On Synchronization Primitive Systems

by

Richard J. Lipton

Research Report #22

This work is a slightly different version of the author's doctoral dissertation, prepared for the Computer Science Department at Carnegie-Mellon University, 1973. A portion of this work was supported by the Sloan Foundation.

October 1973

ABSTRACT

We study the question: what "synchronization primitive" should be used to handle inter-process communication? We present a formal model of the process concept, and then we use this model to compare four different synchronization primitives. We are able to prove that there are differences between these synchronization primitives. Although we compare only four synchronization primitives, our general methods can be used to compare other synchronization primitives. Moreover, in our definitions of these synchronization primitives, we explicitly allow conditional branches. In addition, our model separates the notion of scheduler; this separation allows us to unravel the controversy between Brinch Hansen and Courtois, Heymans, Parnas and to define formally the release mechanism of the PV synchronization primitive.

1. INTRODUCTION

Dijkstra [1968a] has demonstrated how an operating system can be designed and validated by using a "synchronization primitive" to handle inter-process communication. Since this accomplishment, an important issue in the design of an operating system has been:

> (*) What "synchronization primitives" should be used to handle inter-process communication?

Dijkstra used the "synchronization primitive" PV; however, other synchronization primitives have been proposed. Currently, the selection of a synchronization primitive is an ad hoc design decision.

The current attempts to answer the question (*) each show that a given synchronization primitive can "solve" a given "synchronization problem". These synchronization problems include the "mutual exclusion problem" (Dijkstra [1968]), the "first and second reader-writer problems" (Courtois, Heymans, Parnas [1971], Brinch Hansen [1972, 1972a]), and several buffer problems (Habermann [1972], Dijkstra [1972]). The basic assumption of their research is that the capabilities of a synchronization primitive can be determined by trying to solve problems that are found in operating systems. One of the dangers of this approach is that our inability to "solve" a problem can stem from two causes: either our inability to find a solution or the non-existence of a solution. In fact, several people have asserted that the "first reader-writer problem" was not "solvable" by PV; Courtois, Heymans, Parnas [1971] have shown that this "synchronization problem" is "solvable" by PV. Another danger of this approach is that we can never be certain that each researcher is using the same notions of "solve" and "synchronization problem". The controversy between Courtois, Heymans, Parnas and Brinch Hansen over the "second reader-writer problem" can be attributed to the informal nature of their research. (Courtois, Heymans, Parnas [1972], Parnas [private communication], Brinch Hansen [1973]).

We study the question (*) from a formal viewpoint. We will present a formal model of the process concept, and then use this model to compare several different synchronization primitives. We are able to prove that there <u>are</u> differences between several synchronization primitives. Some of these differences have - on a very informal level - been noticed before, i.e., Wodon [1972] and Parnas [private communication]. In addition, our model of the process concept separates the notion of process from the notion of scheduler, this separation allows us to unravel the controversy between Brinch Hansen and Courtois, Heymans, Parnas and to state formally the theorem in Habermann [1972].

We study four synchronization primitives: PV, which is due to Dijkstra [1968, 1968a]; PVchunk, a generalization of PV due to Vantilborgh and van Lamsweerde [1972]; PV multiple, a generalization PV due to Patil [1971] and Dijkstra [unpublished]; and up/down, a generalization of PV due to Wodon [1972]. Although we compare only these synchronization primitives, our general methods can be used to compare other synchronization primitives such as block/wakeup (Saltzer [1966]). In our definitions of these synchronization primitives we explicitly allow <u>conditional branches</u>. Therefore, our results are not related to the results of Patil [1971], nor are they related to the results of Parnas [1972].

We define a relation "→" between synchronization primitives. Informally, x → y means that the synchronization primitive y cannot "solve some synchronization problem" that x can. Our principal results are displayed in Figure 1, where an arrow from x to y means that x → y.

$$\begin{tikzcd}
\text{PVchunk} \arrow[r, bend left] \arrow[dr] & \text{up/down} \arrow[l] \arrow[r] \arrow[d] & \text{PVmultiple} \arrow[l, bend right] \arrow[ll, bend right=40] \arrow[dl] \\
& \text{PV} &
\end{tikzcd}$$

FIGURE 1.

Each of the arrows in Figure 1 can be traced to a particular synchronization problem.

1. up/down → PVchunk, up/down → PVmultiple, up/down → PV. These relations are true because up/down can solve the second reader-writer problem while PVchunk, PVmultiple, and PV cannot. These results do not contradict the "solutions" found in Courtois, Heymans, Parnas [1971] and in Brinch Hansen [1972,1972a]; the notions of solve used by these researchers are weaker than the notion used here. One of our contributions is that we can define several notions of "solve" - all in a precise way.

2. PVmultiple → PVchunk, PVmultiple → PV. These relations are true because PVmultiple can solve the "Five Dining Philosophers" (Dijkstra [1971]) while PVchunk and PV cannot. As in (1) these results do not contradict the "solution" presented by Dijkstra.

3. PVmultiple → up/down, PVchunk → up/down, PVchunk → PV. These relations are true because PVmultiple and PVchunk can "solve" the "first reader-writer problem with a bound on the number of readers that can be reading at one time" while up/down and PV cannot. This is a new "synchronization problem". Although, we can trace these results to particular synchronization problems they are indicative of general phenomena.

This paper is organized into eight sections. In section 2 our model of the process concept is defined. The main result of this section is that processes that use synchronization primitives for inter-process communication satisfy four basic properties: these properties are not dependent on which synchronization primitive the process uses. In section 3 the concept of scheduler is defined. We use the notion of scheduler to state the theorem of Habermann [1972]. In section 4 we define several relations between processes. In defining these relations we formalize the concept of a "synchronization problem". In section 5 and 6 a structure theorem is stated and proved - using just the four basic properties of section 2 - that reduces the question

$$\text{is } x \rightarrow y?$$

to a combinatorial question. In section 7 we use this reduction to compare the synchronization primitives PV, PVchunk, PVmultiple, and up/down. In particular, the results displayed in Figure 1 are proved in this section. In section 8 a summary and a list of open problems are presented.

2. PROCESSES

We now define a general model of the process concept; later on we add additional structure to this model. The resulting model - a C process - is then studied in detail, and several basic properties of C processes are proved.

2.1 A GENERAL MODEL OF PROCESSES

1. Definition. $\mathcal{P} = \langle A,\mathfrak{D},w\rangle$ is a process if A is a set of functions from $\mathfrak{D}$ to $\mathfrak{D}$ and w is in $\mathfrak{D}$. The elements of A are the actions of the process $\mathcal{P}$. The set $\mathfrak{D}$ is the data structure of the process; the element w is the initial data structure element of the process. We will use $\mathcal{P}$ and $\mathcal{Q}$ to denote processes, and we will use f,g,h to denote actions.

Actions - as defined in Definition 1 - are arbitrary functions from $\mathfrak{D}$ to $\mathfrak{D}$. The data structure of a process - as defined in Definition 1 - is an arbitrary set. Later we will place additional restrictions on both the actions and the data structure. Note, $\mathcal{P}$ will usually be a "parallel process", i.e., our concept of a process is what is usually called a set of "sequential processes".

2.2 NOTATION

We will now introduce a shorthand for describing actions. The notion is just a method of describing functions in a convenient and readable way. The theory is unchanged if we change notation. Let $\mathfrak{D}$ be a set, and let '$(x_1,\ldots,x_k)$' be a typical element of $\mathfrak{D}$. The notation

$$\underline{\text{when}}\ B(x_1,\ldots,x_k)\ \underline{\text{do}}\ x_1 \leftarrow f_1(x_1,\ldots,x_k);\ldots;x_k \leftarrow f_k(x_1,\ldots,x_k)$$

where B is a predicate and each f_i $(i=1,\ldots,k)$ is a function, denotes the function g, from $\mathfrak{D}$ to $\mathfrak{D}$, defined by

(1) if $B(y_1,\ldots,y_k)$ is true, then $g(y_1,\ldots,y_k) = (f_1(y_1,\ldots,y_k),\ldots, f_k(y_1,\ldots,y_k))$

(2) if $B(y_1,\ldots,y_k)$ is false, then $g(y_1,\ldots,y_k) = (y_1,\ldots,y_k)$.

