SEMANTICS OF COMMUNICATING PARALLEL PROCESSES

by

IRENE GLORIA GREIF

S.B., Massachusetts Institute of Technology, 1969

S.M., Massachusetts Institute of Technology, 1972

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September, 1975

Signature of Author .....
Department of Electrical Engineering and Computer Science, August 26, 1975

Certified by .....
Thesis Supervisor

Accepted by .....
Chairman, Departmental Committee on Graduate Students

SEMANTICS OF COMMUNICATING PARALLEL PROCESSES

by

Irene Gloria Greif

Submitted to the Department of Electrical Engineering and Computer Sciences on August 26, 1975 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

ABSTRACT

The thesis of this dissertation is that an understanding of the ordering constraints that are introduced among events of parallel processes is essential to the understanding of synchronization and that therefore any language for specifying synchronization of parallel processes should be based on a theory of such orderings. While it is possible to write specifications for systems of communicating parallel processes by reference to the time ordering of some global clock external to the system, such specifications cannot be as useful as ones which are in terms of orderings derivable within the system. Specifications should place constraints on intended behavior of the computer system itself rather than on the possible observations of the system's behaviors from some global viewpoint which may in fact be totally unrealizable.

The dissertation is a development of a specification language. It is based on a model of computation in which an individual process is represented by a totally ordered set of events. Synchronization properties of systems of independent processes are guarantees that in fact the set of events in the system can be ordered by a partial order which properly contains the union of the processes' total orders. This system ordering can be caused by the presence in a system of side-effect primitives or of synchronization primitives. Thus this model applies equally well both to busywaiting synchronization based on coordinated use of storage cells by independent processes and to non-busywaiting synchronization such as that induced by semaphores and structured synchronization primitives. In addition to applying to a range of types of synchronization, the specification language is also used to define a programming language. The meaning of a program is the specification of the behavior of the system into which that program is compiled. Specifications can be written for synchronization problems and for their implementations in terms of various primitives.

THESIS SUPERVISOR: Carl Hewitt

TITLE: Assistant Professor of Electrical Engineering and Computer Science

ACKNOWLEDGEMENTS

I would like to thank Carl Hewitt, Jack Dennis, Barbara Liskov, Benjamin Kuipers and Allen Brown for their contributions during the course of the research and the preparation of the final document.

This research was supported by the Office of Naval Research under contract number N00014-75-C-0522.

Contents

1 Introduction

The thesis of this dissertation is that an understanding of the ordering constraints that are introduced among events of parallel processes is essential to the understanding of synchronization and that therefore any language for specifying synchronization of parallel processes should be based on a theory of such orderings. While it is possible to write specifications for systems of communicating parallel processes by reference to the time ordering of some global clock external to the system, such specifications cannot be as useful as ones which are in terms of ordering derivable within the system. Specifications should place constraints on intended behavior of the computer system itself rather than on the possible observations of the system's behavior from some global viewpoint which may in fact be totally unrealizable.

In this chapter we describe the goals of our research. As the goals are outlined, they will be contrasted with those of related work in semantics of programming languages and parallel processes. We will explain why the underlying semantics of communicating parallel processes has not been well represented in previous theories as well as our reasons for chosing the model on which this research is based. Our theory of semantics is developed to be the basis for a specifications language. Therefore we include some discussion of what specifications are and how this work will differ from certain other work in semantics and verification. The chapter concludes with an outline of this document.

1.1 Physical Constraints on Computer Systems

Our theory of parallel processing, based on the mathematics of partial orders, can have

as its model any kind of parallel processing system. The kind of computer systems from which we have learned the most about underlying semantic concepts are multi-processor systems and computer networks. This is because they expose the importance of a part of the thesis of this dissertation which is that unmotivated assumption of global constraints should be avoided in describing computer systems. In timesharing on a single processor, since at some level of detail there really is only pseudo-parallelism, global views may be meaningful for representing certain properties of that system. By contrast, multi-processor systems, particularly distributed multi-processor systems on networks,[1] do not fit a global state theory as well. In fact, we will claim that part of the difficulty that has been encountered in formalizing properties of these systems arises from just this mismatch between the global state assumptions and the actual time and distance constraints in computer configurations. For most purposes, interesting relations among independent parts of a computer system must be realized in terms of some physical connection between them. It is reasonable to write prescriptive specifications relating operations on computers in, say, San Francisco and New York, if it is understood that some communication will have to take place to enforce it. It is not reasonable to describe an existing system in such a manner unless means for such communication exists.

Partial orders are appropriate for characterization of computer systems since, using them, orderings that must exist among certain operations can be expressed without the hypothesizing of total orders over all operations. Some operations may not be related due, for instance, to lack of physical connection between the processors on which they occur. Other operations, while they can be ordered as they occur, will in fact occur in unpredictable orders each time the computer system is run. Even though quantifying over all possible orders of operations in all parts of a system may in some sense be guaranteed to capture all possible properties, it can in fact obscure the important properties. The important properties will generally be the

---

[1] Or even timesharing systems at the level of abstraction at which users generally deal with them.

properties of ordering relations which are common to all of the postulated total orders. Such common properties should be abstracted from the set of all possible total orders and expressed directly, as a partial order.

1.2 Choosing a Model

The actor model of computation on which this specification language is based, has several properties which relate well to the physical realities of parallel processing. A single sequential process is represented by a totally ordered set of events. Systems of parallel processes are represented by partially ordered sets or events. These ordered sets of events are the behaviors of systems.

Events represent the receiving of a message by an actor. The event of receiving a message is dissociated from both the act of sending and the sender. This avoids the relativistic difficulties that can result from identifying time of sending a message with time of its arrival at its destination. This will be important to the physical reality of the ordering relations specified for actor systems, and will have implications for the realizability of specifications and visibility of properties of systems. For instance, the fact that messages can be guaranteed to be sent in one order does not in general imply anything about the order in which those messages can be expected to arrive at their destinations. In real computer systems the speed at which the messages travel would be a factor. In the actor model that factor cannot be relied on, making it impossible to write specifications for systems which could be realizable only in limited kinds of environments (e.g. on particular computer configurations.)

Individual actors are specified by causal axioms which are properties that will hold for any behavior of any actor system of which the particular actor is a part. They specify relations among events. These relations are then the properties of the behaviors as partial orders.

Besides the sound foundation for a partial order theory of parallel processing, the actor model has properties that are compatible with additional requirements that must be met for development of a specification language. The message passing metaphor together with the uniformity of objects in an actor system (all objects are actor) will be useful for treating objects as "black boxes" for purposes of definition of externally visible properties of systems. Also, when it is necessary to specify implementation approaches or program, we will rely on another feature of actor systems, namely, the flexibility of level of detail of descriptions of behaviors. Behaviors, being partially ordered sets of events, can be described at any level of detail. System properties are properties which at some particular level of detail are common to all behaviors.

1.3 Specifications

The results of the behavioral and causal semantics will be a language for writing specifications for communicating parallel processes. We will be emphasizing uses of specifications for aiding in the problem solving and programming process, etc. but will not treat derivation of specifications from programs, automatic generation of intermediate assertions, or formulation of proof rules for automatic verification. By the writing of simple high level specifications for commonly used terms in synchronization we will illustrate the compatibility of a specifications language about time orderings to our intuitions about systems. Fuller causal axioms of actor systems, specifying how they can affect behaviors of systems of which they are a part, will be shown to relate to important physical properties of computer systems and to determine their realizability. By ordinary reasoning, causal specifications of programs or program parts can often be shown to realize high level specifications. Capturing the meaning of parallelism will not necessarily require the specifications of full details of programs. However, given such details, we can prove compatibility with high level specifications to show relative correctness. Consistency of

specifications can also be investigated largely by relying on information about orderings. Synchronization can be described independently of programming language properties [Hoare, 1972], or implementation details [Milner, 1973] about systems.

In addition to the general need to accurately communicate both the desired properties of computer systems to be built (prescriptive specifications) and the actual properties of existing systems (as documentation, description, etc.), the main reason for wanting formal specifications is that they will then allow formal reasoning. In this area our objectives are somewhat different from those of others.

There has been a growing demand for the ability to prove "correctness" of programs which has lead to very close ties between language definition and "proof rules" for correctness, leaving expression of program meaning implicit in or entirely separated from the effort to represent the language semantics. For example, using Floyd or Hoare type assertional semantics [Floyd, 1967; Hoare, 1969], one defines a programming language by defining how its primitives relate assertions that are made about the programs' meanings. The language for the assertions, generally predicate calculus, is not the subject of verification research. Instead the effort concentrates on formulation of rules for going from assertions about a program to a theorem (or group of theorems) that must be true if the program is correct (relative to the original assertion). The rules are based on blind passing of assertions across statements, with structure of the program being considered only implicitly as the transformations on the assertions are made.[1]

Similarly, semantics based on mathematical approaches rests on relating programs to functions about which formal reasoning e.g. in Milner's LCF [1972], can be done to compare programs or to compare programs and descriptions of the functions they are to realize. Once the semantics are given, reasoning can proceed mechanically and be left to an automatic verifier. This is cited as an important part of the goal since, to quote Cadiou and Levy [1973], " Proofs carried out manually are probably just as likely to contain bugs as the original code."

---

[1] Note that proof rules do not tell you how to prove this theorem but rather how to find it.

Milner [1972] and Bekic [1971] also consider well-defined rules of composition as essential to semantics. For functions this seems to be an appropriate goal since the resulting object is also a function. However, in order to extend this to parallelism, one apparently must restrict oneself to very detailed, low level description. There are times when full details of composition of programs in parallel (represented as all possible interleavings of steps in each program) should be examined. However, this should not be the only representation of the parallel programs available in the formalism.

The goals of our research differ somewhat from the above in that we do not particularly intend that automatic verification be relied on for proving unduly long or unintuitive theorems. We are not even convinced that given formal descriptions of all parts of a complex system we automatically have a satisfactory description of their behavior as a whole. The specifications for whole systems, explicitly capturing their structural properties is the subject of interest. We wish to concentrate on the counterpart of the assertion language in the Floyd proofs to be sure that the language is well-suited for expressing the concepts on which the "meanings" of our systems are to rest. We would like to be able to talk about such meanings in varying amounts of detail and with some hope of being able to relate more detailed specifications to the high level goals expressed in the less detailed specifications. However, we will not expect the task of doing such reasoning to be in any sense mechanical. Neither will we require that it be any more precise than the usual mathematical logic.[1] One reason for this is that we do not believe that proofs should be carried out by machine in a form which is incompatible with natural human reasoning. While our arguments will not be trivially mechanizable, they hopefully will be understandable. Whatever progress is made in mechanizing mathematical reasoning can be put to use in an interactive system where programmer and "understander" work together in analyzing programs, only if the formalism is understandable to the programmer.

---

[1] Most proofs will be about properties of partial orders. We will rely on ordinary mathematical reasoning to prove them. This may be in contrast to what has come to be understood in verification work as formal, namely, checkable by machine.

Thus we will be developing a language for talking about meanings of systems of communicating parallel processes. In addition, by reference to interactions among parts of systems the causality involved can be captured. A rule for composition by interleaving all events is clearly available. Means for comparing high level synchronization specifications to properties which can be guaranteed to hold by causal properties of components will generally prove more important in programming and in analysis, since it relates abstractions to details. We do not expect the abstractions to be an automatic result of composition.

There has been some previous work on specifications. One effort is the work of Robinson and Holt [1975]. It relies on the notion of state described by an invariant on variables. Our reasons for preferring the behavioral approach are discussed in Chapter 6.

The work of Parnas [1972, 1975] on system modules is also meant as a specification language without consideration of proof techniques. It emphasizes what we refer to as external specifications, namely specifications of externally observable properties independent of such internal specifications as particular algorithm used for implementation or full implementation details. Internal definitions become more difficult to understand once side-effects as meaning are introduced. Most people know how to deal with side-effect-free function definition by algorithm (an internal definition saying how to compute the definition) by functional abstraction. It is less obvious how to extract the right side-effect or synchronization properties from a description of an implementation. Therefore techniques for external definition may become more important. We discuss to Parnas' work in Chapter 7.

1.4 Relation to Programming Language Semantics and Alternative Models

As long as programs are written for computers, programming language semantics will remain a subject of interest and importance. Specifications of binding, scope, environments and

closures constitute a large part of this subject. For certain kinds of programs, the mapping of programs and parts of programs onto functions, and the realization of a semantics with a well-defined rule of composition (see [Bekic, 1971; Milner, 1973]) is important. However, as interest in implementing programs on multi-processor computer systems increases it also becomes important for formal semantics to extend to programming languages which include primitives for parallel processing. As Milner [1973] has pointed out, when programs with side-effects include primitives for parallelism, specifications of program meaning must sometimes include more details than input/output specifications. For programming languages with primitives for parallelism and side-effects, I/O relations alone no longer fully describe a program. Two programs with identical I/O specifications might have quite different "meanings" as components of a system of parallel programs, depending on what side-effects they cause while computing output. In fact, it may be that the program is actually being used because of its side-effect and the effect that it can have on the computations of other programs. The I/O relation then becomes less interesting than the specification of these side-effects.

Our view is that, in order to write specifications for languages with side-effects and parallelism semantics of programming languages must reflect semantics of communicating parallel processes. Programs that are meant to be run in parallel should be analyzed in terms that apply to the most general multi-processing environment in which it could run, rather than by artificial mappings onto other formal objects such as functions which may bear no relation to the physical realities of communicating parallel processes. In developing our specification language we will emphasize development of terminology for direct expression of communication among parallel processes rather than programming language issues. When a programming language is formally defined it will be in terms compatible with specifications of parallelism. This will produce no new results for applicative parts of languages, but will allow direct reliance on the theory of parallel processing in describing imperative, synchronization and parallelism primitives of the

language. This approach is a deliberate departure from that of modifying or extending previous work on semantics of sequential programming languages to apply to additional primitives for side-effects and parallelism.

For instance, representations of one side-effect primitive, a cell of read/write storage in a single processor system, when extended to multi-processing environments, can result in a limited representation of a cell's possible uses. Burstall [1972] has proven the correctness of a program for reversing a list. The list is implemented by pointers and cells. In the style of Floyd [1967], he places assertions on programs. His assertions maintain a description of storage cells and pointers with their interpretations as lists. Changes in pointers (values of cells) can be reflected in this description. By tracing pointers from list names one can see the overall result of any single change in value of a pointer. Thus this is a model based on the assumption that there can be a complete description of the state of memory at all times.

Using this approach for parallel processing requires the recording of several possible descriptions -- one for each possible interleaving of the steps of each process. This has the problem of causing a combinatorial explosion in the amount of information that has to be available to specify the meaning of a parallel processing system. Also, it may not represent clearly which information is important to the behavior and which is not. Some changes to storage are important enough to cause further changes in the behaviors of other processes.

Similarly, the extension of mathematical semantics to parallelism, as in Milner [1973], represents parallel composition of processes by the set of all possible interleaving of steps of all the processes. This notion of the meaning of two programs run in parallel can capture some aspects of non-determinism. However, since the only expression of the meaning of parallel programs is in the set of all detailed histories, it is difficult to abstract a property which holds for all those histories. In general, if a system has interesting properties, they will take the form of simpler properties true of all the histories. They may not be at all obvious in the details of

interactions. This is true even if the details must occasionally be examined to prove that the common properties hold.

Milner's approach also has the property that further uninteresting details must always be included to specify how the interleaving is to be accomplished. Parallel processes are run in pseudo-parallel in accordance with oracles which are infinite strings of "process names." Each time the oracle is questioned (examined) it designates the next process to be run. Only fair oracles should be consulted (where fair oracles are ones that always give each process another chance in the future). If new processes are created, the creation of a new oracle which includes the new process will have to be done explicitly.[1] As a result, while the method can be used to state and prove properties of busywaiting synchronization programs [Cadiou and Levy, 1973], there is no means for expressing synchronization without details of busywaiting. We consider it to be important that a theory of parallel processing encompass both busywaiting and forms of synchronization without busywaiting as in systems built from semaphores.

The main approach to parallelism not mentioned so far is Petri nets.[2] While they may afford some insight into causality by graphic representation of interconnections, we contend that the understanding of those interactions is be founded on the same notions of ordering and causality as are used here. Particularly since there is no obvious means for increasing or decreasing amount of detail shown in a net, it may be convenient to make any abstractions about high level properties of nets in statements in another language such as ours. What is more, many assumptions about physics implicit in the Petri net model make it impossible to describe certain things (and perhaps make it too easy to describe possible unrealizable things). A particularly

---

[1] Milner expects that extensions of the work of Scott [1972] (on which Milner's approach is based) to relations will eliminate the need for oracles, but the currently available papers report only the work which depends on the oracle.

[2] One source of references on Petri nets, as well as a short discussion is an Appendix of Tsichritzis [1974].

notable limitation is the inability to refer to cells of storage (other than those with complex indivisible access operations, namely semaphores) which in fact can play a role in synchronization. Although a model which rests entirely on busywaiting for expression of synchronization was the object of criticism above, the alternative of not being able to express busywaiting at all is clearly not acceptable, since busywaiting does arise and must be dealt with at some level of detail in many computer systems.

1.5 Informal Specifications

The area we are moving into, systems of communicating parallel processes, has certainly not been at a standstill due to lack of a formal specification language. On the contrary, many problems have been posed and solved. Some have even been proved "correct" relative to informally stated specifications.[1]

Thus part of our task will be to sort out these informal specifications and to see whether we can make acceptable formal statements of their meanings.[2] This will lead to some further discussion of exactly what limits our choice of model may have placed on the kinds of properties that can be specified. We do not write specifications based on speed of machines, relative ordering of arbitrary (possible unrelatable) occurrences, or relative speeds of implementation. The kinds of properties that can be stated are properties which, if they hold, hold independent of how processors run or where they are located. We will see that informal specifications often confuse such specifications with properties that cannot be invariant over implementations. While both can be of importance, they require separate formulation. This is

---

[1] Even if primitives in programs are specified formally as in Haberman, [1970], the specifications of problems are so closely related to the primitives used in the solution that they could scarcely be called problem specifications.

[2] Clearly this is the best we can do since no precise statement was made previously.

analogous to semantics of programming languages. Programming languages and therefore the programs written in them are generally considered to have some meaning independent of specific compiler used. Compiler dependent properties may also be important and in fact programmers often rely on them. Thus it may be important to be able to express those properties. However, unless the nature of these properties, i.e. compiler dependent, is made explicit, a program can be proved correct and then when used in a different environment (e.g. compiled by a different compiler) have incorrect behavior. We are carefully making the same kind of distinction in starting to write specifications of parallelism and are concentrating our first effort on environment invariant properties.

1.6 Presentation

Chapter 2 is an introduction to the actor model of computation and the definitions of many of the terms which will be used in the following chapters. Chapter 3 contains specifications of a side-effect actor, the cell, and an example of how the cell can be used for communication of time ordering information between two processes. An example of another kind of actor which can affect time ordering is presented in Chapter 4 along with high level specifications of a commonly required kind of synchronization, mutual exclusion. At the end of Chapter 4 there is a review of the uses that have been made of partial orders to that point.

Chapters 5 and 6, through the example of the readers/writers problem, illustrate the different kinds of specifications one might encounter in analyzing a single problem. They range from high level problem specifications to detailed program specifications. The program in Chapter 6 is written in terms of a so-called structured synchronization primitive, and therefore its presentation is followed by some discussion of what structure this might refer to.

Chapter 7 contains definitions of two kinds of semaphores. Examples of

implementations of synchronization actors using semaphores are used to relate the structures of problems (as described in earlier chapters) to structures of programs. In Chapter 8 we consider the adequacy of the the completeness of specifications of programs and primitives on which judgements about power of primitives rest. It also brings us to recognition of areas for related future research into specifications of run-time environment dependent properties of synchronization.

Chapter 9 is a return to the subject of cells for synchronization using busywaiting. It contains proofs of properties of two previously published programs by Dijkstra and Knuth.

While it is important that there be a formal definition of a language for formal arguments about programs which are implementations of specifications, that definition will be presented in the Appendix since for the most part, in this dissertation, the formal language definition serves only the function of formality and will add no insight into parallel processing. Examples will be written in an Algol-like language. Reasoning about those programs will rely on informal definition for all primitives except variables (cells) and synchronization primitives. We hope that this use of familiar programming notation will ease the reading without suggesting inappropriate models of computation. In the Appendix a subset of PLANNER-73 [Hewitt, 1973] is defined. It is a small but powerful language for parallel processing. It includes the particular side-effect and synchronization primitives that are defined formally in this paper.

2 A Model of Computation

The properties of computer systems which interest us are easily formalized in the actor model of computation. The model is presented in this chapter with references to the sources of the concepts on which it is built and explanation of their importance for the purposes of this research. The final section is a preview of the way in which the thesis research builds on the actor model to develop a specification language for systems of communicating parallel processes.

2.1 Actors

All objects are called actors. In general, collections of actors will be used to represent computers systems. Such collections are actor systems. We will often refer to an actor system as an actor since at some level of abstraction a collection of actors is a single object. Activity in an actor system is in the form of one actor being sent to another actor. The sources of energy for these transmissions are referred to as activators and correspond roughly to processes. Any group of actors can be referred to as an actor system. An actor system can have arbitrarily many activators providing energy for the transmissions.

An actor can be thought of as a script to be followed by an activator. The basic activity described in a script is the passing of one actor (the transmitted actor) to another (the target). The activator does this and then follows the script of the target, with information from the transmitted actor available. By virtue of the fact that a script can call for transmissions among other actors, it is clear that an actor also implicitly "knows about" other actors. The actors in any actor system may know about other actors in the system and about other actors not in the system.

Activators are assumed for our purposes to be unlimited sources of energy. Thus we will always be in a multi-process world in which no scheduling of processes onto a smaller number of processors has to be done. The advantage of this property of the model for the purposes of specification of systems of parallel processes is that it will eliminate the possibility of depending on properties such as the number of processes in specifications or proofs of properties of specifications.[1]

2.2 Behaviors

An actor system is characterized by its behavior where a behavior consists of the events that take place and the causal relations among those events. All events in any actor system are transmissions of an actor to a target actor by an activator. These events can be characterized as follows:

> Definition: An event is a four-tuple, $\langle t\ m\ \alpha\ ec \rangle$ where t is a target actor, m is the transmitted actor, α is an activator, and ec is an integer called the event count.

The event count is required in order to distinguish among sets of events of one activator which involve identical actors. The events of a single activator can be considered to be totally ordered. Therefore such events are distinguished by their places in the total order of events of the activator α and are represented by events $\langle t\ m\ \alpha\ ec_1 \rangle$ and $\langle t\ m\ \alpha\ ec_2 \rangle$ where $ec_1 \neq ec_2$. What is more, if there are two events $\langle t_1\ m_1\ \alpha\ ec \rangle$ and $\langle t_2\ m_2\ \alpha\ ec \rangle$ then $t_1 = t_2$ and $m_1 = m_2$. I.e. there cannot be two different events in the same place in the order of a single actor.

If $ec_1 < ec_2$ then $\langle t\ m\ \alpha\ ec_1 \rangle$ --> $\langle t\ m\ \alpha\ ec_2 \rangle$. The ordering --> is referred to as <u>the activator ordering</u>. At a particular level of detail the count after ec is ec'.[2] Thus when two events

---

[1] Such dependencies can exist and may be relied on, but they are not in the scope of this research.

[2] Occasionally, when several sequential events are referred to, the number of primes will be used so that ec''' is represented by $ec^{(4)}$.

occur successively (with no intervening events at a particular level of detail) they are represented by $\langle t_1\ m_1\ \alpha\ ec\rangle$ --> $\langle t_2\ m_2\ \alpha\ ec'\rangle$. The event $\langle t_1\ m_1\ \alpha_1\ ec\rangle$ is said to be an event of activator α_1. If E_1 and E_2 are of activators α_1 and α_2, respectively, where $\alpha_1 \neq \alpha_2$, then E_1 and E_2 are events of different activators. Since any set of events in α is totally ordered by -->, we can define the past of the activator α with respect to an event E with event count *ec* to be the totally ordered subset of those events with $ec_1 < ec$. Similarly, the future can be defined for any E. These terms can be applied to any totally ordered set of events.

Activators and/or actors can be "created in events." If $\langle t\ m\ \alpha\ ec\rangle$ is an event in which α_1 is created, then $\langle t\ m\ \alpha\ ec\rangle$ --> $\langle t_1\ m_1\ \alpha_1\ ec_1\rangle$ for all t_1, m_1, ec_1. Details of how the activator is created are not dealt with in this model.[1] An example of at target actor t, which can cause creation of an activator is a fork actor which creates new processes. Section 2.6 contains specifications for such an actor.

There is similar lack of detail about creation of an actor. It can best be understood by analogy with the closure in programming languages. Interpreting code for a lambda expression ((λ(x) (λ(y) (...))) g), involves binding x to g and using the closure of (λ(y) (...)) with that binding in future computation. Similarly, a target actor's script can contain an actor-creating script, analogous to (λ(x) (λ(y) (...))), but the creation of the closure actor cannot occur until the target actor is sent a message to which it can bind variables. The target actor can create a different closure for every message it receives.

Activator orderings for all the activators form a partial order on all events in a system. The same symbol --> is used to denote the transitive closure of the union of the total orderings for each individual activator and the relations between events of different activators due to creation of an activator. It is still referred to as the activator ordering. A behavior is a set of

---

[1] Thus in an implementation of a computer system that corresponds to an actor system, one must decide on a primitive for process creation and depend on it.

events partially ordered by -->. For an actor system, a set of n initial events with targets in the actor system and with n distinct activators (the initial conditions) will begin one or more behaviors. Any property common to all behaviors and all initial conditions can be considered a property of the actor system. Thus if we refer to properties of an actor's ordering or of an actors's behavior, we refer only to properties that are common to all orderings of all behaviors of that actor.

2.3 Continuations

The basic activity of message passing operates without any control superstructure. There is no implicit memory of what actor or actors caused any given transmission. Thus control structure must be provided for explicitly. The convention for including control information as in the control structure of functions and returns is to structure the transmitted actor in the following way:

(apply: message (then-to: continuation)).

The actor **message** is the argument to the function and **continuation** is the actor to which another actor can be sent. Thus the usual programming language function call with implicit return, as in the call to f in $1 + f(x)$, can be implemented by explicitly sending the continuation to the function along with its arguments, as described in the event:

<f (apply: x (then-to: (λ(y) (1 + y)))) α *ec*>.

The explicit continuation, (λ(y)(1 + y)), is an actor which will take the result of $f(x)$ and add it to 1. This does indeed correspond to the intended computation of $1 + f(x)$.

In addition to functional behavior more general control structures can be represented.

For instance, an actor might create a new process α to do some printing, not caring to hear from that process again. In that case an actor without continuation might be sent:

<print (apply: 3) α *0*>.[1]

There are actors which even when sent an explicit continuation, can choose to ignore that continuation and instead transmit actors to some favored actor of their own. This can produce a control structure similar to a jump out of a procedure body.

The continuation can be important in programming language semantics as demonstrated by Reynolds [1972], Fischer [1972], and others. A programming language definition using continuations is given in the Appendix. The control structures which will be used the most in the following chapters are simply function (procedure) calls and sequential execution of statements in a program. The second example in Section 2.6 illustrates the representation of sequential code in continuation semantics. This example is important since all discussions of "behaviors of programs" in this paper assume translation of sequential code to continuation semantics.

For the purposes of the development of the specification language, the importance of the continuation is the fact that it allows one to take the event view of behaviors, without any other mechanism. A control structure such as sequential execution of statements in a program can be expressed [as well as jump, coroutines, and parallel processing] without reference to program counters or other devices. The ability to characterize systems by orderings on events and thus most of the work to be reported in this paper, rests heavily on the incorporation of the semantic concept of the continuation into the actor model.

2.4 The Data Abstraction

---

[1] The first event in an activator has event count *0*.

Another programming semantics concept which has been developed independently and which is relied on in the actor model is the data abstraction. The cluster in CLU [Liskov, 1974] and the class in SIMULA [Dahl, 1972] are instances of data abstractions in existing programming languages. In a data abstraction, data objects are defined by specifications of all relevant operations and properties of that data rather than by such details as how they are to be stored in a machine. Actors are defined by the relations they introduce into a system depending on the messages they receive. Thus if the different messages to which an actor can respond are considered as procedures or properties in that actor, then the actor can be seen very naturally an a class-like object.

The concept of data abstraction will be relied on in the writing of specifications. Chapter 6 is about a programming language primitive based on the data abstraction, namely the monitor [Hoare, 1974].

2.5 Level of Detail

In describing the behavior of a system one has to decide on a level of detail of interest. Only events which represent activities deemed of interest for the current purposes are included. Although one standard use of level of detail is to successively fill in more and more details of implementation, levels of detail of descriptions of a single actor system do not necessarily form a linear hierarchy. If the details of interest in two different descriptions of an actor are not related, then the behaviors at the corresponding levels of detail may be incomparable.

The "filling in of details" way of thinking of levels of detail is as a choice between information about input/output properties of an actor and information about the means for producing that input/output property. Thus an actor, t, which is generally expected to receive a message and a continuation could be described at the I/O level of detail by describing the

message sent to the continuation in the "next" event. To describe the actor in more detail would require the filling in of other events between the event with the actor t as target and the "output" event. The further level of detail could simply be a direct implementation revealing techniques used (such as storage in local temporaries, etc.) or it could expose unsuspected behavior such as some side-effect of the computation.

There is another way to think of variation in level of detail which is useful for actors that don't have behavior of functions with the well-defined notion of I/O relation. This involves the notion of a boundary around a collection of actors. There are actors $\mathfrak{A}$ in the and actors $\mathfrak{B}$ out of the collection. The highest level of detail of description is one in which only events corresponding to crossings of the boundary are included. This includes only events $\langle t\ m\ \alpha\ ec \rangle$ in which $t \in \mathfrak{A}$ and $m \in \mathfrak{B}$ or vice versa. Also, for two successive events $\langle t_1\ m_1\ \alpha\ ec \rangle$ --> $\langle t_2\ m_2\ \alpha\ ec' \rangle$ if $t_1 \in \mathfrak{A}$, $m_1 \in \mathfrak{B}$ then $t_2 \in \mathfrak{B}$ and $m_2 \in \mathfrak{A}$. This requirement would prevent inclusion at high level of non-essential events like transmissions of the original message to some other internal actors. Thus the events included at highest level represent a conversation between the actor system and the outside.

The way to examine an actor system in more detail is to choose a set of distinguished actors within it and to include all events involving the distinguished actors. Once specifications can be written at varying levels of detail one can define equivalence of actor systems relative to a set of distinguished actors. Then one can distinguish among actor systems which are equivalent (i.e. have the same behaviors) at a high level of detail, according to whether or not they are equivalent in further detail. Two actor systems are equivalent with respect to some set of distinguished actors if and only if they have identical behaviors with respect to that same set of distinguished actors. The choice of distinguished actors can make a difference in equivalence of actor systems. For example, two programs for factorial can be equivalent at the level of input-output transformation (i.e. with no actors distinguished) even if a larger set of distinguished

actors would reveal one to be a "counting-up" factorial and the other to be "counting-down." Consider computation of 4! by both of these factorial actors. If multiplication, represented by *, is considered to be a distinguished actor then in any behavior of one factorial there might be a series of events showing successive multiplication of the partially computed value by the numbers 1, 2, 3, 4:

<* (apply: [1 1] (then-to: c_1)) α ec>
-->
<* (apply: [2 1] (then-to: c_2)) α ec'>
-->
<* (apply: [3 2] (then-to: c_3)) α ec">
-->
<* (apply: [4 6] (then-to: c_4)) α *ec'"*>.

In the other there might be a series showing multiplication by 4, 3, 2, 1:

<* (apply: [4 1] (then-to: c_5)) α ec>
-->
<* (apply: [3 4] (then-to: c_6)) α ec'>
-->
<* (apply: [2 12] (then-to: c_7)) α ec">
-->
<* (apply: [1 24] (then-to: c_8)) α *ec'"*>.

2.6 Specifications of Actors

The properties of a particular actor are specified by axioms which must be satisfied by behavior of any actor system which contains that actor. Thus the actor **add1** corresponding to an instruction in some programming language for computing the successor function has behavior specified by the axiom:

If the event

<c>add1 (apply: x (then-to: c)) α ec</c>

is in a behavior then, the next event in that activator α is

<c>c (apply: z) α ec'</c>

where z = x + 1.

That is, in any behavior of any actor system which contains **add1**, if <c>add1 (apply: x (then-to: c)) α ec</c> is in the behavior then so is <c>c (apply: z) α ec'</c>.

This will often be written as either

<c>add1 (apply: x (then-to: c)) α ec --> c (apply: a) α ec where z = x + 1</c>

or as

<c>add1 (apply: x (then-to: c)) α ec</c>

causes

<c>c (apply: z) α ec'</c> where z = x + 1.

This last form can best be understood if a behavior is thought of as a dynamically increasing set of events. Then the axioms tells what events can be added to that set next. The presence of

<c>add1 (apply: 4 (then-to: c)) α ec</c>

in the behavior, means that

<c>c (aply: 5) α ec'</c>

can be included in the set as well.

This axiom distinguishes add1 from arbitrary functions f for which one can only state

what the next event will be, if there is one. This next event cannot be guaranteed to occur. In other words, addition has been defined as a function which always terminates, whereas, in general, functions may or may not terminate. For functions which are not known to terminate, and the most that can be said is that if <f (apply: x (then-to: c)) α *ec*> is in the behavior, then if there is a next event at this level of detail it will be <c (apply: z) α *ec'*>.

Notice that in the statement of the axiom the fact that actor z can be characterized simply by the mathematical notation z = x + 1 has been used. In general, actors which appear in events in axioms can have arbitrary behavior. When that behavior is not conveniently expressible in conventional notation, its properties must be specified by axioms for its behavior.

The following actor, (λ(x) ((f x) ; (g x)$_{\alpha}$; (h x))) causes events which involve newly created continuation actors. These continuations are in turn defined by axioms about their behaviors. Intuitively, this actor sequentially performs the function calls (f x), (g x), and (h x) after a value to be bound to x is sent to it. The axioms for this actor are:[1]

<(λ(x) ((f x) ; (g x) ; (h x))) (apply: z (then-to: c)) α *ec*>

causes

<f (apply: z (then-to: c_1*)) α *ec'*>

where c_1 is defined by:

---

[1] In axioms, symbolic names of newly created actors are marked by an asterisk in the first events in which they appear. Composite actors will not be so marked. For example, if c_1 is a new actor then the actor (apply: x (then-to: c_1)) must also be newly created since it "knows about" the newly created actor c_1.

$\langle c_1$ (apply: x_1) α $ec_1\rangle$

causes

$\langle$g (apply: z (then-to: c_2*)) α $ec_1'\rangle$

where c_2 is defined by

$\langle c_2$ (apply: x_2) α $ec_2\rangle$

causes

$\langle$h (apply: z (then-to: c)) α $ec_2'\rangle$

Thus the continuation of each statement (i.e. each function call) is the "rest of the sequence of instructions." The value (if any) computed by the preceding statement is ignored, but the value of the last statement is treated as the value of the sequence and returned to the original continuation.

