Abstract:

The general concept of a "process" has become increasingly important both as a tool for gaining an understanding of complex computer systems, and as a basis for organizing their design. This has occurred despite the lack of general agreement on a definition of the concept.

The present paper provides a framework for the definition and extended discussion of processes and techniques for structuring them. It uses the concept to unify a variety of hardware and software tools and methodologies.

PROCESS STRUCTURING

by

J. J. Horning

and

B. Randell

TECHNICAL REPORT SERIES

Series editor: Dr. B. Shaw

Number 31

March, 1972

© 1972 University of Newcastle upon Tyne,
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.
HORNING, James Jay

Process Structuring. [By] J.J. Horning [and]
B. Randell.
Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1972.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 31).

12"

Added entries
RANDELL, Brian
UNIVERSITY OF NEWCASTLE UPON TYNE. Computing Laboratory.

Suggested classmarks (primary classmark underlined)
Library of Congress:
Dewey (17th):
U.D.C.: 621.381952
681.322.02

Suggested keywords:
COOPERATING PROCESSES
CO-ROUTINE
HIERARCHICAL STRUCTURES
INTERPRETER
PROCESSOR
PROGRAM
SEQUENTIAL PROCESS

Abstract
The general concept of a "process" has become increasingly important both as a tool for gaining an understanding of complex computer systems, and as a basis for organizing their design. This has occurred despite the lack of general agreement on a definition of the concept.

The present paper provides a framework for the definition and extended discussion of processes and techniques for structuring them. It uses the concept to unify a variety of hardware and software tools and methodologies.

About the Author
Professor Horning is an assistant professor in the University of Toronto.
Professor Randell has been Professor of Computing Science in the Computing Laboratory of the University of Newcastle upon Tyne since 1969, before which he was at the I.B.M. Watson Research Centre, U.S.A.
## TABLE OF CONTENTS

1. INTRODUCTION .................................................. 4

2. PROCESSES AND PROCESSORS: BASIC DEFINITIONS .......... 9

3. COMBINATION OF PROCESSES ................................. 17
   3.1 Types of Combination .................................. 17
   3.2 Concurrency and Clocking .............................. 23
   3.3 Co-existing Processes .................................. 25
   3.4 Process Interaction .................................... 27
   3.5 Process Switching ...................................... 36

4. ABSTRACTION AND REFINEMENT ............................... 38
   4.1 Interpretations and Images ............................. 38
   4.2 Interpreters and Programmable Processors .......... 43
   4.3 Description of Processes .............................. 47
   4.4 Creation of Processes .................................. 48
   4.5 Combination and Abstraction ........................... 50

5. CONCLUDING REMARKS ........................................ 55

6. ACKNOWLEDGMENTS ............................................ 60

7. REFERENCES .................................................. 60
1. INTRODUCTION

The complexity of 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 problems of gaining an understanding of the behaviour of an existing complex computer system can be immense. One of our best methods of coping with complexity is "divide and conquer". 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 our 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 piece of program text may be simultaneously executed by several different processors for different reasons. A suitable decomposition must clearly reflect this dynamic structure. Several authors [3, 12, 29, 33] have successfully used "process" as the basic unit in their descriptions of complex computer systems (although they have not always used that name).

The concept of process has been in simulation languages for many years. It is known as a "transaction" in GPSS [17]:

"The system represented by the block diagram is operating upon certain basic units that move through the system. ... For convenience the unit is referred to in the simulation as a transaction. The simulation proceeds by creating transactions to represent these units and
moving the transactions through the block diagram in the same manner as the units would progress through the system represented by the block diagram.

and as a "process" in Simula [8]:

"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."

It is the essence of the task concept of OS/360, even though the definition given in [37] for "task" is:

"A unit of work for the central processing unit from the standpoint of the control program; therefore, the basic multiprogramming unit under the control program."

Van Horn [34] gives an informal definition in terms of a "clerk" that can obey the directions given in a program:

"The sequence of actions that a clerk performs in executing a program is called a process."

However, Dennis and Van Horn [9] base their definition on the notion of control:

"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."

