Synchronization: is a synthesis of the problems possible?*

Gerald Belpaire**

Computer Sciences Department and Mathematics Research Center
University of Wisconsin-Madison

INTRODUCTION

Problems of synchronization among processes are encountered in both centralized and distributed computer systems. Conceptually, problems of the same nature arise in both applications (e.g. exclusive access to shared data), but, from the standpoint of implementation, techniques used to solve the problems are quite different.

This paper exposes an idea to define a framework within which this twofold reality will be reflected, together with a method to define with precision the problems of synchronization at their proper level of abstraction. As such, it attempts to give an answer to the question of understanding what characterizes a problem of synchronization, how it is related to a programming method and what are the aspects of its implementation.

PROCESSES AND EVENTS

In this section, we present synchronization rules at the lower level of abstraction as rules governing occurrences of elementary events. These rules correspond to the intuitive notion of "waiting" for something to become possible and "signalling" this possibility.

At first approximation, a process can be seen as a sequence of events defined by the execution of some actions on some data. A typical problem of synchronization is defined as a dependence of the occurrences of some events upon

---

* This work was supported in part by U. S. Army Contract DA-31-124-ARO-D-462.

** Author's present address: Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, New York 10012.

the occurrences of some other events. These dependences are included as coordination rules (e.g. the firing rule in Petri-nets) in all models of synchronization defined in the literature. In this paper, without loss of generality, the following representation of the coordination rules are defined.

To each event e, and at any instant t of time, we attach a binary value called the possibility of that event. The possibility of an event can be either possible or not-possible. If, at time t, an event is possible then it can occur within a finite time after t. Otherwise, it cannot occur.

The possibility of an event can be defined by a time-dependent predicate as follows:

"e is possible at time t" if and only if "$P(e)_t$ is true".

For example, if we consider two processes of three events each:

$$p_1 = (a_1\ a_2\ a_3)\ ,\quad p_2 = (b_1\ b_2\ b_3)$$

the possibility of a_2 can be:

$$P(a_2)_t = S(a_2)_t \wedge C(a_2)_t$$

where:

$S(a_2)_t$ = "a_1 has occurred"

$C(a_2)_t$ = "none of b_1 and b_3 has occurred or both have occurred".

S defines a sequencing rule (internal structure of p_1) and

C defines a coordination rule. In theory, it is not necessary to make a distinction between S and C. In practice however, the distinction is relevant and is therefore kept in the model.

It is possible to give a mathematical formulation of S and C with the following definitions.

Let E be the set of events and T the time (an ordered set of instants). A couple (e,t), $e \varepsilon E$, $t \varepsilon T$ is an occurence of e at time t. A set

$$E_t \subseteq E \times T_t\ ,\quad T_t = \{t' \mid t' < t\ ,\ t' \varepsilon T\}$$

is a timing of the events of E prior to t (i. e. an evolution in time of the processes).

A synchronization problem typically defines some restrictions on the timing E_t. For example:

$$B_1 = \exists t'((b_1 t') \in E_t) \quad B_3 = \exists t'((b_3, t') \in E_t)$$

$$C(a_2)_t = \sim (B_1 \vee B_3) \vee (B_1 \wedge B_3)$$

Since $P(e)_t$ can be an arbitrary predicate, this formulation is very general. But from this generality comes its disadvantage: it cannot be used in practice. It is therefore not proposed as a universal model of synchronization but rather as a foundation for more relevant constructs (cf. next section). The advantage of a common foundation is that it gives insight in the generality of the problem and it enables us to evaluate what restrictions are implied by the definition of ad hoc constructs.

There are in practice synchronization problems where the rule of coordination depends on the result of the operation performed when an event occurs (e. g. a re-read after a parity error). These cases should be provided for in a general model of synchronization. This aspect of the problem is not detailed in this paper.

To summarize, the following relevant concepts should be included in a model of synchronization:

1. A sequencing rule defining a static internal structure of the processes.
2. A coordination rule defining the dynamic interactions between processes.
3. Rules that make explicit the possible dependence of the coordination rule upon the computed result of some operation of a process.

With these concepts and with suitable formal definitions, it is possible to define any problem of synchronization at a very low level of abstraction.

EXCLUSIONS AND COOPERATIONS

Although the principles defined here above could be presented as a synthesis of synchronization problems, their practical applications are somewhat limited. It

would imply the manipulation of a predicate calculus on the set of occurrences of events. An improvement of this difficulty is to define, at a higher level, of abstraction, classes of problems as particular cases of the general low level case. Examples of these classes of problems are the problems of exclusions (i. e. synchronization of accesses to shared data and to re-usable resources) and the problems of cooperation (i. e. producer-consumer scheme, consumable resources, messages transmissions).

The problems of exclusions were treated in [1, 2, 3] and are not detailed here. The problem of cooperation can be illustrated by the following example. Let A and B be two critical sections (i. e. parts of programs whose executions are subject to synchronization) communicating through an infinite buffer. Typically, if A is the producer and B the consumer, the following property must be valid:

the number of portions consumed cannot
exceed the number of portions produced.

Or:

the number of executions (past and present) of B cannot
exceed the number of executions (past and present) of A .

If a_f and b_i are the termination and initiation events of resp. A and B , this becomes:

the number of occurrences of b_i cannot exceed the number
of occurrences of a_f .

i. e. :

$$n(a_f)_t + q \geq n(b_i)_t$$

q being a constant giving the number of initially consumable portions. This relation must be valid at any time t .

