

Process Structuring

J. J. HORNING¹ and B. RANDELL²

The concept of “process” has come to play a central role in many efforts to master the complexity of large computer systems. The purpose of this paper is to discuss useful methods of structuring complex processes, and to relate these to the problems of improving the quality of large computer systems. Two distinct ways of structuring systems are presented, namely, process combination, and process abstraction; these are then used to discuss such topics as concurrency, synchronization, multiprogramming, interpreters, and programmable processors.

This discussion is based on a set of precise definitions for such concepts as “process,” “processor,” “computation,” “combination,” and “abstraction.” The paper relates these definitions to both current research and practical applications, with particular concern for the problems of the performance, reliability, and modifiability of computer systems.

Key words and phrases: sequential process, cooperating processes, asynchronous processes, parallelism, complexity, program, processor, interpreter, hierarchical structures, abstraction, refinement.

CR categories: 4.30, 4.32, 5.24, 6.20.

¹ Associate Professor, Departments of Computer Science and Electrical Engineering and member of the Computer Systems Research Group, University of Toronto, Canada

² Professor of Computing Science, Computing Laboratory, University of Newcastle upon Tyne, England.

Copyright © 1973, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that ACM's copyright notice is given and that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

CONTENTS

1. INTRODUCTION	3
2. PROCESSES AND PROCESSORS: BASIC DEFINITIONS.....	4
3. COMBINATION OF PROCESSES	9
3.1 Types of Combination	9
3.2 Concurrency and Clocking	12
3.3 Coexisting Processes	14
3.4 Process Interaction	15
3.5 Process Switching	18
4. ABSTRACTION AND REFINEMENT	19
4.1 Interpretations and Images.....	19
4.2 Interpreters and Programmable Processors.....	22
4.3 Description of Processes.....	24
4.4 Creation of Processes	25
5. COMBINATION AND ABSTRACTION.....	25
6. APPLICATIONS OF STRUCTURE.....	27
7. ACKNOWLEDGMENTS	29
8. REFERENCES	30
9. INDEX OF DEFINITIONS.....	32

1. INTRODUCTION

The complexity of many current large-scale computer systems, incorporating sophisticated operating systems, is reflected in the difficulties that have been experienced in their design and construction. In fact, even the problem of gaining an understanding of the behavior of an existing complex computer system can be immense. One of the best methods of coping with something that is unmanageably complex is to decompose it into simpler parts. The important question is: What units of decomposition are most appropriate for designing and understanding complex computer systems?

The “subroutine” is the standard unit for decomposing the text of large programs, including operating systems. But this is a static decomposition, while most of difficulties arise from the dynamic structure of computations. On the one hand, what is conceptually a single operation (e.g., output) may invoke a sequence of disparate subroutines, and on the other hand, a single subroutine (e.g., an interrupt handler) may be used for a number of conceptually distinct actions; indeed, in a multiprocessing system the same unit of program text may be simultaneously executed by several different processors for different reasons. A suitable decomposition must clearly reflect this dynamic structure. P. Brinch Hansen [3], E. W. Dijkstra [13], J. H. Saltzer [29], W. M. Turski [33], and others have successfully used “process” as the basic unit in their descriptions of complex computer systems (although that name has not always been used).

The major goal of this paper is to discuss useful methods of structuring complex processes, and to relate these methods to the problems of improving the quality of large computer systems, and our methods of designing and constructing such systems.

We identify two distinct ways of structuring systems. The first is to consider the system as consisting of interacting subsystems, each responsible for some part of the total system’s behavior. Section 3, Combination of Processes, discusses such topics as concurrency, synchronization, cooperation, mutual exclusion, and multiprogramming. The second technique is to represent the system by means of a sequence of models, each conveying details of behavior appropriate to a chosen level of abstraction of its behavior. Section 4, Abstraction and Refinement, deals with levels, interpreters, programs, and programmable processors. As Section 5, Combination and Abstraction shows, these two forms of structuring are not exclusive, and become more powerful when used in conjunction. Finally, Section 6, Applications of Structure, discusses practical benefits that should accrue from well-designed structure, such as improvements in the performance, reliability, and modifiability of systems. These sections are all based on a single conceptual model, which includes both process combination and process abstraction; this model is developed in Section 2.

We have attempted to unify our discussion by means of a coherent set of definitions for the important concepts involved. It has been said that “The concept of process is□□□an abstract one, and may be compared with that of the life of an organism; in neither case is it very easy to pin the concept down in a definition.” [36] Even a casual survey of the literature [e.g., 3, 8-11, 13, 17, 18, 23, 28, 29, 34] shows the lack of any general agreement on a precise definition of “process.” Three distinct aspects of the process concept are emphasized in various definitions. The process concept is used to model:

- 1) The status of a system.

“A *process* is a locus of control within an instruction sequence. That is, a process is that abstract entity which moves through the instructions of a procedure as the procedure is executed by a processor.” [10]

2) Some sequence of status values.

“□□□□ ‘sequential process’ can be considered as a sequence of actions that will be performed when an input tape has been supplied to an abstract machine.” [18]

3) A means of generating a class of such sequences.

“An ALGOL program (block) specifies a sequence of operations on data local to the program, as well as the structure of the data themselves. SIMULA extends ALGOL to include the notion of a collection of such programs, called processes, conceptually operating in parallel.□□The process concept is intended as an aid for decomposing a discrete event system into components, which are separately describable. In general a process has two aspects: it is a data carrier and it will execute actions.” [8]

Although no one of these provides a complete basis for the definition of “process”, each is important to our study, and it is helpful to use different names for each. Thus, we will refer to the “state” of a process, or to a “computation,” which is a sequence of its states, or to the “action function,” which generates its computations. We will discuss each of the concepts in some detail, and then present what we believe to be an adequate and precise definition of “process”.

Since it has not been possible to keep our definitions consistent with all the conflicting definitions in the literature, this paper develops its framework from a set of quite basic definitions, contained in Section 2. The significance of many of these definitions will become more apparent in the later sections. The reader is therefore encouraged to refer back to Section 2 as he reads the rest of the paper.

Our definitions are intended to be precise, but we have introduced formalism only where it seemed to increase clarity and assist understanding. The discussion has been restricted to a few rather basic mathematical concepts, such as sets, sequences, relations, and functions. Examples illustrate both the concepts defined and their notation.

2. PROCESSES AND PROCESSORS: BASIC DEFINITIONS

In this paper we use the concept of “process” as a mathematical tool to explain, predict, and understand the behavior of a class of physical devices exemplified by digital computer systems. These devices (which are viewed as “processors”) are characterized by the fact that their interesting behavior is predictable, and consists of sequences of values of well-defined physical quantities (e.g., voltages on wires, magnetizations of cores, characters printed on paper), which represent the “information” of the system at discrete instants of time. The relation of a process to a processor is similar to the relation of a theory of physics (e.g., the “law of gravitation”) to the objects of that theory (e.g., the motion of planets); the theory is useful to the degree that it provides a sufficiently close approximation to its objects, yet remains understandable. An important difference is that processes are often developed before the processors they model exist. If a process exhibits desirable properties, then it is generally possible to construct a corresponding processor, using the process as a specification.

Our definitions are based on the well-known concepts of state variables, state variable sets, and states [1, 23]. *State variables* are elementary quantities which can assume certain well-defined values. A set of named state variables constitutes a *state variable set*. An assignment

of values to all the variables in a state variable set defines a state of the set; conversely, a state defines a value for each state variable. The set of possible states for a given state variable set is the *state space* of that set. (N. B. The definitions of state and state space are formally equivalent to those used in state machine theory [26]. However, we use the state variables to introduce a structure into states that is absent from the classical theory, but necessary for the definition of combination in Section 3.)

EXAMPLE: Consider the state variable set $V = \{x, y\}$, consisting of two variables named x and y whose values may be any positive integers. If x is assigned the value 2 and y the value 4, this defines the state $(x = 2, y = 4)$; this may be denoted simply by $(2, 4)$ when context makes the respective names of the state variables clear. The state space of this state variable set is the set $S(V) = \{(x = m, y = n) \mid m > 0, n > 0\}$; again, this may be written $\{(m, n) \mid m > 0, n > 0\}$ when context specifies the variable names. Its members are states such as $(2, 2)$, $(2, 4)$, and $(3, 9)$.

EXAMPLE: The state variable set which Bell and Newell [1] associate with the DEC/PDP-8 processor includes state variables such as AC (the accumulator, which contains 12-bit quantities), L (the link bit), PC (the 12-bit program counter), etc. The state space for this set is the set of all possible combinations of values of these variables.

A *computation* in a state space is a sequence of states from that space. The first element of the sequence is its *initial state*, the last (if it is finite), its *final state*.

EXAMPLE: A finite computation in the state space $S(V)$ of the first example above is the sequence $C_1 = \langle (2, 2), (2, 4), (2, 4), (3, 9) \rangle$ for which $(2, 2)$ is the initial state and $(3, 9)$ the final state.

EXAMPLE: An infinite computation in $S(V)$ is the sequence $C_2 = \langle (2, 2^i) \mid i = 1, 2, 3, \dots \rangle$. Its initial state is also $(2, 2)$, but it has no final state.

Computations are central to our study of processes. Particular computations may be specified by a variety of means, two of which have been illustrated in our examples. However, the principal form of specification is in terms of the transitions which occur between states.

An *action* in a state space is a set of assignments of values to some of the variables of its state variable set. If a state is *followed* by an action, the state's *immediate successor* is the new state whose variables all have their old values, except those which have new values explicitly assigned by the action. The *null action* is the empty set (denoted $\{ \}$) and specifies no assignments.

EXAMPLE: If $(2, 2)$ is followed by the action $\{y \leftarrow 4\}$, its immediate successor is $(2, 4)$; if that is followed by the action $\{ \}$, its successor is again $(2, 4)$; if that is followed by the action $\{x \leftarrow 3, y \leftarrow 9\}$ its successor is $(3, 9)$.

EXAMPLE: If $(2, 2^i)$ is followed by the action $\{y \leftarrow 2^{i+1}\}$, its immediate successor is $(2, 2^{i+1})$.

An *action function* in a state space is a mapping from states into actions. We may use an action function to *generate* a computation from an initial state by applying the action function to the initial state (and then to each successive state, as it is obtained) to find the action following that state, and using the action to find the immediate successor state; if, at any point, the action function becomes undefined, the computation terminates.