The definition given by Saltzer [29] is much more detailed, but also appeals to the concept of a device in which the process "runs":

5.
"A process is a program in execution by a pseudo-processor. The internal tangible evidence of a process is a pseudo-processor stateward, which defines both the current state of execution of the process, and the addresses which are accessible to the processor. There is, then, a one-to-one correspondence between processes and statewords, and also between processes and address spaces. It will, in fact, be convenient to make use of this correspondence and identify a process with its address space."

This definition was elaborated, although not made any more precise by Lampson [23].

"... the essential characteristic of a process is that it has, at least conceptually, a processor of its own to run on, and that the state of its processor is more or less independent of all other processors on which all other processes are running."

Brinch Hansen's definition [3] is no more formal, but does make it clear that interactions with the external world can be unified with the process concept:

"An internal process is the execution of one or more interruptable programs in a given storage area. ... An external process is the input/output of a given document identified by a unique process name.

Habermann [18] gives a formal definition in terms of finite state machines and then an informal one:

"... a '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."

Finally, one last quotation, from Dijkstra [10], who used the concept of a sequential process to great effect in his work on the design of the "THE" Multiprogramming System [12]:

6.
"The technical term for what we have called 'rules of behaviour' is an algorithm or program. (It is not customary to call it 'a sequential program' although this name would be fully correct.) Equipment able to follow such rules, 'to execute such a program' is called 'a general purpose sequential computer' or 'computer' for short; what happens during such a program execution is called 'a sequential process.'"

These quotations demonstrate that the general concept of process is gaining acceptance as a tool for understanding any complex activity (the example Parnas [28] uses is that of the apparently patternless sequence of conversations that the listener to a large city's police radio network would hear), and in particular that of a sophisticated computer system. They also illustrate the lack of any general agreement on a precise definition of (or even a term for) the concept, or on methods of structuring a complex system out of a set of processes.

This paper contains extensive discussions both of the concept of processes and of techniques for constructing structures of processes which are adequate to mirror the behaviour of complex computer systems. The major goal of the paper is to identify useful methods of structuring complex processes, and to relate these to the problems of improving the quality of large computer systems, and our methods of designing and constructing them.

We have attempted to unify our discussion by means of a coherent set of definitions for the important concepts involved. Since it has not been possible to keep these definitions consistent with all the conflicting ones in the
literature, this paper builds its framework up from a set of quite basic definitions, contained in Section 2. The significance of many of these definitions will become apparent only in the later sections. We therefore encourage the reader 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. We have restricted ourselves to a few rather basic mathematical concepts, such as sets, sequences, relations, and functions. Examples illustrate both the concepts defined and the notation used for them.
2. **PROCESSES AND PROCESSORS: BASIC DEFINITIONS**

In this paper we present 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 ("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 that 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 that 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.

The quotations in the previous section have variously emphasized three distinct aspects of the process concept: it is used to model the status of a system (e.g., [9]), or to model some sequence of status values (e.g., [18]), or as a means of generating a class of such sequences (e.g., [8]). Although no one of these provides a complete basis for the definition of process, each of them 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 states, or to the action function which generates its computations. We will discuss each of these concepts in more detail before presenting our formal definition of "process."

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

**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)\} | m > 0, n > 0\}; again, this may be written \( \{(m, n) | m > 0, n > 0\} \) 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 on a state variable set is a sequence of states of that set. The first element of the sequence is its initial state, the last (if it is finite), its final state.

**EXAMPLE:** A finite computation on the state variable set \( V \) of a previous example is the sequence \( C_1 = <(2, 2), (2, 4), (3, 9)> \) for which \((2, 2)\) is the initial state and \((3, 9)\) the final state.
EXAMPLE: An infinite computation on V is the sequence \( c = \{(2,2^i) \mid i = 1, 2, 3, \ldots\}\). 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 on a state variable set is a (named) set of new values for some of the variables of that state set. If a state is followed by an action, the state's immediate successor is the new state whose variables have the new values of the action, if they are named in it, and otherwise have their old values from the previous state. The null action is the empty set (denoted \( \emptyset \)) and specifies no changes.

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

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

An action function on a state variable set 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 $f_1(x,y) = \{(y,x,y)\}$.

