Abstract:

This note, using a definition of a sequential process which is based on the concepts of states and state variables, provides a common framework for the discussion of various existing techniques for allowing a set of sequential processes to interact. The techniques discussed include Test and Set-type operations, Dijkstra's P and V operations and the coroutine mechanism.

This working paper was prepared for the NATO conference on Techniques in Software Engineering held in Rome, October 27th – 31st 1969.

Interacting Sequential Processes.

by

Brian Randell

TECHNICAL REPORT SERIES

Series Editor: N.S.M. Cox

Number 4

November, 1969
RANDELL, Brian

Interacting sequential processes. By Brian Randell. (University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, No. 4).

Newcastle upon Tyne, University Computing Laboratory, 1969.

12"

Added entries

NEWCASTLE. University. Computing Laboratory. Technical Report Series. 4

Suggested classmarks (primary classmark underlined)

Library of Congress:
Dewey (17th):
U.D.C.: 510.7834
681.322.0

[Please note that the Dewey scheme is a totally inadequate indexing mechanism for this paper]

Suggested keywords

SEQUENTIAL PROCESSING
PROCESS INTERACTION
SEMAPHORE
CO-Routine
MULTIPROCESSING

Abstract

This note, using a definition of a sequential process which is based on the concepts of states and state variables, provides a common framework for the discussion of various existing techniques for allowing a set of sequential processes to interact. The techniques discussed include Test and Set-type operations, Dijkstra's P and V operations and the co-routine mechanism.

About the Author

The author is Professor of Computing Science in the University of Newcastle upon Tyne. His recent work has been concerned with the design of operating systems.
Introduction

Sequential processes and techniques by which they can be made to interact have been the subject of several recent papers, in particular Dijkstra (1965), Lampson (1968) and Saltzer (1966). These authors have been content to define the concept of a sequential process by appealing to the notions of 'program' and 'processor', a process then being defined as the activity engendered by a computer 'executing' a program. Each author then proposes various special operations to facilitate inter-process communication and synchronization.

The present note provides a rather different viewpoint on the nature of sequential processes, and attempts to identify the common ground which underlies the various proposed schemes for process interaction. It is in the main extracted from one of the sections of a report entitled 'Structuring Complex Processes' (1969) by J. Horning and the present author.

Sequential Processes

The above-mentioned report takes the concepts of state variables, state vectors and states as its starting point. State variables are quantities associated with a system which can assume certain well-defined values. A named set of state variables constitutes a state vector or local state set. An assignment of values to all the variables in a state vector defines a state; conversely, a state defines the values in each of the state variables. The set of possible states for a given state vector is a state space.

The report then defines a sequential process as a deterministic activity whose progress can be represented at discrete points in time by its state, i.e. by the values of a collection of state variables. A history of a sequential process is the complete sequence of its states following from some initial state. Three distinct, but equivalent, methods of specifying a particular sequential process are then:

1. enumerating or characterizing all of its possible histories. This collection is called its history set.
2. giving formal definitions for a local state set, a set of possible initial states and a successor function which, when applied to any state, gives the next successive state of the process.
3. exhibiting a deterministic device (which we call a processor) and an interpretation of its behaviour which indicates at what instants of time and by what means the device represents successive states of the sequential process.

It is this third method of specifying a sequential process which corresponds to the use of the term in connection with the execution of a program. The 'processor' is the program plus the computer on which it is being run - the 'interpretation' identifies which parts of the computer are to be regarded as registers holding data being manipulated by the program. (The discussion of topics such as self-modifying programs, program overlays, etc., and of what is meant by the 'creation' and 'deletion' of sequential processes is beyond the scope of this brief note).

Synchronous Combination

We approach the topic of general interaction of a set of sequential processes by introducing the concept of the synchronous combination of sequential processes. In order to facilitate this we introduce some special subsets of the state variables. For each state of a sequential process, the immediately changed variables are those whose value is different in the successor state. The changed variables of a sequential process are those which are immediately changed in any attainable state.

Under certain conditions (given below), a group of sequential processes, with possibly intersecting sets of state variables, may be joined by the operation of synchronous combination to form a single process. The set of state variables of this process is the union of the sets of state variables of the component sequential processes. The effect of the process on each of its state variables in each attainable state is defined as follows:

1. If the variable is not immediately changed by any of the set of component sequential processes, it is not immediately changed by the resulting process.