EXAMPLE: The computation C_2 is generated from the initial state $(2, 2)$ by the function $f_1(x, y) = \{y \square x \cdot y\}$.

An action function is *strictly deterministic* if it is single-valued wherever it is defined³ (e.g., f_1 in the previous example). Null actions have no essential effect on computations except to change their length; since we use the number of actions as our indication of “time”, they merely “slow down” computations. Thus, we will refer to action functions which differ only in the presence or absence of null actions as *temporal variants*. The *standard form* of a class of temporal variants is the member which generates no null actions; if this action function is strictly deterministic, then all members of the class will be called *deterministic*.

EXAMPLE: As shown in Figure 1, the computation C_1 can be generated from the initial state $(2, 2)$ by the action function g_1 where

$$\begin{aligned} g_1(2, 2) &= \{y \square 4\} \\ g_1(2, 4) &= \{ \} \text{ or } \{x \square 3, y \square 9\} \\ &\text{and } g_1 \text{ is undefined elsewhere.} \end{aligned}$$

This action function is a member of the class of temporal variants whose standard form is g_2 , where

$$\begin{aligned} g_2(2, 2) &= \{y \square 4\} \\ g_2(2, 4) &= \{x \square 3, y \square 9\} \end{aligned}$$

From $(2, 2)$, g_2 generates only the computation $C_3 = \langle (2, 2), (2, 4), (3, 9) \rangle$. Since g_2 is strictly deterministic, g_1 is deterministic.

<u>state</u>	<u>action</u>	<u>new state</u>
(2, 2)	$\{y \square 4\}$	(2, 4)
(2, 4)	$\{ \}$	(2, 4)
(2, 4)	$\{x \square 3, y \square 9\}$	(3, 9)
(3, 9)	undefined	

FIG. 1. The generation of a computation by an action function.

A *process* is a triple (S, f, s) , where S is a state space, f is an action function in that space, and s is the subset of S which defines the initial states of the process.⁴ A process generates all the computations generated by its action function from its initial states. It is (*strictly*) *deterministic* if its action function is (strictly) deterministic. (Note that each computation generated by a strictly deterministic process is uniquely determined by its initial state.)⁵ Similarly, processes are *temporal variants* if they differ only in having action functions which are temporal variants.

³ When an action function is multiple-valued it would be appropriate to call it an *action relation* but this distinction does not seem particularly helpful.

⁴ Wegner [35] defines an *information structure model* $M = (I, I^0, F)$, in which I corresponds to our S , I^0 to our s , and F to our f .

⁵ Although Denning [9] uses equivalent definitions of states, actions, and computations, his definition of process relies on the intuitive notion of a program to implicitly define the (strictly deterministic) action function, and restricts the initial state set to a single state. Thus each of his processes generates a single computation, rather than a class of computations. Our form is required for the definition of process combination to be given in Section 3.

We frequently wish to study processes which are related, but not identical. A process is *weakly contained* in another process if all of its computations are also generated by that process, and *strongly contained* if, in addition, wherever its action function is defined, the action function of the containing process has (at least) all the same values. These definitions simply mean that the containing process can do everything that the contained process can, and possibly more. The distinction between weak and strong containment becomes important only in Section 3, where, as a result of a combination, the state variables of a process may assume values which would never be assigned by the process itself.

EXAMPLE: The process $P_1 = (S(V), g_1, s_1)$ where $s_1 = \{(2, 2)\}$, generates the computations C_1 and C_3 (and an infinite number of others). It strongly contains its temporal variant $P_2 = (S(V), g_2, s_1)$, which generates only C_3 .

EXAMPLE: The process $P_3 = (S(V), f_1, s_1)$ is strictly deterministic, and generates only the computation C_2 . It is strongly contained in the process $P_4 = (S(V), f_1, s_2)$, where $s_2 = \{(n, n) \mid n > 0\}$, which generates all computations of the form $\langle (n, n^i) \mid i = 1, 2, 3, \dots \rangle$. However, P_3 is only weakly contained in the process $P_5 = (S(V), f_5, s_2)$, where $f_5(x, y) = \{y \square 2 \cdot y\}$, which generates all computations of the form $\langle (n, 2^i \cdot n) \mid i = 0, 1, 2, \dots \rangle$.

From any action function, we can deduce the corresponding *successor function*, whose value in any state is the immediate successor (or if the action function is multiple-valued, the immediate successors) of that state, as defined by the action function. Conversely, given a successor function, we can infer a corresponding action function⁶ by noting the changes to values of state variables between each state and its immediate successor(s). The two functions can thus be used interchangeably. We generally find action functions the more convenient, but some definitions in Section 4 are more naturally stated in terms of successor functions.

EXAMPLE: The action function g_2 corresponds to the successor function G_2 , where

$$G_2(2, 2) = (2, 4)$$

$$G_2(2, 4) = (3, 9)$$

and G_2 is undefined elsewhere.

We now turn to the concept of “processor” and its relation to that of “process”. A *processor* is a pair (D, I) , where D is a physical “device” (we leave this term undefined) which can be placed in specified initial states, and I is an *interpretation* of its physical status which indicates at what instants of time, and by what means, the device represents successive states. Each sequence of states following from an initial state is a *computation* of the processor.

The definition of “processor” as a pair may seem somewhat artificial, but is necessary to make a distinction between a mere physical device and a processor (see also Section 4.2). Minsky [26] makes the same distinction, using the term “machine” rather than “processor” .

“The term ‘machine’ cannot usefully be defined as a ‘member of [a certain class of physical objects].’ For the decision as to whether something is a machine depends on what that thing is actually *used for*, and not just on its composition or structure. When we talk about a machine we have in mind not only 1) an object of some sort, but 2) an idea of what the object is supposed to do.”

⁶ This function is not unique since a given transition between two states can be caused by different actions. For example, the null action and the action which assigns to every variable its previous value both leave the state unchanged. By convention we choose the function which gives the action specifying values for only those variables whose value changes.

A process, as a mathematical object, is an abstract, timeless entity. Yet we wish to relate processes to processors, to which the time dimension is essential. We assume that the “interesting” behavior of the device is captured through the interpretation at discrete moments of time, and that the effect of time is adequately modelled by the order in which states (or actions) occur. This assumption is generally valid for “digital” devices, but not for many “analog” devices.

We use a process to model a processor by asserting a relation between the set of possible computations of the processor and the set of computations generated by the process. If the two sets are identical, we call the processor an *exact realization* of the process, or, equivalently, we call the process an *exact specification* of the processor.

EXAMPLE: The process P_4 of a preceding example (see page 8) is an exact specification for the processor $p_4 = (D_4, I_4)$ schematically represented in Figure 2, where the rectangles denote registers which are initially set to the same value and then, at discrete times, the value that is interpreted as y is replaced by the output of the multiplier.

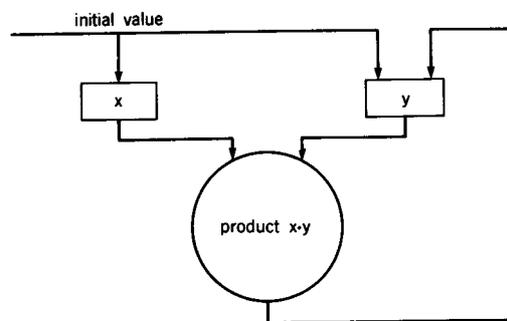


FIG. 2. Schematic representation of the processor p_4 .

More generally, when a process is used as a means of formalizing and/or understanding the behavior, and perhaps the internal structure, of a processor, weaker relations will be acceptable. Thus, if we wish to prove that all computations of a processor have a certain property, we may study a process which is known to generate all computations of the processor (and perhaps more). Conversely, if we need to establish that certain computations are possible for a given processor, we would use a process which generates only (but not necessarily all) computations of that processor.

Different interpretations of a device define different processors, representing different views of the activity of that system. In particular, different interpretations allow us to “subdivide” time as finely or coarsely as we find useful. The choice of a subdivision will determine which changes in the state vector we view as happening “concurrently” and which “sequentially”. It will also determine the number of actions which constitute a given *operation*, i.e., a transition from one given state to some other specified state. There will be no single best interpretation of a complex system, since many different view points are required to provide an adequate understanding of its internal structure and behavior.

EXAMPLE: The CDC 6600 computer [31] can be thought of by the programmer as a high-speed serial CPU with 10 independent PPU’s (peripheral processing units) operating in parallel. However from the logic designer’s viewpoint, the CPU is composed of a number of separate units (e.g., floating add, floating multiply) which operate concurrently, while the 10 PPU’s are implemented by a single set of hardware (which uses the “barrel” to switch among them). Although the 6600

is perhaps an extreme case, there will be similar variations with viewpoint for any multiprogramming or multiprocessing system.

3. COMBINATION OF PROCESSES

3.1 Types of Combination

In studying a complex computer system, it is frequently helpful to view it as a collection of more or less independent processors, each modelled by its own process. Indeed, computer systems are usually put together in just this way. In this section, therefore, we consider processes which can be described as “combinations” of component processes, some of which may themselves be “combinations.” Combination allows us to build processors, whose action functions are extremely complex, from sets of simple processes whose action functions are more easily understood.

The rules governing the combination of processes, of course, depend on the relations among the processors which they model. In the most general case, each processor will act on its state variables (some of which may be shared with other processors) at times which are completely independent of the actions of the other processors. Thus, at any time, the next action of the system may be composed of actions of any number of component processors. Our definition of combination must be adequate for this general case; however, we will first discuss some restricted forms of combination which are more easily structured.

<u>state</u>	<u>action</u>	<u>new state</u>
	<u>by P₅</u>	<u>by P₆</u>
(0, 1)	{x \square 1}	– (1, 1)
(1, 1)	–	{y \square 2} (1, 2)
(1, 2)	{x \square 2}	{y \square 4} (2, 4)
(2, 4)	.	.
.	.	.
.	.	.

FIG. 3. Disjoint combination

Given a collection of processes that have no state variables in common, then the actions of one process can have no effect on any of the others. Thus an action of the combination consists of actions from any or all of the component processes, each determined by its own state variables. More formally, such a *disjoint combination* is the process whose state space is the “direct product” of the component’s state spaces, (i.e., whose state variable set is the union of the components’ state variable sets), and whose initial state set is the “direct product” of the components’ initial state sets (i.e., a state is an initial state of the combination if its components are all initial states of the component processes); the action function of the functions of the combination has, for any state, as its values all unions of values of the action functions of one or more component processes, each applied separately to its own state variables.

