other words, the clarity of a program does not principally depends on the clarity of its individual components, but on the clarity of the relations between them. It is only when those relations are clear that the program becomes as easy to understand as possible. This obviously also holds for a number of programs cooperating for the solving of a problem.

The notion of structure is thus more specific and precise than the notion of clarity, and seems, accordingly, to become the central one: the correctness of programs depends on their structure, since a programmer can only convince himself that the program he has written does what he wanted it to do if it is well structured. The main difficulty of the task of the programmer is then to control the structure of the programs he writes, or to set as much structure as possible in the programs he writes, and finally in the computer itself:

> We know that the von Neumann type machine derives its power and flexibility from the fact that it treats all words in store on the same footing. It is often insufficiently realized that, thereby, it gives the user the duty to impose structure wherever recognizable.[34]

This is, in retrospect, already true for small and medium-sized programs, but it is all the more at the center of the programmer concerns when the problem at stake grows in size:

> I should like to venture the opinion that the larger the project, the more essential the

33. DIJKSTRA, E. W., *Programming Considered as a Human Activity*, p. 11

34. DIJKSTRA, E. W., *Cooperating Sequential Processes*, pp. 83–84

structuring![35]

This gives a better understanding of the goal, but the way to reach it is unapparent, since the programmer's primary tool is a programming language, with which he can choose what the program will do, and in what order, but does not by itself give him the control on the relations between the components of the program. Structure is thus a goal that cannot be reached while the program is actually written, but that should instead be pursued during the conception stage of the program:

> The technique of mastering complexity is known since ancient times: "divide et impera." [...] I assume the programmer's genius matched the difficulty of his problem and assume that he has arrived at a suitable subdivision of the task. He then proceeds in the usual manner in the following stages:
> I) he makes the complete specifications of the individual parts,
> II) he satisfies himself that the total problem is solved provided he had at his disposal program parts meeting the various specifications,
> III) he constructs the individual parts, satisfying the specifications, but independent of one another and the further context in which they will be used.
> Obviously, the construction of such an individual part may again be a task of such a complexity, that inside this part job, a further subdivision is required.[36]

If the solution to a problem is divided into a number of individual parts which can be constructed "independently of one another," it is the sign that those individual parts have clear, well-defined relations with one another, in other words, that the interactions between the individual components are clearly delimited, since otherwise it would obviously not be possible to construct them independently of one another. The "divide et impera" technique seems to provide a good way to attain the goal of structure in programs. It is then worth the effort to try to identify its principle:

> I see the dissection technique as one of the rather basic patterns of human understanding and think is worth-while to try to create circumstances in which it can be most fruitfully applied. The assumption that the programmer has made a suitable subdivision finds its reflection in the possibility to perform the first two stages: the specification of the parts and the verification that they together do the job. Here elegance, accuracy, clarity and a thorough understanding of the problem at hand are prerequisite. But the whole dissection technique relies on something less outspoken, [namely] on what I should like to call "The principle of non-interference." In stage II it is assumed that the correct working of the whole can be established by taking, of the parts, into account their exterior specifications only, and not the particulars of their interior construction. In stage III the principle of non-interference pops up again: here it is assumed that the individual parts can be conceived and constructed independently from one another.[37]

The "principle of non-interference" identifies the precise meaning of the independence of the individual parts of a program: it does not signify that those parts are really independent of each other, since this of course would imply that they are not part of a single program, but it means on the one hand that their specification and their imple-

35. DIJKSTRA, E. W., *The Structure of the "THE" Multiprogramming System*, pp. 344–345

36. DIJKSTRA, E. W., *Programming Considered as a Human Activity*, pp. 3–4

37. DIJKSTRA, E. W., *Programming Considered as a Human Activity*, pp. 4–5

mentation are independent of one another, and on the other hand that their individual implementations can be realized independently of one another. If those two conditions are actually met, then the relations between the components of the program will be clear or, in other words, the program will be effectively structured. And if they are met for the main components of the program and for each one of their sub-components, then the whole program will be structured and easy to understand.