<u>Caution</u>, we consider that the assignments after the 'do' are all done "simultaneously" and not "sequentially". For example, suppose that '(A,B)' is a typical element of the set $\{1,2,3\} \times \{1,2,3\}$, and that $f = \underline{\text{when}}\ A < B$ do $A \leftarrow 1$; $B \leftarrow A$. Then $f(2,3)$ is equal to $(1,2)$. We will delete assignments of the form '$x \leftarrow x$'. For instance, we will shorten <u>when</u> $L = 0$ <u>do</u> $L \leftarrow 1$; $B \leftarrow B$; $C \leftarrow C$ to <u>when</u> $L = 0$ <u>do</u> $L \leftarrow 1$.

We use small Greek letters to denote <u>finite</u> or <u>infinite</u> sequences; the empty or null sequence will be denoted by Λ. If α is a finite sequence and β is a finite or infinite sequence, then the sequence formed by concatenating α and β is denoted by $\alpha\beta$. Define $\alpha \le \beta$ if for some δ, $\alpha\delta = \beta$. Note $\alpha \le \beta$ means that α is an initial part of β. The ith element in the sequence α is denoted by α_i; the first element in the sequence α is α_1, provided $\alpha \ne \Lambda$. The length of the finite sequence α is denoted by length (α). Note, if length $(\alpha) = n$ and $n > 1$, then $\alpha = \alpha_1\ldots\alpha_n$.

Let '$(x_1,\ldots,x_k)$' be a typical element of the set $\mathfrak{D}$. We will use '$x_i[z]$' to denote a_i where $z = (a_1,\ldots,a_k)$.

2.3 TIMINGS

2. Definition. A timing for the process ρ is a finite or infinite sequence of actions of the process ρ.

3. Definition. Suppose that $\rho = \langle A,\mathfrak{D},w\rangle$ is a process. We will define a function $\underline{\text{value}_\rho}$ as follows:

(1) $\underline{\text{value}_\rho}(\Lambda) = w$.

(2) If α is a finite timing for ρ and f is an action in ρ, then $\text{value}_\rho(\alpha f) = f[\text{value}_\rho(\alpha)]$.

When there is no confusion we will delete the subscript 'ρ'. The function value_ρ maps timings to elements in $\mathfrak{D}$. For example, value(fg) =g(f(w)).

We can think of the actions in a given timing as being "executed" in that order. Different timings correspond to "executing" the actions in a different order. Thus, value($\alpha_1\cdots\alpha_n$) is the result of "executing" the actions in the order $\alpha_1,\ldots,\alpha_n$. Note, the result of "executing" Λ is the initial data structure element w.

Any sequence of actions is a timing; hence, in a given process, we may consider some timings as "uninteresting". In the next section we will study the concept of scheduler; this concept allows us to consider certain special sets of timings.

2.4 FEATURES OF PROCESSES USED IN THE SYNCHRONIZATION AREA

One of our goals is to be able to model the processes used in the synchronization area, such as PV processes. In order to achieve this goal we will add additional structure to the model of the process concept as defined in Section 2.1; this structure must reflect the features of the processes used in the synchronization area. Therefore, we will examine the principal features of these processes.

The data structure of the processes in the synchronization area has three basic components.

(1) program counters. These variables are usually implicit in the syntactic representation of a process. For instance, the <u>par begin</u>-<u>par end</u> and <u>co begin</u>-<u>co end</u> notation of Dijkstra [1968] and Hoare [1971] are used to implicitly define several distinct program counters. We uniformly use 'L' with or without subscripts to denote a program counter.

(2) program variables. These variables are usually explicit in the syntactic representation of a process. For example, the processes in Courtois, Heymans, Parnas [1971] and Brinch Hansen [1972] use program variables.

(3) semaphores. These variables are usually explicit in the syntactic representation of a process. They are used exclusively in the inter-process communication of the process.

The actions of the processes in the synchronization area have several important features.

(4) The actions of a process are divided into disjoint sets. Each of these sets is the collection of all actions that use a given program counter.

(5) The actions of a process are also classified into two groups. The first group consists of the actions that handle the inter-process communication of the process. These actions are usually called synchronizing primitives. They are the only actions that can test or set the semaphore variables. The second group consists of the remaining actions. They are the only actions that can test or set the program variables.

(6) The synchronizing primitives are usually actions of the form

(a) $\underline{\text{when}}\ L = \text{address} \wedge p(E)\ \underline{\text{do}}\ L \leftarrow \text{new address};\ E \leftarrow q(E)$

where L is a program counter, E is the part of the data structure that contains the semaphores, p is a predicate, and q is a function. The pairs (p,q) we allow in (a) <u>distinguish the different synchronizing primitives</u>. For example, the pair $(x > 0,\ y \leftarrow y-1)$ is not allowed in PV processes; it <u>is</u> allowed in up/down processes (Wodon [1972]).

(7) The non-synchronizing primitives are usually actions of the form

<u>when</u> L = address <u>do</u> L ← b(D); D ← t(D)

where L is a program counter, D is the part of the data structure that contains the program variables, b is a function, and t is a function. Since b is a function, we allow these actions to branch, i.e.,

<u>when</u> L = 1 <u>do</u> L ← <u>if</u> x = 0 <u>then</u> 2 <u>else</u> 3

is an acceptable action from this group. The actions in this group also satisfy an additional requirement. Suppose that f and g are actions in this group, f and g use different program counters, and f and g "share a variable". (In our model, f and g share a variable iff for some x in the data structure, $f(g(x)) \neq g(f(x))$.) Then the usual definition of processes in the synchronization area forces f and g to be "enclosed in critical sections". (In our model f and g are enclosed in critical sections iff for all finite timings α, if $f(\text{value}(\alpha)) \neq \text{value}(\alpha)$, then $g(\text{value}(\alpha)) = \text{value}(\alpha)$.)

2.5 C PROCESSES

4. <u>Definition</u>. A set of pairs C is a <u>predicate system</u> provided, for each (p,q) in C, there exists a set E such that p is a predicate on E and q is a function from E to E. We will use C to denote a predicate system.

5. Definition. Let C be a predicate system. Also let $\mathcal{P} = \langle A, \mathfrak{D}, w \rangle$ be a process, and let '$(L_1, \ldots, L_n, D, E)$' be a typical element of $\mathfrak{D}$. The process $\mathcal{P}$ is a C process if there is a predicate synchronizer on A, a function program-counter from A to $\{1, \ldots, n\}$, and a function address with domain A such that

(1) If not synchronizer(f), then there exists functions b and t such that

(a) $f = \underline{\text{when}}\ L_k = \text{address}(f)\ \underline{\text{do}}\ L_k \leftarrow b(D);\ D \leftarrow t(D)$ where k = program-counter(f),

(b) for all x in $\mathfrak{D}$, $b(D[x]) \neq$ address (f).

(2) If synchronizer(f), then there exists a $(p,q) \in C$, called the pair of f, and a y such that

(a) $f = \underline{\text{when}}\ L_k = \text{address}(f) \wedge p(E)\ \underline{\text{do}}\ L_k \leftarrow y;\ E \leftarrow q(E)$ where k = program-counter(f).

(b) $y \neq$ address(f).

(3) If not synchronizer(f), program-counter(f) $\neq$ program-counter(g), and α is a finite timing, then

$$f(g(\text{value}(\alpha))) = g(f(\text{value}(\alpha))).$$

(4) If address(f) = address(g) and program-counter(f) = program-counter(g), then f = g.

The definition of C processes is motivated by our desire to be able to model the processes used in the synchronization area. We will now relate the definition of C processes to the discussion in Section 2.4. The data structure of a C process is composed of three parts.

(1) program counters. These variables are $L_1,\ldots,L_n$.

(2) program variables. The program variable is D. Since we make no restrictions on the range of the variable D, there is no loss in generality in considering all the program variables as one composite variable.

(3) semaphores. The semaphores are considered as one composite variable E. As in (2), there is no loss in generality.

Thus, the data structure of a C process corresponds to the data structure of the processes used in the synchronization area. We now focus our attention on the actions of a C process.