EXAMPLE: Let $P_5 = (S_5, f_5, s_5)$ ⁷, where $S_5 = \{x \mid x \geq 0\}$, $f_5(x) = \{x \square x + 1\}$, $s_5 = \{0\}$. Also, let P_6 be defined by $S_6 = \{y \mid y > 0\}$, $f_6(y) = \{y \square 2 \cdot y\}$, $s_6 = \{1\}$. Let

⁷ Henceforth, we will use S_n , I_n , and s_n as the components of P_n without further comment.

$+_d$ denote disjoint combination. Now $P_5 +_d P_6 = P_7$ where $S_7 = \{(x, y) \mid x \geq 0, y > 0\}$,
 $f_7(x, y) = \{x \square x + 1\}$ or
 $\{y \square 2 \cdot y\}$ or
 $\{x \square x + 1, y \square 2 \cdot y\}$,
and $s_7 = \{(0, 1)\}$. As shown in Fig. 3, one computation of P_7 is $\langle(0, 1), (1, 1), (1, 2), (2, 4), \dots\rangle$.

On the other hand, given a collection of processes with the same state spaces⁸, with the property that in each state at most one process is *active* (i.e., its action function is defined and has a non-null value), then the combination performs actions from one process at a time. Such a *serial combination* has the same state space as the components, an action function which is the union of the components' action functions, and an initial state set which is the intersection of the components' state sets. The sequence of actions of a serial combination is an interleaving of actions from the component processes; an interesting special case occurs when processes are in effect "concatenated" because each completes all of its actions before its successor becomes active.

state	action		new state
	by P_8	by P_9	
(0, 2, 1)	$\{x \square 2, z \square 0\}$	–	(2, 2, 0)
(2, 2, 0)	–	$\{x \square 0, z \square 2\}$	(0, 2, 2)
(0, 2, 2)	$\{x \square 2, z \square 1\}$	–	(2, 2, 1)
(2, 2, 1)	$\{x \square 4, z \square 0\}$	–	(4, 2, 0)
(4, 2, 0)	–	$\{x \square 0, z \square 4\}$	(0, 2, 4)
(0, 2, 4)	$\{x \square 2, z \square 3\}$	–	(2, 2, 3)
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮

FIG. 4. Serial combination.

EXAMPLE: Coroutines, as introduced by Conway [7], may be regarded as separate processes, each with its own program counter. Since control is passed among them by explicit calls, only one is active at any time; thus the processes are serially combined.

EXAMPLE: Let P_8 be defined by $S_8 = \{(x, y, z) \mid x \geq 0, y > 0, z \geq 0\}$, $f_8(x, y, z) = \{x \square x + y, z \square z - 1\}$ when $z > 0$, undefined otherwise, and $s_8 = \{(0, y, z) \mid y > 0, z \geq 0\} \square \{(x, y, 0) \mid x \geq 0, y > 0\}$; it develops the product of y and z in x by repeated addition. Let P_9 be defined by $S_9 = \{(x, z) \mid x \geq 0, z \geq 0\}$, $f_9(x, z) = \{x \square z, z \square x\}$

⁸ To combine processes with different state spaces, we first extend each of them to the space involving the union of their state variable sets by retaining the same action function (i.e., the extended process does not change variables that were not in its original state variable set, and the assignments to its original variables depend in the original manner on the original variables) and adjusting the initial state set to contain, for each original state, new states having all possible combinations of values for the new state variables.

when $z = 0$, undefined otherwise, and $s_9 = \{(x, 1) \mid x \geq 0\} \sqcap \{(1, z) \mid z \geq 0\}$; P_9 merely interchanges x and z whenever $z = 0$. Let $+_s$ denote serial combination. Then $P_{10} = P_8 +_s P_9$ has $S_{10} = S_8$, $s_{10} = \{(0, y, 1) \mid y > 0\} \sqcap \{(1, y, 0) \mid y > 0\}$ and develops successive powers of y in x and moves them to z . As shown in Figure 4, one of its computations is $\langle (0, 2, 1), (2, 2, 0), (0, 2, 2), (2, 2, 1), (4, 2, 0), (0, 2, 4), \dots \rangle$.

<u>state</u>	<u>action</u>		<u>new state</u>
	<u>by P_{11}</u>	<u>by P_{12}</u>	
(1, 1)	$\{x \sqcap 1\}$	$\{y \sqcap 2\}$	(1, 2)
(1, 2)	$\{x \sqcap 2\}$	$\{y \sqcap 3\}$	(2, 3)
(2, 3)	$\{x \sqcap 3\}$	$\{y \sqcap 5\}$	(3, 5)
(3, 5)	$\{x \sqcap 5\}$	$\{y \sqcap 8\}$	(5, 8)
.	.	.	.
.	.	.	.
.	.	.	.

FIG. 5. Synchronous combination.

It should be pointed out that even though serial combination is a fairly restrictive form of combination, it results in a process whose computations cannot necessarily be viewed as the “summation” of the separate computations of the component processes. Therefore, the acceptability of a combination is not necessarily ensured by the separate acceptabilities of its components. Rather, it is necessary to treat “cooperation” as an additional problem to be resolved. We will discuss this topic extensively in Section 3.4.

Another interesting special case of combination involves processes that “overlap” in both “space” and “time,” i.e., that “communicate” by means of common state variables and are not strictly serial. A group of processes with the same state space may be joined by the operation of *synchronous combination*. Informally, a state is an initial state of the combination if it is an initial state of each component process. The action function for each state is a composite of one action from each active component process. We call this form of combination “synchronous” (or “parallel”). This is still a fairly restrictive means of combination, but we will show in Section 3.2 that it is useful in representing other, apparently more general, forms of combination.

EXAMPLE: Consider P_{11} defined by $f_{11}(x, y) = \{x \sqcap y\}$ and P_{12} defined by $f_{12}(x, y) = \{y \sqcap x + y\}$, where $S_{11} = S_{12} = \{(x, y) \mid x > 0, y > 0\}$ and $s_{11} = s_{12} = \{(1, 1)\}$. Let $+_p$ denote parallel (i.e., synchronous) combination. Now $P_{13} = P_{11} +_p P_{12}$ has the action function $f_{13} = \{x \sqcap y, y \sqcap x + y\}$ and “computes” the Fibonacci numbers 1, 1, 2, 3, 5, 13, ... as shown in Figure 5.

More formally, the synchronous combination of a set of processes has an initial state set which is the intersection of their initial state sets. The action function for any state is the “union” of the action functions of the active components. (If one or more of the components is nondeterministic, so is the combination; in this case, the action function has as its values all unions of one value from the action function of each active component.)

If synchronous combination produces an action function that assigns distinct values to some variable by the actions of different processes, the processes are said to *conflict* in that variable and the combination is not well-defined. Requiring component processes to be *conflict-free*

(that is, to have no conflicts in any variable for any state) is analogous to avoiding race conditions in hardware.

In serial or synchronous combination of processes, strict ordering of actions is obtained. In many practical systems there may be no strict ordering between the actions of different components; in others, an ordering exists in principle, but is unknown or extremely complex. We may model such systems by *general (asynchronous) combination* of the processes representing its components, which does not involve any assumption about their relative rates. Computations of the combination may result from arbitrary mergings of sequences of actions from the component processes. The action which follows a state consists of the actions of any number of the component processes. Thus, any component process may perform arbitrarily many actions between any two actions of the other component processes; equally, they may perform arbitrarily many actions between any two actions of the given process.

More formally, the general combination of a set of processes has an initial state set that is the intersection of their initial state sets. The action function for each state of the combination consists of all the actions that are unions of actions from the action functions of any subset of the component processes. (If some action is not conflict-free, the combination is not well-defined.)

EXAMPLE: Let $+_g$ denote general combination. Then $P_{14} = P_{11} +_g P_{12}$ has the action function $f_{14}(x, y) = \{x \square y\}$ or $\{y \square x + y\}$ or $\{x \square y, y \square x + y\}$.

The basic “grain of time” represented by general combination is the interval between an action and the “next” action anywhere in the system. For general combination to accurately model real systems, the actions of the component processes must correspond to the “indivisible operations” of the processors they represent. If, on the one hand, the actions are “too large,” then some interactions of the real system will not be reflected in the combination; if, on the other hand, the actions are “too small” the combination will allow interactions which do not occur in the real system.

EXAMPLE: On some computers, multiplication is performed by means of repeated adding and shifting, which may overlap with the execution of other instructions. If the Multiply instruction is modelled by a single action, then the process will fail to model certain instruction sequences in which the operands of the multiplication are modified (perhaps unintentionally) while the operation is in progress.

EXAMPLE: Many computers have an indivisible Add to Memory instruction, which operates in a single memory cycle. If this is modelled by the three-action sequence (Load, Add, Store), then the process (but not the computer) would allow actions by other processes between the Load and the Store.

3.2 Concurrency and Clocking

Synchronous combination provides only a restricted kind of concurrency. However, even this “lockstep” form of combination is adequate to model many useful systems, e.g., the “parallel” operation of many plugboard controlled machines, or the “parallel” operation of array computers.

In general, however, not all processors in a system are active all the time, and different processors proceed at different rates. It might seem that general combination would be necessary to model systems involving processors with different speeds. However, we may

restrict our attention to synchronous combination and model such systems by introducing a new form of process definition.

When considering an isolated processor or a “lockstep” combination of processors, time is adequately represented by the number of actions which have occurred. The notion of rates of processors, however, implies some clock by which to measure rates; the notion that processors are sometimes active (changing state) and sometimes inactive implies some means of control external to the processes which represent them. This may take the form of an *enabling predicate* (depending only on state variables) which, when true, “permits” the process to proceed to the successor state and which, when false, “holds” the process in its current state, i.e., renders it inactive. It is still not necessary to introduce the concept of an “absolute” clock; rather, it suffices to provide means by which the progress of a process can be determined relative to its “environment.” We define the *clocked extension* of a process by an enabling predicate as follows:

When the enabling predicate is true, the action function of the extension is the same as that of the original process; when the enabling predicate is false, the action function of the extension has the null value.

Thus a process and its clocked extension are temporal variants. Their computations differ only in the repetition of some states within computations of the clocked extension, making those computations longer.