An action function is strictly deterministic if it is single-valued wherever it is defined* (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: The computation $C_1$ is generated from the initial state $(2,2)$ by the action function $g_1$, where

$$g_1(2,2) = \{(y,4)\}$$
$$g_1(2,4) = \emptyset \text{ or } \{(x,3), (y,9)\}$$
and $g_1$ is undefined elsewhere.

The standard form of this action function is $g_2$, where

$$g_2(2,2) = \{(y,4)\}$$
$$g_2(2,4) = \{(x,3), (y,9)\}$$
and $g_2$ is undefined elsewhere.

From $(2,2)$, $g_2$ generates only the computation $C_2 = \langle(2,2), (2,4), (3,9)\rangle$. Since $g_2$ is strictly deterministic, $g_1$ is deterministic.

* 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.
A process is a triple \((S, F, s)\), where \(S\) is a state variable set, \(F\) is an action function on that set, and \(s\) is the subset of the state space of \(S\) which defines the initial states of the process. 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.) Similarly, processes are temporal variants if they differ only in having action functions which are temporal variants.

We frequently wish to study processes which are related, but not identical. A process is \textit{weakly contained} in another process if all of its computations are also generated by that process, and \textit{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.

\textbf{EXAMPLE}: The process \(P_1 = (V, \sigma_1, s_1)\), 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 \(P_3 = (V, \sigma_3, s_1)\), which generates only \(C_3\).

\textbf{EXAMPLE}: The process \(P_3 = (V, f_1, s_1)\) is strictly deterministic, and generates only the computation \(C_3\). It is strongly contained in the process \(P_4 = (V, f_2, s_2)\), where \(s_2 = \{(n,n) \mid n>0\}\), which generates all computations of the form \(<(n,n^i) \mid i=1,2,3,\ldots>\). However, \(P_3\) is only weakly contained in the process \(P_5 = (V, f_5, s_2)\), where \(f_5(x,y) = \{(y,2,y)\}\), which generates all computations of the form \(<(n,2^i \cdot n) \mid i=0,1,2,\ldots>\).

From any action function, we can deduce the corresponding \textbf{successor function} whose value in any state is the \textit{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
the corresponding action function 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 function 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

\[
\begin{align*}
G_2(2,2) &= (2,4) \\
G_2(2,4) &= (3,9)
\end{align*}
\]

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.

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" behaviour 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, call the process an **exact specification** of the processor.

**EXAMPLE:** The process $P_4$ of a preceding example is an exact specification for the process $p_4 = (D_4, I_4)$ schematically represented in Figure 1, 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.

![Diagram](image_url)

**Figure 1.** Schematic representation of the processor $p_4$. 

15.
More generally, when a process is used as a means of formalizing and/or understanding the behaviour, 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 of its computations (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 viewpoints are required to provide an adequate understanding of its internal structure and behaviour.

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 multiprocessor 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. In this section, therefore, we consider processes which can be described as (possibly recursive) "combinations" of component processes. Combination allows us to build processes 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.

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 variable set is the union of the component state variable sets, and whose initial state set is the direct
product of the component 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 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.

**EXAMPLES:** Let \( P_5 = (S_5, f_5, s_5)^* \), where
\[
S_5 = \{ x | x > 0 \}, \quad f_5(x) = \{(x, x+1)\},
\]
\( s_5 = \{(0)\} \). Also, let \( P_6 \) be defined by \( S_6 = \{ y | y > 0 \}, \quad f_6(y) = \{(y, 2 \cdot y)\}, \)
\( s_6 = \{(1)\} \). Then their disjoint combination \( P_\gamma = P_5 + d \ P_6 \) is given by \( S_\gamma = \{(x, y) | x \geq 0, y > 0 \}, \)
\( f_\gamma(x, y) = \{(x, x+1)\} \) or
\[
\{(y, 2 \cdot y)\} \) or
\[
\{(x, x+1), (y, 2 \cdot y)\},
\]
and \( s_\gamma = \{(0, 1)\} \).

* Henceforth, we will use \( S_n, f_n, s_n \) as the components of \( P_n \) without further comment.*
On the other hand, given a collection of processes with the same state variable sets, 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 variable set 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 those of the components.