(4) The actions that use program counter L_k are $\{f \mid \text{program-counter}(f) = k\}$. Each action is in exactly one of these sets.

(5) The predicate synchronizer classifies the actions into two groups. If synchronizer(f), then f can test or set E, but it cannot test or set D. On the other hand, if not synchronizer(f), then f can test or set D, but it cannot test or set E.

(6) If synchronizer(f), then f is equal to

$$\underline{\text{when}}\ L_k = \text{address}(f) \land p(E)\ \underline{\text{do}}\ L_k \leftarrow y;\ E \leftarrow q(E)$$

where k = program-counter(f) and $(p,q) \in C$. The pairs allowed in C determine the kinds of "synchronizers" that are allowed. In the next section we will define several predicate systems C.

(7) If not synchronizer(f), then f is equal to

$$\underline{\text{when}}\ L_k = \text{address}(f)\ \underline{\text{do}}\ L_k \leftarrow b(D);\ D \leftarrow t(D)$$

where k = program-counter(f). The additional requirement stated in part (7) of Section 2.4 is reflected in condition (3) of the definition of a C process. In fact, we can prove the following

(a) Suppose that $\mathcal{P}$ satisfies the definition of a C process except that condition (3) is replaced by: if f and g share a variable, then they are enclosed in critical sections. Then condition (3) is true.

For suppose that condition (3) is false; moreover, suppose that not synchronizer(f), program-counter(f) $\neq$ program-counter(g), and $f(g(\text{value}(\alpha))) \neq g(f(\text{value}(\alpha)))$. If synchronizer(g), then $f(g(\text{value}(\alpha))) = g(f(\text{value}(\alpha)))$; hence, not synchronizer(g). By assumption, f and g are enclosed in critical sections. In our model this is equivalent to

(b) if $f(\text{value}(\beta)) \neq \text{value}(\beta)$, then $g(\text{value}(\beta)) = \text{value}(\beta)$.

Informally, if f can "change the data structure element that results after executing β", then g cannot "change the data structure element that results after executing β". The contrapositive of (b) is

(b') if $g(\text{value}(\beta)) \neq \text{value}(\beta)$, then $f(\text{value}(\beta)) = \text{value}(\beta)$.

Thus, by symmetry, we can assume that $g(\text{value}(\alpha)) = \text{value}(\alpha)$. Then $f(g(\text{value}(\alpha))) = f(\text{value}(\alpha))$. Since f cannot change L_k where $k = \text{program-counter}(g)$, $g(\text{value}(\alpha f)) = \text{value}(\alpha f)$. Therefore, $g(f(\text{value}(\alpha))) = f(\text{value}(\alpha))$; hence, $f(g(\text{value}(\alpha))) = g(f(\text{value}(\alpha)))$. This is a contradiction, and hence (a) is true. □

Thus, the actions of C processes have the basic features as outlined in Section 2.4. Additional evidence that C processes are a reasonable model of synchronization processes is contained in Theorem 12.

2.6 EXAMPLES OF PREDICATE SYSTEMS

6. Definition. The pair (p,q) is in the PV predicate systems iff there are non-negative integers i and k such that p is a predicate on $E = \mathbb{Z}^k$ and either ($\mathbb{Z}$ is the set of integers)

(1) $p(x_1 \ldots x_k)$ is always true, and $q(x_1 \ldots x_k) = (x_1 \ldots x_{i-1} y\ x_{i+1} \ldots x_k)$ where $y = x_i + 1$, or

(2) $p(x_1 \ldots x_k)$ is true iff $x_i > 0$, and $q(x_1 \ldots x_k) = (x_1 \ldots x_{i-1} y\ x_{i+1} \ldots x_k)$ where $y = x_i - 1$.

In case (1), we say (p,q) is a <u>$V(x_i)$</u>; in case (2), we say (p,q) is a <u>$P(x_i)$</u>.

Suppose that f is an action in a PV process and that synchronizer(f) is true. If the pair of f is a $V(x_i)$, then f is of the form

$$\underline{\text{when}}\ L_k = \text{address}(f)\ \underline{\text{do}}\ L_k \leftarrow z;\ x_i \leftarrow x_i + 1.$$

On the other hand, if the pair of f is a $P(x_i)$, then f is of the form

$$\underline{\text{when}}\ L_k = \text{address}(f) \wedge x_i > 0\ \underline{\text{do}}\ L_k \leftarrow z;\ x_i \leftarrow x_i - 1.$$

The PV predicate system is essentially due to Dijkstra [1968]. Since he defines PV on an informal level, we cannot prove that our notion of PV processes corresponds exactly to his. However, we feel that our definition is a reasonable one.

7. <u>Definition</u>. The pair (p,q) is in the <u>up/down</u> predicate system iff there are non-negative integers i,k and a subset F of $\{1,\ldots,k\}$ such that p is a predicate on $E = \mathbb{Z}^k$ and either

(1) $p(x_1\ldots x_k)$ is $\sum_{j\in F} x_j \geq 0$, and $q(x_1\ldots x_k) = (x_1\ldots x_{i-1}\,y\,x_{i+1}\ldots x_k)$ where $y = x_i + 1$, or

(2) $p(x_1\ldots x_k)$ is $\sum_{j\in F} x_j \geq 0$, and $q(x_1\ldots x_k) = (x_1\ldots x_{i-1}\,y\,x_{i+1}\ldots x_k)$ where $y = x_i - 1$.

In case (1), we say (p,q) is a <u>$\{x_n | n \in F\}$: up(x_i)</u>; in case (2), we say (p,q) is a <u>$\{x_n | n \in F\}$: down (x_i)</u>.

Suppose that f is an action in an up/down process and that synchronizer(f) is true. If the pair of f is a $\{x_1,x_3\}$: up(x_4), then f is of the form

$$\underline{\text{when}}\ L_k = \text{address}(f) \wedge x_1 + x_3 \geq 0\ \underline{\text{do}}\ L_k \leftarrow z;\ x_4 \leftarrow x_4 + 1.$$

As another example, if the pair of f is a $\{x_1\}$: down(x_2), then f is of the form

$$\underline{\text{when}}\ L_k = \text{address}(f) \wedge x_1 \geq 0\ \underline{\text{do}}\ L_k \leftarrow z;\ x_2 \leftarrow x_2 - 1.$$

The predicate system up/down is essentially due to Wodon [1972]. He defines up/down on an informal level; hence, we cannot prove that our definition corresponds exactly to his definition. Indeed, we do not allow actions of the form

$$\underline{\text{when}}\ L_k = \text{address}(f) \wedge \sum_{j \in F} x_j \geq \underline{\text{do}}\ L_k \leftarrow b(D);\ D \leftarrow t(D)$$

while he does. An action of this form violates what we stated in part 5 of Section 2.4: these actions can test semaphores and test and set program variables. Also, these actions do not satisfy the main theorem of this section - Theorem 12. For these reasons we will not change our definition of up/down processes.

8. Definition. The pair (p,q) is in the PVchunk predicate system iff there are non-negative integers i,k,m such that p is a predicate on $E = \mathbb{Z}^k$ and either

(1) $p(x_1 \ldots x_k)$ is always true, and $q(x_1 \ldots x_k) = (x_1 \ldots x_{i-1} y\ x_{i+1} \ldots x_k)$ where $y = x_i + m$, or

(2) $p(x_1 \ldots x_k)$ is true iff $x_i \geq m$, and $q(x_1 \ldots x_k) = (x_1 \ldots x_{i-1} y\ x_{i+1} \ldots x_k)$ where $y = x_i - m$.

In case (1), we say that (p,q) is a $\underline{V(x_i \text{ with amount } m)}$; in case (2), we say that (p,q) is a <u>$P(x_i$ with amount m)</u>.

Suppose that f is an action in a PVchunk process and that synchronizer(f) is true. For example, if the pair of f is a $V(x_3$ with amount 5), then f is of the form

$$\underline{\text{when}}\ L_k = \text{address}(f)\ \underline{\text{do}}\ L_k \leftarrow z;\ x_3 \leftarrow x_3 + 5.$$

As another example, if the pair of f is a $P(x_2$ with amount 3), then f is of the form

$$\underline{\text{when}}\ L_k = \text{address}(f) \wedge x_2 \geq 3\ \underline{\text{do}}\ L_k \leftarrow z;\ x_2 \leftarrow x_2 - 3.$$

The predicate system PVchunk is essentially due to Vantilborgh and van Lamsweerde [1972]. Again we can only assert that out PVchunk processes are a reasonable model of their processes.