The following actor definition includes creation of new activators and illustrates a type of causal relation by which sets of events are related. It is a definition of the way in which the function (λ (f g) ((f x) • (g x))) computes (f x) and (g x) in parallel once it is sent a pair of functions [F G] to which to bind f and g. On receiving the values of the computations of (f x) and (g x), it adds the two values. Thus transmission of a message to the actor (λ (f g) ((f x) • (g x))) causes two events. The axioms are:[1]

---

[1] The notation [x_1 x_2] indicates an actor which knows about (in the sense that its script refers to) a sequence of other actors, x_1 and x_2. It can distinguish between them in the sense that it knows that x_1 is first and x_2 is second. [See Appendix]

$\langle$(λ(f g) ((f x) • (g x))) (apply: [F G] (then-to: continuation)) α *ec*$\rangle$

causes

$\langle$F (apply: x (then-to: c_1*)) α_1 *0*$\rangle$ and $\langle$G (apply: x (then-to: c_2*)) α_2 *0*$\rangle$

The newly created actors c_1 and c_2 are defined by:

$\langle c_1$ (apply: v_1) β_1 $ec_1\rangle$ and $\langle c_2$ (apply: v_2) β_2 $ec_2\rangle$

cause

$\langle$• (apply: [v_1 v_2] (then-to: c)) α_{β_1,β_2} *0*$\rangle$.

These axioms illustrate several points, some conceptual, others notational. The event count of *0* is used to indicate that the activator is newly created and that this is its initial event. The use of new activator names, α_1, α_2, β_1 and β_2, in the axiom about c_1 and c_2 is necessary for the general case in which additional parallelism is introduced in the computation of (F x) or (G x). (F x) or (G x) could generate several activators operating in parallel. The actors c_1 and c_2 have the property that for <u>every</u> pair of values returned, application of the function • to those values will proceed. In each case the continuation actor used is the same, namely, c, the one which was originally sent as the explicit return point to the function λ (f g) ((f x) • (g x)). However, should it be the case that several different applications must proceed it is necessary at least abstractly to provide for multiple activators to carry them out in parallel.[1] Thus the activator α_{β_1,β_2} is needed to carry out the application based on the evaluations reported from processes β_1 and β_2.

An implementation of these actors might actually represent a special case such as the

---

[1] Otherwise these axioms would be inconsistent in the following sense. If the events $\langle$• (apply: [v_1 v_2] (then-to: c)) α_{β_1,β_2} *0*$\rangle$ were simply required to be the next event in α as in $\langle$• (apply: [v_1 v_2] (then-to: c)) α *ec'*$\rangle$, then the axioms could require two events $\langle$• m_1 α *ec'*$\rangle$ and $\langle$• m_2 α ec'$\rangle$ where $m_1 \neq m_2$.

following one. Process α might create a new process α_2 to do (G x) while doing (F x) itself. Thus $\alpha_1 = \alpha$. Also, the computations might create no further parallelism in which case eventually the events $\langle$F (apply: x (then-to: c_1*)) α_1 0$\rangle$ and $\langle c_1$ (apply: v_1) β_1 $ec_1\rangle$ occur in α and $\langle$G (apply: x (then-to: c_2*)) α_2 0$\rangle$ and $\langle c_2$ (apply: v_2) β_2 $ec_2\rangle$ in α_1. In this case α_{β_1,β_2} might be α itself resuming computation.

2.7 Time Orderings

The actor model can be called a behavioral or event-oriented approach to semantics. The most important feature for the purposes of semantics of parallel processing is that with this approach, a system is characterized by partially ordered sets of events. This feature is so important because it serves as a basis for characterization of systems of parallel processes by partial orderings other than the activator ordering. As stated in the introduction, the underlying concept of synchronization is the imposition of additional ordering constraints in a computer system. This section is about some kinds of orderings on actor systems which are important to synchronization semantics.

An important ordering for expressing high level properties of parallel processing systems is the *time ordering*, written =>. Using it one can state desired properties such as "E_1 must always happen before E_2" which is stated "$E_1 \Rightarrow E_2$." Another property,"E_1 and E_2 cannot be allowed to happen in parallel" can be stated "$E_1 \Rightarrow E_2 \vee E_2 \Rightarrow E_1$." Since => is meant to correspond to some physically meaningful notion of ordering in time, it will be assumed that it is not possible that there are events, E_1 and E_2 such that $(E_1 \Rightarrow E_2 \wedge E_2 \Rightarrow E_1)$.[1]

A kind of time ordering property which will be stated frequently in this paper is the ordering of sequences of events.

---

[1] Proof that a set of specifications for an actor would cause such an ordering is proof of inconsistency of the specifications in the sense that no physical device could realize the specifications.

Definition: S is a sequence of events if $S = \{E_1, \ldots, E_n\}$ and $E_1 \dashrightarrow \ldots \dashrightarrow E_n$.

Definition: $S_1 \Rightarrow S_2$, where S_1 and S_2 are sequences of events $\{E_{1_1}, \ldots, E_{1_n}\}$ and $\{E_{2_1}, \ldots, E_{2_m}\}$, respectively if $E_{1_1} \dashrightarrow \ldots \dashrightarrow E_{1_n} \Rightarrow E_{2_1} \dashrightarrow \ldots \dashrightarrow E_{2_m}$.

This ordering is used in specifications for several kinds of synchronization.

Using => one can write arbitrary time ordering specifications, without concern for their realizability. As a part of the specification language, => is useful for writing high level prescriptive specifications, expressing desired ordering properties. However, as emphasized in the introduction, it is also important to be able to express the means through which orderings can be achieved. This must be expressed in such a way that its relation to the high level specifications can be examined, and so that realizability can be analyzed.

A first example of an ordering which can be used to realize time orderings is the activator ordering -->. If $S_1 \Rightarrow S_2$ is specified, it may be realized by $S_1 \dashrightarrow S_2$, i.e., by building a single processor system which first does S_1 events, then does S_2 events.[1] The activator ordering is an enforcible ordering and thus can be used to realize desired time orderings. Every system has its own time ordering (also denoted by =>) derivable from ordering information such as -->. If this derived => has the specified properties then the system realizes the high level specifications.

Another physically meaningful way to build orderings is based on communication of parallel processes through common actors. For instance, if events E_1 and E_2 are of different activators, but both have the same target, t, then it may be physically meaningful to order those events. It will be possible to order them whenever the causality of the actor t (as embodied in its axioms), is based on the order in which t receives messages. An example of this kind of actor is a cell of read/write storage. It causes behaviors to reflect the order in which messages were received by the cell. Even if the two events E_1 and E_2, corresponding to updates of the cell to

---

[1] That is, providing the specifications did not require parallelism. I.e., the specifications were for sequences of events $\langle t_{1_1}\ m_{1_1}\ ?\ ec_{1_1}\rangle \dashrightarrow \ldots \dashrightarrow \langle t_{1_n}\ m_{1_n}\ ?\ ec_{1_n}\rangle$ and $\langle t_{2_1}\ m_{2_1}\ ?\ ec_{2_1}\rangle \dashrightarrow \ldots \dashrightarrow \langle t_{2_n}\ m_{2_n}\ ?\ ec_{2_n}1\rangle$ where activator identity was not important.

different values, were unordered by -->, the contents of the cell would be left with one value or the other, not both. That value is the value stored in the last update where last is defined with respect to the order in which the messages were received by the cell.

For any actor t, the ordering, $=>_t$, is a total ordering on the events in the behavior with t as target. Orderings such as $=>_t$ will be referred to as *actor orderings*. For any behavior this ordering can be thought of as arbitrary up to consistency with -->. That is, for any E_1 and E_2 which both have t as target, if $E_1 \dashrightarrow E_2$, then $E_1 =>_t E_2$.

If t_i are actors which cause behaviors to depend on their $=>_{t_i}$, then for any actor system containing the actors t_i, that system's time ordering is derived from combining the actor orderings, $=>_{t_i}$, and the activator ordering -->. In following chapters actors are defined by axioms that depend on actor orderings. Each chapter either introduces an actor with a new kind of causality or investigates the way in which a previously introduced actor can be used as part of an actor system to satisfy high level time ordering specifications.

3 Specifications of Cells

This chapter contains the definition of an actor which is a cell for storing arbitrary values. The cell is an example of an actor which causes behaviors to depend on an actor ordering. The definition is followed by an example of how the cell can be used as part of a system to cause a particular time ordering property to hold for all behaviors of that system.

3.1 A Definition of the Cell

A cell is created with some initial contents. It is possible to ask for the contents of a cell. Update messages to that cell can change its contents. Thus it is possible that several events in a single behavior can have identical targets (namely the cell) and identical transmitted actors and yet cause different events to occur. This occurs when events E_1 and E_2 are events

$$E_1 = \langle \text{cell}_i \text{ (apply: 'contents (then-to: c))} \ \alpha \ ec_1 \rangle^1$$
$$E_2 = \langle \text{cell}_i \text{ (apply: 'contents (then-to: c))} \ \beta \ ec_2 \rangle$$

to a cell, cell_i, whose contents have been changed due to some update event, E_3, where E_3 occurs between E_1 and E_2. Thus the actor c is sent different actors after E_1 and E_2, respectively as in the behavior in Figure 3.1.

---

[1] The actor **'contents** is a "quoted" actor, the result of quoting a symbolic name contents.

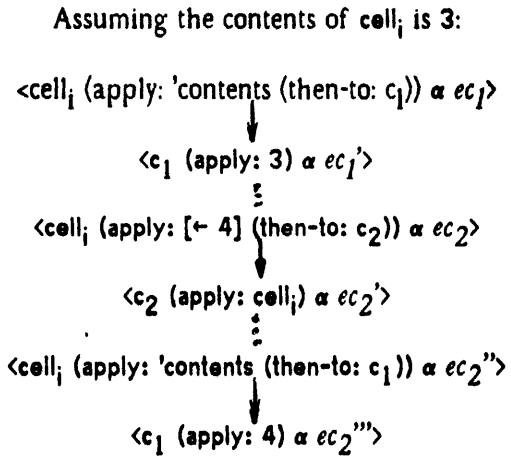


Figure 3.1: An Update Causes Side-effect Behavior

Cells correspond to side-effect primitives in various languages. Therefore, it is likely that most readers have relied on them in programming. Also, it is likely that for any communication between parallel processes one has to rely on cells or something like them. Exactly what properties are being relied on for communication will be brought out in the behavioral definition of a cell.

The actor **cons-cell** creates a cell when it receives a message, namely, the initial contents. Informally, the properties of **cons-cell** and of the cells it creates are:

(1) That creation of a cell (or allocation of space for a cell) with its initial value should be an operation guaranteed to take only a finite amount of time. This is also a desirable property of updates and contents queries. (Any argument that a given program which contains assignment statements always terminates is based on these properties).

(2) Initially we expect contents queries to find the original contents in the cell. This should be true until an update transmission is sent.

(3) Once an update transmission is sent, say [← y], we would expect to find y in the cell until there is another update, where y is the contents stored in the cell in the "last update event."

The "last update event" can be defined with respect to the total ordering $=>_{cell_i}$ as follows:

> Definition: For the event E, the last update event for a cell, $cell_i$, with respect to a total ordering, $=>_{cell_i}$, is defined to be the event E_0 where E_0 is an update event, $E_0 =>_{cell_i} E$ and there is no other event E_3 updating $cell_i$ such that $E_0 =>_{cell_i} E_3 =>_{cell_i} E$.

The use of "last update event" in the cell definition cannot simply be defined with respect to the activator ordering -->. The following example is presented as evidence of the fact that our intuitive use of "last update event" is indeed with respect to the cell's ordering $=>_{cell_i}$.

In a behavior such as that in Figure 3.2[1] can the message place marked by ? be said to definitely be 3?

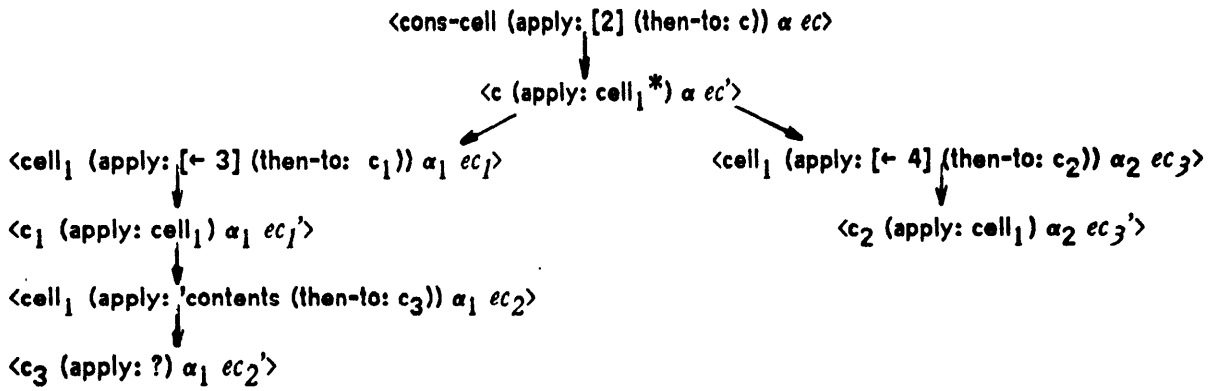


Figure 3.2 Parallel Updates

---

[1] In all such examples we will assume that the behavior segments given show all activators that have access to the actors of interest (in this case $cell_1$). This is an important fact about a behavior segment involving side-effect actors, since if there could be other updates to $cell_1$ in parallel with this segment, we could not meaningfully discuss the set of values that ? can possibly have.

No, it cannot. In fact it can be either 3 or 4 depending on which was the last contents stored in the cell with respect to $=>_{cell_i}$. According to --> the two update events occur in parallel. But since they both involve a common actor, the target actor $cell_1$, the ordering of those events with respect to the arrival of messages at $cell_i$ affects system behavior in the answer to the contents query. Thus this system has two behaviors which are consistent with the partial order so far specified by the consistency with -->. They differ only in the identity of ?.[1] In a representation of the behavior including only -->, there does not seem to be any reason for the two different behaviors, or means for predicting which will occur. Indeed from the point of view which does not include actor ordering, this system is non-deterministic. However, after $=>_{cell_i}$ is included in the representation, the cause of the difference is clear. Two possible orderings are illustrated in Figure 3.3. [There is a third.]

---

[1] That is, it can have two behaviors even given identical *initial conditions* [as defined in the last chapter] each time.

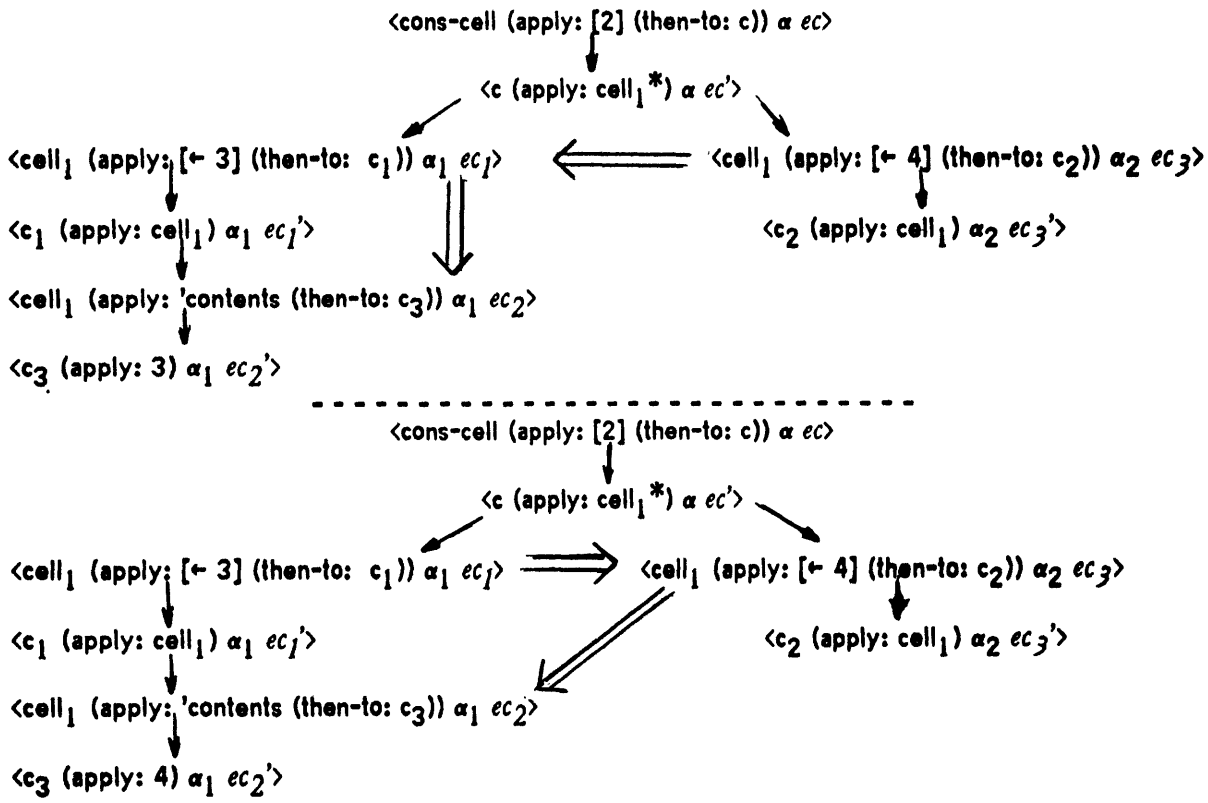


Figure 3.3 The Cell Ordering[1]

As stated in Chapter 2 actor orderings have to be consistent with -->. This requirement prevents ? from being 2. A $=>_{cell_1}$ such that 2 is the last contents stored before the contents query would lead to forming a time ordering => which has a loop, i.e., in which there are two events E_1 and E_2 such that $E_1 => E_2 \wedge E_2 => E_1$. This loop is directly traceable to the

---

[1] In figures we generally indicate the parts of => that are not in --> by =>.

inconsistency of that $=>_{cell_1}$ with -->. Such loops obviously should never happen given the interpretation of => as a time ordering. In fact, in proofs of correctness, undesirable behaviors can often be proven impossible by showing that they could only occur with impossible => orderings.

A further constraint, namely, the constraint that in systems with more than one cell, $cell_i$, the orderings $=>_{cell_i}$ must all be mutually consistent with -->, prevents other pathological => orderings. For example in the behavior segment in Figure 3.4 since $cell_2$ is known to have contents 6, $cell_1$ must have been updated to 5 and the only possible value for $?_1$ is 5.

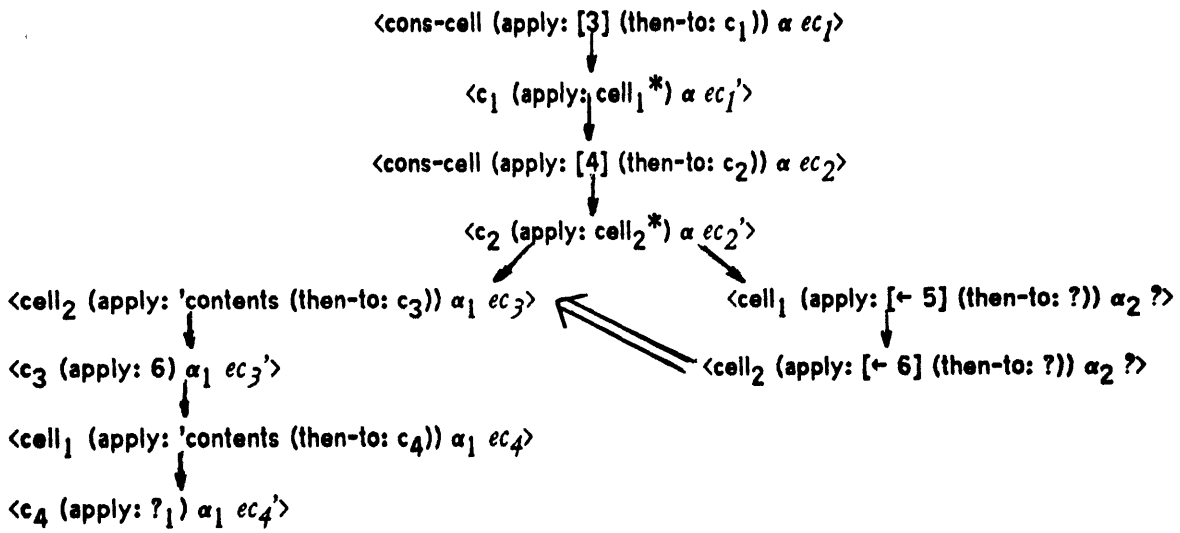


Figure 3.4: Two Cell Behavior

An alternative in which $cell_1$ contains 3 leads to the inconsistent behavior of Figure 3.5.

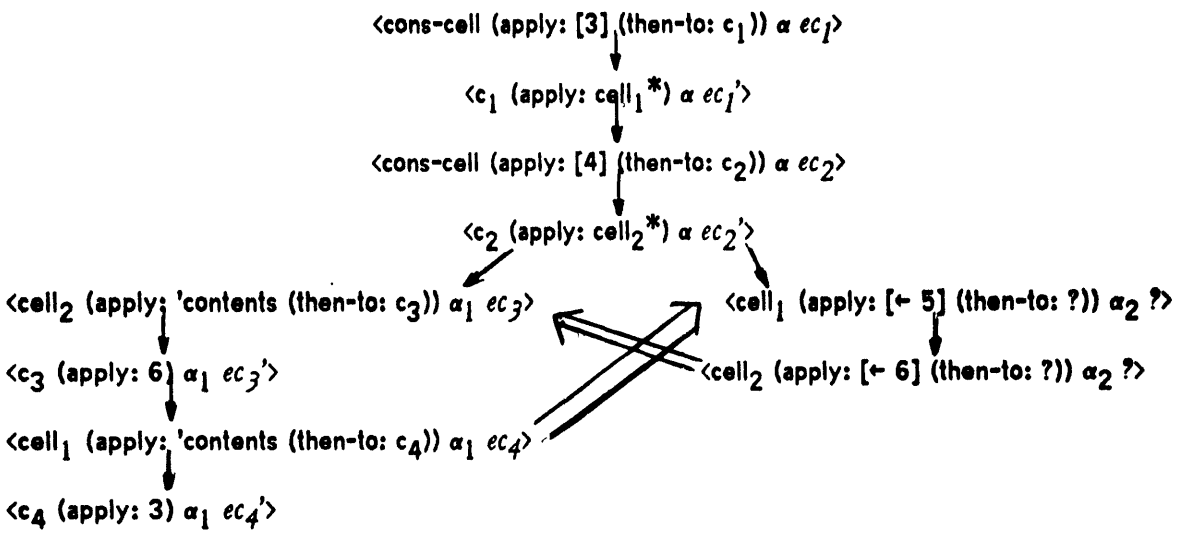


Figure 3.5: An Impossible Behavior Segment

3.2 The Causal Relations Induced by Cells

The full statement of the axioms for cells follow. The cells are defined by a causal relation between the set of events involving cells (and therefore ordered by $=>_{cell}$) and the single event which is the response from the cell. This can be interpreted as a definition of the cell by how it operates as part of a running system. For any behavior of a system containing a cell, if the events occur in the order required by the precondition of an axiom, then events in the postcondition are included in the behavior.

(1) Creation of a cell is guaranteed to terminate, producing a cell, $cell_i$.

$\langle$cons-cell (apply: z (then-to: c)) α *ec*$\rangle$

causes

$\langle$c (apply: $cell_i^*$) α *ec'*$\rangle$

where $cell_i$ induces the following causal relations among events:

(2) Updating a cell is guaranteed to terminate and to cause return of the cell to the continuation.

$\langle cell_i$ (apply: [← y] (then-to: c)) α *ec*$\rangle$

causes

$\langle$c (apply: $cell_i$) α *ec'*$\rangle$

(3) Until the first update, the initial contents is returned in response to any contents query.

E = $\langle cell_i$ (apply: 'contents (then-to: c_5)) α_5 $ec_5\rangle$
where there are no updates before E in $\Rightarrow_{cell_i}$

causes

$\langle c_5$ (apply: z) α_5 $ec_5'\rangle$.

(4) Once there has been an update, the last contents stored is returned.

E_1 = <$cell_i$ (apply: 'contents (then-to: c_2)) α_2 ec_2>
and E_2 = <$cell_i$ (apply: [← y] (then-to: c_3)) α_3 ec_3>
where E_2 is the last update before E_1 in $=>_{cell_i}$

cause

<c_2 (apply: y) α_2 ec_2'>.

The formality of axioms such as the above depends on the fact that phrases such as "E_2 is the last update before E_1" can be stated precisely. That particular phrase has been defined earlier. "There are no updates before E in $=>_{cell_i}$" can be stated formally as "¬ (∃ E_0 ∋ E_0 is an event of the form <$cell_i$ (apply: [← ?] (then-to: ?)) ? ?> ∧ E_0 $=>_{cell_i}$ E)." Throughout the paper similar phrases will be used in place of their full definitions. Definitions will be stated only when a phrase to be used is judged to be significantly different from ones previously defined.

3.3 Uses of Cells

The time ordering => of a behavior, derived from --> and $=>_{cell_i}$ can relate events of different activators other than the events involving the cells. Thus besides being used to store information for its intrinsic value in such a way that arbitrary processes can share it, the cell can be used as a means for enforcing an ordering on otherwise unordered events or for synchronizing otherwise independent processes. In order to characterize the *use* of cells for synchronization, one can state properties of the ordering => which hold for every behavior of that system. This kind of statement can be made as a specification for a program.

In order to realize such an intention, the cells must be used to cause appropriate derived time orderings as the system is running. For example, imagine a system in which a cell is initialized to some value, and in which there can be exactly one other update (to a different

value) in any behavior of the system. Then a contents query finding the initial value can be interpreted as indicating that the update has not yet occurred, while one finding the second value indicates that it has occurred. The example in this section illustrates how by the strategy just described one can use properties of $\Rightarrow_{cell_i}$ and --> to guarantee that an ordering on events of independent parallel processes holds.

The strategy depends on information about $\Rightarrow_{cell_i}$ being determined by contents queries. In the Algol-like program of Figure 3.6, a cell is used to enforce a system ordering in which some "important computation" by process 2 is always done before some "other computation" of process 1.

```
cell_1 := 4;
parbegin[1]
        process 1:
                loop: if contents of cell_1 ≠ 5 then goto loop else other steps ;
        end;

        process 2:
                important steps;
                cell_1 := 5;
        end;
parend;
```

Figure 3.6: A Synchronizing Program.

A sample behavior of this program might be that in Figure 3.7.

---

[1] Starts process 1 code and process 2 code in parallel. I.e., they are executed by separate activators, α_1 and α_2.

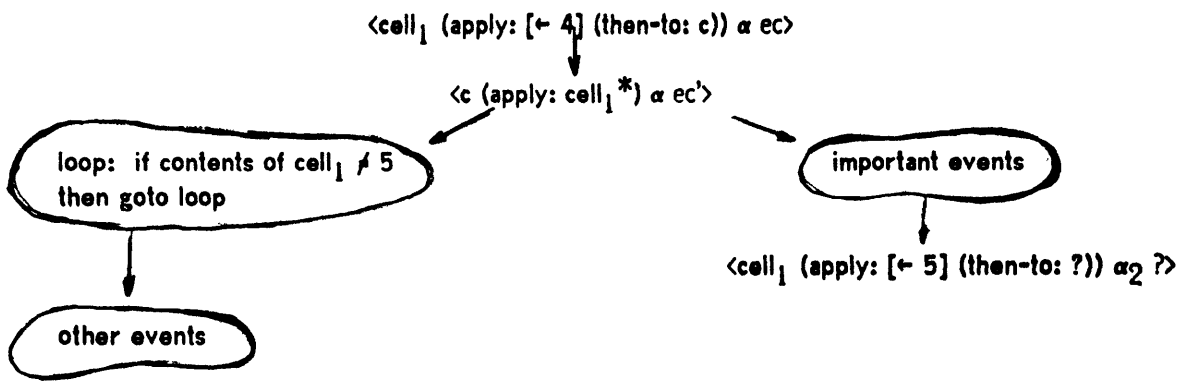


Figure 3.7: Sample Behavior.

The system defined by the program in Figure 3.6 can be said to have behavior in which

$$\text{important events} \Rightarrow \text{other events.}^{1}$$

The reason that it can be shown that this time ordering is realized by the program is that if the contents of $cell_1$ is 5 then <$cell_1$ (apply: [← 5] (then-to: ?)) α_2 ?> must have occurred, and what is more, it is also known from the activator ordering, the "important events" must have occurred. Thus --> and $\Rightarrow_{cell_1}$ together cause sufficient ordering on events other than those involving $cell_1$ to realize the time ordering property.

The justification that this program has this property is in terms of the axioms for behaviors involving cells and properties of the programming language. Besides properties of cells, a proof depends on a definition of the if - then - else construct that allows one to deduce

---

[1] Recall that this means that for the sequence of events $E_{i_1}, \ldots, E_{i_n}$ called "important events" and the sequence of other events $E_{o_1}, \ldots E_{o_m}$, $E_{i_1} \dashrightarrow \ldots \dashrightarrow E_{i_n} \Rightarrow E_{o_1} \dashrightarrow \ldots \dashrightarrow E_{o_m}$.

that other events occur iff the contents of $cell_1$ is found to be 5 by process one.[2] This information is combined with the fact that if any behavior has a contents query which causes a 5 to be sent to the continuation (referred to as c_{test}), then that behavior has a total ordering on $cell_1$ events satisfying

$\langle cell_1$ (apply: [← 4] (then-to: c)) α $ec\rangle$ -->

$\langle cell_1$ (apply: [← 5] (then-to: c_2)) α_2 ?$\rangle$ $=>_{cell_1}$

$\langle cell_1$ (apply: 'contents (then-to: c_3)) α_1 $ec_1\rangle$ -->

$\langle c_{test}$ (apply: 5) α_1 $ec_1'\rangle$.

Regardless of the number of contents queries in the actual behavior this part of the total ordering necessarily is the same [see Figure 3.8]. Since the activator ordering also says that

important events --> $\langle cell_1$ (apply: [← 5] (then-to: c_1)) α_2 ?$\rangle$,

and

$\langle c_{test}$ (apply: 5) α_1 $ec_1'\rangle$ --> other events,

any ordering, =>, constructed from --> and $=>_{cell_1}$ will have important events => other events. Thus this reasoning is independent of the particular behavior examined and is valid for any behavior of that system.

---

[1] Exactly how if-then-else can be defined is discussed in the Appendix. It is equivalent to a cases statement in PLANNER-73.

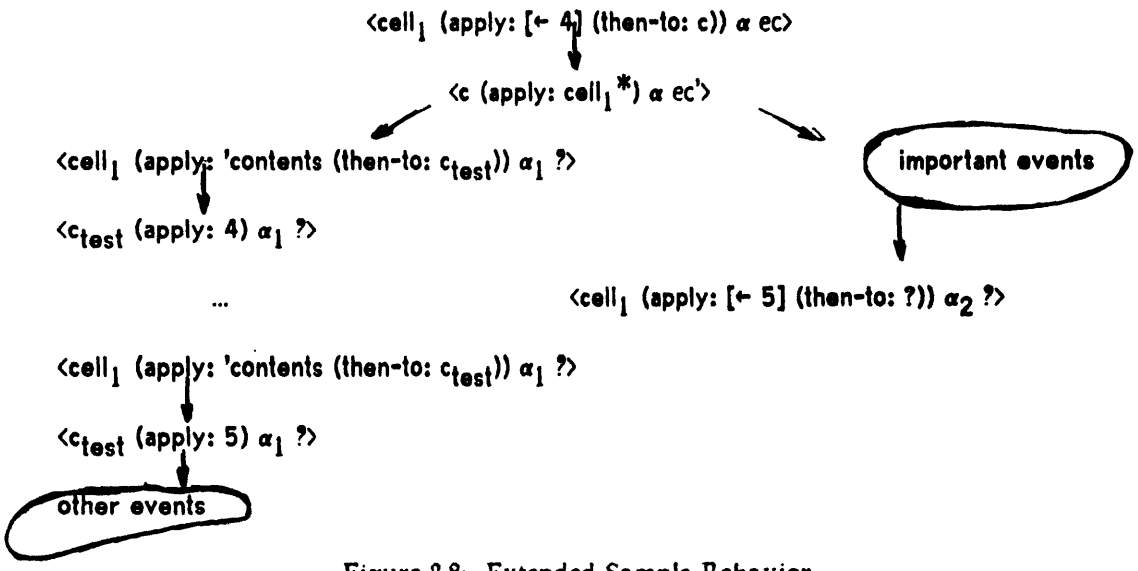


Figure 3.8: Extended Sample Behavior.

The most interesting feature of the system defined by the program in Figure 3.6 above is best described in terms of the time ordering, =>. It describes a property that holds over a class of behaviors which would not be distinguishable by --> alone. The "meaning" of the use of cells in the program is the strategy for ordering sequences of events. This is the structure of the program.

3.4 The Orderings

In summary, this chapter contained the definition of a cell, a side-effect actor which can be used to select a behavior from among several possibilities which would be consistent with the activator orderings and the initial conditions. Given orderings $\Rightarrow_{cell_i}$ for all cells in a system,

each different behavior can be seen to have been caused by differences in the orderings of events involving cells. Each of these behaviors can be more completely characterized by a time ordering derived from --> and $=>_{cell_i}$.

Proof that of a system satisfies a time ordering specification can be done by proof that there are properties common to all derived time orderings of all behaviors of that system. Further examples of this kind of reasoning for systems with cells are contained in Chapter 9.

Some comments can be made on possible physical intuition about $=>_{cell}$. It can be viewed in one of two ways. First, given the constraints already placed by activators on the events in a behavior and their orderings, $=>_{cell_i}$ can be thought of as an incidental ordering, not completely determined by the system. Events of independent processes involving a single actor as target can be incidentally ordered as a system runs, by the order in which the messages of these events arrive at that actor. Then strategies such as the one above can be thought of as making it possible to find out at run time about these incidental orderings and make use of them.

Alternatively, one might note that since two messages could arrive simultaneously, in order for a cell to satisfy its specifications, it may have to be able to arbitrate between them. Thus the cell itself may occasionally determine that ordering. Then rather than simply being an arbitrary ordering given at run time, $=>_{cell_i}$ may be an ordering computed by $cell_i$, with the constraint that it is a total order on all events with $cell_i$ as target and is consistent with -->.

Either interpretation leads to identical deductions with the important events strategy. As long as $=>_{cell_i}$ totally orders all cell events and the place in the ordering is associated with performance of the operations requested in the message, the cell gives the same basis for reasoning about causal links. In Figure 3.8, while $cell_1$ contains 3 one can deduce nothing about important events. $Cell_1$ contains 5 implies important events must have occurred and says nothing about whether events after the update to 5 have occurred. Regardless of exactly how $=>_{cell_i}$ was formed, its consistency with --> indicates a causal link between the important events and the other events.

4 Synchronization

This chapter contains an example of another kind of actor which can be used to form a derived time ordering =>. The first section introduces a time ordering property which must often be relied on in programming for parallel systems. After that the causal axioms for a synchronizing actor, **cons-protection**, are given and it is shown that this actor can be used to realize the time ordering specification. The properties of these axioms as statements about causality will be discussed. The chapter ends with a review of the language for ordering specifications as it has been developed to that point as well as its relation to partial orders.

4.1 Mutual Exclusion as a Time Ordering Specification

One means of protecting a resource from simultaneous use by two processes is "mutual exclusion." This informally means that at most one process can be using the resource at any given time. A precise statement of mutual exclusion can be made in terms of time orderings. If, for instance, the resource to be protected is a data base, the sequences of events between sending a message to the data base and the response from the data base (i.e. the events in one operation on the data base) are the important events. Such sequences can be denoted by

$$S_x = \{E_{x_{in}}, E_{x_1}, \ldots, E_{x_m}, E_{x_{out}}\}$$

where

$$E_{x_{in}} = \langle \textbf{data-base (apply: x (then-to: c))} \; \alpha \; ec \rangle$$
$$E_{x_{out}} = \langle \textbf{c (apply: y)} \; \alpha \; ec^{(m+1)} \rangle$$

and

$$E_{x_{in}} \text{ --> } E_{x_1} \text{ --> } \ldots \text{ --> } E_{x_m} \text{ --> } E_{x_{out}}.$$

At the highest level of detail of examining this data base operation, one would see only $E_{x_{in}}$ and $E_{x_{out}}$ and therefore we will generally use the following notation to represent a data base operation:

$$S_x = \{E_{x_{in}}, E_{x_{out}}\}$$

such that

$$E_{x_{in}} \text{ --> } E_{x_{out}}$$

The definition of mutual exclusion for a system is that the time ordering must have the property that in any behavior of that system, for all sequences S_1, S_2 in that behavior

$$S_1 \Rightarrow S_2 \text{ or } S_2 \Rightarrow S_1.$$

This means that in no behavior of that system is it possible for the events of any S_1 and S_2 to be interleaved.