This analysis seems, however, not entirely in accordance with the way in which Dijkstra actually proceeds when he is faced with a problem. For instance, when he has to identify the individual states needed to implement refined constraints for the allocation of scarce resources to a number of processes (see p. 79), he proceeds by trial and error:

> We are faced with [two] problems:
> a) what structure should we give to the [...] processes?
> b) what states should we introduce? [...]
>
> The problem [...] is that the two points just mentioned are interdependent. [...] The conditions under which the meaning of the state variable values should be applicable is only known, when the programs are finished, but we can only make the programs if we known what inspections of and operations on the state variables are to be performed. In my experience one starts with a rough picture of both program and state variables, one then starts to enumerate the different states and then tries to build the programs. Then two different things may happen: either one finds that one has introduced too many states or one finds that [...] one has not introduced enough of them. One modifies the states and the program and with luck and care the design process converges. Usually I found myself content with a working solution and I did not bother to minimize the number of states introduced.[38]

This is all the more surprising as the general scheme of the solution to that problem is already known, which means that it only has to be adapted to the particular problem at hand. The problem is that the internal arrangement of the individual processes and the states, that is, their common variables, is interdependent and not independent, which means that in a certain way the internal arrangements of the processes interfere. It seems that it is thus not always possible to divide a task and to specify its individual parts without knowing how those parts will be constructed.

Likewise, faced with a difficult problem which he does not know how to tackle, instead of trying to divide it and trying to build an abstract specification of the components resulting from this division, he does what he did to design the central algorithm of the ALGOL compiler (see p. 40): he tries to find a concrete situation resembling to the problem at stake, and by comparison with the concrete solution found for that concrete problem, devises an algorithm or an organization to solve his own problem. The semaphores for instance are directly inspired by the railway semaphores, which can be either open or closed; other examples are the banker algorithm, the problem of the five dining philosophers, or the comparison between the processes running on a computer and the clients of a hotel. This is not, as one may think, only a matter of pedagogical presentation: he actually reasons and investigates the possible alternatives in the terms

38. Dijkstra, E. W., *Cooperating Sequential Processes*, pp. 51–52

of those concrete examples. It seems thus that, confronted with a problem, he does not model it in the terms of an abstract specification, but gives it, on the contrary, a concrete expression.

Those examples are, however, not in a complete opposition with the "divide et impera" technique. Indeed, when he proceeds by trial and error, it does not mean that he actually codes and tests the tentative programs on a computer, but that he investigates by inspection of the possible situations if the states and the programs function in every conceivable situation. Likewise, giving a concrete expression to a problem is, in a way, dividing into sub-problems, since when one do so, one supposes that the individual parts in the concrete solution have a counterpart in individual parts of the programmed solution. Further, this also takes place during the conception phase, and not during the implementation phase, which means that those two phases are indeed clearly separated.

A last aspect of the programming activity is that of the proofs, whose usefulness was hitherto questionable. It still is, but with a slight concession:

> If we are interested in systems that really work, we should be able to convince ourselves and anybody else who takes the trouble to doubt, of the correctness of our constructions. In uniprogramming one is already faced with the task or program verification — a task, the difficulty of which is often underestimated — but there one can hope to debug by testing of the actual program. In our case the system will often have to work under unreproducible circumstances and from field tests we can hardly expect any serious help. The duty of verification should concern us right from the start.[39]

Only in the case of concurrent programs, and only because it is not possible to reproduce the situations arising during their execution, which implies that it is not possible to debug them, are proofs necessary to convince the programmer that the program does verify its specification. If a number of processes with unknown relative speeds are executed concurrently, it is indeed both impossible to try each and every possible sequence of events, and highly unlikely that it will be possible to force a given sequence of events to happen again for debugging purposes. So, for example, the busy waiting solution to the problem of the strict mutual exclusion (p. 61) is accompanied with the following proof:

> We start by observing that the solution is safe in the sense that no two [processes] can be in their critical section simultaneously. For the only way to enter its critical section is the performance of the [two statements before the critical section] without jumping back to Li, i. e., finding all other t's **false** after having set its own t to **true**.
>
> The second part of the proof must show that no [deadlock] can occur; i. e., when none of the [processes] is in its critical section, of the [processes] looping (i. e., jumping back to Li) at least one — and therefore exactly one — will be allowed to enter its critical section in due time.
>
> If the *turn*th [process] is not among the looping ones, $p[turn]$ will be **false** and the looping ones will find $turn \neq i$. As a result one or more of them will find [...] $p[turn] =$ **false** and therefore one or more will decide to assign $turn := i$. After the first assignment $turn := i$, $p[turn]$ becomes **true** and no new [processes] can decide again to assign a new value to *turn*. When all decided assignments to *turn* have been performed,