EXAMPLE: Consider the process P_{15} with the action function $f_{15}(x, y, z) = \{x \square x + y, z \square z - 1\}$, and the enabling predicate $c_1(x, y, z) \equiv z > 0$. Let $P_{15}^{c_1}$ and $f_{15}^{c_1}$ denote the clocked extension of P_{15} by c_1 , and its action function, respectively. Then the action function $f_{15}^{c_1}(x, y, z) = \{x \square x + y, z \square z - 1\}$ if $z > 0$ and $\{ \}$ otherwise. Similarly, the process P_{16} with the action function $f_{16}(x, y, z) = \{x \square 0, z \square x\}$ may be clocked by $c_2(x, y, z) \equiv z = 0$.

Since each clocked extension of a process is a process, our definition of synchronous combination still applies. A clocked extension may be combined with a process which changes variables on which the enabling predicate depends, termed a *clocking process*. The clocked extension may be used to represent the same processor as did the original process, but now its “rate” relative to its environment is controlled by the clocking process. The activity of the clocking process may itself be controlled by some other process. Any number of processes may be controlled by a single clocking process. Alternatively, processes may mutually clock each other.

EXAMPLE: In the synchronous combination $P_{15}^{c_1} +_p P_{16}^{c_2}$, process $P_{15}^{c_1}$ clocks $P_{16}^{c_2}$, and process $P_{16}^{c_2}$ clocks $P_{15}^{c_1}$, ensuring that precisely one of them is active in any state. Note that this achieves precisely the effect of the serial combination $P_{10} = P_8 +_s P_9$ of an earlier example.

This notion of clocking is quite general, allowing as special cases “parallel” operation, operation at fixed “speed ratios,” and processes putting themselves (or other processes) “to sleep,” or “awakening” other processes (but not themselves).

EXAMPLE: In the H800 computer [6], the commutator provides a simple clocking process controlling the rates of up to eight conceptually independent programs. This early example of multiprogramming used hardware to implement the clocking process which switched among instruction streams.

EXAMPLE: Each multiprogramming monitor is a software implementation of a clocking process controlling the rates of several tasks or programs. The above discussion has not related the progress or activity of a process to the existence of an external time continuum.

While such a relation is obviously necessary to determine the physical speeds of processors, it does not seem to be needed for understanding the structure of the processes which represent them. Rather, we are concerned with the order in which actions occur, and the relations among them.

3.3 Coexisting Processes

Processes which have been combined may be regarded as *coexisting*. Each operates on its state variables in an *environment* consisting of the remaining processes. In general, its behavior will be influenced (perhaps strongly) by the changes the environment makes to its state variables. This leads naturally to precise definitions for the intuitive notions of “input” and “output”.

The variables to which a process assigns new values are its “changed variables”; those on which its behavior depends are its “significant variables”. More formally, a variable is *immediately changed* by a process in a state if it is contained in any action following that state. It is *changed* by the process if it is immediately changed in any state. A variable is *immediately significant* to a process in a state if there is a modification of its value that results in a change in the value of the action function for that state. The *significant variables* of a process are those which are immediately significant in one or more states.

EXAMPLE: Recall processes $P_{15}^{c_1}$ and $P_{16}^{c_2}$ with $f_{15}^{c_1}(x, y, z) = \{x \sqcap x + y, z \sqcap z - 1\}$, $f_{16}^{c_2}(x, y, z) = \{x \sqcap 0, z \sqcap x\}$, $c_1 \equiv z > 0$, $c_2(x, y, z) \equiv z = 0$. x and z are the changed variables of P_{15} and P_{16} (and of $P_{15}^{c_1}$ and $P_{16}^{c_2}$). All three variables are significant to P_{15} (and $P_{15}^{c_1}$), while only x is significant to P_{16} . (Because of the clocking predicate, z is also significant to $P_{16}^{c_2}$.)

In a given environment, the *input variables* of a process are those of its significant variables which are also changed variables of its environment. Symmetrically, its *output variables* are those of its changed variables which are significant variables of its environment. Collectively, the input and output variables of a process are called input-output variables, and represent the only means by which coexisting processes may communicate with each other or control each other. An *output action* is one which includes an output variable and an *input action* is one which depends on an input variable. The utility of these actions for communication will depend on prior conventions which assure that each input-output variable is “set” by an output action before it is “used” by an input action, that no outputs are “lost,” by not being used before they are reset by further outputs, etc.

EXAMPLE: In the combination $P_{15}^{c_1} +_p P_{16}^{c_2}$, x and z are both input and output variables of both processes.

Each input-output variable is associated with at least one source process which (potentially) changes its value, and at least one destination process which (potentially) uses its value. The *scope* of a state variable is the set of processes for which it is either a changed variable or a significant variable. If this scope consists of a single process, the variable is local to that process.

EXAMPLE: In the previous combination, y is local to $P_{15}^{c_1}$, and the scope of x and z consists of both processes.

Processors seldom exist in isolation. They are implicitly combined with the external world. When we model a processor by a process, we may model the external world by a coexisting process which provides its inputs and (presumably) uses its outputs.

EXAMPLE: Consider a computer mainframe as a processor which is modelled by a process whose environment contains another process modelling a disk memory unit. The wires which transmit information between these units are represented by input-output variables of their associated processes. However, if we consider the combined process which models the entire computing system, these wires are no longer represented by input-output variables. Rather, the input-output variables correspond to interfaces with the system's users.

3.4 Process Interaction

There are three important categories of interactions among combined processes (and the processors they represent); *cooperation* includes all interactions which are anticipated and desired; *interference* includes those which are unanticipated or unacceptable; and *competition* includes those that, although they are anticipated and acceptable, are undesirable. Competition generally involves resources (state variables) which must be used serially by different processes; there is a considerable body of literature on the problems of "resource allocation" and methods for controlling process competition [e.g., 18].

The topic of this section is the choice of conditions on processes (and thus, implicitly, on the processors they represent) to enable any required amount of cooperation while rigidly excluding all interference. To be useful, these conditions must be easily satisfied by processes modelling actual processors [17]. Our discussion will focus on techniques which are adequate for general combinations. Although special techniques involving the number of actions which have occurred may sometimes be used in synchronous systems, it is generally not very helpful (or realistic) to rely on such conditions.

Interactions, by definition, occur only through input and output variables; thus, we need impose restrictions on only the input-output actions of the component processes. Note that, by our definition, information transfers between processes within a computer, as well as conventional "I/O operations," are considered as input-output actions. For example simple LOAD and STORE operations on shared variables are input and output actions, respectively. However, the undisciplined use of such operations will generally produce interference, i.e., unacceptable interactions, such as overwritten messages, doubly-used messages, multiple recipients of messages, and conflicting simultaneous updates.

More disciplined techniques for communication and control are required to eliminate interference. We will discuss a sequence of successively more sophisticated techniques for cooperation using the comparatively neutral terms SEND and RECEIVE to denote disciplined input-output operations for transferring messages between processes.

First, we note that even the simplest one-way message stream requires two-way ("feedback") communication, to ensure that the source process does not overwrite a message before the destination process has received it.

EXAMPLE: A one-way message stream between two processes may be implemented by means of two shared variables: a buffer variable, through which the messages are passed, and a one-bit flag variable, which indicates the status of

the conversation. The SEND operation can consist of the following actions: test the flag until its value is one; copy the new message into the buffer; and set the value of the flag to zero. Similarly, the RECEIVE operation can consist of the following: test the flag until its value is zero; copy the new message out of the buffer; and set the value of the flag to one. We may establish the adequacy of these operations in four steps: 1) The processes cannot deadlock,⁹ since at most one process can be waiting for the value of the flag to change. 2) There is no conflict on the buffer, since at any time the flag allocates it to either the source or the destination process. 3) There is no conflict on the flag, since at any time it can be changed by only one process. 4) Each message is used precisely once, since the value of the flag changes after each input and output operation on the buffer, ensuring that they strictly alternate.

In only slightly more complex situations, simple feedback schemes are inadequate. For example, if there are two destination processes, and each message is to be sent to either process, but not to both, it is necessary to exclude the possibility that both processes simultaneously perform RECEIVE operations. Even if the processes are serially combined, so that no two actions occur simultaneously, if the RECEIVE operation consists of several actions, interleaving these actions can produce the same interference as produced by simultaneous RECEIVE operations.

EXAMPLE: Suppose that both destination processes implement the RECEIVE operation by the sequence of actions given in the previous example. Consider the following interleaving of actions from the two processes: the first process tests the flag, finds its value to be zero, and copies the message out of the buffer; the second process tests the flag, finds its value to be zero, copies the message out of the buffer and resets the flag to one; finally, the first process also resets the flag to one. Contrary to the stated intention, the message has gone to both processes.

To ensure the cooperation of groups processes, it is generally necessary that certain sets of operations (called *critical operations* by Dijkstra) be *mutually exclusive*, i.e., at any point, at most one of them can be in progress. Actual computing systems use various forms of interlocks, corresponding to enabling predicates, to ensure mutual exclusion of actions when necessary.

EXAMPLE: In many computing systems several processors (e.g., the CPU and I/O channels) have access to a common memory unit. Special priority logic is required in the memory hardware to resolve potential conflicts (generally by delaying all but the highest-priority request).

When all critical operations are single actions, simple interlocks are sufficient to ensure mutual exclusion. However, operations such as SEND and RECEIVE do not correspond to single, indivisible actions on most processors. It is necessary to use actions which actually mirror the hardware to achieve the mutual exclusion of operations involving sequences of actions. The most common technique is to use some available mutually exclusive set of operations to enforce mutual exclusion of other sequences of actions. It has been shown by Dijkstra [12] and Knuth [22] that the simple memory interlock available on virtually all computers is sufficient to achieve any desired mutual exclusions. However, the utter

⁹ A situation where the further progress of each member of a set of processes is dependent on the further progress of some other members of that set, i.e., where all processes in the set are waiting for each other.

simplicity of the clocking process is at the expense of considerable complexity in each of the cooperating processes. This involves a great deal of what Dijkstra has termed “busy waiting,” i.e., activity by the process whose sole purpose is to synchronize with its environment while avoiding deadlock.

Some reduction in the complexity (although not in the busy waiting) of the cooperating processes is facilitated in some computers. A basic instruction (variously known as “Test and Set” [39], “Fetch and Modify Tags” [38], etc.), exploits the memory interlock to allow synchronization by means of a very simple loop. This instruction is based on an extension of the clocking process which resolves conflicting references to the memory unit, so that it allows reading followed by writing as an indivisible operation.