The property of mutual exclusion is important even in cases where the resource being shared by a group of parallel processes is only a single cell. If for instance several processes are sharing a count of some value in a common cell, there must be limitations on how that cell can be accessed if its contents are to remain consistent with the intended value. Due to the restricted means of communication with a cell in which asking for its contents and updating it are separate events, two processes can both find the contents of a common cell to be x and update it to x + 1 when the intended result is an update to x + 2. That is, each should have independently added 1 to it. What is needed is a means for guaranteeing that during the entire time that any one process is performing its sequence of events for examining the cell, computing a new value and

updating it, no other process can be communicating with the cell. For this reason the property S_1 => S_2 or S_2 => S_1 could also be interpreted as making the operations S_i "indivisible" operations.

In contrast with the important event example of Chapter 3 in which the specifications required **important events** => **other events**, in this case, while the sequences must be ordered, they can be ordered either way. It is much more difficult to realize the specifications for mutual exclusion using only cells than it was to implement the important event ordering. If the number of processes is fixed, there are ways to enforce mutual exclusion using cells. Two such programs are examined in some detail in Chapter 9. The next section contains axioms for an actor from whose causal axioms the mutual exclusion time ordering can be derived. If this actor can be relied on as a primitive (or implementaed with primitives other than cells) then the complex cell solution can be avoided in practice.

4.2 A Synchronization Actor

The synchronization actor to be defined is **cons-protection**, an actor which when sent another actor, **data-base**, creates a new actor **protected-data-base** from it. It is actually this last actor, the one created by **cons-protection**, which has the interesting causal axioms. If **protected-data-base** is used in a system instead of the actor **data-base**, then that system will have the time ordering required for mutual exclusion of data base operations. This is accomplished in the manner conveyed graphically in Figure 4.1. Requests will be sent to **protected-data-base** and they will be transmitted to **data-base**. At the end of the data base operation the result is again transmitted through the protective actor, by means of a specially constructed continuation actor.

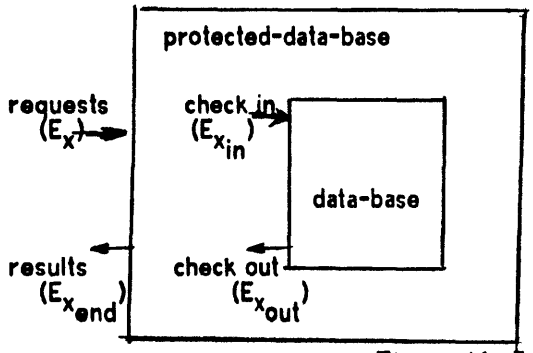


Figure 4.1: Protected Data

The events in a single request and performance of the requested operation are then:

E_x = <protected-data-base (apply: x (then-to: c_i)) α_i ec_i>
$E_{x_{in}}$ = <data-base (apply: x (then-to: s_{c_i}*)) α_i ec_i'>
$E_{x_{out}}$ = <s_{c_i} (apply: y) α_i e_i''>
$E_{x_{end}}$ = <c_i (apply: y) α_i ec_i'''>.

The actor s_{c_i} is a newly created actor that knows about c_i. It is essential to the synchronization as will be shown below. The actor y is defined by the properties of **data-base** which specify responses to the actor x. We will assume that the data base always responds by sending a message to the continuation supplied with the request. The desired effect of **protected-data-base** in one activator is that E_x in the behavior is guaranteed to be followed by $E_{x_{in}}$ --> $E_{x_{out}}$ --> $E_{x_{end}}$. That is, E_x --> S_x --> $E_{x_{end}}$ where S_x is the sequence of events in the data base operation. However, in general E_x alone must not be sufficient for causing $E_{x_{in}}$ or there would be no synchronization.

The actor **protected-data-base** causes orderings $E_{x_{i_{out}}}$ => $E_{x_{j_{in}}}$ to be introduced. The way it does this is by creating a new actor s_c for each process let in to the data base and causing the event in which s_c is sent a message to be necessary to the admission of other activators. This results in orderings S_{x_i} => S_{x_j}.

The axioms for **cons-protection** are:

<cons-protection (apply: data-base (then-to: c)) α *ec*>

causes

<c (apply: protected-data-base*) α *ec'*>

where **protected-data-base** has the causal axioms given below. They are also axioms for the actors s_{c_i} created by **protected-data-base**. We write $=>_{pdb}$ for the ordering $=>_{\textbf{protected-data-base}}$ of the actor **protected-data-base**.

(a) If E_{x_1} is the first request to the protected data base, then E_{x_1} is served first.

E_{x_1}, where E_{x_1} is the first event in the ordering $=>_{pdb}$

causes

$$E_{x_{1_{in}}}$$

(b) If two requests are ordered at the protected data base then the exit of the first is required for the entrance of the second.

E_{x_1}, E_{x_2}, where $E_{x_1} =>_{pdb} E_{x_2}$

cause

$$E_{x_{1_{out}}} => E_{x_{2_{in}}}$$

The axiom (b) is the new type of axiom characteristic of synchronizing actors. It states that the ordering $E_{x_{1_{out}}} => E_{x_{2_{in}}}$ will be added to the ordering formed from $=>_{pdb}$ and -->.

Protected-data-base differs from a side-effect actor in that, rather than causing

different events depending on $\Rightarrow_{cell}$, protected-data-base causes different orderings depending on $\Rightarrow_{pdb}$. *How* this ordering is enforced is not at issue. The use of a synchronizing actor guarantees that the ordering will be enforced, and how it is enforced need not be dealt with at the level of detail at which the synchronization actor is viewed as a primitive operation (i.e. at which we don't look inside it).[1]

The additions to the derivation of $\Rightarrow$ represent causal chains of $\rightarrow$ and $\Rightarrow_a$. To form a time ordering for a system containing protected-data-base one must use $\rightarrow$, $\Rightarrow_{pdb}$, and any additional properties of $\Rightarrow$ added directly. Presumably at some level of detail "inside" the actor, sequences of events occur from which this new $\Rightarrow$ is derived. Thus this is an abbreviation for a causal link, and $E_{x_1 out}$ must occur before $E_{x_2 in}$. Since (b) is true for all E_{x_i} that precede E_{x_2}, all corresponding $E_{x_i out}$ must occur before $E_{x_2 in}$. Once all added conditions are met $E_{x_2 in}$ will occur.

4.3 Properties of Protected Data Base

In a system in which only a protected data base is used, data base sequences S_x contain the events

$$E_{x_{in}} = \langle \text{data-base (apply: x (then-to: } s_{c_i}^{*})) \; \alpha_i \; ec_i' \rangle$$
$$E_{x_{out}} = \langle s_{c_i} \text{ (apply: y) } \alpha_i \; ec_i'' \rangle.$$

It can be shown that the time ordering caused by protected-data-base has the property that for all S_{x_1}, S_{x_2} in a behavior of the system

$$S_{x_1} \Rightarrow S_{x_2} \text{ or } S_{x_2} \Rightarrow S_{x_1}.$$

---

[1] Of course, whether or not protected-data-base is realizable will depend on the implementation of that causal link. Chapter 7 shows an implementation in terms of a synchronization primitive which does exist in real computer systems. Chapter 9 shows an implementation in terms of cells. Thus these axioms are consistent in the sense that there are objects that are described by these axioms.

All sequences S_{x_i} are preceded by requests $E_{x_i} \ni E_{x_i} \dashrightarrow S_{x_i}$. All requests are ordered by $\Rightarrow_{pdb}$. Therefore for any S_{x_1}, S_{x_2} in the behavior, $E_{x_1} \dashrightarrow E_{x_{1_{in}}} \dashrightarrow E_{x_{1_{out}}}$ and $E_{x_2} \dashrightarrow E_{x_{2_{in}}} \dashrightarrow E_{x_{2_{out}}}$ are in the behavior and either $E_{x_1} \Rightarrow_{pdb} E_{x_2}$ or $E_{x_2} \Rightarrow_{pdb} E_{x_1}$.

Asssuming first that $E_{x_1} \Rightarrow_{pdb} E_{x_2}$, by (b), $E_{x_{1_{out}}} \Rightarrow E_{x_{2_{in}}}$ and therefore $S_{x_1} \Rightarrow S_{x_2}$. Similarly, if $E_2 \Rightarrow_{pdb} E_{x_1}$, $S_{x_2} \Rightarrow S_{x_1}$. Thus operations on the data base are mutually exclusive.

This reasoning is trivial given the similarity of the high level specifications and these actor specifications. The point is merely that the derivation of $\Rightarrow$ from $\dashrightarrow$, $\Rightarrow_{pdb}$, and $\Rightarrow$ does indeed give the result, thus an actor so defined can be used to cause the mutual exclusion ordering.

Another property of any system with **protected-data-base** is that if E_x is in the behavior then it is followed by S_x. It means that, in addition to enforcing mutual exclusion, the protected data base uses fair synchronization in the sense that the synchronization can never lock out any activator and all requests are served. This can be stated as

E_x in the behavior $\supset$ S_x is in the behavior

Fairness does not require that E_x be the sole cause of S_x, but only that all requests do get served. It means that the synchronization used to achieve the mutual exclusion ordering must be fair in that it cannot ignore any request that it receives. Not all synchronization is fair. Chapters 7 and 8 are about unfair synchronization.

In the current case it can be shown that for any E_x, if E_x is in the behavior then, since all additional constraints can be met, S_x is also in the behavior.

(1) If E_x is first in $\Rightarrow_{pdb}$ then, by (a), $E_{x_{in}}$ is in the behavior.

$\therefore$ S_x is in the behavior because of the axioms for **data-base**..

(2) Assume $\forall\, i \leq n$, if E_{x_i} is the ith request, then S_{x_i} is in the behavior. Then let E_x be the $n+1$th request. Since by induction $\forall\, E_{x_i} \Rightarrow E_x\, S_{x_n}$ is in the behavior, all $E_{x_{i_{out}}} \ni E_{x_\ell} \Rightarrow E_x$ are in the behavior and $\ni$ by (b), $E_{x_{in}}$ is in the behavior.

$\therefore S_x$ is in the behavior.

4.4 Synchronizing Axioms, Causality, and Partial Orders

In general, for axioms like those for **protected-data-base** it can be arbitrarily difficult to decide what all the additional constraints on an event's occurrence might be. Unless the set of conditions which is implicitly defined by the axioms can be correctly stated explicitly, it may be easy to make errors in arguments which depend on finding the set of necessary and sufficient conditions for events.

One approach to clarifying the necessary and sufficient conditions for the occurrence of an event due to synchronization is to refine axioms such as those given above to equivalent ones which apply to unique events. Thus replacing axiom (b) by (b') below makes it possible to view these specifications as rules for "transference of causality."

(b') Each request other than the first in enabled by the completion of the preceding request.

$$E_{x_1}, E_{x_2}, \text{ where } E_{x_2} \text{ is the first request after } E_{x_1} \text{ in } =>_{pdb}$$

cause

$$E_{x_{1_{out}}} => E_{x_{2_{in}}}$$

By a simple inductive argument one can see that (b') implies (b) and thus the time ordering derived from (a) and (b) is equivalent to that derived form (a) and (b'). The formation of $=>$ from (a) and (b') relies more heavily on $=>_a$ and on the fact that the transitive closure will totally order all data base operations if the axioms order successive pairs. One difference between (b) and (b') is that fewer constraints were explicitly added in (b').

This also means that in (b') E_{x_1} and E_{x_2} transfer causality to $E_{x_{1_{out}}}$, in the sense that now $E_{x_{1_{out}}}$ causes $E_{x_{2_{in}}}$, i.e. if E_{x_1}, E_{x_2} are in the behavior then $E_{x_{1_{out}}}$ in the behavior is necessary and sufficient for $E_{x_{2_{in}}}$ to be in the behavior. It can be important to be able to interpret the axiom this way and therefore we would like to point out the reason why this interpretation is valid.

E_{x_2} and $E_{x_{2_{in}}}$ are of the same activator and the axiom adds $E \Rightarrow E_{x_{2_{in}}}$, where E is of a different activator. Therefore the axiom is about requirements in addition to activator properties necessary for the occurrence of a next event in that activator. What the event $E_{x_{2_{in}}}$ can be is already determined, but there are additional conditions on whether it will occur. The precondition involves a uniquely specified pair of other events, E_{x_1} and $E_{x_{1_{out}}}$. They are the only other events referred to, by virtue of the uniqueness of the definition of "first request after." Since $\Rightarrow_{pdb}$ totally orders events E_{x_2}, there can be at most one E_{x_1} for each E_{x_2}. Except for the E_{x_2} which is the first event in $\Rightarrow_{pdb}$ there will be exactly one such event. The axiom states that once E_{x_1} and E_{x_2} have occurred, $E_{x_{1_{out}}}$ is a necessary and sufficient condition for the occurrence of $E_{x_{2_{in}}}$. This is in contrast with the statement (b) from which one has to deduce the set of events necessary and sufficient for $E_{x_{2_{out}}}$.

These axioms also should be viewed as statements about partial orders. In chapter 2, we suggested that axioms for single activator causality could be viewed as rules for adding events to behaviors. If partially ordered sets of events are viewed as sets of ordered pairs of events, then axioms can also be viewed as rules for adding pairs to the partial order. Even in single activator cases, when the partial order is in fact a total order, the axiom

<addl (apply: x (then-to: c)) α *ec*> --> <c (apply: z) α *ec'*>

says that the pair (<addl (apply: x (then-to: c)) α *ec*>, <c (apply: z) α ec'>) can be added to the activator ordering -->. According to this interpretation the activator ordering is formed from the transitive closure of the pairs determined by the axioms for the actors in the actor system.

When there are multiple activators and cells, behaviors can be specified by their partial orderings, =>. The ordering, =>, is formed from the union of $=>_{cell}$ and -->. Thus, using, {=>}, the set representation of =>, we can state

{=>} is the transitive closure of $\{=>_{cell}\} \cup \{-->\}$.

Since these may be several different sets of events that correspond to possible behaviors of a given actor system with cells, such systems are characterized by sets of partial orders or by properties of => common to all those partial orders. These later properties are generally partial orders on events other than the cell events which are common to all behaviors, e.g. **important events** and **other events** in Chapter 3.

The axioms for **protected-data-base** specify pairs that are to be added to the set of pairs from which the transitive closure is taken. Thus if we refer to the parts of => added by axioms as $=>_{new}$, the ordering {=>} is the transitive closure of $\{-->\} \cup \{=>_{pdb}\} \cup \{=>_{new}\}$. Using the $=>_{new}$ of either axiom (b) or (b') results in formation of the same transitive closure. If the pairs in $\{-->\} \cup \{=>_{pdb}\} \cup \{=>_{new}\}$ are considered as generators of =>, differences among alternative specifications are evident in the size of the set of generators it produces.

To summarize, the new kind of axiom of this chapter introduces further causality, denoted by additional constraints on =>, the derived time ordering. Thus in deriving => for a system involving a synchronization actor s, one uses -->, $=>_s$, and => added in axioms. The axioms for synchronization are different from the axioms for cells in that the properties of $=>_s$ do not directly affect what events will be in the behavior but rather what causal relations will hold on events, if they do occur.

5 Comparing Problem and Solution Specifications

Reinterpretation of the protected data base of the last chapter will reveal still more fundamental statements which can be made about data bases. A totally external definition (one in which the data base is treated as a primitive) of a reliable data base can be given. The actor **protected-data-base** is then seen as an implementation of that data base. By weakening that definition one can look at alternatives which correspond to a well-known synchronization problem -- the readers/writers problem. We develop specifications analogous to those for mutual exclusion which are formal statements of four versions of the readers/writers problem.

Following that, one version is specified more concretely by the writing of its solution specifications. These solution specifications are a step in the direction of implementation. Since they are written as causal axioms of synchronization actors, they include explicit reference to the causality used to achieve the specified orderings. Thus the solution specifications in a sense define the algorithm or structure to be used in programming an implementation. A program with the same structure will generally be more easily compared to the solution specifications than to the problem specifications written in terms of =>. For this reason the proof that these solution specifications are sufficient for realizing the high level specifications is an interesting result. It ensures that any program checked only against these solution specifications will indeed also satisfy the high level specifications. Once again the chapter will conclude with a summary of the ordering information used so far.

5.1 Re-interpreting **Protected-date-base**

One property of a data base which could be desirable is closely related to a property of

cells. The property is that every time a record is read, the result is an actor that was stored by the last write in some total ordering. Since data base operations are generally long sequences of events, a data base may not have this property under all circumstances. If two writes overlapped (or if a read and a write overlapped or even any two operations, depending on implementation) the result of a later read could be a record that was never deliberately stored by any write. It is not usually considered interesting to specify all possible mistaken records that could occur under all circumstances. Therefore data bases are generally specified either by implementation, explaining just what steps are taken in doing an operation, or by expected effect given no interference among operations.

In the specification language we can also write axioms exactly like those of the cell for a data base. They specify a data base which always returns meaningful values, i.e. values that were deliberately stored. An example is the actor rdb, a reliable data base, defined by the following axioms:

(1) A reliable data base is created by **create-data-base** to have records x_i with initial contents y_i.

$$\langle \textbf{create-data-base (apply: [[}x_1\ y_1\textbf{] ... [}x_n\ y_n\textbf{]] (then-to: c))}\ \alpha\ ec\rangle^1$$

causes

$$\langle \textbf{c (apply: rdb}^*\textbf{)}\ \alpha\ ec'\rangle$$

(2) Writes always result in return of the data base.

---

1 Note that in the protected data base specifications of Chapter 4 details such as reference to parts of the data base, the x_i, or to the value to which to update, the y_i, were omitted. This was done for brevity and we will revert to that abbreviated form again after this section.

<rdb (apply: ['write x_i z] (then-to: c)) α *ec*>

causes

<c (apply: rdb) α *ec'*>

(3) Initially, reading record x_i finds y_i, the initial contents.

E_r = <rdb (apply: ['read x_i] (then-to: c)) α *ec*>
where there is no <rdb (apply: ['write x_i z] (then-to: ?)) ? ?> preceding E_r in $=>_{rdb}$

causes

<c (apply: y_i) α *ec'*>

(4) If a record has been written into, a read of that record finds the last record stored.

E_w = <rdb (apply: ['write x_i z] (then-to: ?)) ? ?>
and E_r = <rdb (apply: ['read x_i] (then-to: c)) α *ec*>
where E_w is the last write in x_i before E_r in the ordering $=>_{rdb}$.

causes

<c (apply: z) α *ec'*>

What can this mean? Let us reconsider the cell definition and the reasons why we accepted it. While it specified a useful behavior, our acceptance rested on postulating that a cell could be built with that property. That is usually an accepted assumption about access to memory. Memory is fast enough for it to be unlikely that two operations would ever interfere with each other, or if two do occur simultaneously, a sufficiently good arbitration scheme can be

incorporated in memory accessing to prevent interference. Of course, as machines get faster this assumption can break down. If we had to implement the cell under conditions that would make cell operations lengthy relative to their use, how could we realize the specifications? For instance if, in an implementation, updates require bit by bit updating, and reads, destructive bit by bit read followed by restoring of the contents, part of the implementation specifications would have to be the mutual exclusion of the events which at a lower level of detail occur between the cell event and the event in which the continuation is sent a response. If a behavior contains sequences of the form:

$$\langle \textbf{cell (apply: ? (then-to: c))} \; ? \; ec \rangle \dashrightarrow E_1 \dashrightarrow \ldots \dashrightarrow E_n \dashrightarrow \langle c \; ? \; ? \; ec^{(n+1)} \rangle$$

then all sequences $E_1 \dashrightarrow \ldots \dashrightarrow E_n$ must be ordered with respect to one another.

Thus $\Rightarrow_{\textbf{cell}}$ as an external property of an actor can be the result of deliberate computation at a lower level of detail, i.e., the computation that can enforce the mutual exclusion of the implementation of the operation. We have decided to accept cells as primitive and therefore to accept $\Rightarrow_{\textbf{cell}}$ as being close to an "incidental ordering" which corresponds to the order in which messages arrive at the cell. However, for actors which are not claimed to be primitive specifications relative to $\Rightarrow_a$ must be interpreted as a restriction on the class of implementations possible.

For instance, as an actor, a reliable data base, **rdb**, has an ordering $\Rightarrow_{\textbf{rdb}}$ and properties as illustrated above. This specification requires that if in an implementation the operations are to be side-effect operations, they will be mutual exclusive and will be done fairly (all events with **rdb** as target get ordered). Thus the actor **protected-data-base**, which can enforce fair mutual exclusion and cause data base operations, is one means for implementing such a data base. If **protected-data-base** is the only actor to which data base requests are sent, then at the level of detail at which only events of the forms

<protected-data-base (apply: ? (then-to: c)) ? ?>

and

<c ? ? ?>

appear, the protected-data-base acts like a reliable data base with all requests ordered consistently with --> and all reads finding the result of the last write.

In terms of a computed actor ordering $=>_{rdb}$ fairness means that all events with rdb as target are ordered by $=>_{rdb}$ and that therefore all are followed by next events. Thus in "computing" $=>_{rdb}$, rdb cannot ignore a request. For $=>_{cell}$, an ordering of a primitive actor, we assumed all cell events were ordered. For computed orderings, we may want to specify which events are ordered.

One might actually be interested in the cases where $=>_{rd}$ does not have to include all events with rdb as its target. While it may always be important to find a last update, rather than garbage, it may not matter how last is measured (i.e. with respect to what order). Therefore one might just require that the ordering $=>_{rdb}$ have the property that

E with rdb as target is ordered by $=>_{rdb}$ iff E_{next} is in the behavior

rather than that all events with rdb as target be in the ordering. Together with the axioms given above this can be interpreted as a specification for a data base in which one always finds a last contents stored but in which some requested operations aren't performed. In implementing a reliable data base with this weaker ordering specification, if side-effect operations are used, mutual exclusion might still be necessary. However, it would not have to be enforced fairly.

Similarly, if, for instance, it were known that certain operations, say reads, would not be side-effect operations we might only require $=>_{rdb}$ to be a partial order such the writes are

ordered with respect to other writes and writes and reads are ordered with respect to each other. Given this constraint one can still meaningfully find the last write from any read operation. For any read the value found should be the value stored in the last write. This kind of data base is a readers/writers data base and is the subject of this chapter. The puzzling readers/writers problems are specifications for determining which requests must be served and which can be locked out.

5.2 The Problems of Reading and Writing in a Data Base

Just as in the previous chapter fair mutual exclusion was used to realize a fair reliable data base, properties related to mutual exclusion can be used to realize other kinds of reliable data bases with readers/writers scheduling. Readers/writers scheduling is defined in relation to the readers/writers problem and two of its variants which were first introduced into the literature by Courtois, Heymans, and Parnas [Courtois et al, 1971]. The most important part of the problem is that since writers presumably perform sequences of side-effect actions in the data base, it becomes necessary to protect the data base to preserve its consistency. However, since readers are not ordinarily side-effect operators and therefore cannot interfere with each other, mutual exclusion is generally considered too restrictive. Instead a property which shall be referred to as the Readers/Writers Property is required. Informally, it is the property that there be mutual exclusion of write operations in the data base and that there be mutual exclusion of groups of read operations and single writes.

To state this formally read operations are distinguished from write operations. S_r is a sequence of events in a read operation, S_w a sequence in a write operation. Throughout this chapter subscripts of r or w on events or sequence symbols denote reader or writer events or sequences. Therefore the sequences and events have the following form:

$S_r = \{E_{r_{in}}, E_{r_{out}}\} \ni E_{r_{in}} \text{ --> } E_{r_{out}}$

where

$E_{r_{in}}$ = <data-base (apply: 'read (then-to: c)) α *ec*>

$E_{r_{out}}$ = <c (apply: y) α *ec'*>

and

$S_w = \{E_{w_{in}}, E_{w_{out}}\} \ni E_{w_{in}} \text{ --> } E_{w_{out}}$

where

$E_{w_{in}}$ = <data-base (apply: 'write (then-to: c)) α *ec*>

$E_{w_{out}}$ = <c (apply: y) α *ec'*>

The readers/writers exclusion property is that for every behavior, the ordering => has the property that

$\forall S_r, S_w$ in the behavior, $S_r \Rightarrow S_w$ or $S_w \Rightarrow S_r$
and $\forall S_{w_1}, S_{w_2}$ in the behavior, $S_{w_1} \Rightarrow S_{w_2}$ or $S_{w_2} \Rightarrow S_{w_1}$

This says that all reads and writes must be ordered with respect to one another, and all writes must be ordered. However, it is not necessary for the set of all read sequences to be totally ordered by =>.

This property is sufficient for proving that a reliable data base is implemented. External properties of a reliable data base as described in section 5.1, can be derived, if we can define $\Rightarrow_{rdb}$ to be the following order over events $E_{x_{in}}$ such that S_x is in the behavior:

$E_{x_{1_{in}}} \Rightarrow_{rdb} E_{x_{2_{in}}}$ iff $S_{x_1} \Rightarrow S_{x_2}$.

Then $E_{x_{in}}$ is ordered by $\Rightarrow_{rdb}$ iff S_x is in the behavior (i.e. if $E_{x_{out}}$ is in the behavior and therefore the request was served) and $\Rightarrow_{rdb}$ partially orders those events such that there is a well-defined last write for every read. This is the property suggested at the end of 5.1 as the "problem" in the readers/writers problem.

While this property is the minimal property required for preserving consistency of a data base, it does not yet correspond to any of the popular definitions of readers/writers problems.

What is missing are requirements for either fairness or priority in the serving of requests.[1] Generally some correspondence between time ordering of requests and of operations is required and considered part of the readers/writers problem.

Informal priority requirements are statements to the effect that one kind of operation can be performed before another, etc. That priority can be stated formally only if it is stated relative to some total order on events. If requests arrive in one order and are served in another, then one may be able to say that priority is being given to one kind of request. If there is no measure of the "natural order" from which one is diverging, then there can be no sense in which the claim to actively be giving priority can be made. Thus all the priority specifications will be given relative to a solution structure in which one can refer to the order of events at an actor and then base priority requirements on order of service relative to order of arrival.[2]

Using a scheme similar to the one for **protected-data-base** we can define the solution structure which will enable us to write priority specifications. In Figure 5.1, the activator properties of a readers/writers data base can be seen to include a request event for each operation.

---

[1] In terms of the **rdb**, what is missing is the definition of which requests are in $\Rightarrow_{rdb}$.

[2] At the next level of detail the order at this actor may or may not be given. At some level, $\Rightarrow_s$, the actor order of a synchronizing actor, s, approximates an incidental time ordering. As long as an actor's order is fair, we will assume that we can use that actor as primitive. The specifications are relative to that ordering.

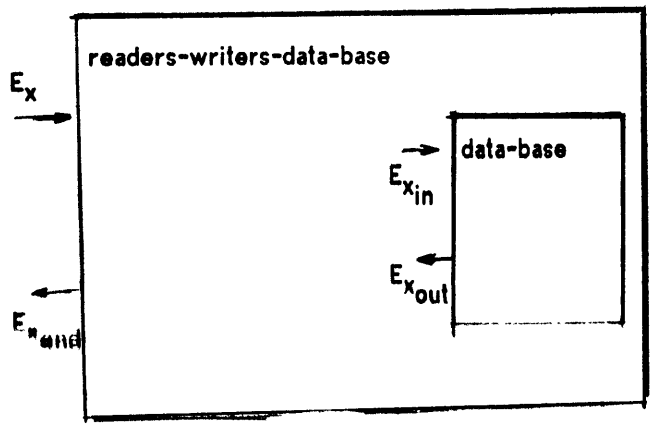


Figure 5.1: Reader/Writers Data Base

Fairness or absence of lockout can be stated as

E_r in the behavior $\supset$ S_r in the behavior [Fairness to readers]
E_w in the behavior $\supset$ S_w in the behavior [Fairness to writers]

Not all of the readers/writers problems are intended to be fair. When fairness to readers or writers is intended we will state it explicitly. The rest of this section contains specifications for various readers/writers problem in terms of =>.

The easiest readers/writers problem to express is Readers/Writers with Queue[1] which serves in the order of requests. It is fair to both readers and writers. Its other specifications are simply:

$$E_r \Rightarrow E_w \supset S_r \Rightarrow S_w$$

$$E_w \Rightarrow E_r \supset S_w \Rightarrow S_r$$

$$E_{w_1} \Rightarrow E_{w_2} \supset S_{w_1} \Rightarrow S_{w_2}$$

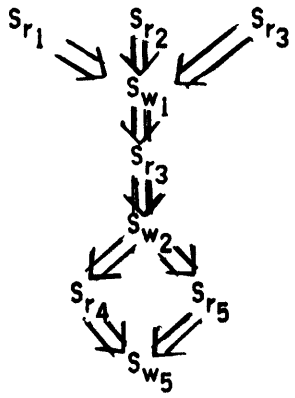
---

[1] The names used to refer to these problems are this author's. This version is not used elsewhere in the literature. For the next three, references are given.

Thus in satisfying the readers/writers property it will have ordering => such as that described in the following scenario. If the requests in order are:

$$E_{r_1}, E_{r_2}, E_{r_3}, E_{w_1}, E_{r_3}, E_{w_2}, E_{r_3}, E_{r_4}, E_{w_5}, \ldots$$

then the behavior will contain the following sequence of events and ordering =>:



The rest of the versions have other properties which in fact amount to giving priority to reads or writes. In all versions the specification

$$E_{w_1} \Rightarrow E_{w_2} \supset S_{w_1} \Rightarrow S_{w_2} \qquad \text{[Write Ordering]}$$

will be assumed ,as well as the readers/writers exclusion and only specification of additional properties will be stated. The kinds of priority special to each problem will be illustrated first by a scenario of possible ordering constraints. This will be followed by definitions of any terms to be used in the specifications and then by the specifications themselves.[1] The most interesting property of all of these specifications is that in order to express them, one must refer to orders on

---

[1] Definitions in this chapter will be given once in English, followed by predicate calculus.

events with different targets. The sub-orderings on sets of events corresponding to requests and ends of operations are depended on. This is important later in the chapter in the causal axioms for actors that realize the time ordering specifications.

In a Write Priority Data Base[1] priority must be given to write requests and the solution must be fair to writers. Thus the following ordering of requests can result in any of a number of behaviors depending on the order in which ends of operations occur:

$$E_{w_1}, E_{r_1}, E_{w_2}, E_{w_3}$$

If the ordering is

$$E_{w_1}, E_{w_{1_{out}}}, E_{r_1}, E_{w_2}, \cdots$$

then $S_{w_1} \Rightarrow S_{r_1} \Rightarrow S_{w_2}$. If the ordering is

$$E_{w_1}, E_{r_1}, E_{w_2}, E_{w_{1_{out}}}, E_{w_{2_{out}}}, E_{w_3}, \cdots$$

then $S_{w_1} \Rightarrow S_{w_2} \Rightarrow S_{r_1} \Rightarrow S_{w_3}$ since the request E_{w_2} will be given priority when $E_{w_{1_{out}}}$ occurs.

Most important is the fact that in the following ordering E_{r_1} can be "locked out" indefinitely:

$$E_{w_1}, E_{r_1}, E_{w_2}, E_{w_{1_{out}}}, E_{w_3}, E_{w_{2_{out}}}, E_{w_4}, \cdots$$

In other words, the reader will only get in at such time as no writer is in the data base and no write requests are waiting.

> Definition[2]: The *data base is empty (or open) for* E if all preceding requests have finished already.

---

[1] This is related to but not identical to the second readers/writers problem of Courtois et al [1971].

[2] Definitions in this chapter will be given once in English, followed by predicate calculus.

The *data base is empty (or open) for event* E if $\forall E_x (E_x \Rightarrow E) \supset (E_{x_{out}} \Rightarrow E)$.

Definition: E *opens the data base to reads* if E is the end of the last write request.

E *opens a data base to reads* if $\exists E_w \ni E_w \Rightarrow E$, $E = E_{w_{out}}$, and $\forall E_{w_1} ((E_{w_1} \Rightarrow E) \wedge (E_{w_1} \neq E)) \supset (E_{w_{1_{out}}} \Rightarrow E)$.

Definition: A *data base is open to reads at* E if no write requests have arrived since the last event that opened the data base to reads.

A *data base is open to reads at* E if $\exists E_1, E_1 \Rightarrow E$, E_1 opens the data base to reads and $\neg (\exists E_w \ni E_1 \Rightarrow E_w \Rightarrow E)$.

Definition: E_r *finds a data base open to reads* if the data base is open to reads at E_r.

The specifications of Writers Priority Data Base are stated as restrictions on fairness to reads. Reads only have to be served if no writes are making demands.

S_r is in the behavior iff E_r is in the behavior and either
E_r finds the data base open to reads
or
$\exists E \ni E_r \Rightarrow E$
$\wedge$ E opens the data base to reads.

The following are two other possible specifications for priority in a readers/writers data base. No later sections refer to them so that the reader who is not interested in all possible variations and their expression in this language may safely skip the rest of this section.

In a Read Priority Data Base [the first readers/writers problem of Courtois et al [1971]], the readers are given priority in the sense that writers only can get in if there is no competitions from a read. The solution should be fair to reads. If requests are (in order)

$$E_{r_1}, E_{w_1}, E_{r_2}, \cdots$$

then the ordering

$$E_{r_1}, E_{r_{1_{out}}}, E_{w_1}, \cdots$$

will result in $S_{w_1} \Rightarrow S_{r_2}$ because the write came while no reader was waiting.

The ordering E_{r_1}, E_{w_1}, E_{r_2}, $E_{r_{1_{out}}}$, $E_{r_{2_{out}}}$ results in $S_{r_2} \Rightarrow S_{w_1}$ since E_{r_2} found no obstacle to its entrance. Note that there are orderings involving more reads in which that write never gets in.

Definition: A *data base is open to writes by* E if there are no read requests preceding it that have not finished.

A *data base is open to writes by* E if $\forall E_r (E_r \Rightarrow E) \supset ((E_{r_{out}} \Rightarrow E) \vee (E_{r_{out}} = E))$

The Read Priority specifications are:

S_w is in the behavior iff E_w is in the behavior and either

the data base is empty at E_w

or

the data base becomes open to writes sufficiently often after E_w to let all preceding writes and then S_w in.

[More formally, $\exists E \ni E_w \Rightarrow E \ni$ the data base is open to writes at E and $\forall E_{w_1} \ni E_{w_1} \Rightarrow E_w$

either $E_{w_{1_{out}}} \Rightarrow E$

or $E_{w_{1_{out}}} = E$.]

The Fair Readers/Writers Data Base [Hoare, 1974] has alternation between readers and writers when both are waiting, so that no lockout can occur. It is fair to both readers and writers. However, the alternation does not necessarily satisfy the Reader/Writer with Queue specifications because when it becomes a readers turn, all waiting reads are enabled.[1] For instance if requests are

$E_{w_1}, E_{r_1}, E_{w_2}, E_{r_2}, \ldots$

then with ends of operations in the ordering as follows:

$E_{w_1}, E_{r_1}, E_{w_2}, E_{r_2}, E_{w_{1_{out}}}, \ldots$

---

[1] This was more natural to implement with Hoare's primitive than the Readers/Writers with Queue, but is more difficult to specify.

it is the case that $S_{r_1} \Rightarrow S_{w_2}$ and $S_{r_2} \Rightarrow S_{w_2}$.