39. DIJKSTRA, E. W., *Cooperating Sequential Processes*, p. 41

> *turn* will point to one of the looping [processes] and will not change its value for the time being, i. e., until $p[turn]$ becomes **false**, namely, until the *turn*th [process] has completed its critical section. As soon as the value of *turn* does not change any more, the *turn*th [process] will wait [...] until all other t's are **false**, but this situation will certainly arise, if not already present, because all the looping ones are force to set their t **false**, as they will find $turn \neq i$. And this, the author believes, completes the proof.[40]

As one sees, this proof does not make use of any other formalism than the ALGOL programming language in which the algorithm is written. It presents the general scheme of the algorithm, and explains how the two requirements of mutual exclusion and absence of deadlock are satisfied, but it is not even complete, since it does not consider the case in which *turn* is not the number of the process which enters its critical section, but the number of the next one, which then loops on Li through the **else** part of the conditional statement. This is in accordance with the way Dijkstra sees the proving activity:

> I [use] the word "proving" in an informal way. I have not defined what formal conditions must be satisfied by a "legal proof" and I do not intend to do so. When I can find a way to discuss [a program] by which I can convince myself — and hopefully anybody else that takes the trouble to doubt! — of the correctness of the overall performance of this aggregate of processes, I am content.[41]

The main purpose of a proof is to convince oneself that the program, or the set of concurrent programs, works correctly: the correctness of a program is thus still ultimately a conviction of the programmer. For this an "informal" proof is often enough, since it gives a feeling of the most important elements of what the corresponding formal proof would be, without bothering oneself with the details or the exceptional cases. Since the correctness proof of the THE operating is outlined even more briefly, a proof even seems to be necessary only when the overall structure is not clear. Its proof only gives the reader a vague feeling that the whole system is indeed correct:

> The sequential processes in the system can all be regarded as cyclic processes in which a certain point can be marked, the so-called "homing position", in which all processes are when the system is at rest.
>
> When a cyclic process leaves its homing position "it accepts a task"; when the task has been performed and not earlier, the process returns to its homing position. Each cyclic process has a specific task processing power (e. g. the execution of a user program or unbuffering a portion of printer output, etc.)
>
> The harmonious cooperation is mainly proved in roughly three stages.
>
> (1) It is proved that although a process performing a task may in so doing generate a finite number tasks for other processes, a single initial task cannot give rise to an infinite number of task generations. The proof is simple as processes can only generate tasks for processes at lower level of the hierarchy so that circularity is excluded. [...]
>
> (2) It is proved that it is impossible that all processes have returned to their homing position while somewhere in the system there is still pending a generated but unaccepted task. (This is proved via instability of the situation just described.)
>
> (3) It is proved that after the acceptance of an initial task all processes eventually will be (again) in their homing position. Each process blocked in the course of task execution

40. DIJKSTRA, E. W., *Solution of a Problem in Concurrent Programming Control*

41. DIJKSTRA, E. W., *Cooperating Sequential Processes*, p. 68

> relies on the other processes for removal of the barrier. Essentially, the proof in question is a demonstration of the absence of "circular waits": process P waiting for process Q waiting for process R waiting for process P.[42]

One may of course object that the proof is presented with all its extensions in Habermann's PhD thesis, and that the interested reader is implicitly referred to it. However, this brief sketch of the proof isn't even part of the original version of the article, and is added only at the request of the ACM editor, which is a sign that it is not viewed as the central argument of the correctness of the system, or in other words, that it only augments the conviction in its correct functioning without being the main argument. On the contrary, what seems to be the main argument, and is discussed in every aspect in the article, is the thorough testing to which the system has been submitted:

> [Our] main contribution to the art of system design [is that we] have found that it is possible to design a refined multiprogramming system in such a way that its logical soundness can be proved a priori and its implementation can admit exhaustive testing. [...] At the time this was written the testing has not yet been completed, but the resulting system is guaranteed to be flawless. When the system is delivered we shall not live in the perpetual fear that a system derailment may still occur in an unlikely situation, such as might result from an unhappy "coincidence" of two or more critical occurrences, for we shall have proved the correctness of the system with a rigor and explicitness that is unusual for the great majority of mathematical proofs.[43]