Somewhat more complex, but rather more elegant, primitives for synchronization have been introduced by Dijkstra [11]. These are the mutually exclusive P and V operations, which affect integer-valued variables called semaphores, which are typically initialized to non-negative values. The operation $V(sem)$ increments the value of the semaphore sem by one; the operation $P(sem)$ always decrements the value of sem by one, but this action is “blocked” when the resulting value would be negative. Thus, the use of a P operation can cause a process to become inactive, and remain inactive until some other process, by means of a V operation, enables it to proceed. No other operations are allowed on semaphores.

Processes can be made to cooperate by judicious use of P and V operations, without having to perform busy waiting, which is relegated to the (still comparatively simple) clocking process which implements P and V . This is a distinct advantage, because busy waiting involves activity without progress. P and V facilitate the task of demonstrating that continued activity produces continued progress, a very important part of ensuring the correct cooperation of a set of processes. (In addition, in multiprogramming systems they result in more efficient use of the hardware than does busy waiting, because they explicitly indicate when the processor should be reassigned, because of the logical blocking of a process.)

Dijkstra describes two rather different uses for these primitives. The first is mutual exclusion, and requires a “binary” semaphore, $mutex$, initialized to the value one. Each critical operation is preceded by $P(mutex)$ and followed by $V(mutex)$. Since every V is preceded by a P , the value of $mutex$ never exceeds one. Since no P operation can reduce a semaphore below zero, no two critical operations can ever be in progress simultaneously. Finally, since a V operation immediately follows each critical operation, no process is needlessly blocked.

The second use of semaphores is to facilitate the implementation of “producer-consumer” relationships among processes. When a process requires a message or a resource from its environment, it performs a P operation on a semaphore whose value indicates the number that are available. Whenever a process releases a message or resource to its environment, it performs a V operation on the corresponding semaphore. Again, the fact that the P operation will not cause the value of the semaphore to become negative ensures that no consumer (or set of consumers) can get ahead of its producer(s). If it also is necessary to ensure that the producers never get more than a certain number ahead of the consumers (the “bounded buffer” problem), then a second semaphore, signaling in the reverse direction, is needed.

Although P and V do not correspond to basic operations on most computers, it may often be easier to establish cooperation among processes by first using the machine operations to implement (mutually exclusive) P and V operations, and then using these operations to synchronize the processes. However, it should be pointed out that P and V are themselves very “primitive” operations, and although they facilitate the demonstration of correctness,

their use does not guarantee correctness. The integrity of a system synchronized by P and V depends on the correctness of each of its components.

EXAMPLE: If a process gets into an infinite loop after performing a $P(mutex)$, but before performing the corresponding $V(mutex)$, then it blocks all other processes from performing any critical operations, i.e., the environment cannot ensure that a process will complete a critical operation, and cannot recover if it does not.

EXAMPLE: If a process performs an extra $V(mutex)$, thereafter two processes will be allowed to perform critical operations simultaneously, thereby destroying mutual exclusion.

The message buffering system used by Brinch Hansen [3] involves somewhat less “primitive” synchronizing operations for facilitating implementation of “producer-consumer” relationships among processes. The responsibility for buffering the transfer of information between a set of processes is removed from these processes, and handed over to a clocking process. The communicating processes are simplified since they do not share variables with each other, but only with the clocking process. Their structure is independent of the complex activity in the clocking process engendered by their simple read and write operations, and recovery procedures can be provided centrally for certain types of error. For example, Brinch Hansen’s system ensures that no process can interfere with a conversation between two other processes.

The best set of sequencing primitives may differ from application to application. Even the comparatively primitive P and V operations will seem specialized and restrictive in a situation where, by virtue of the frequency with which their use causes extensive waiting, the problem of ensuring that they use an appropriate discipline for the queue of waiting processes becomes critical. In differing circumstances quite different choices as to the degree of specialization, and to the security requirements, of a set of sequencing operations will be appropriate. One promising approach is to provide facilities by means of which a hierarchy of sets of sequencing operations can be built up, starting from a very basic set.

This can be done using P and V as the base, as has been shown by Habermann [19]. However, Dijkstra [15] has recently suggested a scheme involving what he terms “secretary processes” (i.e., clocking processes) and “director processes”, which is explicitly intended for the construction of a hierarchy of sequencing operations—it could for example, be used to construct P and V . (A similar scheme has been used by Zurcher and Randell [37].) A full treatment of the various presently competing proposals for synchronization facilities is beyond the scope of this paper—however, the topic of hierarchical synchronization schemes is briefly covered in Section 4.5.

3.5 Process Switching

The notion of process combination provides an illuminating perspective on the nature of conventional sequential programs. A program is the specification of an algorithm as a sequence of elementary actions, chosen from the instruction set of a computer. Each instruction can be regarded as the definition of a “basic” process. The instruction counter and associated sequencing control constitute a clocking process for the combination of these basic processes.

On many computers this clocking process ensures that the processes are in effect combined serially (in the sense discussed earlier). On more sophisticated (e.g., “lookahead” or “pipeline”) machines the clocking process may allow activity associated with different instructions to proceed concurrently.

The clocking process and the combination of basic processes which represent the executing program communicate through a set of common state variables (in fact, in our terms, the input-output variables of the clocking process), which we will term the *program status*. For simple computers, the program status may consist of little more than the instruction counter. However, the clocking process of computers with additional facilities, such as interruption, must share more information, for example the contents of central processor registers. The interrupt facilities of many computers are an example of the implementation of clocking processes partly by hardware (the interrupt system) and partly by software (the interrupt handling routines).

When the same clocking process supervises the operation of more than one program, using the same processor at different times, the program status is central to the switching operation. The act of suspension of the execution of the program associated with a particular process must include saving its status; it can be resumed by restoring its status. Saltzer [29] based his definition of “process” in the MULTICS system on the idea of program status, including address mapping information. This leads to a fixed one-to-one association of address spaces and processes, a somewhat restrictive, but still useful definition. In multiprocessor systems this associates a process with a particular execution of a program, independent of which physical processors are involved in various stages of its activity, how often its execution is interrupted, or how many other processors are simultaneously executing that particular program.

The monitor of a multiprogramming operating system (whether or not it involves multiprocessing) implements a clocking process for the programs being executed. Much of its complexity is derived from the fact that not only must it provide process switching among independent programs, but it must also supervise interprocess communication, as discussed previously.

4. ABSTRACTION AND REFINEMENT

“Just why a scientist has a right to treat as elementary a subsystem that is in fact exceedingly complex is one of the questions we shall take up. For the moment, we shall accept the fact that scientists do this all the time and that, if they are careful scientists, they usually get away with it.” [30]

4.1 Interpretations and Images

We have already discussed how an interpretation may be applied to the behavior of a device to identify the computations of a processor. We will now extend the concept of *interpretation* to include (single-valued) functions which map states from one state space to another. These interpretations are purely mathematical, whereas our previous use of the term involved a mapping from physical quantities into mathematical quantities.

The result of applying an interpretation to a state is the *image state* (under that interpretation), when it is defined; states for which the interpretation is not defined are *unobservable* (under that interpretation). We extend the definition to computations as follows:

An *image computation* is the sequence of image states resulting from the application of the interpretation in turn to each observable state of a computation.

EXAMPLE: The output of a program (for instance, a trace program) is a computation which is very much shorter than the computation defined by the internal states of the program.

The image of a set of computations is the set of image computations. We may associate a new process with such an image in the same way that we associate a process with a processor, that is, by asserting a relation between their computations. However, since both are purely mathematical objects, such an assertion is more amenable to formal proof than a corresponding assertion involving a processor.

Even when a process is simple and well-behaved (e.g., strictly deterministic), our rather general form of interpretation does not ensure that its image can be generated by any process whatsoever. An arbitrary interpretation will not, in general, “retain” the information needed to predict further states of either the underlying or the image computation.

EXAMPLE: From the output of a program we generally cannot predict either its further output or its internal states.

Most images of processes do not exhibit the desirable properties of processes. The reason that we study such a large class of images is that the visible behavior of computer systems often takes such difficult forms, yet we want to analyze them in terms of underlying deterministic processes.

Even though we have not excluded pathological cases, a very important subclass of process images—to which we will devote most of our attention—consists of those whose image computations are themselves generated by processes. We will later discuss means for inferring such *image processes*.

EXAMPLE: Recall the “multiplication” process P_8 with $f_8(x, y, z) = \{x \square x + y, z \square z - 1\}$, when $z > 0$. Under the interpretation $I_1(x, y, z) = (x, y, z)$ if $x = 0$ or $z = 0$, computations of P_8 map into single actions, i.e., the image process has the action function $f_8^{I_1}(x, y, z) = \{x \square y \cdot z, z \square 0\}$, when $z > 0$.

EXAMPLE: The output of a trace program is a computation of an image process which models the execution of the program being traced.

The notion of a process which is an image of an “underlying” process leads naturally to the idea of “levels.” We call the image process an *abstraction* and say that it is at a higher level than the process of which it is an image. Of course, the lower level process, which we call a *refinement* of the image process, may itself be an abstraction of yet another process, and we may construct hierarchies with an arbitrary number of levels of abstraction.

By its very nature, an abstraction can contain no more information than its refinement; in general, it will contain very much less. This is not a disadvantage; rather, therein lies its utility. To comprehend successively longer sequences of activity we require successively less detailed representations, such as those provided by high-level abstractions of a complex process. Of course, these abstractions can yield insight only if they in fact correspond to high-level regularities of the process, e.g., if the image computations are actually generated by simple processes.

EXAMPLE: Complex programs may more easily be understood by studying the subroutines and then the calling structure than by an instruction-by-instruction trace. By contrast, making every 23rd state observable is unlikely to be of much assistance.

EXAMPLE: Dijkstra’s “THE” system has been presented [13] as a five-level hierarchy, in which the first level represents the hardware, and the higher-level processes have successively more complex operations (implemented by software) included among their actions.

Arbitrary interpretations may be constructed by composition of three basic type of function. *Isomorphisms* are one-to-one total functions; they “lose” no information but may be used to rename the variables of the local state set, form “compound variables,” (e.g., $2 \cdot a + b$), perform scaling, etc. *State selection* functions are identity functions whenever they are defined (the *observable states*), thus they “lose” states but do not change state variables. *Projection* functions select a subset of the state variables as observable variables whose values are unchanged, while the values of all the other (*unobservable*) state variables are “lost”, i.e. they project a state space onto a subspace.