Definition: It is *a wrtters turn at* E if E ends the last of the reads that were in the data base and there is a write waiting.

It is a writers turn at E if $\exists E_w \ni (E_w \Rightarrow E) \wedge \neg (E_{w_{out}} \Rightarrow E)$ and E is the last of the events in the set

$$\{E_{r_{out}} \mid E_r \ni E_r \Rightarrow E_w \text{ where } E_w \text{ is first unserved write request preceding E}\}.$$

The specifications of the Fair Data Base are:

$S_w \Rightarrow S_r$ iff $E_w \Rightarrow E_r$ and either
E_w finds the data base empty
or $\exists E$, $E_w \Rightarrow E \Rightarrow E_r$ and
either it is a writers turn at E and there
are no writes before E_w which are waiting
or E ends a write and there are no reads before
it or writes before E_w which are waiting.

5.3 Writer Priority Specifications:

In this section we will write causal axioms for one version of the readers/writers problem, the one with writer priority. With causal axioms we can be specific about the transference of causality required to guarantee derivation of a time ordering with the four properties of Write Priority, Readers/Writers Property, Fairness to Writes, and Write Ordering. Section 5.2 contained writer priority data base specifications for guaranteeing that reads get in only if no writes are trying. They did not specify what event would enable a waiting read. Similarly, in cases where reads are enabled, it is not clear from those specifications how the ordering of those read operations with respect to following writes is achieved. The following axioms will specify the causality necessary to achieve the four properties at once.

Since the properties of Writer Priority depend on a total ordering on the requests and the ends of operations, i.e. on events E_x and $E_{x_{out}}$, the axioms will be written to depend on that ordering. Recalling the forms of those events:

E_x = <rw (apply: x (then-to: c)) α *ec*>
$E_{x_{out}}$ = <s_c (apply: y) α *ec''*>

shows that the ordering of all events E_x and $E_{x_{out}}$ is an ordering over all events with either rw or an s_{c_i} as target. [This might be referred to as $=>_{rw\&s_{c_i}}$.] Since the s_{c_i} are all created by rw, it is possible that this could be a physically realizable ordering just as an ordering $=>_a$ for one actor a can be. How it would be realized is not the issue at this level of detail. However, the fact that it can be realized will have to be part of a proof of correctness of a particular program for implements these specifications. For conciseness we will use $=>_{rw}$ for $=>_{rw\&s_{c_i}}$.

Terms such as open or open to reads when used in these axioms are meant to be defined with respect to $=>_{rw}$ rather than to $=>$ and are repeated here with that substitution.

Definition: The *data base is empty (or open) for* E if all preceding requests have finished already.

The *data base is empty (or open) for event* E if $\forall E_x (E_x =>_{rw} E) \supset (E_{x_{out}} =>_{rw} E)$.

Definition: E *opens the data base to reads* if E is the end of the last write request.

E *opens a data base to reads* if $\exists E_w \ni E_w =>_{rw} E$, $E = E_{w_{out}}$, and $\forall E_{w_1} ((E_{w_1} =>_{rw} E) \wedge (E_{w_1} \neq E)) \supset (E_{w_{1_{out}}} =>_{rw} E)$.

Definition: A *data base is open to reads at* E if no write requests have arrived since the last event that opened the data base to reads.

A *data base is open to reads at* E if $\exists E_1$, $E_1 =>_{rw} E$, E_1 opens the data base to reads and $\neg (\exists E_w \ni E_1 =>_{rw} E_w =>_{rw} E)$.

Definition: E_r *finds a data base open to reads* if the data base is open to reads at E_r.

The causality introduced by the rw actor is defined by the following axioms.

<cons-readers-writers-data-base (apply: data-base (then-to: c)) α *ec*>

causes

<c (apply: rw*) α *ec'*>

where rw, the write priority reader writer actor, has the following properties:

(1) Newly created data bases are open to readers or writers

(a) $E_w \ni E_w$ is the first event in $\Rightarrow_{rw}$

causes

$E_{w_{in}}$

(b) $E_r \ni$ no E_w precedes E_r in $\Rightarrow_{rw}$

causes

$E_{r_{in}}$

(2) A data base which is opened at E_{out} has properties similar to a new data base for readers or a write

(a) $E_w \ni E_w$ is first after an E_{out} which opens the data base

causes

$E_{w_{in}}$

(b) E_r which finds the data base open to reads

causes

$E_{r_{in}}$

(3) When the data base is not open, time ordering must be introduced.

(a) If a read arrives before all preceding writers are finished then the read must wait until such time as the data base becomes open to reads.

$E_r \ni \exists E_w \Rightarrow_{rw} E_r$ and $\neg E_{w_{out}} \Rightarrow_{rw} E_r$

causes

$E \Rightarrow E_{r_{in}}$

where E is the first in $\Rightarrow_{rw}$ of the events in the set

$\{E_{w_{out}} \mid E_r \Rightarrow_{rw} E_{w_{out}}$ and the data base is open to reads by $E_{w_{out}}\}$[1]

---

[1] Note that this event, E, may not exist. All that this axiom says is that if it does occur then it causes $E_{r_{in}}$. If not, $E_{r_{in}}$ does not occur.

(b) If a write arrives before all preceding writes are finished
it must wait until the last write before it finishes.

$$E_w \ni \exists\, E_{w_1} \Rightarrow_{rw} E_w \text{ and } \neg\, E_{w_{1_{out}}} \Rightarrow_{rw} E_w$$

causes

$$E_{out} \Rightarrow E_{w_{in}}$$

where E is the last write request preceding E_w.

(c) If the preceding write did finish already and thereby enabled some reads, the
new write request must be held up until all of them leave the data base.

$$E_w \ni \forall\, E_{w_1} \Rightarrow E_w,\ E_{w_{1_{out}}} \Rightarrow_{rw} E_w$$
$$\text{but } \exists\, E_r \Rightarrow_{rw} E_w \ni \neg\, (E_{r_{out}} \Rightarrow_{rw} E_w)$$

causes

$$E \Rightarrow E_{w_{in}}$$

where E is last in $\Rightarrow_{rw}$ of the events in the set
$\{E_{r_{out}} \mid E_r \Rightarrow_{rw} E_w\}$.

These axioms state the causality of the rw actor. Part (3) deals with waiting and transference of causality to specify how the operations are later enabled. It handles the special conditions of write priority and fairness to writers. The preconditions specify the same situations referred to in the high level specifications. I.e., for writer priority, one situation that lets a reader in before a writer is that the data base was open to reads for the request. This is specified in (3c). However, in Section 5.2 only the correspondence between this situation and the ordering $S_r \Rightarrow S_w$ was specified. The causal axioms specify exactly which event will be the one which now causes S_w, so that the ordering can be derived. This is the reason that solution specifications will be useful criteria against which to measure a program. They indicate a plan for implementation (i.e., what to cause and what event is the cause, without the details of how to cause it).

Note that these really are very specific solution specifications which add very few pairs to the ordering =>. This is due to the fact that although the motivation for introducing $=>_{rw\&s_c}$ was the need to express priority, it has been used, as well, to define "open" data base, thereby rendering unnecessary the addition of constraints in part (2). If preconditions of (2a) or (2b) hold, then ordering of operations holds already, incidentally. A similar approach could have been taken in defining **protected-data-base**. What this amounts to is that additional constraints are added only when "waiting" is involved, i.e. only when the additional requirements are known not to hold already. If, accidentally, the data base has been emptied before a request and if that request can discover that, then there is no need to plan [in planning a program] to implement a causal link between the last operation that was in and the current request.

In other words, minimizing the number of additional constraints means attempting to only add ordering not already available in $=>_{rw}$ or -->. If as the system is running the fact that the data base is empty is derivable from $=>_{rw}$, it is not necessary for any further synchronization to be used. In solution specifications, this minimality of added constraints is desirable, even if it lengthens the specifications, since it constitutes a fairly refined plan (relative to problem specifications or axioms with more added constraints) for implementation. The plan is to check the "state" of the system and to add constraints if there is danger of interference.

5.4 Properties of the Solution Specification

This section contains a proof that the writer priority readers/writers data base specifications written in terms of => are satisfied by the derived ordering of any system which accesses its data base through an actor rw created by **cons-readers-writers-data-base**. That is, Fairness to Writes, Write Ordering, Write Priority, Readers/Writers Property are enforced by **rw**.

Property 1: The **rw** actor causes Fairness to Writes:

E_w in the behavior $\supset$ S_w in the behavior.

Proof: By induction on the number of preceding write requests.

(i) If E_w is the first event in $\Rightarrow_{rw}$, S_w is in the behavior [by (1a)].

(ii) If E_w is preceded only by requests, then there are a finite number of read requests ahead of it. Once they have all finished S_w gets in by either (2a) or (3c)

(iii) If E_w is preceded by exactly one write request, that write request gets in (by ii). Therefore by either (2a), (3b) or (3c) [since there can be at most a finite number of reads enabled when E_w occurs], S_w is in the behavior.

(iv) Assume that if any E_w is preceded by n writes, $E_{w_1}, \ldots, E_{w_n}$, then that E_w gets in.

If E_{w_0} is preceded by $n+1$ writes, $E_{w_1}, \ldots, E_{w_n}, E_{w_{n+1}}$, then by induction, $E_{w_{n+1}}$ gets in and by either (2a), (3b), or (3c) E_{w_0} gets in. □.

Property II: $E_{w_1} \Rightarrow E_{w_2} \supset S_{w_1} \Rightarrow S_{w_2}$ [Write ordering]

Proof: As corollary of proof of Property I. □.

Property III: Writer Priority

S_r is in the behavior iff E_r is in the behavior and either

E_r finds the data base open to reads

or

$\exists E \ni E_r \Rightarrow E$
$\wedge$ E opens the data base to reads.

Proof: If E_r is in the behavior, then by (1a), (2c), (3a) clearly the writer priority holds. □.

Property IV: Readers/Writers Property.

(a) $\forall S_{w_1}, S_{w_2}$ either $S_{w_1} \Rightarrow S_{w_2}$ or $S_{w_2} \Rightarrow S_{w_1}$.

(b) $\forall S_r, S_w$ either $S_r \Rightarrow S_w$ or $S_w \Rightarrow S_r$

Proof (of (a)): Assuming $E_{w_1} \Rightarrow_{rw} E_{w_2}$ or $E_{w_2} \Rightarrow_{rw} E_{w_1}$ proof is straightforward by induction on number of writes between them. □.

Proof (of (b)): Note first that S_r, S_w in the behavior implies E_r, E_w in the behavior, where E_r --> S_r and E_w --> S_w. Now either $E_r \Rightarrow_{rw} E_w$ or $E_w \Rightarrow_{rw} E_r$.

Assume $E_w \Rightarrow_{rw} E_r$.

There are two cases possible [Others are ruled out by activator properties]

A. $E_w \Rightarrow_{rw} E_{w_{out}} \Rightarrow_{rw} E_r$ in which case clearly $S_w \Rightarrow S_r$.

B. $E_w \Rightarrow_{rw} E_r \Rightarrow_{rw} E_{w_{out}}$

Then E_r cannot find the data base open to reads, since this would violate the definition of "open to reads."

Therefore $E_{r_{in}}$ must be caused by some event E_{out}, $E_r \Rightarrow_{rw} E_{out}$ and $\ni$ E_{out} opens the data base to reads (as in (3a)).

By definition of opening a data base to reads, either $E_{w_{out}} = E_{out}$ or $E_{w_{out}} \Rightarrow_{rw} E_{out}$.

In either case $E_{w_{in}} \Rightarrow E_{out} \Rightarrow S_r$ and therefore $S_w \Rightarrow S_r$.

Assume $E_r \Rightarrow_{rw} E_2$.

Again there are two cases.

A. $E_r \Rightarrow_{rw} E_{r_{out}} \Rightarrow_{rw} E_w$, in which case clearly $S_r \Rightarrow S_w$.

B. $E_r \Rightarrow_{rw} E_w \Rightarrow_{rw} E_{r_{out}}$.

There are three axioms about enabling reads, and thus three cases.

(i) E_r enabled as in (1b). Assume there is no write preceding E_r in $\Rightarrow_{rw}$. In this case, let E_{w_1} be the first write after E_r. Then $E_{w_{1_{in}}}$ is caused by the E_{out} which is last in the set $\{E_{r_{1_{out}}} \mid E_{r_1} \Rightarrow_{rw} E_{w_1}\}$. Therefore either $E_{r_{out}} = E_{out}$ or $E_{r_{out}} \Rightarrow_{rw} E_{out}$. Thus $E_{r_{out}} \Rightarrow E_{w_{1_{in}}}$ and therefore $S_r \Rightarrow S_{w_1}$.
If $E_{w_1} = E_w$ then $S_r \Rightarrow S_w$. Otherwise, by Property II, $S_r \Rightarrow S_{w_1} \Rightarrow S_w$.

(ii) E_r enabled as in (2b). If $\exists$ E, E opens the data base and $E \Rightarrow_{rw} E_r \Rightarrow_{rw} E_w \Rightarrow_{rw} E_{r_{out}}$ and there are no write requests between E and E_r, then similarly, find the first write after E_r, and use it as in the last case to show that $S_r \Rightarrow S_w$.

(iii) E_r enabled as in (3a). $E_{r_{in}}$ is caused according to (3a) , then $\exists$E, E opens the data base to reads, $E_r \Rightarrow E$ and E causes $E_{r_{in}}$. Since E opened the data base to reads

either $E \Rightarrow_{rw} E_w$

or $E_w \Rightarrow_{rw} E_{w_{out}} \Rightarrow_{rw} E$.

In the first case $S_r \Rightarrow S_w$

Otherwise $S_w \Rightarrow S_r$. [$E_{w_{in}}$ enabled as in (3b) or (c)] □.

5.5 Kinds of Specifications

In summary, we have analyzed the readers/writers problem and written formal specifications for it. The specifications for priority are explicitly written relative to the order of a synchronizing actor. The next step towards implementation is the refinement of the specifications to the form of causal axioms. The axioms specify causal properties of an actor which can cause a system to have the desired time ordering properties. This is a realizable solution specification because as we shall see in the next chapter there are primitives available which allow us to implement $\Rightarrow_{rw\&s_c}$. Causal axioms were written for an actor implementation of one version of the problem and were then shown to properly meet all the requirements of the time ordering specifications. Since the time ordering specifications consisted of four separate requirements the

proof is of interest in showing that the axioms specify one actor which can realize all four specifications. The next step is program specification, the subject of the next chapter. The steps from the general criteria for reliability assuming side-effect-free reads, to the four properties of =>, to the causal axioms, correspond to a process of refinement of problem statement from properties for consistency of the data base to a partial solution with details of priority specified.

One last note in reference to the reinterpretation of **protected-data-base** is that the chapter is also about the notion of a computed actor order and its specification relative to another actor's total order. All "scheduling" or "priority" specifications that are written in this language are written relative to some total ordering. The realization of these specifications then depends on the use or implementation of an actor which has axioms that depend on its own actor order. Without the relative specifications it is not clear that we are making meaningful statements. This point will be discussed again in Chapter 8 when we compare these specifications with published informal specifications.

6 Comparing Solution and Program Specifications

This chapter is about the further refinement of solution specifications to program specifications. Given solution specifications for the writer priority data base, in which the approach of defining a single actor which is the readers/writers data base is already indicated, it is natural to try to write a program for that actor. One way to do this is to use one of the "structured synchronization primitives" which have already been introduced into the synchronization literature [Dijkstra, 1971; Hansen, 1972; Hoare, 1974; Hewitt, 1974]. First we will re-examine the solution specifications to see how one might expect to implement the actor and what properties one would like a programming language primitive for structured synchronization to have. That is, we will discuss an appropriate program structure.

Once this structure is established, we can look at a primitive that enforces the structure. The primitive chosen, the monitor, can be used to implement the solution.

The reasons for defining such programming language primitives in terms of behaviors become evident on consideration of a proof that a program which uses this primitive satisfies the readers/writers specifications. The chapter ends with some discussion of the sense in which the monitor is a "structured synchronization primitive."

6.1 How to Realize Synchronization Solution Specifications

Once one has a program and compares it to solution specifications, one would like to show that all properties of the solution specification are realized. When these properties are synchronization properties -- namely the adding of ordering constraints based on actor orderings

-- the easiest way to realize properties is to use a synchronizing primitive which has the same properties. However, this strategy cannot always be used unless for general solution specifications there are arbitrarily many special purpose synchronizing primitives available. For instance, for implementation of the writers priority data base one cannot expect to have a primitive which distinguishes two classes of requests and their terminations and favors one type of request in precisely the required way.

Given that there will be limitations on synchronization available the way to proceed is to divide the ordering information about behaviors into two classes -- information which can be picked up by a general purpose synchronizer, and information for which the synchronizing mechanisms will have to be programmed. For this latter class the technique to use is the recording of "state" information in cells which are local to the program. Then the numbers of requests of the two sorts can be recorded, updated and referred to, and ordering information can be left to primitive synchronization.

The important considerations in writing the program for a write priority readers/writers data base (or later in proving it correct) are

(1) How to preserve orderings of requests whether delayed or not. [Use synchronization primitives]

(2) How to map behaviors into states recorded in internal variables. [Use cells]

(3) How to protect sequences of operations which correspond to state changes or to state references, so that the record of the state and all observations of the state are valid. [Mutual exclusion of operations]

(4) How to obtain the ordering over both requests and ends of operations. [The ordering $=>_{rw\&s_c}$ in the solution specifications.]

We clearly can define actors to facilitate the programmer's doing any one of (1), (2), or (3) separately. Certain structured synchronization primitives will allow us to do all three and (4)

combined.[1] This structure will allow mapping of sequences of events in program execution into single events of the solution specifications. I.e., the program is being structured to correspond to the picture in Figure 5.1.

6.2 A Monitor-like Actor

Hoare [1974] defined a primitive, the monitor which is used here to illustrate the completion of a program for the solution specifications. The formal behavioral definition as a programming language is given in the Appendix. The exact form defined incorporates some of the simplifying assumptions which Hoare suggests. He saw no reason to believe these assumptions diminished the power of the primitive. Also, we take some liberties with syntax, mainly in bringing initialization of variables to the beginning of the monitor.

A brief informal definition at this point can be used to show immediately that the monitor can facilitate (1) - (4) above. Further argument is required to actually show that a particular program realizes the full details of readers/writers solution specifications, since that relies on correct programming within the monitor. The syntax of the monitor is shown in Figure 6.1.

---

[1] This can also be accomplished by disciplined use of unstructured primitives discussed later.

```
class monitorname : monitor
    begin declarations of data local to monitor ;
        initialization of data ;

        procedure procname ( ... formal parameters ...) ;
            begin ... procedure body ... end ;

            ...

        procedure procname ( ... formal parameters ...) ;
            begin ... procedure body ... end ;

    end ;
```

Figure 6.1: The Syntax of the Monitor.

Stated informally the main property of a monitor is that at most one procedure can be active at a time. Thus mutual exclusion of sequences of events corresponding to execution of procedure bodies is enforced. What is more, it will be enforced by fair synchronization. If the local variables (cells) are used to record state information, then, due to the synchronization, sequences of events which correspond to state changes are totally ordered. Thus there are obvious means both for the recording of the state information [consideration (2), above] and for its protection [consideration (3)]. Both the operations implementing checking in and those implementing checking out should be procedures in the same monitor, since both must change the state information. The fact that both requests and ends go through the monitor will implement the assumed ordering over both kinds of events [consideration (4)] used in the solution specifications.

In order to provide for the preservation of ordering of requests [consideration (1)], more than just the guarantee of fairness of the monitor as a whole must be provided for. As all requests will get into the monitor, the state can accurately summarize the behavior -- i.e., it can

take into account all requests. The only difficulty is that a request may get into the monitor while the state requires delay. In the write priority case a read request discovering a writer in the data base would have to wait. However, it cannot wait within the monitor, i.e. while maintaining it status as "the one process in the monitor." If it did this, no end of operation could get into the monitor and therefore the state could never be visibly changed and the system would deadlock.

To allow for a process' releasing the monitor without "losing its place in the ordering" Hoare includes "condition variables" among the kinds of data which can be declared local to the monitor. A condition variable is a synchronization actor which on receiving a "request" message causes additional ordering to be added to the behavior (i.e. it has no open state as do **protected-data-base** and **rw**). "Requests" correspond to the event

<cond$_i$ (apply: 'wait (then-to: c)) α *ec*>

represented in code as *condt.watt*. The ordering added is of the form

E => <c (apply:) α *ec*>

where E is of the form

<cond$_i$ (apply: 'signal (then-to: ?)) α ? >.

Specifying exactly which event E is required is somewhat complex. In fact, this part of the "structured" monitor can only be considered a very unstructured synchronization primitive. Detailed specifications of a related unstructured primitive, the semaphore, appear in Chapter 7, and probably should be read before the formal definition of the monitor in the Appendix. The formal specifications of conditions include both the waiting properties of a synchronization and additional releasing properties related to the overall monitor synchronization.

Several important properties of conditions are stated in the precise condition

specifications. First, the conditions are defined to preserve the orderings to which the monitor was exposed. That is, if two read requests E_{r_1}, E_{r_2} are received at the monitor in the order E_{r_1} => E_{r_2} and both find a writer in the data base, then their "wait requests" to the relevant condition

E_{c_1} = <cond (apply: 'wait (then-to: x_1)) α_1 ?>
E_{c_2} = <cond (apply: 'wait (then-to: x_2)) α_2 ?>

will be ordered E_{c_1} => E_{c_2}. This is because α_2 cannot even get into the monitor, let alone have E_{c_2}, until after E_{c_1}.

Furthermore, the events E_{c_i} which are events with conditions as target, and the events with targets that are continuations of such events, act as delimiters of sequences of events which are mutually exclusive. Thus if a procedure body has statements *cond.watt* or *cond.signal* [we assume that if *cond.signal* occurs, it occurs only as the last statement in the procedure body] then the "indivisible" sequences of events are those for code segments such as A, B or C in the following schemas:

```
begin
      A
      cond.watt ;
      B
end
```

or

```
begin
      C
      cond.signal
end
```

One final property of great importance is that if

<cond (apply: 'signal (then-to: c)) α *ec*>

occurs while there is a process waiting on the condition, it will not release the monitor but will instead directly hand it over to the continuation of the waiting process, i.e., a sequence of type B in some procedure body. Thus, in addition to the fact that sequences of events in executing type B code are ordered with respect to all other sequences of type A, B or C, it is also the case that once signalled, B sequences get priority over processes waiting outside the monitor (i.e. sequences of type A or C).[1]

6.3 A Program for Writers Priority

Figure 6.2 contains a program for the monitor for writer priority.

---

[1] Note the monitor requires for implementation several of the very properties for which it will be used in programming. That is, it has mutual exclusion and priority. Given the monitor properties we can implement rw, but this may appear a vacuous result if the monitor is as hard to implement. Implementation of the monitor discussed in Chapter 8 and in Hoare [1974] in terms of still more primitive synchronization shows that this implementation is possible.

```
class[2] writter prtority monitor
    begin
        readcount, writecount : integers ;
        busy : boolean ;
        readers, writters : conditions ;
        readcount := 0 ;  writecount := 0 ; busy := false ;

        procedure startread
            begin
            if writecount > 0 then  readers.wait ;
            readcount := readcount + 1 ;
            readers.signal ;
            end
        procedure endread
            begin
            readcount := readcount - 1 ;
            if readcount = 0 then  writers.signal ;
            end
        procedure startwrite
            begin
            writecount := writecount + 1 ;
            if readcount > 0 ∧ busy then writers.wait ;
            busy := true ;
            end
        procedure endwrite
            begin
            busy := false ;
            writecount := writecount - 1 ;
            if writecount = 0 then readers.signal else writers.signal ;
            end
    end
```

Figure 6.2: Write Priority Monitor.

This monitor contains the checking in and checking out procedures for readers and writers, namely *startread* and *endread* for reads and *startwrite* and *endwrite* for writes. Its operation is based on the following correspondence between the state information in the cells and

---

[2] The class statement according to Hoare [1974] allows us to define many separate instances of the same monitor. We will use this in the next figure.

the behavior desired of the solution in terms of events E_x, $E_{x_{in}}$, and $E_{x_{out}}$. All are stated such that if this state is observed while executing code corresponding to E_x or $E_{x_{out}}$ then the corresponding statement about the E_{x_i} and $E_{x_{i_{out}}}$ that preceded E_x or $E_{x_{out}}$ is true.

(writecount = 0) ≡ (number of E_w = number of $E_{w_{out}}$)[1]

(writecount > 0) ≡ (number of E_w > number of $E_{w_{out}}$)

[this could mean writes in data base and/or waiting]

(readcount = 0) ≡ (number of E_r such that E_r enabled = number of $E_{r_{out}}$)

[this does not include waiting reads]

(readcount > 0) ≡ (number of E_r that were enabled > number $E_{r_{out}}$.)

[whenever this is observed there are no E_r that are not enabled][2]

busy ≡ ((number E_w > $E_{w_{out}}$) ∧ (number of E_r that were enabled = number or $E_{r_{out}}$))

¬ busy ≡ (number of E_w = number of $E_{w_{out}}$) ∨ (number of E_r that were enabled > number $E_{r_{out}}$)).

The states which are checked for are

writecount > 0
readcount = 0
readcount > 0 ∨ busy
writecount = 0.

These correspond to states which are preconditions to one or another of the axioms for the

---

[1] This is an abbreviation, as are all, for the "number of preceding E_w" = number of preceding $E_{w_{out}}$." It also can be seen as implying ∀ E_w ∋ E_w => E implies $E_{w_{out}}$ => E where E is the event at which the state is being checked. Similar interpretations of the other statements hold.

[2] There is a time inside *startread* at which this is not true since technically all E_r on queue are enabled, but not all get signals instantaneously. Since the monitor is not released until it is true, this identity does hold at any time when the contents of *readcount* is examined.

solution and therefore a program based on these tests can have sufficient information about behaviors to implement the actor.

A program for readers/writers data base with write priority should declare its own local instance[1] of this writer priority monitor and use the monitor's procedures explicitly as checking in and checking out routines surrounding its data base operations. That will provide for the required "activator" properties in the program corresponding to those in the solution actor. Once a correspondence between events in the solution and segments of code in the program is made, a proof that the program induces the appropriate behavior can proceed. The ability to map sections of code into single events in the solution specifications comes from reliance on the monitor to implement mutual exclusion such that sequences of events for executing such sections of code are totally ordered and therefore are as "indivisible" as a single event. Figure 6.3 contains this program

---

[1] In some languages there is a question as to how to accomplish this private declaration. See discussion in Appendix and [Hoare, 1975].

```
readers-writers-data-base =
      begin
            y : record ;
            rw : writer priority[2]
            procedure read
                  begin
                  rw.startread ;
                  y := (read operations) ;
                  rw.endread ;
                  return y ;
                  end
            procedure write
                  begin
                  rw.startwrite ;
                  y := (write operations) ;
                  rw.endread ;
                  return y ;
                  end
      end
```

If *startread* corresponds to E_r, then sending a messages to the continuation (which is the rest of the sequence starting with the data base operation) does correspond to $E_{r_{in}}$. What is more, the continuation in $E_{r_{in}}$ includes the call to *rw.endread* and therefore the continuation enables ordering $E_{r_{out}}$ and E_r.[1] Finally, $E_{r_{out}}$ has continuation that can return the result to the continuation which came with the original request (by definition of procedure call). Thus due to a combination of the activator properties enforced by sequential code and the monitor properties, the required ordering $=>_{rw\&s_c}$ is formed.

Details of a proof now depend on validity of the state updates and use of condition definitions to ensure proper transitions once a state is recognized by a test. Now that the program is so strongly committed to the solution structure of rw, the easiest approach is to show that the separate axioms hold. Implicitly this relies on the proof that those causal axioms realize the high

---

[2] This is Hoare's syntax for declaring rw to be an instance of the writer priority monitor. It is declared private to the readers-writers-data-base.

[1] This corresponds to the continuation s_c in the solution specifications.

level specifications.[2] The proof will depend on arithmetic operations for updating states, the correspondence of values and states, and the ability to give priority to waiting processes internal to the monitor.

The following are examples of some of the axioms.

(1a) $E_w \ni E_w$ is first causes $E_{x_{in}}$

If E_w is first, it finds initial conditions, i.e. it finds *readcount* = *0* $\wedge$ *busy* = *false*.

$\therefore$ by property of conditionals and sequential code, this write gets through to the data base

$\therefore$ this events causes $E_{x_{in}}$

(3a) $E_r \ni \exists E_w, E_w \Rightarrow_{rw} E_r$ and $\neg E_{w_{out}} \Rightarrow_{rw} E_r$

causes

$E \Rightarrow E_{r_{in}}$ where E is the first in the set

$\{E_{w_{out}} | E_r \Rightarrow_{rw} E_{w_{out}}$ and the data base is opened to reads by $E_{w_{out}}\}$.

This is equivalent to entering *startread* and finding number of E_w > number of $E_{w_{out}}$, i.e. finding *wrttecount* > *0*.

By definition of conditionals this causes *readers.wait*

Now by definition the reader has to wait until it gets a signal.

Its signal could be from *endwrtte* or *startread*.

If from *endwrtte* it is due to $E_{w_{out}} \ni$ *writtecount* = *0* i.e., $\ni$ the number of E_{w_i} = number of $E_{w_{i_{out}}}$

$\therefore$ the signal is due to opening of the data base.

To show that it is due to the first opening of the data base, note that once the first such signal to readers occurs then due to *startread* and properties of conditions, <u>all</u> readers on the queue get off the queue before the monitor is released again.

---

[1] In a sense the causal axioms correspond to lemmas we would have to prove if trying to prove high level specifications directly. Thus the solution specifications show our ability to express such intermediate level properties. They are not necessarily easy to formulate or derivable automatically.

∴ the first opening is sufficient for enabling this read. This covers the case of its being signalled from *startread* because that also could only happen if the data base became open to reads.

(3b) $E_w \ni \exists E_{w_1} \Rightarrow_{rw} E_w$ and $\neg E_{w_{1_{out}}} \Rightarrow_{rw} E_w$

causes

$E_{out} \Rightarrow_{rw} E_{w_{in}}$ where E_{out} ends the last write before E_w.

The precondition is equivalent to *writtecount* > *0*

If *writtecount* > *0* then *busy* = *true*

∴ the writer gets onto the writers condition queue

∴ the one before it on the condition queue goes before it (by definition of the condition).

If no write is before it on the condition, the one in the data base has to do the signal and since that one is the last write before it, the condition holds.

∴ this causes $E_{w_{1_{out}}} \Rightarrow E_{w_{in}}$ where E_{w_1} is the last write before E_w.

In this manner one can show that the explicit causality of each axiom is enforced by the program. The solution specifications give structure to a proof of the properties. This separates the adequacy of causality arguments (Chapter 5) from implementation of causality arguments (program correctness relative to solution specifications).

6.4 Properties of a Proof of Correctness

The proof may seem no more or less revealing than other proofs of properties of programs that the reader has seen. However, there is at least one important way in which this proof differs from others about the readers/writers problem. That is in the fact that we are proving that a behavioral specification is satisfied by this code given behavioral definitions of the programming language.

The reader may be wondering why the use of behavioral specifications is stressed given

that final implementation depends on finding a state description. In fact even the solution specification depended on the notion of state, since the definitions of the data base being open at E is the definition of a state of the data base. The reasons for favoring the behavioral approach are based on comparison with attempts at specification of readers/writers problem by description of possible states of the system. This is done by means of invariant relations which must always be true of the variables in which the state is encoded. An example of an invariant which applies to the writer priority data base might be:

$$(\text{readercount} = 0 \wedge \text{writecount} \geq 0) \vee (\text{readercount} > 0 \wedge \neg \text{busy})$$

meaning that either a writer and no readers are in the data base or reads are in and no writers are in.

One problem with the state approach is evident in the use of this style of specification by Robinson and Holt [1975] for write priority. The priority property must be stated in terms of state transitions in order to constrain the situation is which a waiting read can be changed to an active read and allowed into the data base. Robinson and Holt can do this by adding the ability to talk about two values of each variable, such as 'readcount' and readcount, the first being the original value, the other the value after the change. For instance he states the condition

$$\min(1, \text{readcount} - \text{'readcount'}) \leq 1 - \min(\text{writecount}, 1)$$

so that state change increasing the number of readers can occur only if writecount is 0, i.e. if there are no waiting writers. Thus they are not really totally depending on properties of states rather than behavior. If the properties were based on sequences of state changes longer than one change, the Robinson and Holt specifications may not be at all useful.[1] At the very least they would encounter notational difficulty.

---

[1] One might also question whether use of min and max etc. has any intuitive value for high level specifications.

Indeed recognition of possible states in which a system can be is an important abstraction. However, there is a correspondence between states and behaviors which allows one to define. the states of a system as an equivalence relation over the possible behaviors. States as abbreviations for classes of behaviors are convenient notations whereas behaviors as sequences of state changes can lead to notational difficulties as in Robinson and Holt. Therefore this work emphasizes the reliance on behavior with states as abbreviations. This is particularly important for systems of communicating parallel processes in which specifications are so often about behavioral properties. Especially in dealing with actors such as the less structured synchronization in Chapter 7 one will need to write specifications dependent on behaviors that correspond to long sequences of "state transitions." For such definitions the behavioral approach is simplest and most direct for implementation independent specifications.

The reason for taking this view is related to our reasons for suggesting that programming language primitives should be defined as systems, and relates to a second problem with the state invariant specifications. In proofs of correctness, state invariants are often used as the assertion which must be shown true of the program [Hansen, 1972; Hoare, 1974]. The danger is in the misinterpretation of such invariants. The invariant as a statement about waiting requests and served requests is about meta-variables of a system (such as numbers of actual requests, etc. to the program being examined). Thus, the invariant should be a property of this relation among the numbers of events of various sorts. Instead, the invariant is almost always interpreted as a property which must hold for explicit program variables which contain state information. Then as long as these variables can be shown to be kept in a consistent state, the program is considered correct. This is independent of the properties of the program as a synchronizer due to synchronization primitives used. Therefore, even if unfair synchronization is used, the program can be proved correct. Although a write request may be locked out of the data base indefinitely by an unfair synchronization, as long as reads are served only when the

program variable for waiting writes is equal to 0, there need be no consideration of this actual waiting write. This approach enabled Hansen to prove correctness of his readers/writers solution using the unfair conditional critical regions [1972] although writer priority within the program can be superceded by lockout at the critical region. This flaw in his program is noted by Courtois et al [1972].

For the above reasons we find that the behavioral approach is better suited to semantics of synchronization than is a state invariant approach, even at the program level. The proof of correctness of the monitor solution to readers/writers includes reference to the properties of the program as it would appear as part of a system which is relying on it for data base protection. It proves correctness as a synchronization device, not just as code for keeping an internally consistent picture of an internal state. It is based on system state rather than data base state and proves the relevance of the program's internal state to the state of the system of which it is a part.

6.5 The Monitor as Structured Synchronization

In examining the details of how a proof of properties of the monitor implementation of readers/writers would be completed, it becomes clear why Hoare [1974] could not extend his proof technique to apply to many monitors. His attempt to find and prove an invariant relation for the monitor as a whole is not well-founded since in practice, the preservation of the invariant may depend on uses of the separate procedures. Thus while a procedure as it is used in the system may not invalidate the invariant, it may also be the case that this cannot be deduced from the invariant alone.