9. <u>Definition</u>. The pair (p,q) is in the <u>PVmultiple</u> predicate system iff there is a non-negative integer k and a subset F of $\{1,\ldots,k\}$ such that p is a predicate on $E = \mathbb{Z}^k$ and either

(1) $p(x_1\ldots x_k)$ is always true, and $q(x_1\ldots x_k) = (y_1\ldots y_k)$ where $y_i = \underline{\text{if}}\ i \in F\ \underline{\text{then}}\ x_i + 1\ \underline{\text{else}}\ x_i$, or

(2) $p(x_1\ldots x_k)$ is true iff [for $i \in F$, $x_i \geq 1$], and $q(x_1\ldots x_k) = (y_1\ldots y_k)$ where $y_i = \underline{\text{if}}\ i \in F\ \underline{\text{then}}\ x_i - 1\ \underline{\text{else}}\ x_i$.

In case (1), we say that (p,q) is a <u>$V(\{x_n | n \in F\})$</u>; in case (2), we say that (p,q) is a <u>$P(\{x_n | n \in F\})$</u>.

Suppose that f is an action in a PVmultiple process and that synchronizer(f) is true. For example, if the pair of f is $V(\{x_1,x_2\})$, then f is of the form

$$\underline{\text{when}}\ L_k = \text{address}(f)\ \underline{\text{do}}\ L_k \leftarrow z;\ x_1 \leftarrow x_1 + 1;\ x_2 \leftarrow x_2 + 1.$$

As another example, if the pair of f is a $P(\{x_1,x_3\})$, then f is of the form

$$\underline{\text{when}}\ L_k = \text{address}(f) \wedge x_1 \geq 1 \wedge x_3 \geq 1\ \underline{\text{do}}\ L_k \leftarrow z;\ x_1 \leftarrow x_1 - 1;$$
$$x_3 \leftarrow x_3 - 1.$$

The predicate system PVmultiple is essentially due to Patil [1971] and Dijkstra [unpublished]. Again we can only assert that our PVmultiple processes are a reasonable model of the processes informally defined by Patil and Dijkstra.

The predicate system PV is a subset of both the predicate system PVchunk and the predicate system PVmultiple. Suppose that (p,q) is a $V(x_i)$. Then (p,q) is a $V(x_i$ with amount 1), and (p,q) is a $V(\{x_i\})$. Thus, (p,q) is in PVchunk and PVmultiple. On the other hand, suppose that (p,q) is a $P(x_i)$. Then (p,q) is a $P(x_i$ with amount 1), and (p,q) is a $P(\{x_i\})$. Therefore, PV $\subseteq$ PVchunk and PV $\subseteq$ PVmultiple. As a consequence,

the set of PV processes $\subseteq$ the set of PVchunk processes and
the set of PV processes $\subseteq$ the set of PVmultiple processes.

The relationship between the predicate system PV and the predicate system up/down is complex. Clearly, PV is not a subset of up/down: the pair $(x > 0,\ x \leftarrow x - 1)$ is not in up/down. However, as Wodon [1972] correctly states: each PV process is "equivalent or isomorphic" to an up/down process. We shall state a relation between PV processes and up/down processes in section 4; in this section we study relations between processes.

Many of the processes used in the synchronization area can be considered as C processes, for a suitable predicate system C. These processes include: fork-join processes (Dennis and Van Horn [1966]), block-wakeup (Saltzer [1966]), conditional critical sections (Brinch Hansen [1972]), and certain Petri Nets (Patil [1971]).

2.7 REPRESENTATION OF C PROCESSES

We will use several conventions when we define C processes. These conventions are best explained by an example. Process EX1 - a PV process - is defined in Figure 2.

```
program counter $L_1,L_2$; (initial value 1)
integer x,y; (initial value 0)
semaphore S; (initial value 1)
SUBPROCESS-1
(1)  when $L_1 = 1 \wedge S \geq 1$ do $L_1 \leftarrow 2$; $S \leftarrow S - 1$
(2)  when $L_1 = 2$               do $L_1 \leftarrow 3$; $x \leftarrow 1$
(3)  when $L_1 = 3$               do $L_1 \leftarrow 4$; $S \leftarrow S + 1$
SUBPROCESS-2
(4)  when $L_2 = 1 \wedge S \geq 1$ do $L_2 \leftarrow 2$; $S \leftarrow S - 1$
(5)  when $L_2 = 2$               do $L_2 \leftarrow 3$; $y \leftarrow x$
(6)  when $L_2 = 3$               do $L_2 \leftarrow 4$; $S \leftarrow S + 1$
```

FIGURE 2. Process EX1: formal representations

Several conventions have been used in Figure 2. First, the data structure of EX1 has been defined in an implicit way. The data structure of EX1 is

$$\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}^2 \times \mathbb{Z}$$

Let '(L_1, L_2, D, E)' be a typical element of the data structure of EX1. Then L_1, L_2, E range over integers while D ranges over pairs of integers. D ranges over pairs of integers because we consider the program variables x,y as one composite variable D. Second, the initial data structure element of EX1 has been defined in an implicit way. The initial data structure element is (1,1,(0,0),1). Third, the actions of EX 1 have been numbered for future reference. Thus, action 3 is

$$\underline{\text{when}}\ L_1 = 3\ \underline{\text{do}}\ L_1 \leftarrow 4;\ S \leftarrow S + 1.$$

Fourth, the sets of actions that use each program counter have been given names. Thus, SUBPROCESS-2 contains the actions 4,5,6.

These conventions will be used whenever we define a C process. Although the conventions are stated informally, the translation to a precisely defined C process should be clear.

The syntactic representation of process EX1 in the style of Courtois, Heymans, Parnas [1971] is displayed in Figure 3. Figure 3 deletes all references to program counters. Moreover, if synchronizer(f), then action f is represented by just the pair of f. Thus,

$$\underline{\text{when}}\ L_1 = 1 \wedge S \geq 1\ \underline{\text{do}}\ L_1 \leftarrow 2;\ S \leftarrow S - 1$$

is represented by P(S). Although process EX1 is a PV process, we can also consider it as a PVchunk process. In this case, action 1 is represented by P(S with amount 1).

```
integer  x,y; (initial value 0)
semaphore S;  (initial value 1)

SUPROCESS-1                        SUBPROCESS-2

(1)  P(S);                        (4)  P(S);
(2)  x := 1;                      (5)  y := x;
(3)  V(S);                        (6)  V(S);
```

FIGURE 3. Process EX1: informal representation

2.8 SUBPROCESS, READY-SET, POINTER-SET

We will now define three notions: subprocess, ready-set, pointer-set. These notions satisfy two requirements.

(1) They are rich enough to allow us to express most of our ideas about C processes.

(2) They suppress a great many of the unnecessary or uninteresting details of C processes.

10. Definition. Suppose that $\wp = \langle A, \mathfrak{J}, w\rangle$ is a C process, and let '$(L_1,\ldots,L_n,D,E)$' be a typical element of $\mathfrak{J}$.

(1) For actions f and g, $\underline{\text{subprocess}_{\wp}}(f,g)$ iff program-counter(f) = program-counter(g). Subprocess is an equivalence relation on the actions of $\wp$. Let SUBPROCESS-i denote the i^{th} equivalence class of the relation subprocess, i.e., $\{f \mid \text{program-counter}(f) = i\}$.

(2) An action f is in $\underline{\text{ready-set}_{\wp}}(\alpha)$ where α is a finite timing iff

$$f[\text{value }(\alpha)] \neq \text{value}(\alpha).$$

(3) An action f is in $\underline{\text{pointer-set}_{\rho}}(\alpha)$ where α is a finite timing iff

$$L_k[\text{value}(\alpha)] = \text{address}(f)$$

where k = program-counter(f).

In each of these notions we will drop the subscript 'ρ' when this will cause no confusion.