No information is lost by an isomorphism since we can use it to determine, from any state or computation of either process, the corresponding state or computation of the other. Thus, any question about a process can be answered by studying its image und any isomorphism. Given a process and an isomorphism, we can derive the image process rather directly, using both the isomorphism and its inverse. The successor function of the image process (from which, as we have previously noted, the action function can be derived) is obtainable by using the inverse of the isomorphism to map an image state into the underlying space, then applying the underlying successor function, and finally applying the isomorphism to map the result back into the image space.

EXAMPLE: Consider the process P_{13} with $S_{13} = \{(a, b) \mid a, b \in \{0, 1\}\}$ and the successor function $F_{13}(a, b) = ((a + b) \text{ modulo } 2, 1 - b)$ and $s_{13} = \{(0, 0)\}$ under the interpretation $I_2(a, b) = 2 \cdot a + b$. The image successor function $F_{13}^{I_2}(c) = I_2(F_{13}(I_2^{-1}(c))) = ((c + 1) \text{ modulo } 4)$, where $c \in \{0, 1, 2, 3\}$. As shown in Figure 6, its image computation is the sequence $\langle (0), (1), (2), (3), (0), (1), \dots \rangle$.

<u>state</u>	<u>new state</u>	<u>image state</u>	<u>new image state</u>
(0, 0)	(0, 1)	(0)	(1)
(0, 1)	(1, 0)	(1)	(2)
(1, 0)	(1, 1)	(2)	(3)
(1, 1)	(0, 0)	(3)	(0)
(0, 0)	(0, 1)	(0)	(1)
(0, 1)	(1, 0)	(1)	(2)
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮

FIG. 6. A computation and its image.

The image of a process under a state selection function is, again, always a process. The successor function of the image is derivable by a procedure similar to that for isomorphism, except that if the immediate successor in the underlying space is not observable, the successor function is reapplied until an observable state results.

EXAMPLE: Recall the process P_8 with the successor function $F_8(x, y, z) = (x + y, y, z - 1)$ when $z > 0$, and the state selection function $I_1(x, y, z) = (x, y, z)$ if $x = 0$ or $z = 0$. The image of the observable state $(0, y, z)$ is $F_8(0, y, z) = (0 + y, y, z - 1)$ if this is observable (i.e., if $z - 1 = 0$), otherwise it is the image of $F_8(y, y, z - 1)$, i.e., it is $F_8^i(0, y, z) = (i \cdot y, y, z - i)$ for the first i which produces an observable state, namely $i = z$. Thus, we deduce that $F_8^{I_1}(0, y, z) = (y \cdot z, y, 0)$. Figure 7 shows a sample computation.

<u>state</u>	<u>new state</u>	<u>image state</u>
(0, 2, 4)	(2, 2, 3)	(0, 2, 4)
(2, 2, 3)	(4, 2, 2)	–
(4, 2, 2)	(6, 2, 1)	–
(6, 2, 1)	(8, 2, 0)	–
(8, 2, 0)	–	(8, 2, 0)

FIG. 7. Unobservable states.

The image of a process under a projection function is not necessarily a process, since a projection does not have a unique inverse. However, we can derive a process which strongly contains this image, by using the immediate successors of all states whose image is a given state to determine its successor function. Thus, we can use abstractions under projection to prove only conjectures about the non-occurrence of certain classes of computations.

EXAMPLE: The image of P_8 under the projection function $I_3(x, y, z) = (x, z)$ is strongly contained in a process $P_8^{I_3}(x, z) = (x + n, z - 1)$ when $z > 0$, where $n > 0$ is “free.” Two computations of this process are $\langle (0, 2), (5, 1), (10, 0) \rangle$ and $\langle (0, 2), (5, 1), (6, 0) \rangle$. (Note that the first computation is an image of a computation of P_8 , while the second is not.) $P_8^{I_3}$ cannot be used to prove that x will attain particular values in computations, but it does show that x never decreases, and that all computations of P_8 are finite.

Although isomorphisms preserve either the determinism or nondeterminism of a process, state selection functions preserve only determinism, and projection functions, in general, preserve neither. Thus, a deterministic process may have a nondeterministic abstraction (i.e., a state which is the image of more than one state has more than one successor), or a nondeterministic process may have a deterministic abstraction (i.e., all the successors of each state have a common image). In fact, deterministic and nondeterministic processes may well have the same abstraction under some interpretation. As Dijkstra has pointed out, one of the principal uses of abstraction is to hide the low-level indeterminisms of the hardware (e.g., in the precise order of I/O operations or interrupts) from users who are not concerned with them.

4.2 Interpreters and Programmable Processors

We have previously noted that a single device may realize a variety of processes under different interpretations. We now present a special case in which two of these interpretations are related. Consider a low-level process realized by a processor (i.e., a device plus an interpretation), and a high-level process which is its image under a second interpretation. Now the image process is realized by a processor which consists of the same device and a new interpretation which is the composition of the two given interpretations.

EXAMPLE: Recall the processor $p_4 = (D_4, I_4)$ of Section 2. The process P_4 , with action function $f_1(x, y) = \{y \square x \cdot y\}$ exactly specifies P_4 . Its image under the interpretation $I_5(x, \log_x y)$ has the action function $f_1^{I_5}(x, y) = \{y \square y + 1\}$, and exactly specifies the processor $p_5 = (D_4, I_5 \circ I_4)$, where \circ denotes functional composition.

Thus, a processor plus an interpretation may be used to define another processor, which may be treated like any other processor. However, if we retain the separate interpretations, we have processes on two levels which simultaneously specify (or explain) the behavior of the

device. We call such a processor-interpretation pair an *interpreter*, even though the distinction between an interpreter and a processor is somewhat arbitrary, as we can decompose almost any processor into an interpretation and a still lower-level processor. *The processor level is, in practice, that level below which we do not choose to identify further structure.*

EXAMPLE: The user of an interactive console language may neither know nor care that his language is interpreted by a program in machine language, which is, in turn, interpreted by a microprogram, which is, in turn, interpreted by the electronics of the computer.□□

The above example represents a conventional use of the term interpreter. Our rather broad definition also includes as special forms of interpreters such techniques as procedures, interrupt systems, and address mapping hardware. The common thread is that in each case an appropriate interpretation of a processor yields another processor, intended to be more easily understood.

In Section 3.5 we described an executing sequential program as being represented by a process which resulted from the (effectively) serial combination of the basic processes corresponding to the instructions of the program. A “general-purpose computer” is a processor with the outstanding characteristic that program instructions, as well as program status, are represented by values of its (changeable) state variables. There is a set of different types of basic processes (the “order code” of the computer). When the clocking process indicates that a particular state variable represents the next basic process to be activated, the current value of that state variable determines which of the basic processes will be performed. Thus, sets of simple basic processes and a simple clocking process can be used to construct (in principle) arbitrarily complex sequences of operations, by appropriate choices for the values of the state variables that represent the program instructions. Our interpretation of the behavior of such a process will, in general, not explicitly reflect the values of these instruction variables, but only those state variables that we choose to regard as containing “data.”

We, in fact, partition the set of significant state variables of a process into instruction variables and data variables to simplify the task of understanding its behavior. The partitioning is also arbitrary, in the sense that different viewpoints can best be accommodated by different partitions.

EXAMPLE: To the author of a program, his input is data, even if he is writing program to accept expressions, to “compile” them, to accept values for variables, and to print the values of the expressions. A user of his program will prefer to think of the expressions as instructions, and only the variable values as data.

Even when both the processor and (a properly chosen) interpretation are fixed, different programs (i.e., different sets of values for the instruction variables) will result in different processes being defined. In other words, we are using the concept of an “image process” (as defined in Section 4.1) to justify the use of the term “process” in describing the activity of a general-purpose computer. More precisely, we use the term *programmable processor* to describe an interpreter with the property that some significant variables of the underlying process are instruction variables, and are not observable at the higher level.

The theoretical importance of a given programmable processor depends on the generality of the class of processes that can be constructed using it.

EXAMPLE: The outstanding example of a theoretically important type of programmable processor is, of course, the Universal Turing Machine [32]. In fact, a computer could not reasonably be claimed to be “general-purpose” unless it could be programmed to be equivalent (aside from considerations of finite memory) to a Universal Turing Machine (cf. [26]).

In addition to any theoretical requirements, there are several factors which determine the practical utility of a given programmable processor for a particular application. These include:

- 1) The ease with which the process required for the application can be realized using the basic processes and sequencing facilities that the processor provides.
- 2) The practicality of constructing or obtaining the processor itself.
- 3) The “efficiency” of the constructed processes.

EXAMPLE: All three of these factors rule out the Universal Turing Machine from practical consideration.

Our definition of programmable processor obviously includes much more than the currently conventional class of general-purpose computers. We think, however, that this definition identifies the most essential characteristics of general-purpose computers, without including the accumulated traditions regarding the forms which their clocking process and order code “must” take. We do not wish to exclude unnecessarily any potentially useful or interesting computer architecture simply because of its disparity with von Neumann’s classic description [5] of a general-purpose computer. His stylized conventions have, by restricting the range of possibilities to be considered, undoubtedly proved themselves remarkably useful in the development of present-day computers. However, it is premature to assume that these conventions will not eventually be modified or superseded.

4.3 Description of Processes

Given a programmable processor, it becomes convenient to define a process by means of a program. A *program* for a particular processor is a set of initial values for its program status and instruction variables. The values of the program status variables indicate to the clocking process where program execution is to start, and the values of the instruction variables represent choices from the order code of the processor.

The forms which a processor requires for its programs (voltages on wires, magnetization of cores, etc.) are generally very inconvenient for human manipulation. It is a practical necessity to have more easily manipulated representations. A *description language* is a set of *descriptions*, each of which represents a process. The function which maps these descriptions into the corresponding processes is the *semantics* of the language. In particular, each programmable processor provides the semantics for the description language which consists of its programs. We frequently rely on such an existing semantics to define the semantics of another language, i.e., we give a mapping, called a *translator*, from the new description language into the old.

EXAMPLE: A compiler is a translator from a “high-level” description language into programs. A loader (perhaps in addition to linking programs) translates programs from “external” into “internal” form. A back-plane wiring machine translates descriptions on paper tape into electrical connections between pins.