2. If the variable is immediately changed by one or more of the component sequential processes and they assign a common value to it, then the resulting process assigns that common value.
The requirement that a common value be assigned to changed variables is imposed to assure determinism of the resulting process. If this condition is always satisfied for all state variables and all attainable states of the combination, the set of sequential processes is termed conflict-free.

Synchronous combination is defined only for conflict-free processes.

This form of combination is termed 'synchronous' because the progress of the resulting process is determined by the composite effect of the component processes acting "at the same time". Although it appears to be a fairly restrictive means of combination, it will be seen that it can conveniently represent other, apparently more general, forms of combination.

It should be pointed out that synchronous combination results in a process whose effects cannot necessarily be viewed as the 'summation' of the separate effects of the component sequential processes. Therefore, the acceptability of a combination is not necessarily ensured by the separate acceptability of its components.

Co-Existing Processes

Processes which have been combined may be regarded as co-existing processes. Each operates on its own 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 concept of significant variables. A variable is immediately significant to a process in a state if some modification of its value results in a successor state which differs in some immediately changed variable from the successor of the unmodified state. The significant variables of a process (in an environment) are those which are immediately significant in one or more states which are attainable (in that environment).

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 co-existing processes may communicate with
each other or control each other. 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.

Although we have not explicitly mentioned the fact, processes seldom exist in isolation. They are implicitly combined with the external world, which may be viewed as a process which provides their inputs and (presumably) uses their outputs.

Parallelism

Synchronous combination provides a restricted kind of parallelism. Even this 'lockstep' form of combination is found in many useful systems, e.g. 'parallel addition' at the register level or the parallel operation of many plug-board controlled machines.

In more general types of process combination, however, not all processes are active all the time, and different processes proceed at different rates. We now show that our definition includes these more intuitive notions of combination and permits us to combine in a natural way processes with different speeds.

To talk sensibly about parallel processes, we must admit the existence of time. When one is considering an isolated process or a 'lockstep' combination of processes, time is adequately represented by the order in which states occur. The notion of rates of processes, however, implies some external clock by which to measure rates; the notion that processes are sometimes active (changing state) and sometimes inactive implies some means of external control. 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. We define the clocked extension of a sequential process by an enabling predicate as follows:

When the enabling predicate is true, the successor function of the extension is the same as that of the original sequential process; when the enabling predicate is false, the successor function of the extension is the identity mapping.

A clocked extension is again a sequential process and 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 rate at which the original sequential process, as represented by its clocked extension, proceeds in this environment may be
controlled by the clocking process. The rate of progress of the clocking process may itself be controlled by some other process. Rates for any number of processes may be controlled by a single clocking process. Alternatively, processes may mutually clock each other. This notion of clocking is quite general, allowing as special cases 'lockstep' operation, operation at fixed 'speed ratios', and processes putting themselves (or other processes) 'to sleep', or 'awakening' other processes (but not themselves).

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 processes themselves. Rather, we are concerned with the order in which events occur, and the relations among them. The progress of the clocking process thus supplies a sufficient representation of the passage of time, and its general nature allows us to 'subdivide' time as finely or as coarsely as we find useful. The choice of a particular subdivision will determine which changes in the state vector we view as happening 'in parallel' or 'in sequence'.

It will also determine the number of state changes which comprise a given activity, i.e. a transition from one given state to some other specified state. In our terms, the choice of a particular process or group of processes to perform the basic clocking function determines the time subdivision.

Different interpretations of a system define different processes that represent different views of the activity of the system. There will be no single best interpretation of a complex system, since many different viewpoints are required to provide adequate understanding of its internal structure and behaviour.

Example: The CDC 6600 (1964) can be thought of by the programmer as a high-speed serial CPU with 10 independent parallel PPU peripheral units. However, from the logic designers viewpoint, the CPU is composed of a number of separate units (e.g. floating add, floating multiply) which operate in parallel, while the 10 PPUs are implemented by a single set of hardware (with the 'barrel' serving as a clocking process 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.

Asynchronous Combination

To this point, we have discussed synchronous combination of processes, where strict ordering of actions (perhaps mediated by clocking processes) is obtained. In practical cases the clocking processes may be unknown or extremely complex. Such situations may be represented by asynchronous combination in which the component processes proceed at arbitrary rates. The resulting process will not be deterministic, since the state does not generally contain sufficient information to predict which process will next be active. It may, however, be equivalent (for example, in its output history) to a satisfactory deterministic process.