**Example:** Let \( f_8(x,y,z) = \{(x,x+y), (z,z-1)\} \) when \( z > 0 \), undefined otherwise; let \( f_8(x,y,z) = \{(x,0), (z,x)\} \) when \( z = 0 \), and undefined otherwise. Further, let \( S_8 = S_9 = [(x,y,z)|x \geq 0, y > 0, z > 0] \) and \( S_8 = S_9 = [(0,y,1)|y > 0] \cup [1,y,0]|y > 0 \). The process \( P_8 \) develops the product of \( y \) and \( z \) in \( x \) by repeated addition. The process \( P_9 \) interchanges \( x \) and \( z \) whenever \( z = 0 \). Their serial combination \( P_{10} = P_8 \circ P_9 \) develops successive powers of \( y \) in \( x \) and moves them to \( z \).

\* To combine processes with different state variable sets, we first extend each of them to 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 changes 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.
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 acceptability of its components. Rather, it is necessary to treat their "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 "time" and "space", i.e. that are not strictly serial and "communicate" by means of common state variables. A group of processes with the same state variable set 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") because the actions of the resulting process are determined by those of all the component processes acting "at the same time" ("in parallel"). This is still a fairly restrictive means of combination, and does not extend our class of processes, 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,y)\} and P_{12}^{*} (x,y) = \{(y,x+y)\}, where S_{11} = S_{12} = \{(x,y) | x>0, y>0\} and S_{11} = S_{12} = \{(1,1)\}. The synchronous combination \(P_{13} = P_{11} + P_{12}\) has the action function \(f_{13}^{*} = \{(x,y), (y,x+y)\}\) and "computes" the Fibonacci numbers 1, 1, 2, 3, 5, 8, 13 ... .
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 has as its values all unions of one value from the action function of each active component. If such a 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 the avoidance of race conditions in hardware.

In serial or synchronous combination or 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. The action which follows an attainable state may consist of the actions of any one or more 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, provided that those actions are conflict-free (otherwise it is not well-defined).

21.
EXAMPLE: The general combination $P_{14} = P_{11} + P_{12}$ has the action function $f_{14}(x,y) = \{(x,y)\}$ or $\{(y,x+y)\}$ or $\{(x,y), (y,x+y)\}$.

The basic "grain of time" represented by asynchronous combination is the interval between an action and the "next" action anywhere in the system. For asynchronous 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; on the other hand, if 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 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 plug-board 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 one is 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 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".

23.
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 $f_{15}(x,y,z) = \{(x,x+y), (z, z-1)\}$. Its clocked extension by the predicate $C_1(x,y,z) \equiv z > 0$ has the action function $f_{15c1}(x,y,z) = \{(x,x+y), (z, z-1)\}$ if $z > 0$ and $\emptyset$ otherwise. Similarly, the process $P_{16}$ with the action function $f_{16}(x,y,z) = \{(x,0), (z, x)\}$ 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 $P_{15c1} +_p P_{15c2}$, process $P_{15}$ clocks $P_{15c2}$, and process $P_{15}$ clocks $P_{15c1}$, ensuring that precisely one of them is active in any state. Note that this achieves precisely the effect of the serial combination $P_{10} = P_8 +_p P_9$ 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 multi-programming 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 behaviour will be influenced (perhaps strongly) by the changes which the environment makes to its state variables. This leads naturally to precise definitions for the intuitive notions of "input" and "output."

These definitions are facilitated by introducing the concepts of changed and significant variables. 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 $P_{15}^{c_2}$ with

$$f_{15}(x,y,z) = \{(x, x+y), \ (z, z-1)\}, \ f_{15} (x,y,z) = \{(x,0), \(z,x)\}$$

$C_1 \equiv z > 0, \ C_2 \equiv z = 0$. $x$ and $z$ are the changed variables of $P_{15}$ and $P_{15}$ (and of $P_{15}^{c_1}$ and $P_{15}^{c_2}$). All three variables are significant to $P_{15}$ (and $P_{15}^{c_1}$), while only $x$ is significant to $P_{15}$. (Due to the clocking predicate, $z$ is also significant to $P_{15}^{c_2}$.)