We have already discussed structuring a process as a hierarchy of interpreters built on an underlying processor. Each interpreter defines a processor at its level. It is particularly

convenient for this processor to be a programmable processor. For each such programmable processor, there is a corresponding description language. There is thus a natural correspondence between levels and description languages. Such a system will be completely defined by giving its base processor and the hierarchy of programs expressed in these description languages.

EXAMPLE: In our previous example of levels, the “console machine,” the “machine,” and the “microprogram machine” are each implemented (defined) by programs on the level below.

The freedom to use at each level a description language appropriate to our conception of the behavior at that level is one of our best tools for mastering complexity. It is, of course, necessary to choose languages carefully, and to separate them cleanly, or we may find the complexity nearly as unmanageable as it would have been with one huge single-level description.

4.4 Creation of Processes

Recall our earlier discussion of a clocking process switching among component processes by saving and restoring their program status variables. It is but a slight extension of this notion for a clocking process to vary the number of processes it controls by initializing or discarding sets of program status variables. This is a further reason why it is helpful to associate closely the notions of “process” and “program status.”

It is possible to use a dynamically varying combination of processes at one level to serve as the interpreter for a dynamic combination at the next level whose variations are completely independent.

EXAMPLE: In the “THE” operating system [13] great care has been taken to structure the monitor explicitly into levels implemented by cooperating processes. Excerpts from the system description indicate their relation: “At level 0 we find the responsibility for processor allocation to one of the processes.” “Above level 0 the number of processors actually shared is no longer relevant. At higher levels we find the activity of the different sequential processes, the actual processor,□□ having disappeared from the picture.” “At level 1 we have the so-called ‘segment controller,’ a sequential process synchronized with respect to the drum interrupt and the sequential processes on higher levels.” “At level 2 we find the ‘message interpreter’□□□□Above level 2 it is as if each process had its private conversational console. The fact that they share the same physical console is translated into a resource restriction of the form ‘only one conversation at a time’.” “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.”

5. COMBINATION AND ABSTRACTION

Abstraction is a means of avoiding unwanted complexity. Previously we indicated how very complex processes could be obtained from simple ones by combination, and noted that their separate correctness was insufficient to guarantee correctness of the combination. Abstraction plays a crucial role in mastering the complexity of such combinations. It allows, for example, a correctness proof for an entire system to be constructed from separate proofs for each process (under certain assumptions about its environment) plus a proof of cooperation (i.e., that all environmental assumptions are satisfied).

The use of abstraction to establish properties of combinations of processes is not new [13]. It is tempting to assume that an abstraction of a combination of processes is the same as the combination of their separate abstractions [20]. Unfortunately this is not generally true.

EXAMPLE: Consider a combined group of processes that are synchronized by means of P and V operations on semaphores, and an interpretation that “loses” the value of the semaphores. Now the abstraction (under this interpretation) of the combination will still reflect this synchronization, while the combination of the abstractions cannot be synchronized by the “lost” semaphores.

Recently, some attention has been devoted to the problem of finding restrictions which ensure that a combination of abstractions accurately models the abstraction of the combination, or, equivalently, that combinations of refinements actually are a refinement of the intended combination [24].

Suppose that we wish to establish the correctness and cooperation of a group of combined processes. First, we may use abstractions which select only the state variable sets of single processes; each image process now represents a single component of the combination, which may be studied separately. (The image will, of course, reflect nondeterministic changes in the input variables caused by other processes.) Next, we may study the abstraction which reduces each sequence of actions within a single process that do not involve input-output to a single action. Finally, if we ensure the mutual exclusion of the sequences of actions which constitute the input-output operations of separate processes, we can safely use the abstraction in which each such operation becomes a single action.

There are two basic ways of achieving mutual exclusion of operations in a system involving asynchronous combination. Recalling the techniques discussed in Section 3.4, the availability of even fairly simple operations which are mutually exclusive may be used to ensure the mutual exclusion of operations consisting of arbitrarily many actions. Thus, this aspect of the correctness of a system can be treated as a recursive problem, with the mutual exclusion of operations on each level dependent on the achievement of mutual exclusion on a lower level. Of course, this recursion must terminate. It seems that the only technique for achieving mutual exclusions which is not based on a lower-level mutual exclusion involves an active clocking process which “polls” the processes it is clocking, and allows the critical operations to proceed one at a time.

To date, practical applications of abstraction and combination in structuring complex systems have relied on informal conditions to assure that arguments about abstractions could be carried over to their refinements. This has, for example, been the case in the work of Dijkstra [13] and of Zurcher and Randell [37]. Both papers concern design methodologies in which the concept of levels of abstraction plays a central role.

The former paper describes the design and structure of the “THE” multiprogramming system. The outstanding feature of this design methodology is the careful use of structure (in particular, levels) to enable the designers to satisfy themselves, a priori, as to the logical “correctness” of the system. The aim is to show that whenever a process is presented with a task, it will, under all circumstances, complete the task within a finite time and return to its “homing position,” ready to accept a new task. The proof proceeds in three stages: no process, while performing a single task, can lead to the generation of an infinite number of further tasks; when all processes have returned to their homing positions, no uncompleted tasks remain; there is no possibility of deadlock, so all processes must ultimately return to their homing positions.

The feasibility of proofs of conjectures about systems as complex as the “THE” system depends strongly on the degree to which reliance on enumerative reasoning can be minimized [14]. The concept of multilevel processes is very useful in this regard. One can represent a group of sequential processes by a single image process, and prove that if this can progress, so can each of the set of processes of which it is an image. In further arguments it is then sufficient to satisfy oneself that the image process will always be able to progress. This technique can substantially reduce the number of situations which must be considered at each stage of the proof. Dijkstra also notes that this approach has significant advantages in testing a system as it is implemented. “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.” [13]

The use of multilevel processes described by Zurcher and Randell [37], on the other hand, grew out of the desire to simulate the design of a complex system as the design took shape. Thus, the simulation would gradually evolve and grow, and possibly become the actual system. This naturally placed very severe demands on the understandability and modifiability of the simulation program, which were met, at least in part, by constructing it as a set of distinct levels. Each level represented, at an appropriate degree of abstraction, the state of the system and those actions of the system best described in terms of that particular abstraction. The number of sequential processes on a given level would be chosen independently of the number on any other level. (For example, one level might represent each of the dynamically varying number of jobs in the system as a sequential process; another level, each of the hardware processors as a sequential process.)

There was a distinct methodological difference between these two efforts. Dijkstra’s approach consisted of successively forming simpler images of lower-level operations, whereas Zurcher and Randell’s approach consisted of successively forming refinements of higher-level actions. There is probably no single “correct” order in which to take a series of design decisions, though it can usually be agreed that some orderings are better than others. Almost invariably, some early decisions (thought to have been clearly correct when they were made) will turn out to have been premature. A more extensive discussion of this topic is contained in [27].

6. APPLICATIONS OF STRUCTURE

“The fact, then, that many complex systems have a nearly decomposable, hierarchic structure is a major facilitating factor enabling us to understand, to describe, and even to see such systems and their parts. Or perhaps the proposition should be put the other way round. If there are important systems in the world which are complex without being hierarchic, they may to a considerable extent escape our observation and our understanding. Analysis of their behavior would involve such detailed knowledge and calculation of the interactions of the elementary parts that it would be beyond our capacities of memory or computation.” [30]

Structuring techniques, and formalisms for their description, are of value only as they are applied. Our formalism has been developed because it facilitates the careful consideration of both combination and abstraction within a uniform conceptual framework; we have concentrated on these two techniques because they are of profound importance in structuring

the design and implementation of complex systems. Previous sections have discussed applications of particular techniques; in this concluding section we turn to more general uses.

The importance of structuring is a result of its usefulness in mastering complexity. This applies whether one is trying to understand an existing system, or to design a proposed new system. The goal is to profit from this “mastery” by finding better ways of producing better systems, and, as an almost automatic by-product, better methods of documenting systems. However, it is important to recognize that structuring in itself is not necessarily beneficial; bad or excessive structuring may be valueless or even harmful.

EXAMPLE: A program which has been divided into too many subroutines may not only be unreadable, but may also execute very inefficiently.

The appropriate use of structure is still a creative task, and is, in our opinion, a central factor of any system designer’s responsibility.

“When we cannot grasp a system as a whole, we try to find divisions such that we can understand each part separately, and also understand (in that framework) how they interact. When we make such a division for the purpose of analysis, each part is treated in turn as the machine of interest and the remainder as its environment. One cannot usefully make such divisions completely arbitrary because an unnatural division of a system into ‘parts’ will not yield to any reasonable analysis.” [26]

There are as yet few generally accepted guidelines for developing appropriate structures. Parnas [28] has suggested: “In designing your system you should decompose it into a set of cooperating sequential processes, designing each of these processes separately and considering their cooperation as a question separate from their design.” How to choose the component processes is a question which still lacks a general answer, even though there have been some very successful attempts at following this guideline in the design of specific systems, e.g., by Brinch Hansen [3], Dijkstra [13], and Turski [33].

A useful guideline for the choice of effective levels of abstraction may well be implied by a characteristic of the “THE” system, recently discussed by Dijkstra [16]. He points out that, in retrospect, one of the crucial characteristics of the set of levels of abstraction used in the “THE” system is that each level is concerned with events that occur on a substantially longer time scale than that of its underlying level. Level 0 effects processor switching, on a 50 μ sec time scale; level 1 performs memory management, using a drum with a 40 msec revolution time; level 2 communicates with the operator, on a time scale of seconds; level 3 performs peripheral assignments, every few minutes; and level 4 handles job submission (several minutes). It appears that this sort of relationship is essential for a given level of abstraction to safely ignore the detailed sequencing on lower levels—otherwise, efficiency considerations will force the incorporation of interlevel scheduling interactions. (A similar relationship between levels and time scales has been used by Lynch to structure the CHIOS operating system [25].)

If a design team has been successful in using structuring to help them cope with the complexities of the system they are constructing, then we would expect an improved final product. There are also more direct ways in which structuring, as described earlier, can influence the quality of a system. It is beneficial to mirror the conceptual structuring of the process that represents the system’s behavior by the actual physical structure of the hardware and software that constitute the computer system.

Some of the possible benefits of retaining structure are fairly obvious, such as increased ease of modification, and decreased shelf-footage of required documentation. It is also clear that if what might otherwise have been defined as a single, rather complicated, sequential process is instead defined as the asynchronous combination of a set of simpler processes, then the conceptual concurrency might well become actual concurrency (given appropriate processor facilities), with the result that considerable performance gains ensue.