There is a question of what structure the use of the monitor can impose on programs. One measure of the structure of a program is the modularity of the code. The ability to take a block of code, prove it satisfies an invariant and then only use that invariant in place of the code

for proof of correctness of the rest of the program indicates the validity of that module as a block in the semantic as well as syntactic structure. Apparently for this reason Hoare wishes to prove invariants of monitors, thus emphasizing their meanings as modules in a system. However, there are intersecting logical divisions of the program as seen in the program (Figures 6.2 and 6.3) which foil this plan. For the purposes of mutual exclusion *startread, endread, startwrtte, endwrtte* must all be in one monitor. However, for the purposes of preserving reasonable properties of the data base, the blocks

startread ; read ; endread

and

startwrtte ; wrtte ; endwrtte

are appropriate "modules." When data base invariants for these block are proved relative to the synchronization properties of the monitor as a module, the program can be proved correct. However, no interesting invariant about variables private to the monitor need be provable for the monitor as a whole.

The invariant $\mathfrak{I}$

$$(\text{readercount} \geq 0 \wedge \text{writecount} = 0 \wedge \text{busy} = \text{false}) \vee (\text{readercount} = 0 \wedge \text{writecount} \geq 0 \wedge \text{busy} = \text{true})$$

might be a reasonable property to be maintained over the variables in the monitor. However, *endread* itself cannot be proved to preserve $\mathfrak{I}$. This is because if a process could enter *endread* when *readercount* = 0, then *readercount* could be reduced to less than 0. In fact, by the structure of the program in Figure 6.3 if we ever enter *endread* at all it is only under the condition (readercount $\geq$ 0 ...). This is not a property of the monitor but rather of the program which

forces the monitor procedures to be used appropriately. The invariant is true but it is not strong enough for proving that the invariant will be preserved.

Unless the program is structured so that all read operations are forced to be of the form *startread ; read ; endread*, we would have to prove that for all instances scattered throughout the system of operations for checking in and checking out there is a stronger $\mathfrak{I}'$ such that

$$\{\mathfrak{I}'\}\ \text{endread}\ \{\mathfrak{I}\}$$

and $\mathfrak{I}'$ always is true for *endread*. Thus the localization of this synchronization structure can seriously change the difficulty encountered in understanding a system. This part of the structure is the part most important for realizing goals of structured programming relating to producing understandable code.

7 Unstructured Synchronization

This chapter contains a definition of the semaphore. The definition is independent of implementation and illustrates the "lack of structure" of this primitive, in the sense that its presence alone in a system does not introduce easily abstracted time ordering properties such as mutual exclusion as does the presence of protected-data-base or rw. It can be used to build systems which do cause interesting time ordering properties, and one program for such a system is discussed.

Another definition, that of an unfair semaphore, i.e. one that can continually overlook some waiting requests, is given as well. By restricting the ways in which it can be used in a system one can see that structure and even fairness can be achieved under certain carefully chosen conditions.

The purpose of this chapter is to show first that the specification language so far developed is not prejudiced towards definition of structured synchronization, and second that the language is sufficiently powerful to give truly implementation independent specifications of this primitive. The value of such specifications is made evident by their use in specifying properties of programs which use the primitive. These specifications can convey the structure of the program directly without details of the implementation of the semaphore.

The chapter ends with a discussion of structure of synchronization as discussed here and its relation to properties of "structured programming language synchronization primitives" and the programs which use them.

7.1 A Definition of the Semaphore

The reader can find a number of definitions of a semaphore in [Dijkstra, 1968; Tsichritzis, 1974; Shaw, 1974; Hansen, 1972a; Haberman, 1970]. What most have in common is a dependence of the description on an implementation, generally in terms of "indivisible" sequences of events for updating and testing a cell's values.[1] As long as the cell's value is greater than 0 that value can be decremented Its value can always be incremented. A general semaphore is usually defined first and a binary semaphore, one which has maximum value 1 in its internal cell, given as a restricted semaphore. However, in implementing semaphores, the binary semaphore is usually built first [Shaw, 1974] from cells (or test and set)[2] operations and the general semaphore from it.

The binary semaphore contains the basic locking and unlocking which is characteristic of synchronization primitives and is the one we will examine. As a "gate," the semaphore becomes "locked" by any process that passes through. When locked, the semaphore causes others to wait. The gate remains locked until someone unlocks it. Unlocking an open gate has no effect.[3]

A fair binary semaphore, s, can be defined behaviorally, by reference to ordering relations induced on events of the following forms:

E_p = <s (apply: 'P (then-to: c)) $\propto ec_1$>
E_v = <s (apply: 'V (then-to: c)) $\propto ec_2$>

Since all events E_p, E_v have s as target they can all be ordered by $\Rightarrow_s$. In a single activator if there are further events then

---

1 Haberman [1970] is most like our definition by behavior since it summarizes history in an invariant and meta-variables. However, it is not clear that the meta-variables correspond to numbers of any physical events in a system. Also, he must have difficulty with a binary semaphore where the order of "events" (real or not) is of more importance than the numbers.

2 An indivisible group of updates with no conditions for waiting or queuing implicit, thus more primitive than a semaphore.

3 In a general semaphore unlocks can be "saved" so that if n unlocks are done while the gate is open, n processes can get through the gate before it will become locked again.

E_p --> <c (apply:) α ec_1'>
E_v --> <c (apply:) α ec_2'>.

We will refer to these events as $E_{p_{next}}$ and $E_{v_{next}}$, respectively. E_v always causes a next event, but for any E_p, there may be additional events required to cause $E_{p_{next}}$. These depend on the "state" of the semaphore where the state is as usual a class of (pasts) histories.[1] Unlike the cell in which one only has to look back as far as the last update to determine its state, with a semaphore one often has to look back over its entire history. The update to a cell changes its state and that state is stable and determines responses until the next update. Similarly, if the semaphore is "unlocked" or "open" then the response of an E_p is clear. However, the designation of all other histories as putting the semaphore in a "closed" state is less useful. It can stay closed indefinitely. In that case the effect of an E_p or an E_v is determined not by the fact that the semaphore is closed, but rather by the order of events since the last time it was open.

Not only must the total history be relied on, but in addition, no simplifying assumptions about this history can be made to facilitate its description. The kinds of assumptions that one might want to make are things like the fact that there will be no unlocks before there are locks, that all activators that get through a gate take responsibility for unlocking it later, or that no one unlocks a gate unless he locked it, etc. These are assumptions which are reflected in possible orderings of P's and V's in histories. What is more they are the kinds of assumptions on which structured specifications tend to rely. With **protected-data-base** or **rw** rather than **data-base** certain properties of => hold in <u>any</u> behavior. I.e., those properties would hold independent of the system of which the actor was a part and the kinds or numbers of messages it received.

This is not true of the semaphore. While it does have inherent properties which can be

---

[1] As per discussion in Chapter 2, when a set of events is totally ordered (in this case by $=>_s$) we can refer to the set of totally ordered events as a history, the events before an event E in $=>_s$ as the past, those after it as the future.

specified, any abstractions that can be made about behaviors of systems containing a semaphore are extremely dependent on the system of which that semaphore is a part. Whether or not a system has interesting time ordering properties will depend on whether the semaphore is used in a manner that enforces the orderings.

A standard way to enforce a property such as $E_1 \Rightarrow E_2$, where E_1, E_2 have different targets, and are in different activators, is to have an event, E_a, after E_1 which must be necessary to the occurrence of an event, E_b, before E_2. With a semaphore one might have $E_a = E_p$ and $E_b = E_v$. However, unless the semaphore is guaranteed to keep E_2 waiting until after E_1, no guarantee about time ordering can be made. For instance if a system with a semaphore has a process that is in a loop continually sending V's to that semaphore, then it is unlikely that anything can be said about the orderings imposed elsewhere in the system by that semaphore. If another process can unlock the semaphore, the ordering of E_1 and E_2 is not enforced.

Protected-data-base and rw had no control over how many or what kinds of requests they receive, but once they received messages they had complete control over setting up possible unlocking messages. Each s_c was good for one unlock and meant the end of the data base operations (this last depends an assumptions such as no parallelism in the data base, etc.). In other words, they imposed structure on locking and unlocking. The semaphore by itself is subject to receipt of arbitrary messages in arbitrary order with no structure necessarily imposed. Whether or not this means semaphores are "bad" is a subject for debate related to structured programming. The point of concern here is that since no pattern of the "locking" and "unlocking" messages in the history of a semaphore can be assumed, definition independent of a particular use and independent of implementation is a somewhat complex task.

First of all, since unlocks (V's) can no longer be assumed to mean "end of operation" and therefore to correspond naturally to some previous request, a means for talking about the effect of a V must be introduced. If the semaphore is locked, it is locked due to some P being let through. Thus the V terminates the locking due to that P.

Definition: E_1 terminates E_2 [written $E_1 = \mathfrak{T}(E_2)$] if

(1) E_2 is first E_p in $\Rightarrow_s$ and E_1 is first E_v after E_2

or (2) E_{p_1} is the last E_p before E_2 and E_1 is first E_v that is both after $\mathfrak{T}(E_{p_1})$ and after E_2.

Definition: E_2 is a terminating V event iff $\exists\, E_p \ni E_2 = \mathfrak{T}(E_p)$.

Examples of E_v's that are not terminating V events are E_{v_1}, E_{v_6} and E_{v_7} in the following sequence where E_{v_1} is first event in $\Rightarrow_s$:

$$E_{v_1}, E_{p_1}, E_{v_2}, E_{p_2}, E_{p_3}, E_{p_4}, E_{v_3}, E_{v_4}, E_{v_5}, E_{v_6}, E_{v_7}, E_{p_5}, \cdots$$

where $\mathfrak{T}$ is:

$$E_{v_2} = \mathfrak{T}(E_{p_1})$$
$$E_{v_3} = \mathfrak{T}(E_{p_2})$$
$$E_{v_4} = \mathfrak{T}(E_{p_3})$$
$$E_{v_5} = \mathfrak{T}(E_{p_4}).$$

The definition of semaphore will illustrate causality as "a relation between relations." Not only can sets ordered by $\Rightarrow_s$ be in preconditions. Now the relation of $E_v = \mathfrak{T}(E_p)$ is used. In a sense this kind of relation was implicit in the rw specifications, where E_w and $E_{w_{out}}$ were similarly related. However, since the event, $E_{w_{out}}$, was itself identifiable as the "matching" end, no explicit definition of the relation was necessary. In order to refer to an $E_w \ni \mathfrak{T}(E_w)$ was in the behavior, we simply spoke of an $E_w \ni E_{w_{out}}$ was in the behavior.

One further difficulty arises when attempting to define the semaphore to be "fair." The definition of fairness is somewhat complicated by the fact that there is no guarantee in the semaphore itself that there will ever be V operations.[1] If there are only P operations then clearly

---

[1] This contrasts with the other aspect of the lack of structure, namely the lack of matching referred to earlier in due to which there can be too many V's, some of which will not be terminating V's

all activators but the first will be stopped forever (i.e., will not have $E_{p_{next}}$'s and therefore will have no further events). Thus one cannot say the semaphore is unfair simply because there are conditions under which there can be events E_p with no $E_{p_{next}}$ in a behavior. However, in any behavior in which there are a "sufficient number" of E_v's it must be the case that all E_p's are followed by $E_{p_{next}}$. "Sufficient" may be defined relative to some bound on the number of E_v's that occur after E_p such that $E_{p_{next}}$ may still not be in the behavior. This bound may be a function of the number of E_p's in the past. For any behavior in which $\exists\, E \ni E_{p_1} \Rightarrow_s E$ and the number of V's between E_{p_1} and E is greater than $\mathfrak{B}(n)$ where n is the number of unenabled E_p's before E_{p_1}, then $E_{p_1\,next}$ is in the behavior. For instance, as we shall see in section 7.2, given a structure in which every $E_{p_{next}}$ eventually causes an E_v, there should be no E_p that is not followed by an $E_{p_{next}}$.

The easiest way to anticipate fairness criteria for any use of semaphores is to rely on the actor ordering $\Rightarrow_s$ in specifying the choice of the next process to be let in.[1] In section 7.3 we will discuss the alternatives.

The axioms for a semaphore are:

<cons-semaphore (apply: (then-to: c)) α *ec*>

causes

<c (apply: s^*) α *ec'*>

where the following properties of s hold.

---

[1] If $\Rightarrow_s$ is interpreted as the "incidental order" in which messages arrive at the semaphore, this corresponds to a FIFO queuing algorithm based on time of arrival of request. It also corresponds to $\mathfrak{B}(x) = x$. Since all P events are of the same form there is no other grounds for providing fair selections in a semaphore. There can be fair algorithms other than FIFO provided other properties are visible, either by extending the model or by having events of differing forms. We have seen a non-FIFO, but fair algorithm in Hoare's fair readers/writers which while it doesn't necessarily serve requests in order, does serve all requests. When algorithms other than FIFO are used, argument that the algorithm is indeed fair is required.

(a) V's always are completed.

$$E_{v_1}$$

causes

$$E_{v_{1_{next}}}.$$

(b) A new semaphore is open.

$$E_1$$

where E_1 is the first E_p in $\Rightarrow_s$

causes

$$E_{1_{next}}$$

(c) Every V that ends a P enables the next P. V's that don't end P's have no effect. [The V that ends a P either reopens the semaphore or releases the next waiting P.]

$$E_{p_1}, \ni \exists E_p \text{ preceding } E_{p_1}$$

causes

$$E_v \Rightarrow E_{p_{1_{next}}}$$

where $E_v = \mathfrak{T}(E_{p_1})$ for E_{p_1} the last E_p before E_{p_1}.

This last clause includes the orderings $E_v \Rightarrow E_{p_{1_{next}}}$ which could have been derived from $\Rightarrow_s$. I.e., if $\forall E_p \Rightarrow_s E_{p_1}$, $\mathfrak{T}(E_p) \Rightarrow_s E_{p_1}$, then E_{p_1} causes $E_{p_{1_{next}}}$. However, since all we care about in this definition of semaphore is being able to characterize the $\Rightarrow$ it induces, and not a description of when an implementation would have to actually add constraints, this definition suffices. The

reason that we are only interested in this kind of definition is that the semaphore will only be used as a primitive in this paper. Therefore all that need be known are the time ordering properties that can be relied on if it is used. The definition by causal axioms clarifies the reliance on the actor's ordering to assure that all E_p and E_v events are ordered in any time ordering derived form a system including semaphores.

The reliance on the definition of $\mathfrak{T}(E_p)$ to pinpoint the "matching" unlocks is essentially the point on which our distinction between structured and unstructured synchronization hangs.[1] With the **protected-data-base** defined previously we could predict the form of the enabling event to the extent that it was known to correspond to the termination of an operation. In the **protected-data-base** we knew exactly which event it would be. The end of the preceding operation would occur in a predictable activator, etc. With a semaphore all that is known about the terminating V is that it will be of the form E_v.

A point to note about these specifications is that while they rely on more than just what preceding events occurred, (i.e., they rely on relations among those events), the fact that all these events have **s** as target should ensure the realizability of the specifications. An implementation of the semaphore has to be able to recognize the fact that a past E_p has been terminated. Since the recording of the number of E_v's is not sufficient to derive that presumably a record of the relation must be kept. Although the record of this relation is theoretically ever increasing in size as more pairs are added, one should note that this does not have to imply a requirement of ever increasing space in an implementation. The recording of this relation can be implemented in any of a number of ways. A queue from which E_p is removed once $\mathfrak{T}(E_p)$ occurs could be used. Then "E_p on queue" ≡ "$E_p \ni \mathfrak{T}(E_p)$ has not occurred yet." Similarly, if being implemented using only cells doing busywaiting, the responsibility is shifted. The "recording of the relation" could be implemented by the fact that process α of E_p ceases its busywaiting examination of cells once it finds an appropriate condition is set [i.e. once $\mathfrak{T}(E_p)$ occurs].

---

[1] It will be seen again in the definition of the condition in the monitor.

7.2 The Semaphore Implementation of Protected Data Base

The semaphore can be used to implement the protected data base by using it in a program such as that in Figure 7.1.

```
procedure protected-data-base (x)
    semaphore x ;
    P(s) ;
    y := data-base (x) ;
    V(s) ;
    return (y) :
    procend
```

Figure 7.1: Semaphore Implementation of Protected Data Base.

Thus when *protected data base* is sent a request, x, with continuation c, it sends a message 'p with continuation, c_i, to the semaphore. This c_i is the rest of the sequence of statements in the procedure body. On receiving acknowledgement from the semaphore, in the form of the message (apply:), c_i causes transmission of x to the data base, followed (on return of response put into y) by V'ing the semaphore, s, and returning y to the original continuation, c.

When the fair semaphore as defined above is used in this manner, the mutual exclusion of data base operations is implemented. What is more, there can be no lockout. This property is derived partly from the semaphore definition and partly from the particular use here. The fact that every P operation that is released will eventually be followed in that activator by a V operation, combined with the knowledge that V's can only happen under that circumstance, (i.e. after a P), makes two points clear. First, there will always be more V's in the "future" than there were unenabled P's in the past, allowing any process that does a P operation to continue. Also, $\forall$ E_p, $\mathfrak{T}(E_p)$ is now easily identified, making a description of how this program works considerably simpler than that of the semaphore to write.

These points can be shown to be true as follows. Since it is the case that

<s (apply: 'P (then-to: c)) α *ec*> --> data base sequence of events --> <s (apply: 'V (then-to: r)) α $ec^{(x)}$>

we can also say that

$$\langle s \text{ (apply: 'P (then-to: } c_i)) \ \alpha_i \ ec_1 \rangle \Rightarrow_s \langle s \text{ (apply: 'P (then-to: } c_j)) \ \alpha_j \ ec_2 \rangle$$

causes

$$\langle s \text{ (apply: 'V (then-to: } r_i)) \ \alpha_i \ ec_1^{(x)} \rangle \Rightarrow \langle c_j \text{ (apply:)} \ \alpha_j \ ec_2' \rangle$$

where $\langle s \text{ (apply: 'V (then-to: } r_i)) \ \alpha_i \ ec_1^{(x)} \rangle$ is the first V in α_i after that P operation.

If the first V after E_{p_1} in the same activator is referred to as $E_{p_{1_{out}}}$ then this can be stated

$$E_{p_1} \Rightarrow_s E_{p_2}$$

cause

$$E_{p_{1_{out}}} \Rightarrow E_{p_{2_{next}}}$$

I.e., if E_{p_1} in α_i is E_x then the event $E_{x_{out}}$ is the event $\langle s \text{ (apply: 'V (then-to: } r_i)) \ \alpha_i \ ec_1^{(x)} \rangle$ which releases releases the next E_p. This clearly is related to the **protected-data-base** structure axioms and should be recognizable as mutual exclusion specifications. It can also be interpreted as a specification of the structure of a mutual exclusion solution. That is, the crucial events which delimit the critical sections are mentioned and causal relations among them which are necessary for enforcing mutual exclusion are specified. This specifies the structure of a program. The implementation of the causality can then be left to semaphore which can cause waiting and addition of dependencies. What this specification says is exactly where the semaphore operations should appear in the program.

The structure of the program guarantees that there could be no other E_v in the behavior $\ni E_{p_1} \Rightarrow_s E_v \Rightarrow_s E_{p_{1_{out}}}$. This axiom shows clearly that this program enforces mutual exclusion of data base operations. There also can be no extraneous V's, since due to the fact that all V's occur only after P's, it can be shown quite easily that every E_v is a terminating V event.

Fairness in the sense of protected-data-base, namely that every request is served, can also be shown to hold. The argument is based on the activator properties due to the structure of the solution and the properties of the semaphore as it was defined in terms of $\Rightarrow_s$. By induction all E_{p_1}'s preceding any E_p have $\mathfrak{T}(E_{p_i})$. Therefore the conditions for $E_{p_{next}}$ are always met and the data base operation can proceed. The structure of the program ensures that at any point in the history of the semaphore the number of V's in the "future" is always greater than or equal to the number of unenabled P's in the "past." This combined with the fact that the semaphore was defined to be fair relative to that condition, ensures fairness of the structure synchronization here enabled.

To repeat, the structure of the program is the structure of mutual exclusion. The property

$$E_{p_1} \Rightarrow_s E_{p_2}$$

cause

$$E_{p_{1_{out}}} \Rightarrow E_{p_{2_{next}}}$$

is a property of this program. It is not the high level mutual exclusion specification. Nor is it simply the definition of a semaphore. It is a description of the events of this particular program which involve the semaphore -- a description of *how the semaphore is being used* in this case. It captures the meaning of the program template

```
procedure ...
        semaphore s ;
        P(s) ;
            ...
        V(s) ;
            ...
        procend.
```

7.3 An Unfair Semaphore

Often in definitions of the semaphore it is only said that if processes are waiting when a V operation occurs, then that V will enable some waiting process (e.g. [Courtois et al, 1970]). The point is to avoid committment to any particular scheduling algorithm. However, if one cannot even assume that the algorithm that will be used is in any sense fair then the usefulness of the semaphore might be impaired.

Taking the statement of choice literally, a behavioral definition of a semaphore can be written in which V messages simply cause the enabling of one of the set of already waiting processes. We can then investigate the resulting restrictions on the properties of the semaphore which can be relied on.

With the fair semaphore the choice was by first in $=>_s$ of a set of elements E_p $\ni$ there is no $\mathfrak{T}(E_p)$ in the behavior. Each time an E_p was chosen, the next E_v became $\mathfrak{T}(E_p)$, therefore E_p was removed from the set. If instead some arbitrary element in the set is chosen, the FIFO discipline is eliminated and in fact the semaphore so defined is demonstrably unfair. If every V in a behavior finds more than one E_p waiting there is no guarantee, even given an unbounded future number of V's, that all P's get through. [I.e. there is no such thing as a sufficient number of V's] The ordering

$$E_{p_1}, E_{p_2}, E_{p_3}, E_{v_1}, E_{p_4}, E_{v_2}, E_{p_5}, E_{v_3}, \cdots$$

can cause behavior in which $E_{p_{3_{next}}}$ never occurs if the following relation is defined:

$$E_{v_1} = \mathfrak{T}(E_{p_1})$$
$$E_{v_2} = \mathfrak{T}(E_{p_2})$$
$$E_{v_3} = \mathfrak{T}(E_{p_4})$$

One way to represent the choice of next enabled P which a $\mathfrak{T}(E_p)$ makes is to define another relation $E_v = \mathfrak{R}(E_p)$. It is defined in the axioms for s, and depends on a definition of the set $W(E_v)$ of waiting events and a revised definition of $\mathfrak{T}$.

Definition: $E_1 = \mathfrak{T}(E_2)$ if either

(1) E_2 is first E_p, and E_1 is first E_v after E_2

(2) E_2 finds the semaphore open and E_1 is the first E_v after E_2

or (3) E_2 is such that $E_2 \Rightarrow \mathfrak{R}(E_2)$ and E_1 is the first E_v after $\mathfrak{R}(E_2)$.

Definition: $E_{p_1} \in W(E_{v_1})$ if $E_{p_1} \Rightarrow_s E_{v_1}$ and

(1) E_{p_1} is not the first E_p in $\Rightarrow_s$

(2) the semaphore was not open for E_{p_1}

and (3) not $\exists\, E_v \ni E_v \Rightarrow_s E_{v_1} \wedge \mathfrak{R}(E_{p_1})$.

Then the axioms for the unfair semaphore s are:

(a) The first P always gets in

$$E_1 \ni E_1 \text{ is the first } E_p \text{ in } \Rightarrow_s$$

causes

$$E_{1_{next}}$$

(b) If all preceding P's are terminated, and the semaphore is open, the first P gets in and the last V that terminated a P is the releasing V.

$$E_1 \ni E_1 \text{ is not the first } E_p$$
$$\text{and } \forall\, E_p,\ E_p \Rightarrow E_1 \supset \mathfrak{T}(E_p) \Rightarrow_s E_1$$

causes

$$E_{1_{next}}$$
$$\text{and } E_v \text{ last in the set } \{\mathfrak{T}(E_p) \mid E_p \Rightarrow_s E_p\} \text{ has } \mathfrak{R}(E_{p_1})$$

(c) A terminating V that finds waiting P's releases one of them

$$E_v \ni \exists E_p,\ E_v = \mathfrak{T}(E_p)$$

causes

$$E_{P_{next}} \text{ for unique } E_p \in W(E_v)^1$$
$$E_{V_{next}}$$
$$\text{and } E_v = \mathfrak{R}(E_p).$$

Note that this definition implies that if $W(E_v) = \{E_p\}$ then E_p is released by E_v. I.e., whenever exactly one process is waiting it is enabled by the next V. Also, whenever no process is waiting, a terminating V opens the semaphore and therefore the next P is guaranteed to get in. Thus the only way in which a process can get stuck at a semaphore is if there is always more than one process waiting.

7.4 Using Unfair Semaphores

The same protected data base implementation as in section 7.2 can be done with unfair semaphores and clearly the solution cannot be proved fair. If the specifications were simply guaranteed mutual exclusion, this might be satisfactory.

Unfair semaphores can cause ordering properties identical to those of a fair one for certain restricted behaviors. If, due to properties of the system causing messages to be sent to the semaphore, it can be guaranteed that at most one P will ever be waiting at a locked semaphore, then V operations are always guaranteed to release that one P.[2] Within this restricted set of histories a terminating V always releases the "next" P after the one it is terminating. Examples of

---

[1] Compare to Parnas' [1975] hidden functions definition which depend on set operations.

[2] Similarly, fairness although not necessarily FIFO, can be guaranteed relative to any usage guaranteed to bring the number waiting down to zero arbitrarily often in any behavior.

this kind of use might be in a semaphore being used in mutual exclusion style to protect a resource which is held for an extremely short time relative to the times involved in computations outside the resource and known frequency of use of the resource.[1] Such assumptions can break down, and a system built with unfair synchronization and relying on appropriate use, can begin to have "incorrect" or undesirable behavior once usage changes. If fairness is important, then such a system cannot be considered correct except relative to certain conditions. It is not guaranteed to behave correctly in all environments and under all conditions of use.

The next chapter contains a program of Courtois et al [1971] which relies on unfair semaphores for certain properties. In order to guarantee certain properties extra semaphores are used to ensure that at a certain key semaphore at most one process will ever be waiting. This restriction of behaviors involving that semaphore is independent of usage of the program as a whole. Thus that semaphore can process the messages it receives exactly as if it were a fair semaphore.

7.5 Unstructured Synchronization

The semaphore is an unstructured synchronization primitive in the sense that while it has "lock" and "unlock" operations, there is no relation enforced by the primitive itself between the locking and unlocking. I.e., there is no property comparable to that of **protected-data-base** or **rw** which allowed prediction of what event would unlock the synchronization for a waiting process. In **protected-data-base** it was always the end of the last data base operations. Thus the entire event, including its activator could be predicted. By contrast, with semaphore, all one can say is that a "V" operation, if it occurs will enable a waiting activator. What is more the property

---

[1] This is the assumption Hansen makes in conditional critical regions solutions for which fairness is not guaranteed. He claims that the only reasonable assumption about usage is that variables determining use of the data base are not scarce resource [Hansen, 1973]. I.e. that it is not likely that lots of waiting would occur for access to variables determining the state of the data base.

of whether an event represents a waiting process or not can no longer be indirectly specified by whether or not its ending event has occured. Events must be designated as having ended particular P operations.

The information available in these other actors (**protected-data-base** and **rw**) can be a property of a program in which semaphores are used in a particular manner, but they cannot be said to be properties of the semaphore itself.

8 Using Unfair Synchronization

This chapter contains an analysis of a program meant to solve a Writer/Priority Readers Writers Problem using semaphores which cannot be assumed fair. The specifications of this program and of the solution it implements are quite different from those stated in Chapter 5. The differences and the reasons for them are discussed. Some formal properties of the solution specifications can be shown to hold. Other properties are environment dependent specifications which the high level behavioral semantics given earlier were not meant to anticipate. The wide range of specifications that could be relevant to evaluating synchronization programs is sampled with consideration of how our specification language might have been used had the particular points about the running environment been included in the writer's picture of his needs.

The chapter continues with an examination of another solution written with synchronization primitives which have been specified with varying degrees of formality and "completeness." Implementation of environment dependent specifications, comparison of power of various primitives, and styles of programming are discussed.

8.1 The Original Problem

Courtois et al set themselves a problem, defined carefully to be implementable with semaphores which could not be guaranteed to be fair. Specifically they did not require that any sort of fairness hold, since they knew that there was no guarantee that their semaphores could provide it.[1]

---

[1] They mention explicitly, that if they did not restrict themselves to simple semaphores with no fairness guarantee they might be able to accomplish more.

In terms of => it is still the case that $\forall S_r, S_w$ in a behavior $S_r \Rightarrow S_w$ or $S_w \Rightarrow S_r$ and $\forall S_{w_1}, S_{w_2}$ in any behavior $S_{w_1} \Rightarrow S_{w_2}$ or $S_{w_2} \Rightarrow S_{w_1}$. However, there is no guarantee of fairness to readers or to writers and no guarantee that all requests can be totally ordered. The only requirement in addition to the readers/writers property is the following statement of writer priority: "Once a writer is ready to write, he performs his 'write' as soon as possible. ... to meet this requirement a reader who arrives after a writer has announced that he is ready to write must wait *even if the writer is also waiting*."

The phrase "as soon as possible" is a typical informal problem specification phrase which does not convey sufficient information by itself to be implementable. There is no specification of the notion of "time" being used or of what comparisons are to be made in determining that the amount of delay has indeed been minimized (or speed maximized). There could be many interpretations of this phrase. If readers are already in the data base when a writer announces, then it may be that the "quickest" way to serve the writer is to interrupt all the reads and let the writer in immediately after. It may be faster to let him in immediately and to simply warn all readers that were in the data base that they probably got bad information [Lamport, 1974] and that they should reread.[1] Courtois et al evidently did not mean to maximize speed in this sense. Also, since they explicitly accepted the fact that once a writer has announced, that writer may still get locked out if many other writes are waiting, they clearly did not even mean by "as soon as possible" to imply guarantee of writing.[2]

This is an example of mixture of program specifications and specifications of runtime properties. There is no sense of "as soon as possible" which will be meaningful for this program over all possible running situations. Some interpretations of "good" solutions can be specified but

---

[1] No fairness to reads is required so that there is no need for concern over the fact that this could potentially keep readers rereading forever as new writers appear.

[2] Regarding this property, the best that can be said is that as long as a writer is trying, some writer (not necessary the one trying) gets in.

no one solution is particularly specified by this informal phrase. For the time being we will analyze the program relative to the solution specifications implicit in the statement " ... a reader who arrives after a writer has announced that he is ready to write must wait ... ."

The main point besides the reader/writer property to be captured in the Courtois specifications is the fact that the solution is deliberately not meant to assume a single actor solution. Readers and writers have their own entrance actors which happen to use a common channel of communication for checking writer priority. All writers must be announced and all readers must check. Writers cannot be prevented from announcing (even if a particular writer can be locked out).

Thus Courtois et al really were specifying the use of a communication channel between readers and writers as well as mutual exclusion of writers. The communication channel must be used in a particular way. If properly used, the communication channel will cause the following properties to hold. If E_{aw} is the event in which a group of writes announce themselves and E_r the one in which a read checks for writes:

> (1) S_r is in the behavior iff E_r is in the behavior and either E_r found no announcement or was released due to there being no writers still claiming the data base.
>
> (2) If $E_w \ni E_{w_1} \Rightarrow E_w \Rightarrow E_{w_{1_{out}}}$ and E_{w_1} is an E_{aw}, then no reads that didn't precede E_{w_1} get in before E_w.[1]

This last requirement could be realized by the following property, where $\Rightarrow_c$ is the ordering observed at the communication channel.

> $E_r \Rightarrow_c E_w \supset S_r \Rightarrow S_w$,
>
> $E_w \Rightarrow_c E_r \supset S_w \Rightarrow S_r$ (if S_r in the behavior).

---

[1] All reads are ordered with respect to announcing writes. Therefore all reads either do or do not precede E_{w_1}.

and the requirement that all writes either announce themselves or satisfy themselves that writes are announced. Thus the requirement (2) is

$$\text{If } E_w \ni E_{w_1} \Rightarrow E_w \Rightarrow E_{w_{out}} \text{ and } E_{w_1} \text{ is an } E_{aw}, \text{ then } S_r \Rightarrow S_w \text{ iff } E_r \Rightarrow_c E_{w_1}$$

Write announcements can cause reads to wait, but if a read does check before a writer announces, then that read can get in. There is no implication in such specifications of total ordering of all requests or of fairness to any individual request. The priority can be passed along from write to write by letting each write that arrives while writes are still in the data base "inherit" the current announcement. Thus writes only have to race with reads for the common channel when the data base has no waiting writes. By having writers communicate through the same channel when giving up the data base, the events necessary for S_r to be in the behavior, i.e. the "unannouncing" of writer, can be specified and causal specifications given. A particular solution would have to include a communication channel through which a read can relinquish the data base after it gets it.

8.2 The Courtois, Heymans, and Parnas Program

The solution that Courtois et al present requires cooperation of the following sorts. All writers must use a single semaphore to achieve mutual exclusion and readers and writers must announce themselves through some common semaphore so that priority can be achieved. Since the program is written in terms of semaphores which may be unfair, readers and writers cannot simply pass through one common semaphore. If they did, then whenever a group of readers and writers were waiting, there would be nothing that could be said about which was selected next after a V operation. No priority could be given, since any attempt in the program to give priority to writers would be foiled by the possibility of writers accidentally never getting through the common semaphore. Thus Courtois et al go to great lengths to guarantee that at most one process can ever be waiting at any common semaphore. The solution is presented in Figure 8.1.

The only common semaphores, *r* and *w*, must be protected so that at most one process can be waiting at either. Thus *mutex 3* is used to let only one reader at a time to the common semaphore *r*. *Mutex 1* protects the *readcount* variable, *mutex 2* the *writtecount* variable. The semaphore *w* is used to indicate that the data base is busy. The data base can be busy either due to some readers or one writer. The semaphore *r* is used for writes to announce their intention to take priority. Reads all check for that and are stopped if writes are already waiting. If the first in a set of reads and a writer are both making progress towards entering, due to *r*, only one will. Thus the ordering at *r* can determine which gets through first (this only applies in cases where the data base is opened instantaneously and is used in deciding whether the data base is recaptured by the writers or not.) No more than one of that group of reads can get in before the competing write, once both get to *r*. Readers can only examine *readcount*, writers only *writtecount*. To implement a solution in which both kinds of processes examine both variables, a common semaphore would have to be used reintroducing the lockout problems.

```
integer readcount, writecount ; (initial value = 0)
semaphore mutex 1, mutex 2, mutex 3, w, r ; (initial value = 1)

READER                                  WRITER
P(mutex 3) ;                              P(mutex 2) ;
    P(r) ;                                  writecount := writecount + 1 ;
        P(mutex 1) ;                          if writecount = 1 then P(r) ;
        readcount := readcount + 1 ;        V(mutex 2) ;
        if readcount = 1 then P(w) ;      P(w) ;
        V(mutex 1) ;
    V(r) ;
V(mutex 3) ;
        ...                                 ...
    reading is done                         writing is done
        ...                                 ...
P(mutex 1) ;                              V(w) ;
readcount := readcount - 1 ;            P(mutex 2) ;
if readcount = 0 then V(w) ;              writecount := writecount - 1 ;
V(mutex 1) ;                              if writecount = 0 then V(r) ;
                                        V(mutex 2) ;
```

Figure 8.1: Courtois, Heymans and Parnas Solution to Second Readers/Writers Problem

Note that only the first in a group of readers and the first in a group of writers communicate through both of the common semaphores. (All reads go through r but only first through w, all writes go through w but only on through r.) Since the state variables containing numbers of readers (*readcount*) and numbers of writers (*writecount*) are used only by readers and writers, respectively, a reader does not check explicitly for number of waiting writers. It only checks the number of readers and coordinates with writers implicitly through the semaphores.