The notions subprocess, ready-set, pointer-set have intuitive meanings. Suppose that $\rho = \langle A, \mathfrak{D}, w\rangle$ is a C process, and let '$(L_1,\ldots,L_n,D,E)$' be a typical element in $\mathfrak{D}$. Recall that $L_1,\ldots,L_n$ are program counters.

(1) Subprocess divides A into the disjoint sets SUBPROCESS-1,..., SUBPROCESS-n. SUBPROCESS-i is the set of actions that use the program counter L_i.

(2) Ready-set(α) is the set of actions that can "change the data structure element that results after executing the timing α". For example, if f is

$$\underline{\text{when}}\ L_3 = 2 \wedge S \geq 3\ \underline{\text{do}}\ L_3 \leftarrow 3;\ S \leftarrow S - 3,$$

then f is in the ready-set(α) iff

$$L_3[\text{value}(\alpha)] = 2 \quad \text{and}$$
$$S[\text{value}(\alpha)] \geq 3.$$

As another example, suppose that f is a "nop or null statement", i.e.,

$$f = \underline{\text{when}}\ L_k = \text{address}(f)\ \underline{\text{do}}\ L_k \leftarrow z.$$

Then f is in ready-set(α) iff L_k[value(α)] is equal to address(f). This example is interesting because f never changes any program variables or semaphores. However, in our model f can change the data structure; for our concept of data structure includes the program counters.

(3) Pointer-set(α) is the set of actions f such that the program counter of f "points to f". For example, if f is

$$\underline{\text{when}}\ L_3 = 2 \wedge S \geq 3\ \underline{\text{do}}\ L_3 \leftarrow 3;\ S \leftarrow S - 3,$$

then f is in the pointer-set(α) iff

$$L_3[\text{value}(\alpha)] = 2.$$

The notions subprocess, ready-set, pointer-set are sufficient to express the basic concepts of the synchronization area. For instance, these notions can express the concept of "PV release mechanism" as defined by Dijkstra [1968] and the concept of "fairness"; this is demonstrated in Section 3. In addition we will show, in Section 4, how we can use these three notions to express the concepts of "safe" and "deadlock free". One of the contributions of our model is the understanding that the notions subprocess, ready-set, pointer-set are sufficient to express the central features of the synchronization area.

2.9 BASIC PROPERTIES OF C PROCESSES

We will now state and then prove four basic properties shared by all C processes. These properties do not depend on the predicate system C. The proof of these properties should help motivate each decision that has been made in the definition of C processes.

11. Definition. The timing α is active in a C process provided, if $\beta f \leq \alpha$, then f is in ready-set (β).

The timing $\alpha_1 \ldots \alpha_n$ is active if and only if for each i, α_i is in ready-set $(\alpha_1 \ldots \alpha_{i-1})$. Informally, the timing $\alpha_1 \ldots \alpha_n$ is active if and only if each action α_i can change the data structure element that results "after executing $\alpha_1 \ldots \alpha_{i-1}$".

(I) For any finite timing α, ready-set$(\alpha) \subseteq$ pointer-set(α).

This property is an immediate consequence of the defintion of a C process. Informally, if the action f can "run", then the program counter of f must "point" to f.

(II) For any finite timing α, the pointer-set(α) has at most one action from each SUBPROCESS-i.

This property is a direct consequence of condition 4 of Definition 5. For suppose that f and g are in pointer-set(α) and subprocess (f,g). Then by definition of subprocess, program-counter(f) = program-counter(g). By the definition of pointer-set,

$$L_{\text{program-counter}(f)}[\text{value}(\alpha)] = \text{address}(f) \text{ and}$$
$$L_{\text{program-counter}(g)}[\text{value}(\alpha)] = \text{address}(g).$$

Thus, address(f) = address(g), and hence by condition (4) of Definition 5, f = g.

We have excluded actions of the kind

$$f = \underline{\text{when}}\ L_1 = 1 \wedge S > 0\ \underline{\text{do}}\ L_1 \leftarrow 2$$
$$g = \underline{\text{when}}\ L_1 = 1 \wedge S = 0\ \underline{\text{do}}\ L_1 \leftarrow 3$$

from C processes. We have to express f and g as

$$\underline{\text{when}}\ L_1 = 1\ \underline{\text{do}}\ L_1 \leftarrow \underline{\text{if}}\ S = 0\ \underline{\text{then}}\ 2\ \underline{\text{else}}\ 3.$$

(III) If $\alpha f\beta$ and $\alpha\beta$ are active finite timings, then

$$\text{pointer-set}(\alpha f\beta) \cap \text{SUBPROCESS-i} = \text{pointer-set}(\alpha\beta) \cap \text{SUBPROCESS-i}$$

provided f is not in SUBPROCESS-i. [$\cap$ is set intersection]

This property is non-trivial; it will be proved in detail in Theorem 12. This property allows us to compare the pointer-sets of distinct active timings. It states that the program counters are in some sense "local". Actions from SUBPROCESS-i cannot change the program counters of SUBPROCESS-j, provided $i \neq j$.

(IV) If $\alpha f\beta\delta$ and $\alpha\beta f\delta$ are active finite timings, then

$$\text{pointer-set}(\alpha f\beta\delta) = \text{pointer-set}(\alpha\beta f\delta).$$

This property is also non-trivial; it will be proved in detail in Theorem 12. This property also allows us to compare the pointer-sets of distinct active timings. It states that the pointer-set has a certain kind of "order invariance".

12. Theorem. Every C process satisfies properties I-IV.

Proof. Let $\mathcal{P} = \langle A, \mathcal{D}, w\rangle$ be a C process. Let '$(L_1, \ldots, L_n, D, E)$' be a typical element in $\mathcal{D}$.

Lemma 1. Suppose that x is in $\mathcal{D}$, f is an action, and $1 \leq k \leq n$. Then

(1) if program-counter(f) $\neq$ k, then $L_k[x] = L_k[f(x)]$

(2) if $f(x) \neq x$, then $L_{\text{program-counter}(f)}[x] = \text{address}(f)$.

Proof of Lemma. This is an immediate consequence of the definition of a C process. □

Lemma 2. Suppose that x is in $\mathcal{D}$, y is in $\mathcal{D}$, and $1 \leq k \leq n$. Also suppose that $D[x] = D[y]$ and $L_k[x] = L_k[y]$. Then if $f(x) \neq x$ and $f(y) \neq y$ where f is an action, then $D[f(x)] = D[f(y)]$ and $L_k[f(x)] = L_k[f(y)]$.

Proof of Lemma. This is also an immediate consequence of the definition of a C process. □

Lemma 3. Suppose that $\alpha\beta$ and $\lambda\beta$ are finite active timings. Also suppose that $D[\text{value}(\alpha)] = D[\text{value}(\lambda)]$ and $L_k[\text{value}(\alpha)] = L_k[\text{value}(\lambda)]$. Then $D[\text{value}(\alpha\beta)] = D[\text{value}(\lambda\beta)]$ and $L_k[\text{value}(\alpha\beta)] = L_k[\text{value}(\lambda\beta)]$.

Proof of Lemma. This follows by repeated applications of Lemma 2. □

Lemma 4. Suppose that $\alpha f\beta$ and $\alpha\beta$ are finite active timings. Then for $1 \leq i \leq \text{length}(\beta)$, program-counter(f) $\neq$ program-counter(β_i).

Proof of Lemma. Let k = program-counter(f). Assume that the lemma is false, and let i be the least integer such that k = program-counter(β_i).

By Lemma 1, part 2, since αf and $\alpha\beta_1\ldots\beta_{i-1}\beta_i$ are active,

$$L_k[\text{value}(\alpha)] = \text{address}(f) \text{ and}$$
$$L_k[\text{value}(\alpha\beta_1\ldots\beta_{i-1})] = \text{address}(\beta_i).$$

By Lemma 1, part 1 and the definition of i, an inductive argument shows that

$$L_k[\text{value}(\alpha)] = L_k[\text{value}(\alpha\beta_1\ldots\beta_{i-1})].$$

Hence, address(f) = address(β_i). Therefore, by the definition of a C process, $f = \beta_i$. By the definition of a C process (1b and 2b),

$$L_k[\text{value}(\alpha f)] \neq \text{address}(f).$$

By Lemma 1, part 1 and the definition of i,

$$L_k[\text{value}(\alpha f\beta_1\ldots\beta_{i-1})] = L_k[\text{value}(\alpha f)] \neq \text{address (f)}.$$

But $\alpha f\beta_1\ldots\beta_{i-1}\beta_i$ is active, so by Lemma 1, part 2,

$$L_k[\text{value}(\alpha f\beta_1\ldots\beta_{i-1})] = \text{address}(\beta_i).$$

This is a contradiction, for $f = \beta_i$. □

By what we have already shown, we need only prove that a C process satisfies properties III and IV.