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 $P_{15}^{c_1} + P_{15}^{c_2}$, $x$ 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 \(P_{10}^{2}\), and the scope of x and z consists of both processes. Processors seldom exist in isolation. They are implicitly combined with the external world, and we may model the external world as a process combined with the process for any given processor being studied, 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, those 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 two 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. 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. 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 over-write 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; set the value of the flag to zero. Similarly, the RECEIVE operation can consist of: test the flag until its value is zero; copy the new message out of the buffer; set the value of the flag to one. We may establish the adequacy of these operations in four steps: 1) The processes cannot deadlock,* 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

---

*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.*
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 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 tests the flag, finds it zero, and copies the message out of the buffer; the second tests the flag, finds it zero, copies the message out of the buffer, and resets the flag to one; finally, the first also resets the flag to one. Contrary to the stated intention, the message has gone to both processes.

To ensure the cooperation of groups of 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 [11] and Knuth [22] that the simple memory interlock available on virtually all computers is sufficient to achieve any desired mutual exclusions. However, the utter simplicity of the clocking process is at the expense of considerable complexity in each of the cooperating processes, and 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" [38], "Fetch and Modify Tags" [36], 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 [10]. These are the P and V operations, which affect integer-valued variables called semaphores. The operation V(S) increments the value of the semaphore S by 1; the operation P(S) decrements the value of S by 1 as soon as the resulting value would be non-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 most computer systems they result in more efficient use of the hardware than does busy waiting.)

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. 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, signalling 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 co-routine concept, introduced by Conway [7], and the message buffering system used by Brinch Hansen [3], involve somewhat less "primitive" synchronizing operations for facilitating implementation of "producer-consumer" relationships using 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 errors. For example, Brinch Hansen's system ensures that no process can interfere with a conversation between two other processes.

There is almost certainly no single best set of sequencing primitives. Even the comparatively primitive P and V operations will seem specialized and restrictive in a situation when, 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 [14] 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 [35].) 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 returned to briefly 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 finite 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 multi-processor 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 springs from the fact that not only must it provide process switching among independent programs, but must also supervise interprocess communication, as exemplified by the coroutine techniques discussed above.
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 behaviour 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 space to another. These interpretations are purely mathematical, whereas our previous use of the term involved a mapping from physical quantities into mathematical ones.

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 is one 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.

The rationale for studying such a large class of images, most of whose members do not exhibit the desirable properties of processes, is that we wish to study computer systems—whose visible behaviour often takes such difficult forms—in terms of (at least conceptually) 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 $f_8(x, y, z) = \{(x, x+y), (z, z-1)\}$, when $z > 0$. Under the interpretation $I_1(x, y, z) = (x, y, z)$ if $x \leq 0$ or $z = 0$, computations of $P_8$ map into single actions, i.e., the image process has the action function $f_8 I_1(x, y, z) = (y \cdot z, y, 0)$ 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 only yield insight 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. However, making every $n^{th}$ state observable would be no help at all!
Arbitrary interpretations may be constructed by composition of three basic types of function. Isomorphisms "lose" no information, but may be used to re-order the variables of the local state set, form "compound variables," (e.g., x+y), perform scaling, etc. State selection functions are identity functions whenever they are defined (the observable states), thus they "lose" states but not 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 any process which is isomorphic to it (provided the isomorphism is known). In particular, 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 the successor function $P_{13}(x,y) = (y, x+y)$, $S_{13} = \{(x,y) | x>0, y>0\}$, and $s_{13} = \{(1,1)\}$ under the interpretation $I_2(x,y) = (x+y,y)$. The image successor function $P_{23}^{I_2}$ $(x,y) = I_2(P_{13}(I_2^{-1}(x,y))) = (x+y^2, x/y+y)$, and the image computation is the sequence $<(1,1), (2,2), (6,3), (15,5), (40,8), ...>$. 

41.
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 re-applied until an observable state results.

**EXAMPLE:** Recall the process \( P_a \) with \( F_a(x, y, z) = (x+y, y, z-1) \) when \( z>0 \) and the state selection function \( I_a(x, y, z) = (x, y, z) \) if \( x=0 \) or \( z=0 \). The image of the observable state \((0, y, z)\) is \( F_a(0, y, z) = (0, y, z-1) \) if this is observable \((z-1=0)\), otherwise it is the image of \( F_a(y, y, z-1) \), i.e., it is \( F_a^i(0, y, z) = (y, y, z-1) \) for the first \( i \) which produces an observable state, namely \( i=2 \). Thus we deduce that \( P_a^{I_2}(0, y, z) = (y, z, y, 0) \).