The arguments required for proving that this program satisfies the specifications depend on the definition of a semaphore and properties of the particular uses of semaphores within this program. To prove mutual exclusion of writes the mutual exclusion pattern of $P(w)$ and $V(w)$ in the writer program must be recognized and it must be shown that the $P(w)$ and $V(w)$ in the readers program cannot cause mutual exclusion to could be violated. In particular, while

the reader's $P(w)$ can hold up writers, the reader's $V(w)$ can never occur while a writer is in the data base and therefore cannot cause violation of mutual exclusion of writers.

Now before the communication channel property can be handled, some properties of its use must be checked. This requires proof that at most one process can ever be waiting at r and that writes always either go through r or check that some other write did. Due to properties of conditionals and since writes are ordered at *writtecount*, if *writtecount* is greater than 1 then there is an E_{aw} preceding the write. Given these facts, proof of the following are straightforward:

$E_r \Rightarrow_r E_{aw} \supset S_r \Rightarrow S_w$

$E_{aw} \Rightarrow_r E_r \supset S_w \Rightarrow S_r$

$\forall E_w \ni E_{aw}$ is its announcement, $S_r \Rightarrow S_w$ iff $E_r \Rightarrow_s E_{aw}$.

Thus with appropriate filling in of details in the preceding outline, one can prove that this program does satisfy those communication requirements that we can extract from the informal Courtois specifications and formalize. The question remains whether these extracted properties are all the specifiers had in mind. Presumably even a single person writing specifications and implementing them can make a mistake and not implement his intended specifications. If the implementation is taken as the precise statement of the meaning then it cannot be incorrect. If the precise meaning simply was not well expressed in the informal specifications, and the program is meant to implement some precise specifications, how can we judge correctness? The following section notes one interpretation taken by readers of Courtois specifications by which the solution is inadequate.

8.3 An Alternative Interpretation

As a result of the placement of communication operations in the sequence of operations in entrance code there can be some discrepancy between the orderings by which one might

naturally try to judge performance and orderings acknowledged by the system. If one assumes that by some global clock, a read coming to *mutex 3* can be ordered with respect to a read coming to *mutex 2*, then one can find fault in this solution since a writer can request entrance (by coming to *mutex 2*) before a read does and yet fail to enter the data base before that read does. [The reader should be able to see this happening when exactly one writer has been in the data base and is checking out while these requests appear.][1] This violates the scheduling specification which states that a reader cannot get in while a writer is waiting.[2]

To evaluate this "failing" we must consider what specifications we are using. If the high level specifications were of the Chapter 5 variety in which priority was defined relative to some total ordering on all requests, indeed this solution may be incorrect. However, the writers specifically said they did not expect to have readers and writers go through a common semaphore. Thus they deliberately used independent actors leaving this program with no physically meaningful basis for an ordering $=>_{mutex2\&3}$.[3] This ordering is not a system property and cannot be used to determine time orderings.

If the high level specifications require relative ordering, then this solution could be ruled out on the grounds that the two independent semaphores do not implement such an

---

[1] This timing sequence was pointed out to the author by Ellis Cohen.

[2] The reader could compare this to the situation in the monitor solution in which at a time when a write is finishing, a queue E_r, E_w, ... is formed outside the monitor. The read being first will enter to find the data base empty and will get in even though there is a writer waiting on the queue. Somehow waiting on that queue is still not equivalent to "waiting." Similarly, It should be clear that Courtois et al did not intend writers waiting at the semaphore *mutex 2* [or even writers simply inside their first critical section] to be considered "waiting writers." Those semaphores are there for preserving the validity of the current count of waiting writers. Only after a writer has both changed this count and explicitly put itself into a waiting state on one of the semaphores for communications (*r* or *w*) is it recognized as waiting (i.e., has it "announced" its readiness to write).

[3] Compare to $=>_{rw\&s_{c_i}}$. There we were writing prescriptive specifications and requiring that *rw* and s_{c_i} be implemented by actors that had common ordering. In this case we would be taking two independent actors which already have specifications (they are unfair semaphores) and basing descriptive specifications on $=>_{mutex2\&3}$. There is no basis for that in the specifications of the primitive.

ordering. However, recall that this is just one possible interpretation of what the "real" high level specifications are. In the next section we consider programs which are claimed to satisfy the stricter specifications.

8.4 Relative Power of Unfair Primitives

With only the unfair semaphore available Courtois et al had to set fairly modest requirements in stating their problem. However, even those requirements, when interpreted as stated above, have been are used in support of arguments that semaphores are insufficiently powerful to solve this readers/writers problem. The basis for this power difference is not related to fairness but rather to the number and kinds of operations that can be performed as single operations. One standard "extended" semaphore is the semaphore array where $P(s_1,s_2)$ is completed iff s_1 and s_2 are both greater than zero. Similar primitives can be made available in a programming language or can be programmed using a monitor. In this and the next section an extended semaphore solution to the readers/writers problem is examined for how well it satisfies the Courtois specifications. This section contains an unfair synchronization version, the next a fair one.

No claim has been made that if one takes the view that the priority specifications are relative to a single total ordering on requests, unfair semaphores can solve the readers/writers problem. However, there are claims [Presser, 1975; Lipton, 1974] that solutions with more powerful primitives can "solve" the readers/writers problem.

For instance, using his own representation of a primitive up/down which can test and change arbitrarily many variables in one step, Lipton presents the following solution: [Each line has an indivisible operations on it.]

READERS

(1) if $s = 0 \wedge b = 0$ then $a := a + 1$;
(2) read
(3) $a := a - 1$;

WRITERS

(4) $s := s + 1$;
(5) if $a = 0 \wedge b = 0$ then $b := b + 1$;
(6) write
(7) $b := b - 1$;
(8) $s := s - 1$;

This program satisfies the following specifications: (4) "stops" (1), but (1) cannot stop (4). Taking this as a specification[1] Lipton shows that no semaphore program can have two steps with that property.[2]

Our main question about the claim that semaphores cannot solve the readers/writers problem is whether the up/down primitive as specified does in fact enable a better solution. The possible misleading factor in this is that no specification of scheduling is made for up/down. The reason that this may be important is that certain implementations of the up/down can be subject to the same criticisms that can be directed at Hansen's conditional critical regions. Conditional critical regions are primitives for programming in which the programmer can specify tests and operations to be performed on shared variables with mutual exclusion enforced but in which he has no scheduling facilities. The conditional critical region enforces mutual exclusion

---

[1] It is not clear whether it is meant as a full specification of readers/writers or as a single property common to all programs for readers/writers.

[2] Thus Lipton is implicitly only accepting implementations of stopping specifications in one step. Results might be different if he allowed for the stopping of sequences by sequences of steps. Lipton and Snyder are currently working on generalizing their definitions to include such solutions, with measures of the distance of such solutions from single step ones.

of any regions that access common variables, but scheduling of waiting processes is deliberately totally uncontrolled and unfair.

The reason the up/down primitive must be similar to the conditional critical region is that it too provides for mutual exclusive access to variables (a, b, s in this case) without scheduling specified. The program can be rewritten to emphasize the relations among the independent blocks of code due to access of common variables.

READERS

```
(1)   shared s,b,a
            if s = 0 ∧ b = 0 then  a := a + 1 ;
(2)   read
(3)   shared a
            a := a - 1 ;
```

WRITERS

```
(4)   shared s
            s := s + 1 ;
(5)   shared a, b
            if  a = 0 ∧ b = 0 then  b := b + 1 ;
(6)   write
(7)   shared b
            b := b - 1 ;
(8)   shared s
            s := s - 1
```

Given this representation we can see that there must be mutual exclusion among (1), (4) and (8) due to access to s. Now the reader can see that if exclusion is enforced unfairly, the program has exactly the same race as noted in the Courtois solution. That is, if the last write in the data base is checking out and is doing (8), a write can precede a read at the synchronization for (1) and (4) and yet due to unfair scheduling the reader could proceed first with (1). Its test will succeed (a and b are 0) and a read will have won an advantage over a write.

Thus the "stopping" property of the up/down solution, given no scheduling algorithm, is a property of execution of the step, not of attempt at execution. Within a system of readers and writers, accessing a data base through these procedures, the priority arrangements can be overridden due to both readers and writers implicitly passing through a common unfair synchronization.[1]

Thus while the up/down seems to have more "expressive" power in that it allows expression of the indivisible operations necessary for readers/writers, it is not clear that it has any associated increase in power as an implementation mechanism since it does not allow for statement of those details necessary to ensure that this program can be a useful part of a system.

8.5 A Power Difference

If instead the up/down primitive could be defined with a fair scheduling algorithm, two other points, should be considered. First, specifying a fair algorithm may prove sufficiently difficult as to render it an impractical device regardless of its apparent power. Second, it is only reasonable to compare the fair up/down to a fair semaphore. Once released from the constraints under which Courtois et al operated, there may be better semaphore programs. Thus while we do not claim to disprove the Lipton results we may introduce properties of synchronization not considered in their model that may bear weight on grounds of practicality against the literal interpretation of the result favoring up/down. What is more, given that a fair semaphore is still not adequate, we can show that the full power of the extended semaphore or up/down is not required for the readers/writers problem. In fact, only that part least likely to produce serious

---

[1] In fact, this may be seen as a sacrifice in clarity of a program due to using a more structured primitive. Since the synchronization mechanism is implicit, it is harder to see this flaw than it would be in a semaphore program in which all entrance code involved passing through a common semaphore (i.e. if the Courtois program had only one semaphore *mutex* in place of *mutex 3* and *mutex* 2.)

practical implementation difficulties is actually necessary. We will deal with the implementation by fair semaphores first.

Given fair semaphores one can have all requests and ends use the same semaphore for protection. Since all four operations can be made mutually exclusive both the *readcount* and *writtecount* variables can be accessed in each. Every request can have access to a record of the complete history and no ordering between reads and writes can be lost. The reader might like to try his hands at the fair semaphore solution . There is no reason why it should fail in the manner described by Lipton. I.e. writes can always stop later reads.[1]

There is a difference between semaphores and extended semaphores which will probably become apparent in any fair semaphore solution to this problem. The most obvious fair semaphore solution to this problem is to build a monitor from semaphores and to use the monitor solution for the rest. Hoare shows how to implement the monitor given semaphores. We would like to point out the following problem, however. If one expects now to be able to fully realize the original solution specifications, one would be depending on the ability to guarantee that order of waiting on internal queues corresponds to order on external ones (see Section 6.2).

There seems to be a difficulty in the implementation of monitors with semaphores which prevents this, and which in general makes it necessary to release shared semaphores before P'ing private ones. Hoare implements his monitor using a semaphore *mutex* for the "outside" of the monitor, and semaphores *conditton* for conditions. Thus to implement the wait operation one must do

```
V(mutex) ;
P(conditton) ;
```

---

[1] Of course, if E_r, E_{w_2} are on the queue, E_r will be treated depending on past behavior and not taking into account the fact that a write is waiting behind it on the queue. The only published solution which apparently gives "complete" priority to writes in the sense that a write only has to wait for preceding writes is Lamport's [1974].

Clearly the opposite order would not work since if *condition* is locked, while the process waits *mutex* could not be released and deadlock results. However, with this ordering there can be no guarantee that another process does not get ahead on the condition.[1]

Thus the semaphore does have a weakness in the separation of these two operations. A single operation for enqueuing on one semaphore and releasing another would be a more powerful primitive than the semaphore for the purpose of achieving the fairness property.

Since this increase in power over the semaphore does produce a correct solution it appears to be sufficient to solve the readers/writers problem. What is more, the added power of general up/down may come only at unduly high cost. The reason for this is in the difficulty in defining up/down to ensure fair synchronization. Depending on the particular use of up/down this can become arbitrarily complex.

In an extended semaphore which allows indivisible P and V on arbitrary groups of semaphores, scheduling is difficult enough. Then a P operation part in say [P s_1 ... s_k V s_1 ...s_m], causes waiting if any of s_1 ... s_k are 0. Other V operations may change the values of arbitrary s_i's. If for instance s_1 and s_3 are 1, s_2 and s_4 are 0 and P(s_1, s_2), P(s_2, s_3), and P(s_2, s_4) occur in that order, who should be enabled by the sequence of operations V(s_4); V(s_2)? What is a fair algorithm for a particular set of semaphores and what is a general fair algorithm over all possible extended semaphores? It would appear there should be one queue for each set of values waited for and an algorithm for rotating among those queues when a single value is changed.

In this case all P's simply require non-zero values. With up/down (or conditional critical region) arbitrary predicates can be required. We refer the reader to Hansen [1972a, 1972b] for discussion of possible implementation of scheduling. It may be that reliable programming of synchronization by relinquishing all power to program scheduling may in fact generally cost too

---

[1] If Hoare meant this to be his definition then the ordering correspondence was not at issue. However, if his informal specifications (or our Chapter 6 interpretation of them) is to be the criteria, this is an error.

much in efficiency and in clarity due to heavy reliance on implicit scheduling. Even if the programmer who is concerned about "good" as well as "reliable" programs does not decide to rely on semaphores alone, we suspect he is unlikely to go further in the structured direction than the monitor.[1] The monitor makes only the first level of synchronization implicit and has facilities for programmed scheduling. If the programmer writes in deadlock or some other undesirable feature at least his code is subject to debugging. With unscheduled primitives he has no recourse when the program proves ineffective.

8.6 Properties in Run-time Environment

Some properties of run-time environment are quite deliberately excluded form our language by choice of model. Most notable is the deliberate representation of processes by sequences of events leaving each process with no measure the passage of time finer than the event count.[2] This is important because while at a given level of detail the events in a behavior of a system should be identical independent of the physical machine used as activator, the time for any two machines may be grossly different.[3] Therefore if we state and prove a property of a system program, it holds over all possible speeds at which that program runs, including arbitrary relative speeds of its parts.

There are some timing properties in terms of numbers of events between events that

---

[1] In fact in [Hansen, 1972b] Hansen suggests similar means for introducing ability to schedule.

[2] Another deliberate choice is to associate events with receipt of message rather than sending. Intuitively, two messages could be sent in one order but received in another depending perhaps on the channel over which the messages travel. For this reason the two occurrences, namely sending and receiving, cannot both be represented in one event without rendering the model inapplicable to real systems. Receiving rather than sending is chosen in this model to avoid dealing with properties internal to the sender.

[3] Time for a process on the same machine could be different from one use to the next if the process is part of a timesharing system.

can be stated in our language and which may be of interest in specifying system. For instance, there is a difference between the Courtois and monitor solutions to the readers/writers problem that is not measured by the current specifications. However, if more is acknowledged about implementation, it can be expressed. The difference relates to the time involved in enabling a waiting read. In the monitor solution once the data base is opened to reads all reads on the condition queue are let in. This was in keeping with our specifications that said that all waiting reads were enabled by the opening of the data base. If that could be accomplished instantaneously, then that would be a fine specification and the monitor program an adequate implementation. However, in that program, the monitor is kept closed while the read loops through a sequence of signals and additions, letting one at a time off the queue. If a large number of reads were waiting it is quite likely that writes are accumulating outside the monitor during the releasing. By contrast, Courtois et al who have all reads but one pile up "outside" can account for this actual time involved in releasing by checking with writes (through *r*) for each read. If duration of instruction execution is taken into account, comparison of these two solutions could show the Courtois one to be better in the sense that when reads and writes are competing, on the average it lets writes in sooner. *Startread* in the monitor defined in Chapter 6 cannot be modified to check for newly arrived waits without giving up the monitor. Then if there are no writes there would be no way to get back in to release waiting reads.

The fact that enabling is not instantaneous in implementation could be represented by distinct events for requests and being acknowledged as waiting. The solution specification for this improved solution could allow for this timing problem by considering the separate events in which reads request and in which reads are enqueued.[1] The specifications would be about events E_r, $E_{r_{enq}}$, $E_{r_{in}}$, $E_{r_{out}}$, $E_{r_{end}}$. E_r causes $E_{r_{enq}}$ only when no reads or writes are still waiting. $E_{r_{enq}}$

---

[1] The Courtois program specifications have separate events, namely, the *mutex 3* or *mutex 2* events and the P'ing of *r*.

causes $E_{r_{in}}$ only when no writes are ahead. Therefore the readers check twice at E_r and $E_{r_{enq}}$ on whether they can proceed. If a write comes along before the second check, it loses. This can be done in the Courtois style with the first check being for preceding reads and the second for writes. This leaves some reads and writes unordered. Since proceeding from E_r to $E_{r_{enq}}$ is independent of the number of writes, E_r need not be ordered with respect to writes.

Nested monitors can be used to pre-order reads, letting only one at a time be order with respect to writes and conditions. This solution is not significantly different from the Courtois solution if fair semaphores were substituted and may yield a better performing data base than one which uses the fair semaphore (or monitor) to totally order all requests.[1]

Given more precise notions of the measure of "as soon as possible" we can specify behavioral properties that must hold to satisfy specifications. Speed requirements phrased in terms of events are within the range of expressible properties of our language. Whether or not these properties will be observable when measured by arbitrary external criteria is dependent on the running environment of the programs. In any of these solutions, if writers are running in machines which are significantly slower than the readers machine then even priority may not be apparent to the user. [See related discussion at end of Chapter 9].

---

[1] Hansen's patch to his conditional critical regions solution [1972a] is similar.

9 Busywaiting Synchronization

This chapter does not continue the develop of the specification language, but instead steps back to the uses of cells. It contains two busywaiting solutions to the problem of mutual exclusion of critical sections. They are analyzed carefully for correspondence of parts to conditions in the mutual exclusion specifications given in Chapter 4. Also, the properties of deadlock-free and absence of lockout are stated formally and proofs that these properties hold are outlined. This serves the purposes of both utilizing the definition of cells and giving further instances of mapping of specifications onto details of implementation. Reasoning about partial orders is the basis for most arguments.

Few programmers of systems with parallelism will ever have to write busywaiting code since most are likely to be writing programs in languages containing synchronization primitives -- whether semaphores or some structured primitive. However, it may still be that for serious understanding of synchronization one should study a few busywaiting implementations, if only to appreciate the value of a synchronization primitive. Also, it provides a good basis from which to abstract the causality of synchronization actors, as represented by the direct addition of => to orderings. These solutions show how that order can be added.

These programs have both been proven correct already [Dijkstra, 1965;Knuth, 1966]. However, in neither case was a formal statement of the property which had to hold for "correctness" made. Thus these proofs are interesting in that there is a precise statement and it is compatible with our language for understanding programs and for reasoning about them.

9.1 The Structure of the Dijkstra Solution

Dijkstra [1965] published a solution which without synchronization primitives enforced mutual exclusion of critical sections in "A Problem in Concurrent Programming Control." To represent the fact that in general processes may need to enter critical sections arbitrarily often, the structure he chose was for each program to be in a loop performing operations some of which are designated as critical, others not. With the additional constraint that only one process at a time may be executing its critical section, there is need for communication among processes before and after the critical section. Thus the solution takes the form:

```
Lt:  entrance code ;
     critical section ;
     exit code ;
     remaining code ;
     goto Lt ;
```

It is properties of the cells which are updated and queried in the entrance and exit code sections that are relied on to guarantee the enforcement of the desired ordering. Thus the critical section ordering will be due to a property like:

if entrance events in α_1 => entrance events in α_2

then exit events in α_1 => critical section events of α_2.

We will abbreviate "critical section events of α" by "CS in α."

The particular solution Dijkstra wrote is to be analyzed here in the two process case and is shown in Figure 9.1. In it each process has private variables, b_t and c_t and both share a common variable k. The variable k points to one process (by containing the number t of the process)[1] and has some relation to which process is in its critical section. Each process i tries to set

---

[1] Note that these cell solutions require knowledge of number of processes and their "names" (i.e., the numbers) whereas for synchronization primitives the ability to name processes (an arbitrary number of them in fact) was assumed as primitive and internal to the actor whose external behavior we were defining.

k to t before getting into its critical section. The b_t can be interpreted as signalling whether a process is in its remaining code or not. (The alternative includes being in any of entrance, critical section or exit). The variable c_t indicates whether or not process i "thinks" it has the go ahead. Entrance code consists of setting b_t to *false* and then trying to establish $k = t$. While doing this c_t must be true and the process can potentially stay in a loop (between L11 and L13). Once the process i finds $k = t$ it sets c_t to false (to say it thinks it is set to go) and then checks that no one else thinks they can go (L14). Failure (finding a conflict) results in failing back to the beginning (L11). The exit code consists of setting both variables to *true*. Since the program in Figure 9.1 is just for two processes all the polling of other processes is simplified.[1]

```
L10:   b1 := false;
L11:   if k ≠ 1 then
L12:         begin c1 := true;
L13:         if b2 then k := 1;
             goto L11
             end
       else
L14:   begin c1 := false;
             if not c2 then goto L11;
             end;
       critical section;
c1 := true; b1 := true;
       remainder of cycle;
goto L10
```

```
L20:   b2 := false;
L21:   if k ≠ 2 then
L22:         begin c2 := true;
L23:         if b1 then k := 2;
             goto L21
             end
       else
L24:   begin c2 := false;
             if not c1 then goto L21;
             end;
       critical section;
c2 := true; b2 := true;
       remainder of cycle;
goto L20
```

Figure 9.1: Mutual Exclusion of Critical Sections.

This solution has the property that one process can loop forever in its entrance section while the other does an arbitrary number of critical sections. The ways in which the entrance and exit codes enforce orderings can be analyzed in terms of events.

In order to enter a critical section an activator must have the events:

---

[1] For process 1, instead of checking all processes ≠ 1 it can simply check process 2, etc.

<c_i (apply: [← 'false] (then-to: x_1)) α *ec*> -->
    <x_1 (apply: c_i) α *ec'*> -->
        <c_j (apply: 'contents (then-to: x_2)) α *ec''*> -->
            <x_2 (apply: 'true) α *ec'''*> where i ≠ j.

I.e. it must have

E_1 = <c_i (apply: [← 'false] (then-to: x_1)) α *ec*>
E_2 = <c_j (apply: 'contents (then-to: x_2)) α *ec''*>
and no event
E_3 = <c_i (apply: [← 'true] (then-to: ?)) α ?>
such that E_1 --> E_3 --> E_2.

Thus for every process which enters a critical section, there is such a sequence of events in the behavior before each critical section. This is the property of the solution that can be said to hold for each activator. (Analogous to activator properties of synchronization actors of earlier chapters.) Each behavior segment corresponding to entrance events which succeed in entering the critical section ends in this sequence.

Due to the properties of cells it can also be said that if two processes contain such sequences they can be ordered with respect to each other. Thus if E_1, E_2, E_3 and E_4 are in the behavior where

E_1 = <c_i (apply: [← 'false] (then-to: ?)) α_i *ec*$_1$>, E_2 = <c_j (apply: 'true) α_i *ec*$_1$'''> and j ≠ i

E_3 = <c_k (apply: [← 'false] (then-to: ?)) α_k *ec*$_2$>, E_4 = <c_l (apply: 'true) α_k *ec*$_2$'''> and l ≠ k

then either E_1 => E_2 => E_3 => E_4
    or E_3 => E_4 => E_1 => E_2.

What is more, if E_1 => E_2 => E_3 => E_4 then E_2 => E_5 => E_3 => E_4
where E_5 = <c_j (apply: [← 'true] (then-to: ?)) α_i ?>.

This says that this program causes correspondences in orderings => over *events of different activators* (the ordering of E_1, E_2 is related to the ordering of E_3, E_4), and that it uses a release mechanism for enforcing that ordering (E_5 in the case of the first ordering). It does not say that every process that wants to get into its critical section does get in. It is not the case that having

E_1 = <b_1 (apply: [← 'false] (then-to: ?)) α_1 *ec*>

in a behavior necessarily implies that E_1 --> CS in α_1 in that behavior. This solution can lock out a process indefinitely.

The ordering property just described is one of the program as an actor, not of the cells. It is not the same as the overall intention either. That statement was that all critical sections should be executed in some order. This says something about how that ordering is achieved.

9.2 A Proof of Properties of the Dijkstra Solution

Dijkstra stated in English four conditions for correctness of this solution. He then proceeded to prove its correctness. However, this proof must in fact be considered at most an informal argument of correctness since the statements of the things to be proved were so vague. It is not clear that all of the requirements can be formalized.

In the actor model there is a formal interpretation of the statements in this program, and of the meaning of a cell as a communication device.[1] Also, some of Dijkstra's intentions can be formalized, and it can be proved that these intentions are realized by the program. This solution can be proved correct quite formally, but once again only details pertaining to cells are

---

[1] One might note that models for parallel processes such as Petri nets can only model semaphore communication, but cannot model directly this use of cells for communication. Petri nets model control structure and would reveal only the sequential and loop control structures of these programs, not the implicit interprocess control.

given. Others can be filled in by using the full formal translation of the Algol language to actor code. The interesting reasoning depends on the cell as defined above. The rest is based on definitions of sequential control and if-then-else.

Dijkstra's additional conditions are:

"(a) The solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority.

(b) Nothing may be assumed about relative speeds of the N computers; we may not even assume their speeds to be constant in time.

(c) If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

(d) If more than one computer is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity. In other words, constructions in which 'After you' - 'After you' - blocking is still possible, although improbable, are not to be regarded as valid solutions."

As far as (a) is concerned, there is no explicit fixed priority. However, there appears to be some priority implicit in the numbering of processes even in Dijkstra's solution. Since we cannot interpret this restriction clearly enough to see why Dijkstra felt he was observing it, we will ignore it. The assumption (b) is taken care of by the use of the actor model in which one can talk only about events and not about the "time" it takes to carry out an action or the "time" that elapses between events. For the two process case, condition (c) simply requires that if either process runs alone in the state in which the other process' variables are all set to "true," then that process can get into its critical section as often as it likes. This is because (assuming c_i, b_i are not accessed in the remainder code) when a process stops in its remaining code section its variables are left in this state. A process i stopping in its remainder section will be represented in the actor

model by the possibility of there being no further events in α_i after an event starting the remaining code section. It can be interpreted as meaning that either program running alone can have infinite behavior, which is easy to confirm.

Condition (d) is quite interesting to prove. It can be stated, formally, as:

> For every set of critical section events, CS_i, there is another set of critical section events CS_j such that $CS_i \Rightarrow CS_j$.

This simply means that in any behavior in which at least one of these programs is running, there will be an infinite number of critical section events. It can't be the case that no process can get into its critical section. This of course is based on the fact that the remainder of the cycle cannot have infinite behavior at any given level of detail. For our purposes here, we can represent that remainder by a single event with a target actor that is not known to always send a message to its continuation. If it does, that continuation is the code corresponding to the code following the label Lil.[1]

> **Theorem (Mutual Exclusion)**: The Dijkstra solution has the property that
>
> $\forall\, CS_1, CS_2\ (CS_2 \neq CS_1) \supset (CS_1 \Rightarrow CS_2) \vee (CS_1 \Rightarrow CS_2)$.

Proof: The argument for mutual exclusion is straightforward. We start with the events known by the activator ordering to precede and succeed the critical sections and then build $\Rightarrow$ from observing the requirements of the axioms for cells. Thus the behavior fragment in Figure 9.2 is the basis for our reasoning:

---

[1] This assumption might appear to contradict the assumption that a process could "stop" in its remainder code unless we assume that "stop" refers to an activator property rather than an actor one as in the experiments for parallelism.

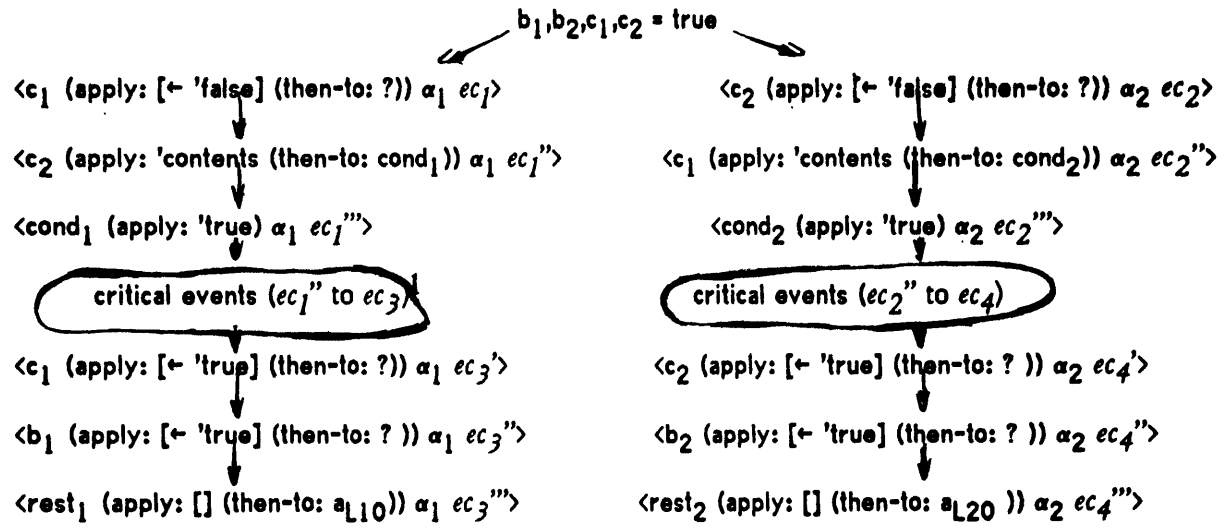


Figure 9.2: Mutual Exclusion Behavior.

It is quite easy now to see that the only possible orders derivable from $\Rightarrow_{c_1}$ and $\Rightarrow_{c_2}$ are the ones in Figure 9.3. In either of these cases the two sets of critical section events are ordered by $\Rightarrow$. □

---

[1] I.e. the first event in this sequence has event count ec_1 and the last ec_3.

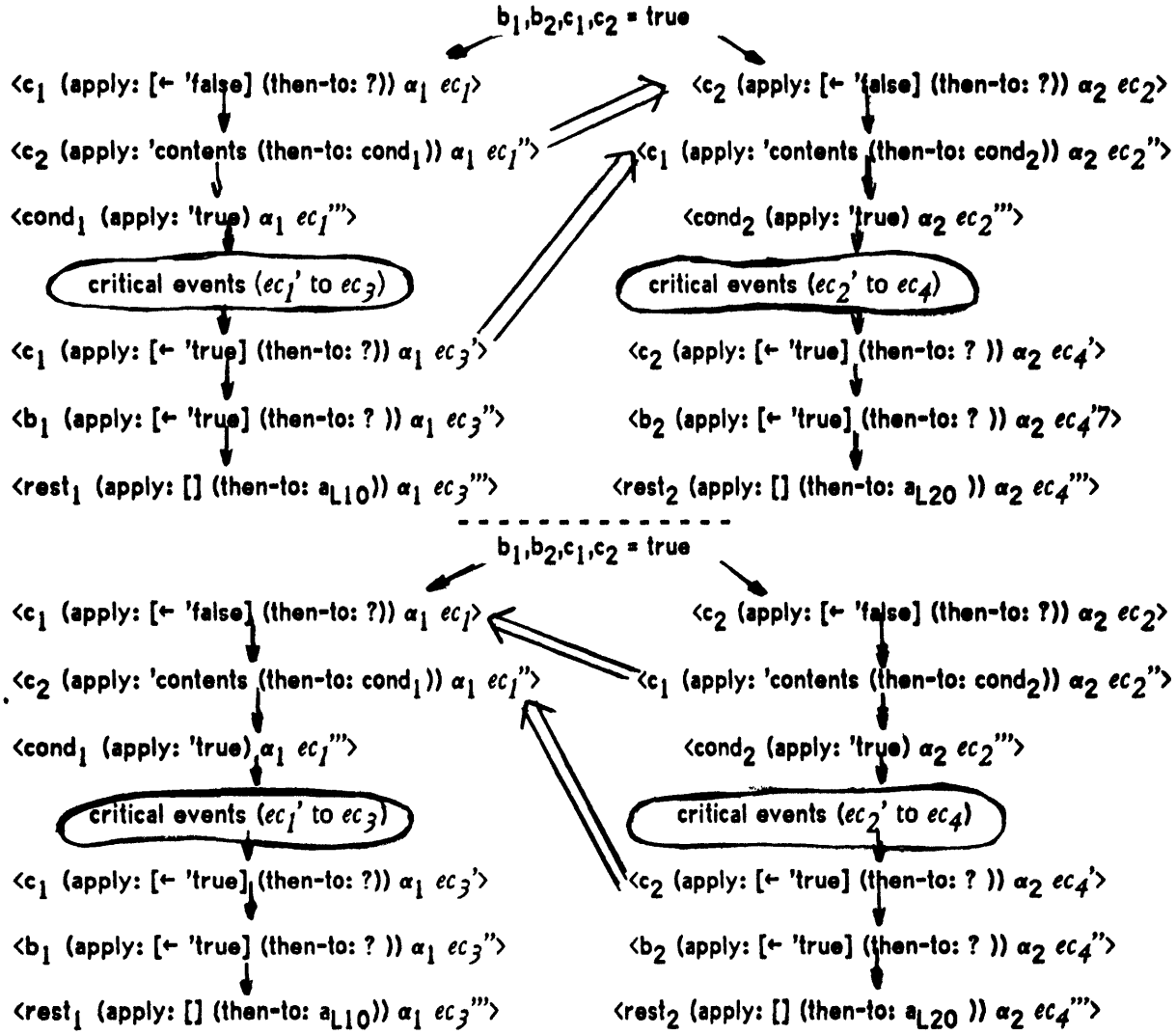
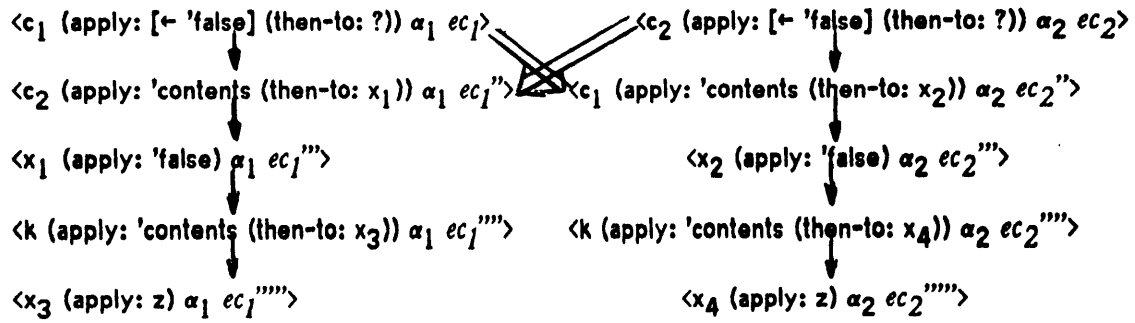


Figure 9.3: Orderings for Mutual Exclusion.

The way to prove that there is no deadlock is to prove the impossibility of both processes looping forever. We would like to prove that as long as there are events E_1 = <b_1 (apply: [← 'false] (then-to: ?)) α_1 ec_1> and/or E_2 = <b_2 (apply: [← 'false] (then-to: c_2)) α_2 ec_2> (i.e. the first events in the entrance code) in a behavior, then there is some CS in the behavior where E_1 --> CS in α_1 or E_2 --> CS in α_2.[1] Which process gets in seems to be independent of both the order in which E_1 and E_2 occur and the value of k.[2] However, once both processes have gotten as far as statement Li4 and then still had to loop the choice is determined by the value of k. The following Lemma is a statement of that fact.

> **Lemma**: In a behavior in which the following events and orderings prevail, whatever value z has is the value of the next activator to get into its critical section.
>
> 
>
>
> I.e., then there is CS in α_z in a following segment of the behavior.