We will now show that property III is satisfied by P. Suppose that $\alpha f\beta$ and $\alpha\beta$ are finite active timings. In order to prove property III, it is sufficient to prove

(3) for $i \neq k$, $L_i[\text{value}(\alpha f\beta)] = L_i[\text{value}(\alpha\beta)]$

where k = program-counter(f). For suppose that (3) is true. Let g be an action such that not subprocess (f,g). Now g is in pointer-set($\alpha f \beta$) iff

$$L_j[\text{value}(\alpha f \beta)] = \text{address}(g)$$

where j = program-counter(g). Also g is in pointer-set($\alpha\beta$) iff

$$L_j[\text{value}(\alpha\beta)] = \text{address}(g).$$

Since $k \neq j$ and (3) is true,

g is in pointer-set($\alpha f \beta$) iff g is in pointer-set($\alpha\beta$).

Therefore, property III is true; hence condition (3) implies property III. In proving (3), there are two cases depending whether or not synchronizer(f) is true. First, suppose that not synchronizer(f). By repeated applications of condition (3) in the definition of a C process and the fact that, by Lemma 4,

for $1 \leq j \leq \text{length}(\beta)$, $k \neq \text{program-counter}(\beta_j)$,

we can conclude that value($\alpha f \beta$) = value($\alpha\beta f$). By Lemma 1, part 1,

for $i \neq k$, $L_i[\text{value}(\alpha\beta)] = L_i[\text{value}(\alpha\beta f)]$.

Thus, in this case (3) is true. Second, suppose that synchronizer(f). By the definition of a C process and Lemma 1, part 1,

$D[\text{value}(\alpha)] = D[\text{value}(\alpha f)]$ and
for $i \neq k$, $L_i[\text{value}(\alpha)] = L_i[\text{value}(\alpha f)]$.

By Lemma 3,

$D[\text{value}(\alpha\beta)] = D[\text{value}(\alpha f\beta)]$ and

for $i \neq k$, $L_i[\text{value}(\alpha\beta)] = L_i[\text{value}(\alpha f\beta)]$.

Thus, in this case (3) is true. Therefore, ρ satisfies property III.

We will now prove that ρ satisfies property IV. Suppose that $\alpha f\beta\delta$ and $\alpha\beta f\delta$ are finite active timings. In order to prove property IV, it is sufficient to prove that

(4) for all i, $L_i[\text{value}(\alpha f\beta\delta)] = L_i[\text{value}(\alpha\beta f\delta)]$.

For suppose that (4) is true. Let g be an action. Then g is in pointer-set$(\alpha f\beta\delta)$ iff

$L_j[\text{value}(\alpha f\beta\delta)] = \text{address}(g)$

where j = program-counter(g). Also g is in pointer-set$(\alpha\beta f\delta)$ iff

$L_j[\text{value}(\alpha\beta f\delta)] = \text{address}(g)$.

Since (4) is true,

g is in pointer-set$(\alpha f\beta\delta)$ iff g is in pointer-set$(\alpha\beta f\delta)$.

Therefore, property IV is true; hence, condition (4) implies property IV. In proving (4), there are two cases depending whether or not synchronizer(f) is true. First, suppose that not synchronizer(f). By repeated applications of condition (3) in the definition of a C process and Lemma 4, we can conclude as before that value$(\alpha f\beta)$ = value$(\alpha\beta f)$. Thus, value$(\alpha f\beta\delta)$ = value$(\alpha\beta f\delta)$, and hence (4) is true. Second, suppose that synchronizer(f). As in the proof of property III we can show that

$D[\text{value}(\alpha\beta)] = D[\text{value}(\alpha f\beta)]$ and

$$\text{for all } i \neq k,\ L_i[\text{value}(\alpha\beta)] = l_i[\text{value}(\alpha f\beta)]$$

where k = program-counter(f). By the definition of a C process and Lemma 1, part 1,

$$D[\text{value}(\alpha\beta f)] = D[\text{value}(\alpha f\beta)] \text{ and}$$
$$\text{for } i \neq k,\ L_i[\text{value}(\alpha\beta f)] = L_i[\text{value}(\alpha f^{\beta})].$$

Since synchronizer(f), there is a constant, say z, such that

$$\text{if } f(x) \neq x, \text{ then } L_k[f(x)] = z.$$

Thus, $L_k[\text{value}(\alpha f)] = z$ and $L_k[\text{value}(\alpha\beta f)] = z$. By Lemma 4 and Lemma 1, part 1,

$$L_k[\text{value}(\alpha f\beta)] = L_k[\text{value}(\alpha f)].$$

Therefore,

$$D[\text{value}(\alpha\beta f)] = D[\text{value}(\alpha f\beta)] \text{ and}$$
$$\text{for all } i,\ L_i[\text{value}(\alpha\beta f)] = L_i[\text{value}(\alpha f\beta)].$$

Finally, by Lemma 3,

$$\text{for all } i,\ L_i[\text{value}(\alpha\beta f\delta)] = L_i[\text{value}(\alpha f\beta\delta)].$$

Thus, in this case, (4) is true. Thus, ρ satisfies property IV. □

2.10 REMARKS ON THE PROPERTIES I-IV

We will now present an example of a PV process that shows that certain generalizations of properties III and IV do not hold for PV processes. The formal representation of process EX2 is in Figure 4.

```
program counter $L_1, L_2$; (initial value 0)
integer x,y; (initial value 0)
semaphore m; (initial value 1)

SUBPROCESS-1

(1) when $L_1 = 0 \wedge m > 0$ do $L_1 \leftarrow 1$; $m \leftarrow m - 1$
(2) when $L_1 = 1$              do $L_1 \leftarrow 2$; $x \leftarrow 1$
(3) when $L_1 = 2$              do $L_1 \leftarrow 3$; $m \leftarrow m + 1$

SUBPROCESS-2

(4) when $L_2 = 0 \wedge m > 0$ do $L_2 \leftarrow 1$; $m \leftarrow m - 1$
(5) when $L_2 = 1$              do $L_2 \leftarrow 2$; $y \leftarrow x$
(6) when $L_2 = 2$              do $L_2 \leftarrow 3$; $m \leftarrow m + 1$
(7) when $L_2 = 3$              do $L_2 \leftarrow$ if $y = 0$ then 4 else 5
(8) when $L_2 = 4$              do $L_2 \leftarrow 5$
(9) when $L_2 = 5$              do $L_2 \leftarrow 6$
```

FIGURE 4. Process EX2: formal representation

The informal representation of $\mathcal{P}$ is displayed in Figure 5.

```
integer x,y; (initial value 0)
semaphore m; (initial value 1)

SUBPROCESS-1

(1) P(m);
(2) x := 1;
(3) V(m);

SUBPROCESS-2

(4) P(m);
(5) y := x;
(6) V(m);
(7) if y = 0 then goto A else goto B;
(8) A:;
(9) B:;
```

FIGURE 5. Process EX2: informal representation

First, let us consider the following generalization of property III:

(1) if $\alpha\lambda\beta$ and $\alpha\beta$ are finite active timings and each action in λ is in SUBPROCESS-i, then pointer-set($\alpha\lambda\beta$) $\cap$ SUBPROCESS-j = pointer-set($\alpha\beta$) $\cap$ SUBPROCESS-j, provided $i \neq j$.

By the definition of the process EX2, pointer-set(4567) = {1,8} and pointer-set(1234567) = {9}. Thus, (1) is false, since pointer-set(4567) $\cap$ SUBPROCESS-2 = {8} and pointer-set(1234567) $\cap$ SUBPROCESS-2 = {9}. Therefore, this generalization of property III does not hold for PV processes.