In general, the image of a process under a projection function is not 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 \( P_a \) under the projection function \( I_a(x, y, z) = (x, z) \) is strongly contained in a process \( P_a^{I_2} \) with the successor function \( F_a^{I_2}(x, z) = (x+n, z-1) \) when \( z>0 \), where \( n \) is "free." Two computations of this process are \(<(0,2), (5,1), (10,0)> \) and \(<(0,2), (5,1), (6,0)> \). (Note that the first computation is an image of a computation of \( P_a \), while the second is not.) \( P_a^{I_2} \) cannot be used to prove that \( x \) will attain particular values in computations, but it does show that \( x \) never decreases, and that all

42.
computations of \( P_0 \) are finite.

Although isomorphisms preserve either the determinancy or nondeterminancy of a process, state selection functions preserve only determinancy 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 indeterminacies of the hardware (e.g., 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 realise 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., by 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 \( f_2(x, y) = \{(y, x+y)\} \) exactly specifies \( P_4 \). Its image under the interpretation \( I_5(x, y) = (x, \log_x y) \) has the action function \( f_5(x, y) = \{(y, y+1)\} \), and
exactly specifies the processor $p_B = (D_x, I_t \circ I_s)$.

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 behaviour 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 always decompose a 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 micro-program, which is in turn interpreted by the electronics of the computer...

The previous example is a conventional use of the term interpreter. Our rather broad definition also includes such techniques as procedures, interrupt systems, and address mapping hardware as special forms of interpreters. 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 behaviour of such a process will generally 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 in order to simplify the task of understanding its behaviour. This 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 a program to accept expressions, "compile" them, accept values for variables, and 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. [27]).

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:

(i) The ease with which the process required for the application can be realized using the basic processes and sequencing facilities that the processor provides.

(ii) The practicality of constructing or obtaining the processor itself.

(iii) The "efficiency" of the constructed processes.

**EXAMPLE:** It is, of course, these factors which rule out processors as simple as a 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 translates programs from "external" into "internal" form. A backplane 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 "micro-program 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 behaviour 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" operation system [12] 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."
4.5 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 [12]. 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 values of the semaphores. Now the abstraction (under this interpretation) of the combination will reflect their 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 non-deterministic 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 to structure 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 [12] and of Zorcher and Randell [35]. Both of these 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 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 [13]. The concept of multi-level 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 his 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." [12]

The use of multi-level processes described by Zurcher and Randell [35] 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 eventually 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.)

"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 behaviour 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." [20].
5. CONCLUDING REMARKS

Our aim in this paper has not been to develop yet another computational formalism. Rather, our efforts have been motivated by a belief in the profound importance of structuring in the design and implementation of complex computing systems, and of the need for a sound conceptual framework within which to discuss and develop structuring techniques. Two of the essential structuring techniques for processes seem to be combination and abstraction; our formalism permits the consideration of both within a uniform conceptual framework.

Our belief in the importance of structuring is based on 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 recognise that structuring in itself is not necessarily beneficial; bad or excessive structuring may be valueless or positively harmful.

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

The appropriate use of structure is still a creative task, and is in our opinion a central portion 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." [27]

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 [12] and Turski [33].

A useful guideline for the choice of effective levels of abstraction may well be implied by a characteristic of the "THK" system, recently discussed by Dijkstra [15]. He points out that, in retrospect, one of the crucial characteristics of the set of levels of abstraction used in the "THK" 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-microsecond time scale; level 1 performs memory management, using a drum with a 40-millisecond revolution time; level 2 communicates with the operator, on a time scale of seconds; level 3 performs peripheral assignments, every few minutes; 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 inter-level scheduling interactions. (A similar relationship between levels and time scales has been used by Lynch to structure the CHIOS operating system [26].)

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 behaviour 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 self-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 parallelism might well become actual parallelism (given appropriate processor facilities), with the result that considerable performance gains ensue.

Another example is a process which is defined 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 if it were all 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 two types of structuring 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, 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 whole 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 it 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.
6. ACKNOWLEDGEMENTS

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, 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 due to Lauer.

7. REFERENCES