This is clearly a predicate on the occurrences of events determining the possibility of b_i . Predicates of this form characterize the problems of cooperation between processes and are particular cases of the general arbitrary predicate defined here above.

Combinations of such rules can lead to arbitrary complex problems of cooperation. For instance adding the following predicate for the possibility of a_i :

$$n(a_i)_t \leq n(b_f)_t + p$$

defines a finite buffer problem of length $p + q$. The practical character of such constructs is evident and has been emphasized (e.g. [12]) by the use of "invariants". These definitions are consistent with this other approach and therefore exhibit the same advantages.

PROGRAMMING TOOLS

It is beyond the scope of this paper to study in a comparative way all the programming primitives introduced to "solve" particular problems of synchronization.

Primitives are frequently defined by their implementation rather than by a suitable model expressing the intended synchronization effects. For this reason, different language constructs may appear fundamentally different although they actually describe the same effects and are therefore applicable to the same problems.

By using a system of definitions as presented in this paper, it is possible to draw a comparison between the different primitives and between their powers of expression. It is important to notice that synchronizing primitives are not definitions of synchronizing problems but only descriptions of the problems in some programming language.

The formal model must be considered as the semantics of the synchronization, and therefore of the primitives. The linguistic aspects of the primitives (e.g. ease of use, elegance, error-catching properties etc.) depend on language design decisions and cannot be evaluated by the tools defined here.

Examples of primitives are given in e.g. [1, 2, 3, 4, 5, 6, 11, 12]. Methods similar to the one defined in this paper were used to analyze synchronizing primitives in [3, 7, 9].

IMPLEMENTATION

The means of analysis defined in the three preceding sections are defined at a given level of abstraction where the issues of implementation are not represented. No conceptual differences are presented between synchronization among internal processes (belonging to the same computer system) or among external processes (belonging to different nodes of a network).

At the level of the implementation, however, operating systems synchronization and network synchronization will differ drastically. Assumptions about the hardware structure will require different approaches to the problem. Two classes of systems can be distinguished. First centralized computer systems where there is either a general interrupt facility and/or a general interlock mechanism for the memory (e.g. test-and-set instruction) . These cases are frequent in single processor operating systems. Second, networks where there are communications facilities and local interlock mechanisms. In the latter case, synchronization will be implemented by means of communication mechanisms that will be assumed to work properly. This means that synchronization and interprocess communication are at two different levels of abstraction: interprocess communications establish the media upon which synchronization is constructed.

Implementation of particular primitives can be based on decisions regarding the kind of problems to solve, the requirements of efficiency and the physical constraints of the system. It seems hopeless to try to design an implementation for the general arbitrary predicates on the possibility of events. It would certainly be unefficient, if ever possible. Until recently, it was believed that only very elementary synchronizing primitives could be implemented efficiently and this was presented as an argument against the definition of any new synchronizing tool. New researches [3,7] show that efficient implementation can be devised for more complex primitives.

CONCLUSION

The preceding considerations are showing that a synthesis of the current problems of synchronization is possible. The method suggested is based upon a definition of elementary coordination rules upon which higher level constructs are

defined. These definitions are not presented as a universal model of synchronization. On the contrary, they characterize a variety of problems connected to a common root of elementary concepts. A strict separation is maintained between abstract concepts and the practical problems of defining synchronizing primitives and of designing implementations.

The method is of a semantic nature, i. e. it attempts to represent intuitive concepts and to fit practical situations and it is not confined to a mere definition of formal objects.

The results obtained so far are promising and there is, as far as we know, no problem of synchronization that cannot be analyzed by this kind of approach.

REFERENCES

1. Belpaire, G. & Wilmotte, J. P. Semantic Aspects of Concurrent Processes. ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices 8, 9, pp 42-45, Sept. 73.
2. Belpaire, G. & Wilmotte, J. P. A Semantic Approach to the Theory of Parallel Processes. Proceedings of the International Computing Symposium 73, North Holland Pub. Co., pp 159-164, 1974.
3. Belpaire, G. On Programming Dependences between Parallel Processes. Technical Report #244, Department of Computer Sciences, University of Wisconsin-Madison, 1975. (D. Sc Thesis).
4. Dijkstra, E. W. Cooperating Sequential Processes. In Programming Languages. F. Genuys Ed., Academic Press, New York, 1968, pp. 43-112.
5. Brinch Hansen, P. A Comparison of two Synchronizing Concepts. Acta Informatica 1, 3 (1972), 190-199.
6. Vantilborgh, H., & van Lamsweerde, A. On an extension of Dijkstra's Semaphore Primitives. Information Processing Letters 1, 5 (October 72), 181-186.
7. Sintzoff, M. & van Lamsweerde, A. Constructing Correct and Efficient Concurrent Programs. MBLE Research Lab. Report R266. September 1974.
8. Lamport, L. A New Solution to Dijkstra's Concurrent Programming Problem. Comm. ACM 17, 8 (August 74), 453-455.
9. Lipton, R. J. On Sychronization Primitive Systems. Report of the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (1973).
10. Lipton, R. J. Schedulers as Enforces in Synchronization Processes. In Operating Systems, Proc. of an Int'l. Symposium, Rocquencourt, April 23-25, 1974, E. Gelenbe and C. Kaiser Ed., Lecture Notes in Computer Science 16, Springer-Verlag, Berlin-Heidelberg (1974).
11. Campbell, R. H. The Specification of Process Synchronization by Path Expressions. Ibidem.
12. Brinch Hansen P. Operating Systems Principles. Prentice-Hall, Englewood Cliffs, New Jersey (1973).