Second, let us consider the following generalization of property IV:

(2) if α and β are finite active timings and α is a permutation of β, then pointer-set(α) = pointer-set(β).

By the definition of process EX2, pointer-set(1234567) = {9} and pointer-set(4567123) = {8}. Since 1234567 is a permutation of 4567123, this generalization does not hold for PV processes.

These two examples show how delicate the two properties III and IV are. The chief difficulty we had in proving these properties, in Theorem 12, was due to the fact that we allow <u>conditionals</u> in our C processes. Any reasonable theory or model of PV processes must allow conditionals. For example, conditionals are used in the processes defined in Courtois, Heymans, Parnas [1971]. In addition, conditionals allow us to show that our definition of a PV process incorporates the notion of a "PV with array semaphores". A P on the array semaphore s(n) is equivalent to the actions

```
when $L_k = a$ do $L_k \leftarrow$ branch(n)
when $L_k = a_i \wedge S_i > 0$ do $L_k \leftarrow a'$; $S_i \leftarrow S_i - 1$   $[1 \le i \le \max]$.
```

The value of branch(n) is equal to a_n. We have replaced an array P by a "branch action" and several P's; the same construction can be used for array V's.

The properties I-IV and the notions ready-set, pointer-set, and subprocess are of central importance. In fact, all our basic theorems are proved using only these properties and these notions. The only exceptions are the specific results in Section 7, i.e., our theorems about PV processes, PVchunk processes, PVmultiple processes, and up/down processes. Our insistence on using only the properties I-IV and the notions ready-set, pointer-set, and subprocess will now be demonstrated.

13. __Definition.__ A set of actions in a process is __sequential__ provided for all finite timings α, at most one action from this set is in the ready-set(α).

14. __Theorem.__ Each subprocess-i of a C process is sequential.

__Proof.__ Suppose that f and g are in SUBPROCESS-i, and suppose that f and g are in ready-set(α). By property I, f and g are in pointer-set(α). Thus, by property II, f = g. □

3. SCHEDULERS

We will now define the concept of scheduler. This concept is used to state the theorem of Habermann [1972]. More importantly, the concept of scheduler displays the generality of our model.

1. Definition. Suppose that ρ is a process. Then S is a scheduler for ρ provided, S is a predicate on the timings of ρ.

We can use a scheduler S to select timings that satisfy a number of criterion. First, the scheduler S might "enforce a priority" among the actions of the process. For example, the actions that control a hardware device may be given priority over the actions of a user's program. Second, the scheduler S might enforce "fairness" among the actions of the process. For instance, consider a disk queuer that uses the "least arm movement criterion" to select the next request. Often the scheduler is designed so that requests for a distance part of the disk do not wait forever. Third, the scheduler S might "enforce the release mechanism of PV" (Dijkstra [1968a]).

The separation of the notion of scheduler from the notion of process gives us a great deal of freedom. We are able to study the same process with respect to a variety of schedulers. We do not present one scheduler as the "correct one"; instead, we present and study several different schedulers.

3.1 SEMI-ACTIVE TIMINGS

2. Definition. A timing α in a C process is semi-active provided, if $\beta f \leq \alpha$, then f is in pointer-set(β).

3. Theorem. Suppose that ρ is a C process and that the relation subprocess$_\rho$ has m equivalence classes. Also suppose that $N_1,\ldots,N_k$ are integers from the set $\{1,\ldots,m\}$. Then there is at most one semi-active timing α such that

$$\text{for } 1 \leq i \leq k,\ \alpha_i \text{ is in SUBPROCESS-}N_i.$$

Proof. Immediate from property II. □

Theorem 3 is implicitly used in operating systems. They usually consider timings as sequences of "subprocess names" instead of as sequences of actions. The key observation is: as long as we are only interested in semi-active timings, it is immaterial whether we consider timings as sequences of subprocess names or as sequences of actions.

3.2 BLOCKING AND THE PV RELEASE SCHEDULER

It is instructive to consider the timing 13521 of the process EX3 (a formal representation of EX3 is in Figure 6). This timing is semi-active; it is not active. The "execution" of this timing is informally

1	this action changes L_1 to 2 and a to 0
3	this action does not change the data structure - it is now blocked
5	this action does not change the data structure - it is now blocked
2	this action changes L_1 to 1 and a to 1
1	this action changes L_1 to 2 and a to 0

```
program counter $L_1, L_2, L_3$; (initial value 1)
semaphore a; (initial value 1)
SUBPROCESS-1
(1)  when $L_1 = 1 \wedge a \geq 1$ do $L_1 \leftarrow 2$; $a \leftarrow a - 1$
(2)  when $L_1 = 2$             do $L_1 \leftarrow 1$; $a \leftarrow a + 1$
SUBPROCESS-2
(3)  when $L_2 = 1 \wedge a \geq 1$ do $L_2 \leftarrow 2$; $a \leftarrow a - 1$
(4)  when $L_2 = 2$             do $L_2 \leftarrow 1$; $a \leftarrow a + 1$
SUBPROCESS-3
(5)  when $L_3 = 1 \wedge a \geq 1$ do $L_3 \leftarrow 2$; $a \leftarrow a - 1$
(6)  when $L_3 = 2$             do $L_3 \leftarrow 1$; $a \leftarrow a + 1$
```

FIGURE 6. Process EX3: formal representation

The usual literature definition of PV (Dijkstra [1968a]) rules out this timing. The informal reason that the timing 13521 is not allowed is: "P's that are blocked, i.e., 3 and 5, must be given priority over P's that are not blocked, i.e., 1". The way our model handles this restriction is that we can define a scheduler S such that

> if a timing satisfies S, then the timing "gives blocked P's priority over other P's".

In particular, 13521 will not satisfy S.

4. <u>Definition</u>. Suppose that α is a finite timing in a C process. Define <u>blocked-set</u>(α) inductively as follows.

(1) blocked-set(Λ) is the empty set.

(2) If α_k is not in ready-set$(\alpha_1 \ldots \alpha_{k-1})$ then

blocked-set$(\alpha_1 \ldots \alpha_k)$ = blocked-set$(\alpha_1 \ldots \alpha_{k-1}) \cup \{\alpha_k\}$.

(3) If α_k is in ready-set$(\alpha_1 \ldots \alpha_{k-1})$, then

blocked-set$(\alpha_1 \ldots \alpha_k)$ = blocked-set$(\alpha_1 \ldots \alpha_{k-1}) - \{\alpha_k\}$.

Note, $\cup$ is set union, and - is set difference.

Informally, the action α_k is "added" to block-set$(\alpha_1 \ldots \alpha_{k-1})$ if α_k cannot "run"; the action α_k is "removed" from blocked-set$(\alpha_1 \ldots \alpha_{k-1})$ if α_k can "run". For example, consider again the timing 13521 of the process EX3. Then blocked-set(1352) = $\{3,5\}$; this formalizes the intuitive statement made earlier.

In order for an action to be in blocked-set$(\alpha_1 \ldots \alpha_k)$, it must have been "tried". More exactly, blocked-set$(\alpha_1 \ldots \alpha_k)$ is a subset of $\{\alpha_1, \ldots, \alpha_k\}$. For instance, consider the timing 13 in process EX3. Clearly, blocked-set(13) = $\{3\}$. Note, action 5 is not in ready-set(13). However, action 5 is not in blocked-set(13); for it has not been tried.

We will only ever consider blocked-set(α) for semi-active α. For timings that are not semi-active, blocked-set(α) can behave in strange ways. For instance, since (in process EX3) action 2 is not in ready-set(Λ), blocked-set(2) = $\{2\}$. However, it is definitely "pathological" to have a V(a) inserted into the blocked-set. This "pathology" can be avoided if we restrict our attention to semi-active timings. In fact we can prove in a PV process that

if f is in blocked-set(α) and α is semi-active, then f is a P(b), for some b.

(2) If α_k is not in ready-set($\alpha_1 \cdots \alpha_{k-1}$) then
blocked-set($\alpha_1 \cdots \alpha_k$) = blocked-set($\alpha_1 \cdots \alpha_{k-1}$) $\cup$ $\{\alpha_k\}$.