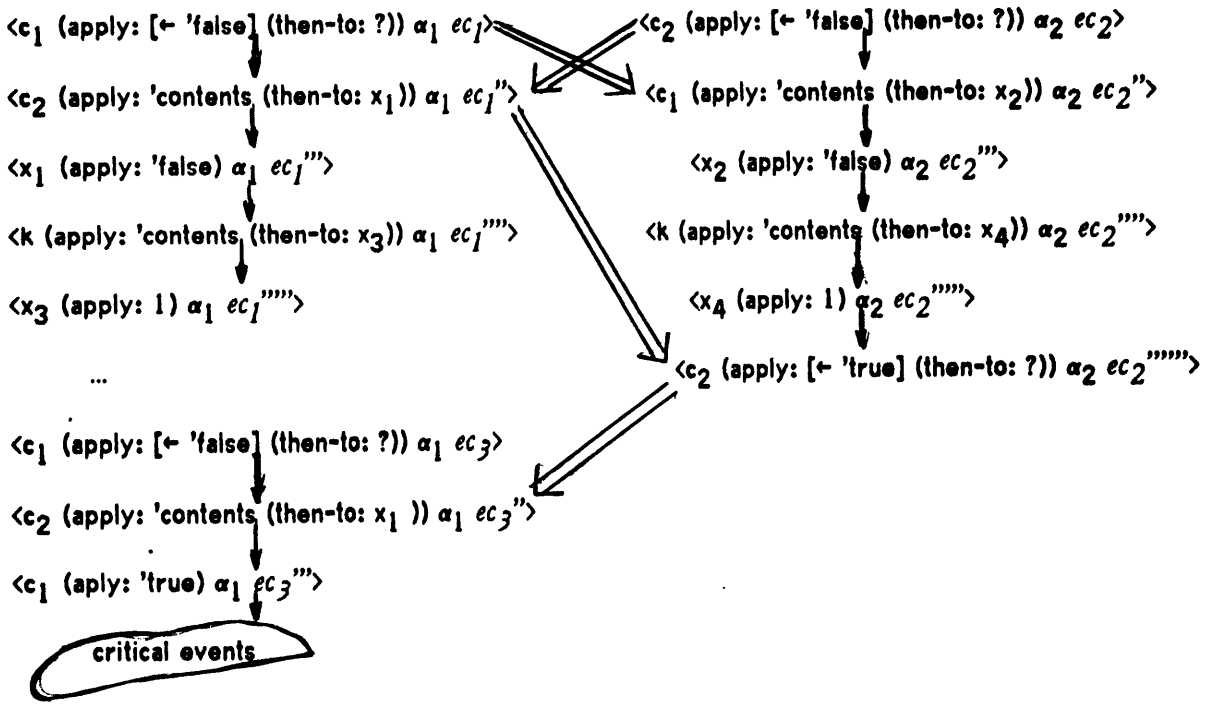
Proof: Process z will now be looping between the test of k (Li1) and the test of c_f, $f \neq z$ (after Li4), waiting for c_j = true. Meanwhile process $j \neq z$ now has to try to change k to f. In doing so it

---

[1] I.e. as long as there isn't a failure in remainder code, so that there is a next attempt at entrance, there will be another critical section.

[2] This does not necessarily mean that given particular timing constraints the solution is completely unbiased. However, since we know nothing about timing between events, all we can refer to is the fact that for any combination of ordering of E_1, E_2 and value of k there are timings such that either process can get in first.

changes c_1 to *true* so that eventually process z can get to its critical section. Thus if $z = 1$ the behavior is:



In the unspecified (...) part of α_1 there may be arbitrary repetitions of the first 5 events (or none) in all of which the contents query to c_2 precedes the update in α_2. □

Now to prove that the program is deadlock-free it is necessary to show that if both processes are looping they cannot avoid the situation in the hypothesis of the Lemma.

Theorem: The program is deadlock-fee.

Proof: Assume that both processes are looping indefinitely. The value of k cannot keep changing indefinitely while α_1 and α_2 loop since once both are in their entrance sections, both b_i are false

and both tests that have updates to k in their continuations fail. Thus at some point one process, say i, finds k = i and goes to Li4. If it passes the test it gets out of the loop. Since we are assuming that both loop it must be the case that process j got to Lj4, changed c_j to false and caused process i to fail. Similarly, if j is to continue looping it must fail. But then we have the condition for the Lemma and some process will get in. □

In this example the reasoning and arguments used were quite similar to those originally used by Dijkstra. The difference is that they now have meaning in a formal model and are being used to prove a result which is stated formally about that model.

9.3 The Knuth Solution

Knuth [1966] wrote a solution to the mutual exclusion of critical sections using only cells. His solution has the property that there is no lockout. Any process which starts the entrance protocol gets into its critical section. Knuth's solution for 2 processes is shown in Figure 9.4. In this solution each process has a single private variable, c_t. This variable can have one of three values: 0, interpreted as being in the remainder section; 1, interpreted as just being in the entrance code; and 2, interpreted as thinking $k = t$. After finishing a critical section a process i changes the value of k to $j \neq t$.

```
L10:  c1 := 1;
L11:  if k = 1 then goto L12;
      if c2 ≠ 0 then goto L11;
L12:  c1 := 2;
      if c2 = 2 then goto L10;
L13:  k := 1;
            critical section;
            k := 2;
L14:  c1 := 0;
L15:  remainder;
            goto L10;
      end
```

```
L20:  c2 := 1;
L21:  if k = 2 then goto L22;
      if c1 ≠ 0 then goto L21;
L22:  c2 := 2;
      if c1 = 2 then goto L20;
L23:  k := 2;
            critical section;
            k := 1;
L24:  c2 := 0;
L25:  remainder;
            goto L20;
      end
```

Figure 9.4: Knuth's Mutual Exclusion Solution

Besides enforcing mutual exclusion this solution has the property of no lockout:

> For all i, if E_1 = <c_i (apply: [← 1] (then-to: ?)) α_i *ec*> (the first event in an entrance) is in the behavior, then so is a series of events CS in $\alpha_i \ni E_1$ --> CS.

However it still cannot be said that there is a guarantee that certain obvious measures of ordering of attempts guarantees an order of entrance. For instance if E_1 and E_2 are the following events

> E_1 = <c_1 (apply: [← 1] (then-to: ?)) α_1 ec_1>
> E_2 = <c_2 (apply: [← 1] (then-to: ?)) α_2 ec_2>

it cannot be said that E_1 => E_2 (where => is built from $=>_{cell_i}$) means that process 1 gets into its critical section before process 2 does. The reader can see that this does not hold by thinking about how this ordering information could be made available to the system. It simply is not available in some cases. In others it is not used consistently. For instance, in the case where k = 2, one place where the value of c_2 is checked is in "if $c_2 \neq 0$ then goto L11;" At this point $c_2 = 0$ would say that indeed E_1 => E_2. However, by the next time c_2 is tested process 2 could have changed the value of c_2 to 2 causing process 1 to loop. I. e., process 2 could get to its critical section first since k pointed to it, even though it can be known to have tried later. Similarly, if k pointed to 1, process one would have the advantage regardless of ordering information which is explicitly checked for.

Therefore, this first event in a sequence is not the sole measure of who is let in first. The ordering depends both on the orderings $\Rightarrow_{c_i}$ and on the ordering $\Rightarrow_k$. This ordering $\Rightarrow_k$ reflects who was in its critical section last.[1]

9.4 A Proof of Properties of the Knuth Solution

One fact of use in proving absence of lockout is stated in the following Lemma (stated here for Process 1).

> **Lemma**: If process 1 can be established as trying before a critical section of process 2 ends, then once k contains 1, process 1 can get in. What is more, process 1 will then get into its critical section before process 2 does another critical section.
>
> In terms of the behavior, if the following events are in the behavior
>
> $\langle c_1$ (apply: [← 1] (then-to: ?)) α_1 *ec*$\rangle$ => $\langle$k (apply: [← 1] (then-to: ?)) α_2 $ec_1\rangle$
>
> then it must also be the case that
>
> $\langle c_1$ (apply: [← 1] (then-to: ?)) α_1 *ec*$\rangle$ => CS_1 in activator α_1
>
> and that there is no CS_2 in activator α_2 such that
>
> $\langle$k (apply: [← 1] (then-to: ?)) α_2 $ec_1\rangle$ => CS_2 => CS_1.
>
> We will carry out all arguments from the point of view of process 1 and omit the symmetric argument for process 2.

Proof: The behavior in Figure 9.5 has the events and the ordering postulated and then a possible sequence of events for process 2 to have. We will show that that is the only sequence of events it could have until after process 1 has indeed done another critical section. If this is the sequence of events, then it is possible for process 1 to get into its critical section without another critical section of process 2.

---

[1] In the Dijkstra solution ordering also depended on k but k's value was not as strongly correlated with the identity of the last activator to be in its critical section.

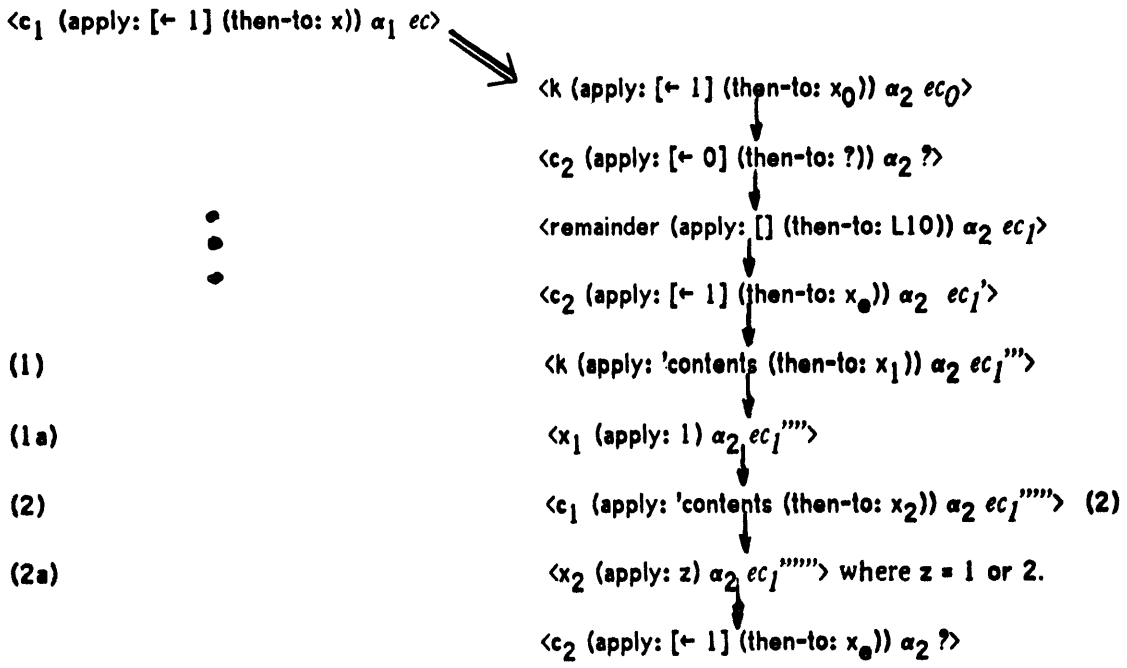
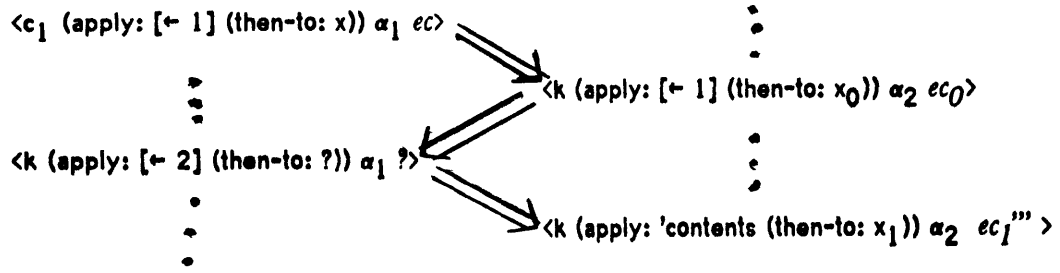


Figure 9.5: Hypothesized Behavior

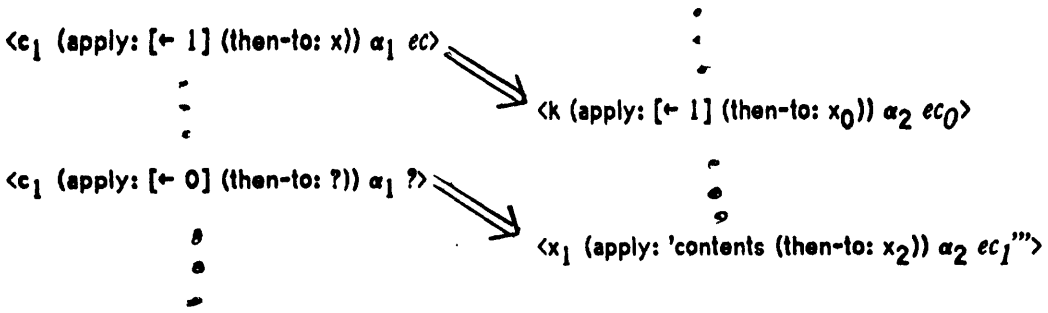
In (1) and (1a) could k be found equal to 2? Not unless there is an update to k such that



but that is only possible if process 1 did its critical section before the update. [Assumes no change

to k possible in remainder.]

Could c_1 be 0 at (2) and (2a)? Not unless the behavior is



and again that is only possible if process 1 did its critical section.

So process 2 will have this loop of events until after process 1 does a critical section. Clearly now process 1 will do a critical section since if it finds k = 1, all it has to do is find $c_2 \neq 2$ before entering. Once process 2 does <c_2 (apply: [← 0] (then-to: ?)) α_2 ?> it cannot do an update until process 1 does a critical section. Thus process 1 will eventually find $c_2 \neq 2$ and the Lemma is proved. □

This property can now be combined with two more facts to prove the impossibility of lockout. The facts are

(1) the activator property of each process which says that any (CS in α_i) --> <k (apply: j) α_i ec'> where $j \neq i$.

(2) There is no deadlock in this solution.

Theorem: There is no lockout in Knuth's solution.

Proof: By (2) some process always gets into its critical section. Thus if process i starts the entrance protocol either it or the other process does get into its critical section. If it gets in it hasn't been locked out. If the other process gets in, then once process i has started the entrance, by (1) the situation in the hypothesis of the Lemma must eventually occur. Then by the Lemma, the first process is to be let in. Thus no process can be locked out of its critical section indefinitely.[1] □

The proof of the deadlock free property is straightforward and will not be done here.

9.5 High Level Properties of Busywaiting Solutions

The definition of fairness of synchronization that we have been using is equivalent to absence of lockout, i.e. every process that tries to get into its critical section will get in. It does not necessarily correspond to a first in first out algorithm. In fact it is not clear in Knuth's program how to measure who is first. One point in understanding Knuth's solution is that only ordering that is computed is relied on. Therefore only once a process is verified as having started its entrance code before another finished its critical section does the first take its place in line. In fact, in the n-process case of the Knuth solution, the second process can get through its critical section several times before the first gets its place established. [See Knuth, 1966]. As long as our specification of guaranteed absence of lockout is met, the solution satisfies the criterion of fairness. We would like to point the reader to other solutions such as Eisenberg and McGuire [1972] or Lamport [1974] which do much better in terms of the number of critical sections of other processes that can happen between start of entrance code of one and its entrance. These differences are not reflected in our specifications and we think for good reason. As an

---

[1] In fact, it can be shown that any process starting its entrance has to wait at most for one further execution of the critical section of the other process.

experiment from outside the solution, one cannot tell anything about how "close to FIFO" a solution is.[1] While in Knuth's solution one can have to do arbitrarily many busywaiting steps while the bounded number of other critical sections occurs, from the outside this cannot be distinguished form very slow execution of the bounded number of steps required to secure a waiting place in a better solution such as Lamport's. Since the place is not secured until that point, one could see all of the same behaviors from outside. Thus our specifications say the most they can without making assumptions bout machines on which programs are running.

This is not to say that considerations of use of resources may not make these interesting measures. I.e., reducing the maximum number of critical sections of other processes could reduce the amount of busywaiting a process will have to do. There should be means for formal statement of the difference between say Knuth and Lamport solutions. However, since the properties are not externally observable it may be necessary to go into more detail to specify them.[2] If a property can only be stated relative to internal properties then it will be shown to be a property which is significantly different from properties such as fairness which we have specified.

---

[1] However, as per discussion in Chapter 8, we could express these properties in terms of events.

[2] As in readers/writers where extra events of enqueuing reads must be used if we are to refer to fact that internally some effort is involved in enabling a waiting read.

10 Conclusions and Future Research

A specification language for time ordering properties of communicating parallel processes has been presented. It was used to write precise specifications for several problems that have not previously been adequately formalized. The time ordering concepts directly expressible in the language are apparently adequate for stating high level properties in a form which is close to intended meaning. A problem specification describes behaviors which we hope to realize through a program or a system of parallel programs. The solution specification describes more detailed behaviors (i.e. with additional events) which can be proved to realize the problem specifications. This means the solution specification is complete. Consistency of solution specifications is demonstrated by showing that the axioms for the primitive (at the current level of detail) actors of the actor system are themselves realizable.

In this final chapter a concluding review and evaluation is made of the approach to specification taken in this research. This is followed by consideration of some properties of synchronization which have been exposed in the process of developing this language and speculation on applications of this work in the understanding of both programs and computer systems.

10.1 Time Ordering Specifications

Two criteria for a language in which to write specifications for synchronization have been applied throughout this work. They are that there be both means for expressing high level properties and means for specifications about causality which emphasize physical realities of

implementation. Using the time ordering => one can make arbitrary statements about time orderings. Among these arbitrary statements are some which clearly correspond to typical synchronization terms such as mutual exclusion. Using a combination of activator orderings, actor orderings, and causal properties of actors, we can also express the causes of orderings from which derived time orderings can be built. Specifications of time ordering properties of an existing actor system are about this derived time ordering. Specifications to be met by an unbuilt system indicate desired properties of the derived ordering of the implementation. Solution specifications in the form of causal axioms define the generators of the derived order =>. Proof of completeness is accomplished by showing that desired time ordering properties hold for the transitive closure of the union of the sets of generators associated with each actor in the system.

Specifications for an actor can be about the actor as it appears from the outside or about its internal operations. From the outside, treating the actor as a black box, one can first specify relations among events with the actor as target, and events outside the actor which are caused by the actor (such as the event in which a response is sent to a continuation). The cell and data bases were specified in this way.

Alternatively, one can give specifications about events internal to a system. These can be given in varying amounts of detail. They may or may not have implications for the external appearance of the actor. While there is no general way to abstract from a specification of internal properties "all" externally manifested behavior, one can check whether a particular desired external property does hold. Thus for the readers/writers actor, rw, derived in Chapter 5, the external property of "reliable data base behavior" (though not necessarily fair) is proved to hold since the readers/writers property and the data base definition hold. That is, no operations that can cause inconsistent results to appear are possible.

In addition, the actor does have relative write priority and fairness to writers. These last two are internal properties. Priority (as in the readers/writers problem) is considered an internal

property since it cannot be observed (or specified) from outside. A difference such as that between fair and unfair implementations, may be observable from the outside, but it is not clear how one could distinguish two different unfair algorithms, such as the monitor implementation of writer priority and the Courtois one. Even though the priority algorithms are different when compared in detail, their differences are not externally visible since there is no difference in their behaviors excluding actor ordering information. All of the readers/writers solutions have the property that the possible behaviors of systems involving them are identical (up to fairness). I.e., even if writers are being given priority, if E_r and E_w are in parallel then if E_r finds results of a particular E_w it is not clear if that E_w happened before E_r because of priority of because of timing accidents. Similarly, looking at any two fair solutions, say the one Hoare implemented and the queued one defined in Chapter 5, no difference can be seen from outside. In both cases all requests are served.

Our theory of parallelism is deliberately meant to include among its models the most general form of parallelism, namely, distributed multi-processing systems. There may be meaningful total orders in a case in which all processes are situated locally and perhaps even sharing processors. However, in general this is not the case. This is the reason for our concentration on properties which can be proved independent of machine configuration. Less general properties of particular computer systems may be of interest as well, and perhaps should be expressible. Statements of such properties should be considered as relative to the assumptions about configurations, just as specifications for priority are given relative to an actor ordering Only in systems in which this ordering is available can the priority specifications be met.

Very detailed timing considerations, such as those which might be related to a specifications "as soon as possible" may be considered optimization properties. In areas other than synchronization, there is a distinction between "correctness" and such performance questions as speed, use of space (perhaps even termination, although that is often considered part of

correctness). To date there are no very good means for specifying properties desired of optimized code.[1] Yet optimization criteria are intuitively separable from correctness. In sequential code, assertions against which correctness is measured seem always to measure properties of functions. One reason why side-effect formalisms are not as well understood as side-effect-free ones (besides their difficulty) is that for functional programs, correctness independent of implementation or optimization can generally be expressed in a side-effect-free form. For sequential programs it may be reasonable for side-effects and the problems of dealing with correctness given possible interference among side-effect operations to be put off as an issue for understanding implementations of programs rather than their meanings. As long as their meanings are embodied in relations between input values and the values produced, this distinction works and prevents confusion of meaning and implementation. Once side-effects are introduced and can be considered part of the meaning of a program, the separation of "important" side-effects from ones which are simply "useful" for implementation becomes an issue.

In synchronization, no properties have yet been singled out as correctness properties. All properties are intimately related to time orderings and various intuitive notions of time and there is no distinction among correctness, partial correctness, implementation dependence and optimizing considerations. Thus one of the considerations in beginning to formalize properties of systems with synchronization is the delineation of types of properties. The properties dealt with in this paper have been entirely of the sort that can be assumed to hold independent of physical computer configuration. They are about time orderings related to causality not about time as a measure of speed or efficiency. Care is taken to use only time ordering properties that are meaningful even if extremes of distances and speeds are introduced.

As additional information becomes available, more properties relative to that

---

[1] This is a serious deficiency which must be overcome if research in optimization is to develop from the study of program transformation to program transformation towards a goal.

information can be stated. For instance, if it became possible to refer to uses of resources such as processor time, then a side-effect of the number of events at a lower level of detail, namely, the amount of processing done, becomes visible. One might then want to be able to specify further requirements about solutions to synchronization problems.[1]

10.2 Why Specifications

We have emphasized ability to express properties of communication among parallel processes. As stated in the introduction, we have not produced the kinds of evidence usually deemed important in justifying an approach to semantics. Reasoning about properties expressed in this language is not trivially mechanizable, nor can we mechanically produce statement of auxiliary properties that are required in proofs. While we have written specifications for various stages of implementation of problems, we have not attempted to find automatic procedures for formulating statements of one level of abstraction from statements which are descriptions at a different level of abstractions.

What we have done is to develop a formalism which, due to its basis in event-oriented semantics and the mathematics of partial orders, can be used to express precisely certain properties of communicating parallel processes. The formalism is not so far removed from usual informal characterization of systems as to require long informal justification of high level specifications. That is, the representation of properties such as mutual exclusion and fairness are not difficult to accept. This is important since there can be no formal justification at the highest level of abstraction that specifications do capture the specifier's intentions. As statements of properties become harder to understand proofs that those properties hold become less convincing

---

[1] This increase in describable properties with increase in available information can be related to similar observations in describing protection relative to externally observable information [Jones and Lipton, 1975].

as "proofs of correctness." Statements such as Robinson and Holt's specifications in terms of the functions min and max [shown in Chapter 6] or the kinds of statements Cadious and Levy [1973] make about functions and oracles, require more justification that they do capture the intended meaning.

For analyzing programs one may not be able to avoid some amount of detail and reliance on properties far removed from the high level goal. The subproblems of any problem can require implementation specifications that are not as obviously related to intended high level properties of the overall program. For this reason it is important to be able to show that high level properties are realized by more detailed specifications.. For properties of communicating parallel processes this takes the form of deriving the time ordering of a system from the causal axioms of its parts. Since this derived ordering is compatible with the time ordering of high level specifications, comparison can be made. While abstraction from (or implementation of) specifications is not automatic, it is possible to prove relations hold between specifications and particular abstraction or implementations. Thus programming can be done by stepwise refinement from high level specifications, with comparisons of more and more detailed specifications making the conceptual transitions manageable. Also, even though examination of all possible interleavings of events may be necessary for analyzing a system, once it is analyzed, if there is a property consistent for all behaviors, we can express it at a higher level. Thus we are not limited to the detailed expression for description.

Detailed specifications are usually axioms for certain actors which will be in an implementation (provided they themselves are implementable). They can be proved to realize the high level specifications. The next considerations is whether these actors themselves are implementable. We have suggested several criteria for realizable such as reasonable assumptions about ordering, etc. However, all specifications depend on the implementability of some primitives. Unless the primitives are physically impossible, the realizability of the specifications

must be analyzed relative to the specifications of the primitives. Thus in **protected-data-base** we are assuming that it is physically possible to order events at **pdb**, in **rw** that there can be an ordering on events with **rw** and s_{c_i} as target. These are realizable because **pdb**, **rw**, s_{c_i} have no specifications other than this ordering which conflicts at this level of detail. By contrast, a specification of the Courtois program saying both that *mutex 1* and *mutex 2* events should be ordered and that *mutex 1* and *mutex 2* are separate semaphore actors is inconsistent and not realizable.

The guiding principle in writing specifications that will be realizable is of course not to overspecify. Having two different events of some activator with same event count is a conflict (a danger in parallelism), inconsistent time orders is another. The clear nature of inconsistency enables proof of certain properties (namely, impossibility of bad properties) to proceed. In this paper we have illustrated the later technique and not really exercised testing of specifications. However, Chapter 3 and the Appendix have examples of common types of inconsistencies.

Another important point about specifications of flexible level of detail is that they enable explicit statements about parts of programs and abstraction about program schemes (templates or structures). Thus they allow us to describe real code, as it is actually written, at all levels. This is not always possible in formalisms which stick to one level of abstraction. We feel the ability to represent separately the parts of a program is extremely important to the understanding of programs. Besides clarity to humans, another goal of structured programming is easy checking of code. Styles of programming which cause difficulties in placement of assertions or in modularization are considered to contribute to "unrealizable" programming since they make automatic verification techniques more difficult to apply. Our position is that rather than restrict the kinds of programs one can expect to rely on, one should learn to express one's understanding of code. People do write machine language code and know what it is about. Some of it is even quite reliable. How is it understood? We suggest that in a formalism in which

various abstractions can be expressed, there is hope that by combining statements of different levels one can express both how primitives (such as semaphores and goto) are being used (the structure of the program) and what their properties are. Particularly if we are ever to deal with low level code in which goto's and weird side-effects and semaphore-like primitives cannot be avoided, we must learn to represent the programmers' understanding of his code, its structure, its parts.

The structured programming approach and mechanizable proving approach attempt to reduce hard problems to mechanical ones. We see no point in trying to represent parallel processing as a simple subject. This work is an attempt to develop a full and precise theory of the subject.

10.3 Future Research

Future research of three sorts can be projected. The areas are extensions of the uses of this model to other problems, programming language issues relating to synchronization and the use of specification formalisms in the wide open field of development of programs that understand programs.

We would like to try to use the notions of partial orders to analyze single processor implementations of side-effect programs and coroutines. While we understand that side-effects may be avoidable at some level of understanding of sequential code and coroutines, we feel that in fact it may be beneficial to view procedures with side-effects or coroutines in the same way as we do communication among independent processes. Side-effects as communication devices an be viewed positively and can perhaps be analyzed through partial orderings even in the absence of true parallelism.

Other topics which bear further study are the full range of problem type and

implementation properties that can be expressed in the language. This would include consideration of modifications of the model for applicability to particular known configurations, introducing additional activator properties.

While using this language to specify problems and their solutions, we have touched on several issues in programming of synchronization. Particularly important are relative power of primitives, structure implicit in certain primitives. and possible misinterpretation of uses of existing primitives (e.g., conditional critical regions and the semaphore implementation of the monitor). The fact that through specifications in this language we have exposed or found reasons for the difficulties indicates the power of this approach. We feel there is considerable work to be done in understanding what is desirable in synchronization primitives, how existing ones can be used, and how they can be implemented. Progress in this area will depend on both precise and adequately expressive formalisms in which to specify the objects (primitives or programs) under consideration.

Finally, we would like to see how this approach could be used as a basis for implementing a theory of parallel processing. A program which can understand programs will need to understand parts of programs and interaction among parts. If it is to be helpful as a program manipulater [Knuth, 1974] it must have representation of both structured and unstructured programming. Even if programmers are found to "think" better and to program more reliably in structured languages it is unlikely that optimal code will ever be of the same structured form. Some understanding of alternatives must be available if such code is to be understood.

Appendix Behavioral Definition of PLANNER-73

Programming language primitives can be defined by specifications of properties of their behaviors. The difference between an actor system and the program implementing that system can be viewed as a difference in level of detail of description.. In the actual system, messages are sent to actors and this is reflected directly in events in the behavior. The names used in the events are the names of the actual actors involved. In a programming language, there have to be additional conventions for assigning names to actors so that the behavior of a target actor can be defined before it is known what particular message it may be sent. The definitions of primitives for writing programs describe how these names are used by saying how the program can be interpreted when it is sent a message. The programming language is a means for invoking certain known actors in the particular circumstances required by the programmer. The behavior of interpreting this code contains events in which names are given meaning and events involving the intended actors are caused. If all events involving code are eliminated from a behavior one should be left with the corresponding behavior of the system which the programmer originally envisioned, but which he had to describe indirectly through the programming language. This is the relation between a program and the system to be realized.

Programming language constructs will be defined as part of a subset of PLANNER-73 [Hewitt, 1973] in this Appendix. The language also includes primitives which have already been defined, such as cells and synchronizers. The programming language primitive **cons-cell** is simply a means for invoking the actor **cons-cell** as defined already. Programs are run in environments in which names are given meaning. **Cons-cell** always[1] means the actor with the behavior defined in Chapter 3, just as λ always has the same meaning in a λ-expression.

---

[1] Unless "rebound," as PLUS could be in LISP.

Certain actors can change the environment by associating names with actors. (I.e., a procedure is an actor that changes the environment by binding argument names to arguments.) Others can specify how to change the environment should it become necessary. In the case of a procedure definition in which it is specified how arguments are received by a procedure. The procedure definition actor says how to go about changing the environments before using the program for the procedure body.

Programming language definition based on continuation semantics is not original, since continuations have been used for semantic definitions of languages by Reynolds [1972] and others. It is somewhat different in that there is no further structure, such as functions, beneath the continuation semantics. The continuation is not an argument to a function which gives the semantics. One addition is the use of a continuation-like complaint department to which error messages are sent, thus facilitating more explicit definition of error handling in the specifications (and in programming at the lowest level of detail).

The presentation separates certain basic templates for expressing transmissions from others which correspond to more familiar programming language primitives. The primitives "P_{kernel}" are of this first type. They are described in English in Section A.1 and by axioms in Section A.2. Section A.3 contains definitions of the conventional primitives for side-effect-free programming in languages such as LISP and Algol. This is where sequential code, **goto**'s and **if-then-else** can be found. Section A.4 contains the other primitives of interest for this work, namely, the **cell**, a primitive for parallelism and a monitor-like actor.

A.1 The Primitives of P_{kernel}

The simplest programming language for actors has in it primitives for describing side-effect-free transactions among actors run by a single activator. The primitives of such a

language include a transmission actor, a receiver actor, and a sequence actor. The transmission actor is required to include transmissions in the scripts of actors, the receiver to write the script of an actor by specifying what the activator will do when this actor receives a message, the sequence actor to describe complex actors (like the transmitted actor in most events we have described, i.e. **(apply: [x y] (then-to: c))**). In addition, the programming language must have variables and quoted constants for naming actors.

As an actor a program must know how to interpret or evaluate itself so that these names and programs for scripts can eventually cause the transmissions which they describe. The polymorphic function Eval, when applied to an actor, x, and an environment, **env**, causes the actor **['eval env]** to be sent to x. The interpretation of names is made by reference to an environment. I.e. if

(x <<== y)

is in a program and if the environment for evaluating the program has x and y evaluate to actors a_x and a_y, then eventually in evaluating itself in activator α, the actor **(x <<== y)** must cause the event $\langle a_x\ a_y\ \alpha\ ?\rangle$.

Receivers (the programs for actors as target) are evaluated to closures -- combinations of program and environment -- so that their code can be interpreted in the appropriate environment when they do receive messages. Definitions of programming language primitives are mainly descriptions of how the primitives evaluate themselves.

Environments are changed when new bindings must be added. This can be done without side-effects (i.e., without really changing the behavior of any actor) by simply creating a new environment and using it in the future. New bindings arise when a target actor receives a transmitted actor and must set up a correspondence between it (or its parts) and the names which are used in the program for the script of the receiver (target actor). To accomplish this, receivers

have patterns of variables and other actors which are matched against the transmitted actor. The function match is also a polymorphic operator. Match(x y env) results in sending ['match y env] to x. Thus some actors must know how to match themselves.

The most common form of a transmitted actor is a particular sequence actor for which we will use a special notation to make clear what its elements are. Its elements are "arguments," the continuation and the complaint department. This complaint department is like a continuation in that it can be sent messages. It is generally used as an error continuation. For any actor which we define here by saying what events are caused by receipt of particular messages, we will assume that for all other messages, the event <cd 'not applicable α ?> is caused. Thus these primitives can be assumed to respond (in the sense of causing more events) to all messages received. The complaint department can be explicitly programmed (see definition of **cases** statement) so that there is a direct means of expressing the flow of control on error conditions and treatment of errors. The expected form of a transmitted actor will now be

(apply: x (then-to: c) (else-to: cd)).

The reader should keep in mind that this is just an abbreviation for a sequence with quoted constants denoting interpretations of elements: ['apply: x 'then-to c 'else-to cd].

Evaluation is assumed to be done with a continuation and complaint department. However, if a piece of code evaluates to a transmission, the continuation and complaint department that accompanied the 'eval message, will generally not be remembered. The system of actors to which this code evaluates is an independent system of actors which will determine its own flow of control. One can define actors which do evaluate to values and which send those values to the continuation that came with the 'eval message. This is a functional kind of semantics similar to the λ-calculus and is discussed in section A.3. Code actors all know how to evaluate themselves when sent 'eval, an environment and a continuation.

One further point should be made about these programming language actors. In order for them to behave like ordinary programming languages they must know how to "read" or "parse" themselves, so that they get grouped into reasonable syntactic blocks for evaluation. For example, we don't really want to talk about how ==>> evaluates itself, but rather about how (t ==>> m) evaluates itself. We will assume this parsing is done and that the parsed form of an entire program, x, is sent the following message, to start evaluation:

(apply: ['eval empty-environment] (then-to: default) (else-to: default)).

Default can be interpreted as an I/O device if ever sent a message.

Sequence

The expression $[x_1 \ldots x_n]$ evaluates itself by evaluating x_1, x_2, and so on in order until x_n. Its value is the sequence of the values to which the x_i evaluate. This sequence actor also has the properties that it knows its ith element and that it can match itself against other sequences, element by element.

Transmitter

Evaluation of the expression **(expression-for-target <<== expression-for-transmission)**, will cause evaluation of **expression-for-target** and of **expression-for -transmission** to say **target** and **transmission**, followed by sending of transmission to target. For example the actor

(a-sequence <<== (apply: [3] (then-to: c)))

or, equivalently, the actor

((apply: [3] (then-to: c)) ==>> a-sequence)

represents the transmission of the result of evaluating **(apply: [3] (then-to: c))** to the actor to which **a-sequence** evaluates. It is left up to **a-sequence**, not the evaluator, to send a message to **c**. A

transmitter evaluates itself by evaluating its target expression (a-sequence in the example), then evaluating its transmission (the actor (apply:[3] (then-to: c)) here), and then sending the value of the transmitter to the value of the target.