Another example is a process which is realized by means of a hierarchy of interpreters built on an underlying programmable processor. Each interpreter will be defined by a program in some description language. Appropriate design of description languages will normally result in the total description of the process being much smaller than a description in terms of the order code of the underlying processor. Furthermore, the levels of program may be reflected in the choice of memories for their storage (e.g., a microprogram level in high-speed storage, and a program level in core storage).

However, one of the most interesting potential benefits of extensive use of the structuring techniques that have been described in this paper concerns the problem of achieving system reliability, in the face of hardware malfunctions and/or software errors. All error detection is based on the provision of redundant information, whose consistency can be checked. The subdivisions of a complex process into levels and into groups of cooperating processes can provide guidelines as to what redundancy should be provided, where it should be checked, and what recovery should be attempted when inconsistencies are found.

Subdivisions that are merely conceptual are of little assistance in error recovery. For example, the levels used by Dijkstra, though essential to his technique of establishing the initial correctness of his system [13], are not particularly helpful in coping with hardware problems. This is because their physical realization in the corresponding program text is by means of conventions, just some of which are enforced by the compiler which generated the program text (and even here it is necessary to assume that the compiler is error-free, and was run while the machine was functioning correctly). However, this question of the use of the concept of process structuring for achieving system reliability is a subject in itself—detailed discussion is beyond the scope of this paper.

Finally, it is worth mentioning that users of a computer system can benefit greatly if the system is carefully structured into several levels. To a certain extent this already happens—different classes of users of a computer system may have quite different means of communicating with the system, depending on whether they are, for example, maintenance engineers, system programmers, FORTRAN programmers, or users of standard application packages. However, as has been pointed out by Bridger [2] in his comments on a paper by Bryant [4], these benefits can be greatly diminished if the separation between levels is not properly maintained. Levels which should be invisible to a particular user have an embarrassing tendency to show up when there is a malfunction—the task of minimizing these occurrences and their effects should be regarded as an important responsibility of a system designer.

7. ACKNOWLEDGMENTS

This paper is largely based on an earlier report by the same authors [20]. Its preparation has been greatly aided by the many useful and stimulating discussions that the authors have had with E. W. Dijkstra, G. C. Driscoll, A. N. Habermann, C. J. Kuehner, H. C. Lauer, W. M. McKeeman, D. L. Parnas, D. Tsichritzis, and F. W. Zurcher. Their valuable comments and criticisms (and those of the referee) on earlier drafts of this paper are largely responsible for

any merits the present version might have. In particular, our original, somewhat unsatisfactory, definition of “process” has been replaced by a superior one suggested to us by Lauer.

8. REFERENCES

- [1] BELL, C. G.; AND NEWELL, A. *Computer structures: readings and examples*. McGrawHill, New York, 1971.
- [2] BRIDGER, D. A. “Comments on levels of computer systems.” *Comm. ACM* **10**, 5 (1967), 260.
- [3] BRINCH HANSEN, P. “The nucleus of a multiprogramming system.” *Comm. ACM* **13**, 4 (1970), 238-241, 250.
- [4] BRYANT, P. “Levels of computer systems.” *Comm. ACM* **9**, 12 (1966), 873-876.
- [5] BURKS, A. W.; GOLDSTINE, H. H.; AND VON NEUMANN, J. *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*, Part I. Vol. 1. Institute for Advanced Study, Princeton, New Jersey, 1946.
- [6] CLIPPINGER, R. “Multiprogramming on the Honeywell 800/1800.” In *Information Processing 1962. Proceedings of IFIP Congress 62*, C. M. Popplewell (Ed.), North-Holland, Amsterdam, The Netherlands, 1962, 571-572.
- [7] CONWAY, M. E. “Design of a separable transition-diagram compiler.” *Comm. ACM* **6**, 7 (1963), 396-408.
- [8] DAHL, O. J., AND NYGAARD, K. “SIMULA—an ALGOL-based simulation language.” *Comm. ACM* **9**, 9 (1966), 671-678.
- [9] DENNING, P. J. “Third generation computer systems.” *Computing Surveys* **3**, 4 (1971), 175-216.
- [10] DENNIS, J. B., AND VAN HORN, E. C. “Programming semantics for multiprogrammed computations.” *Comm. ACM* **9**, 3 (1966), 143-155.
- [11] DIJKSTRA, E. W. Cooperating sequential processes. Report EWD 123, Mathematical Department, Technological University, Eindhoven, The Netherlands, September 1965. [Reprinted in *Programming Languages* (F. Genuys Ed.) Academic Press, London, 1968.]
- [12] DIJKSTRA, E. W. “Solution of a problem in concurrent programming Control.” *Comm. ACM* **8**, 9 (1965), 569.
- [13] DIJKSTRA, E. W. “The structure of the “THE” multiprogramming system.” *Comm. ACM* **11**, 5 (1968), 341-346.
- [14] DIJKSTRA, E. W. Notes on structured programming. Report EWD 249, Mathematical Department, Technological University, Eindhoven, The Netherlands, August 1969.
- [15] DIJKSTRA, E. W. “Hierarchical ordering of sequential processes.” *Acta Informatica* **1**, 2 (1971), 115-138.
- [16] DIJKSTRA, E. W. Lecture at International Summer School in program structures and fundamental concepts of programming, Marktoberdorf, Germany, July 1971.
- [17] GILBERT, P.; AND CHANDLER, W. J. “Interference Between Communicating. Parallel Processes.” *Comm. ACM* **15**, 6 (1972), 427-437.
- [18] HABERMANN, A. N. “On the harmonious cooperation of abstract machines.” PhD Thesis, Technological University, Eindhoven, The Netherlands, 1967.
- [19] HABERMANN, A. N. “Synchronization of communicating processes.” *Comm. ACM* **15**, 3 (1972), 171-176.

- [20] HORNING, J. J.; AND RANDELL, B. Structuring complex processes. Report RC 2459, IBM Thomas J. Watson Research Center, Yorktown Heights, N. Y., 1969.
- [21] JOHNSTON, J. B. "The structure of multiple-activity algorithms." In *Proc. 2nd ACM Symposium on Operating Systems Principles*, October 1969.
- [22] KNUTH, D. E. "Additional comments on a problem in concurrent programming control." *Comm. ACM* **9**, 5 (1966), 321-322.
- [23] LAMPSON, B. W. "A scheduling philosophy for multi-processing systems." *Comm. ACM* **11**, 5 (1968) 347-360.
- [24] LAUER, H. C. "Correctness in operating systems." PhD Thesis, Carnegie-Mellon Univ. Pittsburgh, Pa. 1972.
- [25] LYNCH, W. An operating system design for the computer utility environment. Paper presented at the International Seminar on Operating System Techniques, Belfast, N. Ireland, Aug.-Sept. 1971.
- [26] MINSKY, M. *Computation: finite and infinite machines*. Prentice-Hall, Englewood Cliffs, N. J., 1967. .
- [27] NAUR, P.; AND RANDELL, B. (Eds.) *Software engineering*. NATO Scientific Affairs Division, Brussels, Belgium, 1969.
- [28] PARNAS, D. L. On identifying processes within an operating computer system. Dept. of Computer Science, Carnegie-Mellon Univ. Pittsburgh, Pa. (unpublished).
- [29] SALZTER, J. H. "Traffic control in a multiplexed computer system." PhD Thesis, MIT, Cambridge, Mass., July 1966.
- [30] SIMON, H. A. "The architecture of complexity." In *Proc. Am. Phil. Soc.* **106**, (1962), 467-482. (Reprinted in *The Sciences of the Artificial*, MIT Press, Cambridge, Mass., 1969.)
- [31] THORTON, J. E. "Parallel operation in the Control Data 6600." In *AFIPS Conference Proceedings, Vol. 26, Part 2, 1964- Fall Joint Computer Conf.*, Spartan Books, Washington, D. C., 1964, 33-40.
- [32] TURING, A. M. "On computable numbers, with an application to the entscheidungsproblem." In *Proc. London Mathematical Society* (2) **42** (1936-37).
- [33] TURSKI, W. M. "SODA-A dual activity operating system." *Computer J.* **11**, 2 (1968), 148-156.
- [34] VAN HORN, E. C. Computer design for asynchronously reproducible multi-processing. MIT Project MAC Technical Report, MAC-TR-34 November 1966.
- [35] WEGNER, P. "Data structure models for programming languages." *SIGPLAN Notices* **6**, 2 (February 1971).
- [36] WILKES, M. V. *Time-sharing computer systems*. Macdonald & Co., London, 1968.
- [37] ZURCHER, F. W., AND RANDELL, B. "Iterative multi-level modeling—a methodology for computer system design." In *IFIP Congress 68*, Edinburgh, Scotland, August 1968 (preprints), D138-D142.
- [38] Instruction operations for the B8501 central processor module. Reference Manual BJ-27 Burroughs Corporation, Defense, Space and Special Systems Group, July 1966.
- [39] IBM System/360 principles of operation. Form A22-6821-7, IBM Corporation, Data Processing Division, White Plains, N. Y., 1968.

9. INDEX OF DEFINITIONS

abstraction, 21
action, 5
action function, 6
active, 10
changed, 15
clocked extension, 13
clocking process, 14
coexisting, 14
competition, 16
computation, 5, 8
conflict, 12
conflict-free, 12
cooperation, 16
critical operations, 17
description, 25
description language, 25
deterministic, 6
disjoint combination, 10
enabling predicate, 13
environment, 14
exact realization, 8
exact specification, 8
final state, 5
general combination, 12
generate, 6
image computation, 20
image process, 21
image state, 20
immediate successor, 5
immediately changed, 15
immediately significant, 15
initial state, 5
input action, 15
input variables, 15
interference, 16
interpretation, 7, 20
interpreter, 24
isomorphism, 22
mutually exclusive, 17
null action, 5
observable state, 22
observable variable, 22
operation, 9
output action, 15
output variables, 15
process, 4, 6
processor, 7
program, 25
program status, 20
programmable processor, 25
projection, 22
refinement, 21
scope, 15
semantics, 25
serial combination, 10
significant variables, 15
standard form, 6
state selection, 22
state space, 5
state variable set, 5

state variables, 5

strictly deterministic, 6

strongly contained, 7

successor function, 7

synchronous combination, 12

temporal variants, 6, 7

translator, 26

unobservable, 20

weakly contained, 7