Asynchronously combined processes may or may not have any means of interaction, i.e. any shared state variables. It is advantageous to study the combination only if they do in fact interact. Dijkstra (1965) has used the term 'loosely connected processes' in such a case.

We continue to disallow the case where two or more processes make different changes to a state variable 'simultaneously'. This corresponds to a 'race condition' in a device, leading to unpredictable behaviour. Actual computing systems use various forms of interlocks, corresponding to enabling predicates, to prevent its occurrence. In our terms, if we wish to combine processes with common output variables, and cannot otherwise rule out the possibility of such conflicts, we can do so in the environment provided by a clocking process whose function is to resolve these conflicts.

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).

In general, the motivation for combining processes - synchronously or asynchronously - is to obtain their co-operation, i.e. behaviour which they would not exhibit separately. The resulting process may achieve an effect which it would be difficult (perhaps to the point of impracticality) to achieve with a single sequential process.
Example: On a computer with interrupt capabilities it is often much simpler to code I/O routines and a main program separately — letting the interrupt system switch among them as devices require attention — than it would be to code the same degree of buffering using in-line testing and I/O instructions in the main program.

It is in theory harder to obtain a desired degree of co-operation among asynchronously combined processes than among synchronously combined ones, due to the lack of knowledge about their relative rates. However in practice, for all but the simplest of processes, the knowledge that each process has completed the same number of steps (which would be the case with synchronous combination) is of little use in determining the progress of one process given a knowledge of the progress of the other process. Thus the co-ordination implied by synchronism must in general be supplemented by, or in the case of asynchronous combination be replaced by, explicit inter-process communication indicating relative progress.

Techniques for process interaction

It has been shown by Dijkstra (1965) and Knuth (1966) that the simple memory interlock of our earlier example is a sufficient means for achieving any desired amount of inter-process synchronization for purposes of co-operation. However, the utter simplicity of the clocking process is at the expense of considerable complexity in each of the co-operating processes, and involves a great deal of what Dijkstra has termed 'busy waiting', i.e. activity by one process whose sole purpose is to synchronize that process with the other processes while avoiding the peril of total blocking.*

Some reduction in the complexity (although not in the 'busy waiting') of the co-operating processes is facilitated in some computers. A basic instruction (variously known as 'Test and Set' (1968), 'Fetch and Modify Tags' (1966), 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 action.

* A situation where the further progress of each process is dependent on the further progress of some other, i.e. where all processes are waiting for each other.
Somewhat more complex, but rather more elegant, primitives for synchronization have been introduced by Dijkstra (1965). These are the P and V operations, which use 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 halt, and remain halted until some other process, by means of a V operation, enables it to proceed. Processes can be made to co-operate 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 gives the appearance but not the reality of progress, and the problem of determining the amount of progress made by each process is central to the problem of ensuring their correct co-operation.

Dijkstra describes two rather different uses for these primitives. The first he terms 'mutual exclusion', which is required whenever a process performs a sequence of operations on output variables which must appear to the environment as a single action — for example, the updating of several related elements of a shared data structure. The second use is to facilitate the implementation of 'producer-consumer' relationships among processes. Conway (1963) has introduced the concept of co-routines (a term whose similarity to 'sub-routine' is deceptive) to describe process combinations with these relations. He transfers the responsibility for buffering the transfer of information to the 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.

Summary

The methods of defining sequential processes, and their synchronous combination given in this brief note have provided a framework for describing the apparently more general concept of the asynchronous combination of two or more sequential processes. Furthermore, the concept of a clocking process has been shown to be a convenient common denominator of the memory interlock mechanism, the Test and Set type of instruction, Dijkstra's P and V operations, and the co-routine mechanism. Each of these mechanisms is seen as providing
a different allocation of the activity needed to achieve co-operation between
the set of sequential processes which are required to co-operate, and the
environment (provided by a clocking process) in which they exist.

Acknowledgements

As stated earlier, this note is in large part extracted from a
joint report which documents the results of a very pleasant collaboration
that the present author had with Jim Horning, of Stanford University, during
and following the summer of 1968, at the T. J. Watson Research Centre.
REFERENCES


   (Reprinted in Programming Languages (Ed. F. Genuys) Academic Press, London (1968)).