Those last words may suggest that the correctness of the system, the fact that it is "guaranteed to be flawless," is founded on the proofs developed in Habermann's thesis. However, since he suggests that the proof will be completed only when the system, and its testing, will be completed, it is rather its "exhaustive testing" that should to be considered as the proof itself. This is somewhat surprising, since the possibility of testing the correct behavior of concurrent programs is precisely what was initially denied, because of their mutual behavior is not reproducible. This inherent limitation can however be circumvented by a careful design of the whole system:

> Starting at level 0 the system was tested, each time adding (a portion of) the next level only after the previous level had been thoroughly tested. Each test shot itself contained, on top of the (partial) system to be tested, a number of testing processes with a double function. First, they had to force the system into all the different relevant states; second, they had to verify that the system continued to react according to specification. I shall not deny that the construction of these testing programs has been a major intellectual effort: to convince oneself that one has not overlooked "a relevant state" and to convince oneself that the testing program generate them all is no simple matter. The encouraging thing is that (as far as we know!) it could be done. This fact was one of the happy consequences of the hierarchical structure.[44]

If the system is designed with a hierarchical structure, then it is obviously possible to write and to test it layer by layer. If each one of those layers is small enough and well defined, then the number of "relevant states" of this layer can become extremely small,

42. Dijkstra, E. W., *The Structure of the "THE" Multiprogramming System*, p. 346

43. Dijkstra, E. W., *The Structure of the "THE" Multiprogramming System*, p. 342

44. Dijkstra, E. W., *The Structure of the "THE" Multiprogramming System*, p. 344

and it is then possible to "force the system" into all of them, or in other words, to test each one of those states and each transition from one state to another. The task of testing is "no simple matter," for even with a reduced number of relevant states, one has to "convince oneself" that none of them has been left out, and that the testing program "generates them all," but it becomes possible. It is thus possible to submit a system that is well designed, for instance with a hierarchical structure, to an "exhaustive testing," and to guarantee that it is flawless:

> In testing a general purpose object (be it a piece of hardware, a program, a machine, or a system), one cannot subject it to all possible cases: for a computer this would imply that one feeds it with all possible programs! Therefore one must test it with a set of relevant test cases. [...] It seems to be the designer's responsibility to construct his mechanism in such a way — i. e. so effectively structured — that at each stage of the testing procedure the number of relevant test cases will be so small that he can try them all and that what is being tested will be so perspicuous that he will not have overlooked any situation.[45]

The very possibility of this testing depends on the fact that the program is "effectively structured," namely, that its individual parts have such clear and simple relations with one another that their writing and testing can be completed totally independently of one another. If it is not possible to attain this total independence, then at least one of the components should be implementable without the other ones, and another one with only that first one, and so forth: this is precisely what a hierarchical system proposes. In other words, testing a program is only impossible if one considers the whole problem at once.

The true benefit of a layered structure, be it in a whole hierarchical system, an abstract machine, a library, or a single program, is thus that it simplifies the problem, in that it reduces the number of states to consider, both when writing and when testing the program. The task of the programmer, when he writes a program or a set of programs to solve a given problem, is thus to define new primitives, depending both on the problem he has to solve and on the underlying machine. These primitives will improve that machine, and this can roughly happen in two different ways: in writing a library subroutine, the machine is extended, hopefully with a useful primitive; in writing a virtual machine, the machine is replaced, hopefully with a better one. In both cases, the number of states to consider at the higher level will then be sufficiently low to be tested exhaustively.

The conclusion that can be drawn from those observations is that the concept of structure, which appeared only sporadically in Dijkstra's earlier works, becomes central and seems to replace the notion of clarity, which is a more general one: structure is indeed understood as the clarity of the relations between the individual components of a program. Well-structured programs have the nice property that they admit exhaustive testing, which is obviously the best way to be convinced that the program actually does what it is supposed to do. Proving the correctness of programs is therefore still regarded as a secondary concern.

45. Dijkstra, E. W., *The Structure of the "THE" Multiprogramming System*, p. 344

CONCLUSION

Don't strive for recognition (in whatever form):
recognition should not be your goal,
but a symptom that your work has been worthwhile.
Dijkstra

This concludes our present study. Before closing it, we would like to have a second look at the four limits we had identified in our introduction, in order to examine if we didn't surpass them in one way or another, and to the twofold aim we have set ourself, in order to determine to what degree it has been fulfilled.