Receivers

The expression (==> pattern body) evaluates to an actor which can receive transmitted actors and make use of the actors in those tranmsitted actors. Since the expression body is meant to corresponds to an actor's script, the code in the body must be interpreted in an environment in which variables correspond to the intended actors. Thus the receiver must evaluate itself by forming its closure with an environment. When this closure is sent a transmission, it matches the expression pat and the transmission, and then evaluates body in the environment consisting of the closure environment and any new bindings caused in matching. Thus the pattern must be an actor which can match itself against other actors (most often it will be a sequence). The following expression

```
( (==>>
        (apply: =x (then-to: =the-continuation))
        (x <== (apply: [2] (then-to: the-continuation))))
                                        <==
                                              (apply: [1 5 6] (then-to: c)) )
```

represents the actor which when evaluated in some environment in which c is bound, send 2 to c. In more detail what happens is that the target (the receiver) and the transmission are evaluated after which the a transmission is caused. That transmission causes pattern matching in which x is bound to [1 5 6] and the-continuation to c. Now the body of the receiver is evaluated in an environment containing that binding. This causes sending the message, [2], and the continuation, c, to the sequence [1 5 6]. By the axioms for sequence this will result in c being sent 5. Any further properties of the behavior depend on properties of c.

Pattern Matcher

The polymorphic operator match bears a strong analogy to **eval**. Actors used as patterns must know how to receive messages of the form:

(apply: ['match x environment] (then-to: ...) (else-to: ...)).

For instance a sequence matches a message sequence by matching each of its parts against the parts of the message sequence. The pattern =x matches anything and creates an environment in which x is bound to the actor it matched to. Thus the pattern [=x 3 =y] matches the actor [2 3 4] and produces an environment with x bound to 3 and y bound to 4. There have to be primitives for testing equality and identity of actors. The pattern ? matches anything.

A.2 Axioms for Primitives of P_{kernel}

This section contains axiomatic behavioral definitions of primitives of P_{kernel}. These axioms are presented assuming a single activator since there are no actors for introducing parallelism in this language. See section A.4 for an explanation of how these definitions should be modified to apply to a parallel processing environment.

Evaluation in PLANNER-73

> Cons-Sequence. The way this actor evaluates itself is to ask each of its elements to evaluate itself and then to create a sequence actor from them. The newly created sequence has the property that it knows its ith element.

$\langle [x_1 \ldots x_m]$ (apply: ['eval env] (then-to: c) (else-to: cd)) α $ec_1\rangle$
--> $\langle x_1$ (apply: ['eval env] (then-to: c_1*) (else-to: cd)) α $ec_1'\rangle$
and for $1 \le i < n$
$\langle c_i$ (apply: v_i) α $ec_{i+1}\rangle$
--> $\langle v_{i+1}$ (apply: ['eval env] (then-to: c_{i+1}*) (else-to cd)) α $ec_{i+1}'\rangle$
and $\langle c_n$ (apply: v_n) α $ec_{n+1}\rangle$
--> $\langle c$ (apply: sequence*) α $ec_{n+1}'\rangle$

The actor **sequence** satisfies axioms:

<sequence (apply: [i] (then-to: c) (else-to: cd)) α ec>
--> <c (apply: v_i) α ec'> for $1 \le i \le n$

and

<sequence (apply: [i] (then-to: c) (else-to: cd)) α ec>
--> <cd (apply: 'not-applicable) α ec'>.

Transmitter. The transmission template **(t ==>> m)** is used to write a program to cause the sending of a transmission to another actor.

<((t ==>> m) (apply: ['eval env] (then-to: c) (else-to: cd)) α ec_1>
--> <t (apply: ['eval env] (then-to: c_1*) (then-to: cd_1*)) α ec_1'>

where c_1 satisfies
<c_1 (apply: v_1) α ec_2>
--> <m (apply: ['eval env] (then-to: c_2*) (else-to: cd_2*)) α ec_2'>

where c_2 satisfies
<c_2 (apply: v_2) α ec_3>
--> <v_1 v_2 α ec_3'>.

Notice that the assumption is that both t and m will eventually evaluate to values returned to the continuation. If this is not the case then the completion of the evaluation of this transmission, i.e., the causing of

$$\langle v_1\ v_2\ \alpha\ ec_3 \rangle$$

may not occur. (If t is a receiver and m a sequence, the values will be returned.) It might be the case that $cd_1 = cd_2 = cd$ would be a sufficient specification. However, one might prefer to know whether trouble is in t or m evaluation. Thus cd_1 might have behavior

<cd_1 'not applicable ? ?>
--> <cd 'target trouble ? ?>

while cd_2 reports message trouble. Also notice that v_2 is sent as the entire transmitted actor. Hopefully it will be of a form which v_1 accepts.

Receivers. A receiver template, (==>> pat body), constructs an actor which is a closure, including the environment in which it is evaluated. Receivers (closures) can receive transmissions and make use of their messages. Since they do this by evaluating the code in their bodies, they must bind variables and update the environment.

<(==>> pat body) (apply: ['eval env] (then-to: c) (else-to: cd)) α ec_1>
--> <c (apply: receiver*) α ec_1'>

where receiver satisfies
<receiver trans α ec_2>
--> <pat (apply: ['match env trans] (then-to: c_1*) (else-to: default)) α ec_2'>

where c_1 satisfies
<c_1 (apply: env_1) α ec_3>
--> <body (apply: ['eval env_1] (then-to: default) (else-to: default)) α ec_3'>.

The behavior of env_1 is specified by the pattern matcher axioms which follow.

Once a receiver is created, it tries to act like the actor the programmer meant to implement by writing the program. Thus it can receive transmitted actors after which control follows its "script." Since this script is still only implicit in the program of the body section, creation of a new environment and evaluation must be part of the behavior. However, this evaluation (probably better termed interpretation in this case) is not for producing a value but rather for finding out what activities should be going on. Thus in evaluating the body the default continuation and a complaint department are used. If a message is ever sent to default, it means that body evaluated to a value (or an error condition arose). Either is cause for "outside"

notice and we are leaving open for interpretation what the form of this notice is by using the actor **default** for which we give no specifications. The point is that the receiver is supposed to be totally responsible for determining future control. If **trans** and **pat** are of the form **(apply: x (then-to: ?) (else-to: ?))**, then **body** has variables bound to actors which it can use as continuation or complaint department. If it doesn't use them, something is strange and **default** should hear about it. **Default** can be considered a convenient representation for an I/O interface with the outside of the actor system realized by the program.

Pattern Matching

If an actor wants to match another actor against a pattern it sends that pattern a message containing the atom 'match, the actor to be matched, and the environment in which the matching is to be done. The environment is necessary since in general there will be atoms in the pattern which will have to be "looked up" in the environment. In the definition of a receiver, there is no reason to assume **pat** is evaluated before matching. A behavioral description of pattern matching includes descriptions of the primitives **identical**, **cons-variable**, **cons-atom** and additional behaviors of sequences, and environments. **Cons-variable** and **cons-atom** will be treated somewhat casually. We will simply use =x (or 'x) to refer to the actor created by sending x to **cons-variable**, written =, (or x to **cons-atom**, written ').

Matching Behavior of Sequences

$$\langle [x_1 \ldots x_n]\ \textbf{(apply: ['match env } [m_1 \ldots m_n]\textbf{] (then-to: c) (else-to: cd))}\ \alpha\ ec\rangle$$

causes

$$\langle x_1\ \textbf{(apply: ['match env } m_1\textbf{] (then-to: } c_1\textbf{) (else-to: cd))}\ \alpha\ ec'\rangle$$

and for $1 \le i < n - 1$

where

$\langle c_i$ (apply: env_i) α ec$\rangle$

causes

$\langle x_{i+1}$ (apply: ['match env_i m_{i+1}] (then-to: c_{i+1}*) (else-to: cd)) α $ec_1'\rangle$

and

$\langle c_{n-1}$ (apply: env_{n-1}) α $ec_{n-1}\rangle$

causes

$\langle x_n$ (apply: ['match env_{n-1} m_n] (then-to: c) (else-to: cd)) α $ec_{n-1}'\rangle$.

The Behavior of =x

$\langle$=x (apply: ['match env y] (then-to: c) (else-to: cd)) α ec$\rangle$

causes

$\langle$c (apply: env_1*) α ec'$\rangle$

in which env_1 has the same behavior as env except that

$\langle env_1$ (apply: ['lookup x] (then-to: c) (else-to: cd)) α ec$\rangle$

causes

$\langle$c (apply: y) α ec'$\rangle$.

The Behavior of 'x

<'x (apply: ['match env y] (then-to: c) (else-to: cd)) α ec>

causes

<identical (apply: [x y env] (then-to: c_1*)(else-to: cd)) α ec'>

where

<c_1 (apply: 'yes) α ec>

causes

<c (apply: env) α ec'>.

The Behavior of Identical

<identical (apply: [x y env] (then-to: c) (else-to: cd)) α ec>

causes

<c (apply: 'yes) α ec'>
iff x = y
or the result of looking up x in env is identical to y.

Otherwise it sends 'no to cd. [Note that a newly created actor is not identical to any existing actor, i.e. if x is newly created there is no y ∋ y = x]

Behavior of ?

<? (apply: ['match env x] (then-to: c) (else-to: cd)) α ec>

causes

<c (apply: env) α ec'>.

Behavior of empty-environment

<empty-environment (apply: ['lookup x] (then-to: c) (else-to: cd)) α ec>

causes

<cd (apply: 'not-applicable) α ec'>

and

<empty-environment (apply: 'empty (then-to: c) (else-to: cd)) α ec>

causes

<c (apply: 'yes) α ec'>.

A.3 Relationship to Other Languages -- More Familiar Primitives

Many systems corresponding to more familiar and convenient language constructs can be shown to be in Σ_{kernel}. Σ_{kernel} is as large as the set of systems realizable in say PURE LISP, Lambda Calculus, etc. However, although the same systems can be realized, if one tried to compare the means of realization, in some cases they would not be the same. That is, programs involving primitives such as functional application and composition have behaviors which at the

level of detail at which 'eval messages are apparent, are very different from the behaviors of P_{kernel} programs for the same systems. Others, such as functional abstraction (λ) are just abbreviations for code in P_{kernel} and would have identical behavior. In this section, several of these more familiar primitives are introduced and explained either as abbreviations or by behavioral specifications. The language extended by the lambda-like receiver, ≡>, a functional applications transmission, ≡>, cases, let, and labels, can be referred to as $P_{functional}$. Only informal argument that Σ_{kernel} equals $\Sigma_{functional}$ will be given along with references to work in programming language semantics which presents more formal versions of essentially the same arguments (they are translatable to behavioral semantics).

The actors cons-receiver, ≡>, and cons-transmitter, ≡>, can be used to write programs without explicit continuations. They correspond to λ-expressions and combinations of the λ-calculus. Thus

```
([6] ≡> (≡> [≡x] (x * 2)))
```

evaluates to 8.

The difference between ≡> and ≡≡>> is just that with ≡> there is no explicit binding to the continuation or complaint department. Thus these actors cannot be referred to explicitly in the body code and so cannot be used for any purpose other than to be the continuation or complaint department. Thus

```
(≡> pat body)
```

is an abbreviation for

```
(≡≡>> (apply: pat (then-to: ≡c) (else-to: ≡cd)) (c <<≡≡ body))
```

provided c and cd are not free in the body. The value of the body will be sent to the actor bound to c but there can be no jump of control from the middle of the body to that actor.

The reasons for believing that adding '=>' to the language won't increase the set of systems are based on work already done in continuation semantics [Fischer, 1972; Reynolds, 1972]. It is known that functions and composition can be computed in a continuation style. A translation from y = (f (g x)) to continuations language would be something like

```
(g ==>> (apply: x (then-to: (==>> (apply: =z) (f <<== (apply: z (then-to: actor-for-y) (else-to: cd)) ) )
                  (else-to: cd)) )
```

I.e. the innermost application is done first with a continuation which can do therest of the nested functions. In fact, even though y = (f (g x)) is written in reverse order, even its semantics as nested functions require computation in the continuation program ordering. This is why we claim there is no difference between Σ_{kernel} and $\Sigma_{functional}$. The actor system which the program implements is the same in either case.

The primitive => is best described by axioms rather than as an abbreviation, because at the 'eval level it does have different behavior. It must somehow pick up the continuation which comes in with the eval messages.

<(t => m) (apply: ['eval env] (then-to: c) (else-to: cd))α ec_1>
--> <t (apply: ['eval env] (then-to: c_1*) (then-to: cd_1*)) α ec_1'>

wher c_1 satisfies
<c_1 (apply: v_1) α ec_2>
--> <m (apply: ['eval env] (then-to: c_2*) (else-to: cd_2*)) α ec_2'>

where c_2 satisfies
<c_2 (apply: v_2) α ec_3>
--> <v_1 (apply: v_2 (then-to: c) (else-to: cd)) α ec_3'>.

In PLANNER-73 both the => and =>> transmissions can be used in the same program. The result is often that the =>> transmissions correspond to goto's or the J-operator [Landin, 1965]

in that they change the flow of control by dropping the evaluation continuation. Thus in this model of computation both kinds of control flow -- nested function calls, and the "unstructured" goto -- have direct semantic analogues in behaviors. This can be seen in sequential style of programming (usually written $x_1 ; \ldots ; x_n$ in Algol-like languages).

The actor for sequential control evaluates itself by evaluating sequentially each of x_1 to x_n and returning the value of x_n as its value. The form $[x_1 ; \ldots ; x_n]$ will be an abbreviation for

$$(x_1 \Rightarrow (\equiv\equiv\rangle\rangle\ ? \quad (x_2 \Rightarrow (\equiv\equiv\rangle\rangle\ ? \quad \ldots \quad x_n) \ldots))\ )$$

Thus for n = 3 we have

$$[x_1 ; x_2 ; x_3] \equiv (x_1 \Rightarrow (\equiv\equiv\rangle\rangle\ ? \quad (x_2 \Rightarrow (\equiv\equiv\rangle\rangle\ ? \quad x_3))))$$

In each transmission any transmitted actor is acceptable (i.e. matches the pattern ?) and causes evaluation of the body. Only the value of evaluating the last expression in the sequence is returned.

The "goto" as a transfer of control can be examined now. Perhaps instead of a function call x_2 is a jump. Say x_2 is (t $\langle\langle\equiv$ m), a goto[1]. Then we have

$$(_0\ x_1 \Rightarrow (_1\ \equiv\equiv\rangle\rangle\ ? \quad (_2\ (_4\ t\ \langle\langle\equiv\ m)_4\ \Rightarrow (_3\ \equiv\equiv\rangle\rangle\ ? \quad x_3\ )_3\ )_2\ )_1\ )_0.$$

with subscripted parentheses for future reference. Some of the events in the evaluation of this functional transmission are

---

[1] It is a goto because it drops the rest of the sequence for its continuation.

(a) $\langle(\ldots)_0$ (apply: ['eval env] (then-to: c_0) (else-to: cd_0)) α?>

(b) $\langle(\ldots)_1$ (apply: ['eval env] (then-to: c_1*) (else-to: cd_1*)) α ?>

(c) $\langle c_1$ (apply: $closure_1$) α ?>

(d) $\langle x_1$ (apply: ['eval env] (then-to: c_2*) (else-to: cd_2*)) α ?>

(e) $\langle c_2$ (apply: v_1) α ?>

(f) $\langle closure_1$ (apply: v_1 (then-to: c_0) (else-to: cd_0)) α ?>

...

(g) $\langle(==\rangle\rangle$? x_3 $)_3$ (apply: ['eval env] (then-to: c_3*) else-to: cd_3*)) α ?>

...

(h) < (t <<== m) (apply: ['eval env] (then-to: c_4) (else-to: cd_4)) α ?>

...

(i) $\langle v_t\ v_m\ \alpha$?>
where v_t and v_m are results of evaluating t and m

and the rest of the sequence is forgotten, i.e., neither v_t nor v_m knows about c_4, therefore neither knows about c_0 and therefore there is no further carrying around of the "functional " return, c_0, which is no longer to be returned to anyway.[1]

This suggests considering the directness of the semantics. If the implementation is as described, it clearly is efficiently using space, since it only holds on to return points while they may be useful. As a semantic model it is also "efficient" in the sense that this dropping of

---

[1] The event (a) is evaluation of the whole form, (b) evaluation of the target, (d) of the message. In (f) the transmission is done after which $(\ldots)_1$ gets evaluated, leading to evaluation of its body ... . In (g) the target of that body is evaluated, and in (h) the message. With (i) control "jumps" due to the structure of the message.

continuations can be expressed directly rather than being put on top of some functional notation which in fact has implicit return points not reflecting important semantics concepts. [Compare to comments by Landin [1966] to the effect that this does not matter since when not an implementation, efficiency doesn't matter. We feel this is a mistake since clarity is closely related to eliminating extraneous details.]

Cases

The responses of some actors depends on the transmissions they receive. Thus it becomes convenient to have a constructor for programs that can make choices. The **cases** actor creates an actor which is related to the **if-then-else** construct. The actor

```
(cases
      f
      g)
```

represents an actor which will act like f if it receives messages which f can accept (i.e. which matches f's pattern, if f is a receiver) and otherwise will act like g if it receives messages which it accepts.

Cases evaluates itself by evaluating each of its parts and creating a new actor from their values. This new actor can send any transmissions it receives to each of its parts successively until one accepts it.

Cases is not primitive since **(cases f g)** is just an abbreviation for the following:

```
(==>> [=f =g]
   (==>> (apply: =m (then-to: =c) (else-to: =cd))
      (f <<== (apply: m (then-to: c)
                        (else-to:
                              (==>> 'not-applicable
                                    (g <<== (apply: m
                                                (then-to: c)
                                                (else-to: cd))))))))))
```

Thus **(cases <= [f g])** would be equivalent to the cases statement described above. Notice that if f

and g are receivers, the evaluation of f and g before forming the cases statement should form closures of f and g. Thus for this use of cases there is no danger of evaluation of f and g not terminating and preventing the formation of the cases statement.[1] To create a primitive in which evaluation is delayed for any actor one would have to add **cases** as a primitive (rather than as a macro which is essentially what the abbreviations are).

An Algol "if P(x) then f else g ;" statement where P(x) is of the form "x = a", would translate to

```
( [x] => (cases
            (=> [a] f)
            (=> ? g) ) ).
```

Let

Let is not primitive. The expression

```
(let {[x = y]} A)
```

is an abbreviation for

```
([y] => (=> [=x] A))
```

Labels

Labels is convenient for defining actors in terms of themselves. Its general form is:

```
(labels {[x = D]}
        body).
```

---

[1] In general one does not want to evaluate the arguments to a conditional until one has to. In if a then b else c. If c is nonterminating and a true, evaluating first leads to nontermination, evaluating after leads to b.

For example, the definition of factorial to be in the body is:

```
(labels {[factorial ≡
          (≡> [=n]
              (rules n
                  (≡> 0
                      1)
                  (≡> ?
                      (n * (factorial (n - 1)))))))]}
        body)
```

The labels actor does not extend the class of systems implementable since the recursion which it introduces can be simulated. The above program could be replaced by:

```
(let {[factorial ≡
          (let
              {[internal-factorial ≡
                        (≡> [=n =f]
                            (rules n
                                (≡> 0
                                    1)
                                (≡> ?
                                    (n * (f (n - 1) f)))))]}
              (≡> [=n]
                  (internal-factorial n internal-factorial))))]}
        body)
```

Notice that the principle of this rewrite procedure is that the actor being defined is sent as an argument in each recursive call. Unfortunately this does not mean that there is a simple form of which a labels statement can be considered to be an abbreviation. Thus for convenience labels is usually contained as primitive in a language (like LISP).

> Axioms for Labels: Evaluation of a labels expression causes evaluation of the body of the expression in a special environment. In that environment not only are x_i bound to D_i, but evaluation of D_i proceeds in that same special environment. This is the source of the recursion, by which if D_i contains x_i, then that x_i will be evaluated correctly (namely, to the same D_i).
>
> <(labels {[x_1 ≡ D_1] ... [x_n ≡ D_n]} body) (apply: ['eval env] (then-to: c) (else-to: cd)) *αec*>
> --> <body (apply: ['eval env_1] (then-to: c) (else-to: cd)) *α ec'*>

where env_1 has the same axioms as env except for

<env_1 (apply: ['lookup x_i] (then-to: c_0) (else-to: cd_0)) α ec_0>
--> <D_i (apply: ['eval env_1] (then-to: c_0) (else-to: cd_0)) α ec_0'>

In summary, an evaluated receiver (a closure) formed from an expression (=> pat body) always sends a continuation with the eval message that it sends to its body. This continuation is the continuation which comes with the transmitted actor which it receives. Thus it is all right for it to evaluate to a single value with no further specification of control. This is made meaningful by the related conventions about continuations observed by the transmitter =>. When an unevaluated transmitter formed from => is evaluated with continuation c, the resulting transmission includes as it then-to part the actor c. Therefore the transmission is done with a continuation. Cases is an extended form of if-then-else (ALGOL) or cond (LISP) which forms an actor from a sequence of receivers (either => or ==>>) which when later sent as transmission actor evaluates the body of the first of those receivers to have a pattern that matches. Let is an abbreviation for binding, labels facilitates recursive definitions.

A.4 Cells, Parallelism and Synchronization

If cons-cell is added to our language, the expression (cons-cell x) does not have to be treated as a programming language primitive in the sense of knowing how to evaluate itself. P_{kernel} can be considered extended by the ability to refer to the actor cons-cell. This can be accomplished by the inclusion in the "empty environment" of the binding of the *expresston* cons-cell to the *actor* cons-cell. Then the program statement (cons-cell x) can be treated as any other transmission[1] causing the event

---

[1] Technically, this leaves the possibility of rebinding cons-cell later, as can happen to atoms like PLUS or 3 in LISP.

<cons-cell (apply: a_x (then-to: ?)) ? ?>.

The axioms for the actor cons-cell are as given in Chapter 3.

There are many primitives for parallel processing which we could add to our language. We will choose one, the parallel sequence actor, and use it as the basis of our discussion of systems with parallelism, pointing out those properties due to this particular style of parallelism. A parallel sequence evaluates itself by evaluating each of its elements in parallel. That is, it creates t parallel processes, one for each element and starts each off evaluating one element. If these evaluations produce values, they are placed together in a new sequence, and the processes are terminated.[1]

Axioms for Parallel Sequence

$$\langle [\!|\, x_1 \ldots x_n \,|\!] \text{ (apply: ['eval env] (then-to: c) (else-to: cd) } \alpha \; ec \rangle$$

causes

the n events
$\langle x_i$ (apply: ['eval env] (then-to: c_i) (else-to: cd_i)) $\alpha_i \; 0 \rangle$
for $1 \leq i \leq n$

where the c_i have behavior

for every set of events $\{E_1, \ldots, E_n\}$
where $E_i = \langle c_i$ (apply: v_i) $\beta_i \; ec_i \rangle$

there is an event
$\langle c$ (apply: $[v_1 \ldots v_n]$) $\alpha_{\beta_1, \ldots, \beta_n} \; 0 \rangle$.

---

[1] One variation on this primitive might not include this "joining" aspect, but just start off multiple processes, as the basic transmission primitives starts off transmissions without providing for continuation.

Thus the α_i are the new activators. Since additional parallelism could arise in evaluation of any or all of the x_j, the events E_i are specified with names of arbitrary activator β_i. For every such set of events the actor c must receive the sequence of values. In case there is more than one, a newly created activator will be used for each transmission of values.

Just as there had to be provision for unexpected parallelism in evaluating x_i, technically the axioms for the Σ_{kernel} primitives must be changed to specify how they behave in the presence of parallelism. Since an evaluated receiver might be the target of an event in any activator not just the one in which it was evaluated, some parts of the definition must be changed to indicate it. The following is a revised receiver definition.

<(==)> pat body (apply: ['eval env] (then-to: c) (else-to: cd)) α_1 ec_1>
--> <c (apply: receiver*) α_1 ec_1'>

where receiver satisfies
<receiver (apply: trans) α_2 ec_2>
--> <pat (apply: ['match env trans] (then-to: c_1*) (else-to: default)) α_2 ec_2'>
where c_1 satisfies
<c_1 (apply: env_1) α_3 ec_3>
--> <body (apply: ['eval env_1] (then-to: none) (else-to: default)) α_3 ec_3'>.

Other axioms should be modified similarly.

Finally, we define an actor cons-monitor to which the expression cons-monitor evaluates, extending PLANNER-73 to include programs of monitor-like objects. We give a general definition of the properties of cons-monitor first and then show how with particular forms of exp and exp_2, the PLANNER-73 equivalent of the monitor is written.

If the form is

(cons-monitor exp_1 exp_2)

Exp_1 must evaluate to a number, exp_2 to an actor, m, which accepts sequences and which returns an actor inside to the continuation. The programming language primitive **cons-monitor** evaluates to **cons-monitor** and is sent the sequence [n m].

<cons-monitor (apply: [n m] (then-to: c)) α *ec*>

<m (apply: [$cond_1$* ... $cond_n$*] (then-to: c_1)) α *ec'*>

<c_1 (apply: inside*) α *ec''*>

<c (apply: a-monitor*) α *ec'''*>

Some events of interest in describing the monitor can be abbreviated as follows:

E = <a-monitor (apply: x (then-to: c)) α *ec*>

E_{in} = <inside (apply: x (then-to: s_c*)) α *ec'*>

E_{out} = <s_c y α *ec''*>

E_{end} = <c y α *ec'''*>

E_{wait} = <$cond_i$ (apply: 'wait (then-to: c)) α ec_1>

E_{signal} = <$cond_i$ (apply: 'signal (then-to: c)) α ec_1>

The one restriction we will assume on m and therefore on inside is that if an event E_{signal} ever happens, E_{signal} actually always happens with a continuation s_c.[1] It causes either

---

[1] How to "enforce" or check for this in general is not at issue. It can be checked for trivially in the restricted syntax of exp_2 which corresponds to Hoare's monitor with restriction that signals are last statements. See syntax after this definition.

$E_{out} = \langle s_c \text{ (apply:) } \alpha\ ec_1' \rangle$

or

$E_{next_s} = \langle c \text{ (apply:) } \alpha\ ec_1' \rangle$[2]

$Cond_i$ have definitions like semaphores. E_{wait} is followed by the event (if any)

$E_{next_w} = \langle c \text{ (apply:) } \alpha\ ec' \rangle$

The E_{signal} property depends on whether there are waiting processes or not. Then it causes either an E_{next_s} or an E_{out}. All processes wait on signals.

(a) $E_{1_{wait}}$

causes

$E_{2_{signal}} \Rightarrow E_{next}$

where $E_{2_{signal}}$ is first signal $\ni$ all $E_{i_{wait}} \Rightarrow E_{1_{wait}}$ are released.[2]

(b) $E_{signal} \ni \exists\ E_{1_{wait}} \ni E_{1_{wait}}$ not enabled

causes

E_{next_s}

(c) $E_{signal} \ni$ no unenabled waits precede it

causes

E_{out}

---

[1] Note that conditions know how to decompose s_{c_i}'s.

[2] To state this formally we need a relation like $\mathfrak{T}$ in the semaphore definition.

The rest of the properties of a-monitor are defined by induction on order of requests:

(1) E_1 first in $\Rightarrow_{a\text{-}m}$

causes

$$E_{m_{1_{in}}}$$

(2) $E_2 \ni E_2$ second request in $\Rightarrow_{a\text{-}m}$

causes

$$E \Rightarrow E_{2_{in}}$$

where $E = E_{1_{wait}}$ or $E_{1_{out}}$, whichever is first.
This E is the event $\mathfrak{F}(E_2)$

(3) $E_3 \ni E_3$ is next request after E_2

causes

$$E \Rightarrow E_{2_{in}}$$

where E is first event after $\mathfrak{F}(E_2) \ni E = E_{wait}$ or $E = E_{out}$.

(1) and (2) are the basis, (3) the induction. It uses wait and signals to release the monitor as well as E_{out} events, thus allowing for transference of control of the monitor. This makes it difficult to predict exactly which event will enable a waiting process.

Hoare's monitor can be realized by the syntax:

```
(cons-monitor  n  (≡> [=x1 ... =xn]1
                     (let  {y = ai}2
                           (cases
                                 (≡> pat1  body1)

                                 ...

                                 ( ≡> patm bodym)))))
```

Presumably the x_i will appear in the receiver bodies in condition statements. **Body$_i$ can contain**

```
(xj 'wait)
(xj 'signal)
```

when evaluated these cause events

```
<condj (apply: 'wait (then-to: ?)) ? ?>
```

and

```
<condj (apply: 'signal (then-to: ?)) ? ?>.
```

Body$_i$ is a sequence of statements. If (x$_j$ 'signal) is in a body$_i$ it is the last in the sequence.

We repeat the events in creation of a monitor given this syntax:

```
<cons-monitor (apply: [n (≡> [=x1 ... xn] ...)] (then-to: c)) α ec>

<(≡> [=x1 ... =xn] ... ) (apply: [cond1* ... condn*] (then-to: c1)) α ec'>

<c1 (apply: inside*) α ec''>

<c (apply: a-monitor*) α ec'''>
```

---

1 These are condition names, so this corresponds to the declaration of conditions.

2 These are local variables.

Inside is the closure of the cases statement with x_i bound to $cond_i$ (created by cons-monitor) and y_i bound to a_i. Since all the $cond_i$ and a-monitor are created by cons-monitor it is reasonable to assume that events with any of $cond_i$ or a-monitor as target can be ordered by the same synchronization.[1]

Thus an expression of the form

```
(let [rw = (cons-monitor 2
             (=> [=reader =writer]
                 (let {[readcount = 0]
                       [writecount = 0]
                       [busy = false]}
                 (cases
                       (=> 'startread ...)
                       (=> 'startwrite ...)
                       (=> 'endread ...)
                       (=> 'endwrite ...)) ) )))
```

can be used to create an instance, rw, of the PLANNER-73 version of the write priority monitor of Chapter 6.

---

[1] See unstructured synchronization chapter for assurance that this can be implemented using synchronization dependent only on one actor's ordering.

Bibliography

Bekic, H. 1971. Towards a Mathematical Theory of Processes. Technical Report TR25.125. IBM Laboratory, Vienna.

Burstall, R. 1972. Some Techniques for Proving Correctness of Programs which Alter Data Structures. *Machine Intelligence 7.* ed. D. Michie. Edinburgh University Press.

Cadiou, J. M.; and Levy, J. J. 1973. Mechanizable Proofs about Parallel Processes. *Proceedings of the 14th Annual Sympostum on Switching and Automata Theory.*

Courtois, P. J.; Heymans, F.; and Parnas D. L. 1971. Concurrent Control with "readers" and "writers." *Comm. ACM 14* 10. pp. 667-668.

_____. 1972. Comments on "A Comparison of Two Synchronizing Concepts." *Acta Informatica 1.* pp. 375-376.

Dahl, O-J.; and Hoare, C. A. R. 1972. Hierarchical Program Structures. *Structured Programming.* O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. Academic Press. New York.

Dijkstra, E. W. 1965. Solution of a Problem in Concurrent Programming Control. *Comm. ACM 8* 9. p. 569.

_____. 1968. Cooperating Sequential Processes. *Programming Languages.* Academic Press. New York.

_____. 1972. Notes on Structured Programming. *Structured Programming.* A.P.I.C. Studies in Data Processing No.8. Academic Press. New York. pp. 1-81.

_____. 1971. Hierarchical Ordering of Sequential Processes. *Acta Informatica 1.* pp. 115-138.

Eisenberg, M. A.; and McGuire, M. R. 1972. Further Comments on Dijkstra's Concurrent Programming Control Problem. *Comm. ACM 15* 11. p. 999.

Floyd, R. W. 1967. Assigning Meaning to Programs. *Proceedings of Sympostum in Applied Mathematics. Vol. 19.* pp. 19-32. (ed. J. T. Schwartz) Providence, Rhode Island, American Mathematical Society.

Fischer, M. J. 1972. Lambda Calculus Schemata. *Proceedings of the ACM Conference on Proving Assertions about Programs.* Las Cruces, New Mexico. (Jan. 1972)

Haberman, A. N. 1970. Synchronization of Communicating Processes. *Comm. ACM 15* 3. pp. 171-176.

Hansen, P. B. 1972a. A Comparison of Two Synchronizing Concepts. *Acta Informatica 1.* pp. 190-199.

_____. 1972b. Structured Multi-progamming. *Comm. ACM 15* 7. pp. 574-578.

_____. 1973. A Reply to Comments on "A Comparison of Two Synchronizing Concepts." *Acta Informatica 2.* pp. 189-190.

Hewitt, C. 1974. Protection and Synchronization in Actor Systems. MIT Artificial Intelligence Working Paper.

Hewitt, C.; Bishop P.; and Steiger, R. 1973. A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI-73. Stanford, Calif. Aug, 1973.

Hoare, C. A. R. 1969. An Axiomatic Basis for Computer Programming. *Comm. ACM 12* 10. pp. 576-580.

_____. 1972. Towards a Theory of Parallel Programming. *Operating Systems Techniques.* Academic Press. New York.

_____. 1974. Monitors: An Operating System Structuring Concept. *Comm. ACM 17* 10. pp. 549-557.

_____. 1975. A Structural Approach to Protection. Queen's University of Belfast.

Knuth, D. E. 1966. Additional Comments on a Problem in Concurrent Programming Control. *Comm. ACM 9* 5. pp. 321-322.

_____. 1974. Structured Programming with GOTO Statements. *Computing Surveys Vol 6.*

Jones, A. K. and Lipton, R. J. 1975. The Enforcement of Security Policies for Computation. Department of Computer Science. Carnegie-Mellon University.

Lamport, L. 1974. A New Solution of Dijkstra's Concurrent Programming Problem. *Comm ACM 17* 8. pp. 453-455.

_____. 1974. On Concurrent Reading and Writing. Massachusetts Computer Associates Report CA 7409-0511.

Landin, P. J. 1965. A Correspondence Between ALGOL 60 and Church's Lambda-Notation. *Comm. ACM 8.* pp. 89-101, 158-165.

_____. 1966. A Formal Description of Algol 60 *in Formal Language Description Languages for Computer Programming* ed. T. B. Steel, Jr. North Holland Publishing Company.

Lipton, R. J. 1974. Limitations of Synchronization Primitives. *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing.*

Liskov, B.; and Zilles, S. 1974. Programming with Abstract Data Types. *Proceedings of ACM SIGPLAN Conference on Very High Level Languages. SIGPLAN Notices 9* 4.

Milner, R. 1972. LCF Logic for Computable Functions -- Description of a Machine Implementation. Stanford AI memo 169. Stanford University.

_____. 1973. An Approach to the Semantics of Parallel Programs. *Proc. Convegno di Informatica Teorica.* Pisa.

Reynolds, J. C. 1972. Definitional Interpreters for Higher-Order Programming Languages. *Proceedings of the ACM National Convention.*

Robinson, L. and and Holt, R. C. 1975. Formal Specifications for Solutions to Synchronization Problems. Stanford Research Institute.

Parnas, D. L. 1972. A Technique for Software Module Specification with Examples. *Comm. ACM.*

Parnas, D. L.; and Handzel, G. 1975. More on Specification Techniques for Software Modules. Technische Hochschule Darmstadt Fachbereich Informatik.

Presser, L. 1975. Multiprogramming Coordination. *Computing Surveys Vol* 7 1. pp. 22-44.

Scott, D. 1972. Lattice-Theoretic Models for Various Type-Free Calculi. *Proc. Fourth International Congress for Logic, Methodology, and Philosophy of Science.* Bucharest.

Scott, D.; and Strachey, C. 1971. Towards a Mathematical Semantics for Computer Languages. *Proceeding of the Symposium on Computers and Automata* Microwave Research Institute. Symposia Series Vol 21. Polytechnic Institute of Brooklyn.

Shaw, A. C. 1974. *The Logical Design of Operating Systems* Prentice-Hall. Englewood Cliffs, New Jersey.

Tsichritzis, D. C.; and Bernstein, P. A. 1974. *Operating Systems.* Academic Press. New York.