The most flagrant limit of our work is that it does not seem to contribute to the progress of the discipline. Indeed, each one of the ideas developed here is already present in Dijkstra's writings, which are publicly available. Moreover, even the method we have used to analyze them is not new: it has been used for example by A. Koyré in his *From the Closed World to the Infinite Universe* to study the evolution of scientific and philosophical thought in the sixteenth and seventeenth centuries. However, on the one hand, most of the ideas we have developed are exclusively present in Dijkstra's writings, and often in a very sketchy way: a hundred page synthesis of the essential ideas spread over about two thousand pages of technical writings seems a valuable achievement. On the other hand, to the best of our knowledge, the method we have used has never been applied before to study the history of computing science, and, contrary to Koyré, we tried to enter as much as possible in the technical details to provide a solid basis for our later observations. Our main contribution to the progress of the discipline is thus that we propose a novel way to study the history of science; we hope to have shown by this example that it is a fruitful one.

A second limit of our work is that it is partial and subjective, in that it is centered around a single author. One may now even object that it is so partial and so subjective that it gives the perhaps misleading impression that Dijkstra invented everything *ex nihilo*. To this we would like to reply with three remarks. Regarding the development of the X1's interrupt handler, it is reckoned that Dijkstra "independently developed [that is, invented] an interrupt system that enabled buffering of input and output in 1957 and

1958."[1] Regarding the writing of the ALGOL 60 compiler, the very fact that the MC compiler was by far the first one to be completed, one year before its closest concurrent, seems to be a serious and objective indication that he indeed invented all the compiling techniques it made use of. Finally, regarding the development of the THE operating system and the invention of the first synchronizing primitives, it is also reckoned that "without the foundations laid by Dijkstra we would still be unable to separate principles from their applications in operating systems,"[2] and that "most multiprogramming concepts evolved during the design of THE multiprogramming system; that is, critical regions, semaphores, deadlock prevention, and hierarchical program design."[3]

A next limit is that our work may give an impression of generality, because it does not cover all the technical details, and because it does not make use of mathematical formulæ. However, on the one hand, without entering into all the technical details, we have made an effort to fill pages with facts and arguments, rather than with vague observations. On the other hand, the total absence of any formal notations in our work is not related to the fact that our aims were to give an insight into some of the core concepts of computing science and to illustrate a way of thinking, which could be interpreted as an way to exclude beforehand any rigorous developments, but to the total absence of any formal notations, other than flowcharts and ALGOL programs, in Dijkstra's writings, at least until 1968.

A last limit is that our work is not exhaustive. Some of Dijkstra's achievements during the two first decades of his professional life are entirely passed over in silence. We believe, however, that nothing essential has been neglected. Further, it does not examine the possible influences to which Dijkstra was exposed during those years, nor the posterity of his discoveries. This could have been done, and it would certainly have been interesting, but it would not have fit in the imposed limits for our work: we believe that we have dealt with as much material as possible.

Our first aim was to gain an insight into some of the central concepts of computing science. If we take stock, we encountered in the course of our work the notions of machine, subroutine, interrupt, block, procedure, recursion, stack, programming language, concurrency, synchronization, semaphore, sequential process, ... as well as the notions of clarity, elegance, abstraction, correctness, structure, testing, independence, proof, ... Each one of these concepts has been studied with great care, and we have seen how and why they can be used to write programs. We believe that we may conclude that our first aim has been entirely fulfilled, far beyond our expectations. We even venture the opinion that our method is not only and not primarily a way to study the history of a discipline, but a way to study the discipline itself.

Our second aim was to illustrate a way of thinking, a way of approaching new prob-

1. Knuth, D. E., *The Art of Computer Programming*, § 1.4.5

2. Brinch Hansen, P., in Hoare, C. A. R., Perrott, R. H. (eds.), *Operating System Techniques*, p. 34

3. Cf. Brinch Hansen, P., *Operating System Principles*, p. 286

lems. We have discovered that rigor can be achieved simply with the use of the natural language, together with the help of carefully chosen concrete images of the problem at stake. In other words, precision can be achieved without the use of formal methods, that is, without formal notations and without formal proofs. This is true both for small and well-defined problems, and for large and loosely defined ones. This is one of the most surprising conclusions of our thesis: contrary to the common belief, Dijkstra was actually opposed to the use of formal methods, at least during the first two decades of his life as a programmer. One may of course object that the use of formal methods was uncommon at that time. This is, however, untrue, especially in Dijkstra's immediate environment: it suffices for instance to have a look at the *Report on the Algorithmic Language ALGOL 68*, established under the direction of his first mentor, van Wijngaarden, or to recall that he was teaching in a mathematical department, to convince oneself that he could not be unaware of the usage of formal methods. It remains of course to be understood why he became a proponent of formal methods, although he had achieved quite advanced projects which were nearly flawless, with very few resources and without any formalism. But for the time being we believe that the ideas, the problems and the solutions presented throughout this work may be of some inspiration, and that our second aim has thus also been fulfilled.

While considering Dijkstra as one of the founding fathers of computing science seems fully justified, we believe that this question, like the one of the paternity of ideas, is relatively unimportant. What is important for science is that it progresses, and not to determine who invented this or that, or to decide who deserves this or that title. Dijkstra wasn't alone when he invented all these solutions: he devised all of them in collaboration with other people, and he never demanded to be considered as their inventor. What is important for science is that we all cooperate with the truth.

BIBLIOGRAPHY

DIJKSTRA, E. W., *Functionele beschrijving van de ARRA*, Mathematisch Centrum, MR12 (1953)

—, *Handboek voor de programmeur — FERTA*, Mathematisch Centrum, MR17 (1955)

—, *Programmering voor de ARMAC*, Mathematisch Centrum, MR25a (1957)

—, *et al.*, *Programmering voor Automatische Rekenmachines*, Mathematisch Centrum, CR9 (1957)

—, *Communication with an Automatic Computer*, PhD Thesis, University of Amsterdam, Excelsior, Rijswijk (1959)

—, *Recursive Programming*, Num. Math. 2, pp. 312–318 (1960)

—, *Making a Translator for ALGOL 60*, ALGOL-Bulletin Supplement 10 (1961)

—, *An ALGOL 60 Translator for the X1*, ALGOL-Bulletin Supplement 10 (1961)

—, *On the Design of Machine Independent Programming Languages*, Mathematisch Centrum, MR34 (1961)

—, *Defense of ALGOL 60*, CACM 4(11), pp. 502–503 (1961)

—, *Operating Experience with ALGOL 60*, Comp. J. 5(2), pp. 125–127 (1962)

—, *A Primer of ALGOL 60 Programming*, Academic Press, London (1962)

—, *An Attempt to Unify the Constituent Concepts of Serial Program Execution*, Mathematisch Centrum, MR46 (1962)

—, *Some Meditations on Advanced Programming*, EWD 32 (1962)

—, *Over de sequentialiteit van procesbeschrijvingen*, EWD 35 (1962)

—, *Multiprogrammering en de X8*, EWD 54 (1963)

—, *Some Comments on the Aims of MIRFAC*, CACM 7(3), p. 190 (1964)

—, *Programming Considered as a Human Activity*, EWD 117 (1965)

—, *Cooperating Sequential Processes*, EWD 123 (1965)

—, *Solution of a Problem in Concurrent Programming Control*, CACM 8(9), p. 569 (1965)

—, *Documentatie over de communicatieapparatuur aan de X8*, EWD 149 (1966)

—, *An Effort Towards Structuring of Programmed Processes*, EWD 198 (1967)

—, *The Structure of the "THE" Multiprogramming System*, CACM 11(5), pp. 341–346 (1968)

—, *Hierarchical Ordening of Sequential Processes*, EWD 310 (1970)

GREEN, J., *et al.*, *Recommendations of the SHARE ALGOL Committee*, CACM 2(10), pp. 25–26 (1960)

HABERMANN, A. N., *On the Harmonious Co-Operation of Abstract Machines*, PhD Thesis, Technological University of Eindhoven (1967)

KRUSEMAN ARETZ, F. E. J., *The Dijkstra-Zonneveld ALGOL 60 compiler for the Electrologica X1*, Centrum voor Wiskunde en Informatica, SEN2 (2003)

LOOPSTRA, B. J., *The X1 Computer*, Comp. J. 2(1), pp. 39-43 (1959)

NAUR, P. (ed.), *ALGOL-Bulletin* 7 (1959)

—, *ALGOL-Bulletin* 8 (1959)

—, *Report on the Algorithmic Language ALGOL 60* (1960)

NAUR, P., *The European Side of the Last Phase of the Development of ALGOL 60*, ACM SIGPLAN Notices 13(8), pp. 15–44 (1978)

SAMELSON, K., BAUER, F. L., *Sequential Formula Translation*, CACM 3(2), pp. 76–83 (1960)