(3) If α_k is in ready-set($\alpha_1 \cdots \alpha_{k-1}$), then
blocked-set($\alpha_1 \cdots \alpha_k$) = blocked-set($\alpha_1 \cdots \alpha_{k-1}$) - $\{\alpha_k\}$.

Note, $\cup$ is set union, and - is set difference.

Informally, the action α_k is "added" to block-set($\alpha_1 \cdots \alpha_{k-1}$) if α_k cannot "run"; the action α_k is "removed" from blocked-set($\alpha_1 \cdots \alpha_{k-1}$) if α_k can "run". For example, consider again the timing 13521 of the process EX3. Then blocked-set(1352) = $\{3,5\}$; this formalizes the intuitive statement made earlier.

In order for an action to be in blocked-set($\alpha_1 \cdots \alpha_k$), it must have been "tried". More exactly, blocked-set($\alpha_1 \cdots \alpha_k$) is a subset of $\{\alpha_1, \ldots, \alpha_k\}$. For instance, consider the timing 13 in process EX3. Clearly, blocked-set(13) = $\{3\}$. Note, action 5 is not in ready-set(13). However, action 5 is not in blocked-set(13); for it has not been tried.

We will only ever consider blocked-set(α) for semi-active α. For timings that are not semi-active, blocked-set(α) can behave in strange ways. For instance, since (in process EX3) action 2 is not in ready-set(Λ), blocked-set(2) = $\{2\}$. However, it is definitely "pathological" to have a V(a) inserted into the blocked-set. This "pathology" can be avoided if we restrict our attention to semi-active timings. In fact we can prove in a PV process that

if f is in blocked-set(α) and α is semi-active, then f is a P(b), for some b.

5. Definition. A timing α is a release timing in a C process provided, for $\beta f \le \alpha$,

> if blocked-set(β) $\cap$ ready-set(β) is non-empty, then f is in blocked-set(β) $\cap$ ready-set(β).

Suppose that α is a timing in a C process and $\beta f \le \alpha$. Informally, the set of actions in the set blocked-set(β) $\cap$ ready-set(β) is the set of actions that satisfy

(1) they can "run",
(2) they were at "one time blocked",
(3) they have never been "unblocked".

The release restriction is that if this set is non-empty, then f must be in blocked-set(β) $\cap$ ready-set(β). A release timing "enforces a kind of priority rule": the actions in blocked-set(β) $\cap$ ready-set(β) have a "priority" over all other actions. However, there is no priority among the actions in blocked-set(β) $\cap$ ready-set(β).

Let us return once again to the timing 13521 of process EX3. The objection stated earlier is: 13521 is not a release timing. Both 13523 and 13525 are release timings. The release restriction enforces no priority based on "the order an action is added to blocked-set".

Most discussions of PV processes implicitly assume that the scheduler always selects release timings. An example of this assumption is the theorem found in Habermann [1972]. In our model this theorem is a property of PV processes <u>and</u> the release scheduler. Some people have stated that there are <u>two</u> kinds of PV. In our model there is one kind of PV process; PV processes behave differently for different schedulers.

We will now use our model to state the theorem found in Habermann [1972]. Suppose that $\mathcal{P}$ is a PV process and that S is a semaphore. Also suppose that α is a semi-active release timing. Define three functions as follows.

(1) ns(i) = the number of k with $1 \le k \le i$ such that $\alpha_k \in$ ready-set$(\alpha_1 \ldots \alpha_{k-1})$ and α_k is a V(S).

(2) np(i) = the number of k with $1 \le k \le i$ such that $\alpha_k \in$ ready-set$(\alpha_1 \ldots \alpha_{k-1})$ and α_k is a P(S).

(3) nw(i) = the number of k with $1 \le k \le i$ such that $\alpha_k \notin$ blocked-set$(\alpha_1 \ldots \alpha_{k-1})$ and α_k is a P(S).

Say the integer i <u>performs a release</u> iff α_i is a V(S) and blocked-set$(\alpha_1 \ldots \alpha_{i-1})$ contains a P(S). Then Habermann's theorem - in our model - states that

(4) if i does not perform a release, then

$$np(i) = MIN(nw(i), ns(i) + S_0)$$

where S_0 is the initial value of S.

Habermann's theorem can be rephrased into a more intuitive form. First, n(w) - np(i) is the cardinality of the set blocked-set$(\alpha_1 \ldots \alpha) \cap \{f \mid f \text{ is a } P(S)\}$. Second, ns(i) + S_0 - np(i) is equal to S[value$(\alpha_1 \ldots \alpha_i)$]. Thus, (4) is equivalent to

(5) if i does not perform a release, then either

(a) blocked-set($\alpha_1 \ldots \alpha_i$) contains no P(S), or

(b) S[value($\alpha_1 \ldots \alpha_i$)] = 0.

Informally, if i does not perform a release, then either there are no "blocked P(S)'s" or the "semaphore S is 0".

For example, let $\mathcal{P}$ = EX3 and let α = 13523. The values of the functions ns, np, nw are in Table 1. Since 5 does not perform a release,

$$np(5) = MIN(nw(5), ns(5) + 1).$$

Note, np(4) $\neq$ MIN(nw(4), ns(4) + 1): this is true since 4 does perform a release. The relation np(i) = MIN(nw(i), ns(i) + 1) does not hold "when a release is in progress"; it does hold at all other places.

integer i	ns(i)	np(i)	nw(i)
1	0	1	1
2	0	1	2
3	0	1	3
4	1	1	3
5	1	2	3

TABLE 1. The functions ns, np, nw for the timing 13523 in EX3

3.3 "FAIRNESS"

We will now define a type of timing that is related to the intuitive concept of "fairness".

6. Definition. Say the timing α in a C process is pointer-bounded provided, if f is in pointer-set($\alpha_1 \ldots \alpha_k$) and f is not in

pointer-set($\alpha_1 \ldots \alpha_{k-1}$), then for some $m > k$, $\alpha_m = f$.

Note, if $k = 0$, then by convention: pointer-set(α_{-1}) is the empty set. Thus, if α is pointer-bounded and f is in pointer-set(Λ), then for some $m > 0$, $\alpha_m = f$.

A timing is pointer-bounded if no action "waits forever for a chance to be tried". For example, consider the timing 12... (12 repeated forever) in process EX3. This timing is active and a release timing; it is not pointer-bounded. In fact, if α is a semi-active release pointer-bounded timing in EX3, then

"at least two of the subprocesses of EX3 must cycle an infinite number of times".

Caution, the timing 135231413... (231413 repeated forever) is a semi-active release pointer-bounded timing. In this timing

"one of the subprocesses of EX3 never cycles".

Thus, while the notion of pointer-bounded is a notion of "fairness", it is a weak one. Clearly, we can define schedulers that would enforce a stronger notion of "fairness".

4. RELATIONS BETWEEN PROCESSES

4.1 SYNCHRONIZATION PROBLEMS

One of the issues that we must face in our study of "synchronization problems" is: how are we to represent "problems"? The usual method in the synchronization area is to state a problem in some informal language. A "solution" to this kind of problem is some process and some assertions that the process satisfies. This method, however, is hard to formalize; in addition, this method cannot easily compare different solutions to the same problem.

We consider the study of synchronization problems as the study of the relationships between processes. More exactly - in our theory - a synchronization problem is:

(1) a process $\mathcal{P}$ and

(2) a relation R between processes.

A solution to a synchronization problem is a process $\mathcal{Q}$ such that $R(\mathcal{Q},\mathcal{P})$ is true. The components of a synchronization problem - $\mathcal{P}$,R - are present in the literature definitions of synchronization problems. The process $\mathcal{P}$ is usually defined informally: often in English. This process defines the "behavior" that is desired. The relation R is usually defined informally. For instance, the chief source of the controversy between Brinch Hansen and Courtois, Heymans, Parnas can be attributed to their informal definitions of R. Brinch Hansen considered the second reader-writer problem to be $\mathcal{P}$,R while Courtois, Heymans, Parnas considered the second reader-writer problem to be $\mathcal{P}$,R' where $R \neq R'$. We will say more about this controversy when we study the second reader-writer problem.

There are several advantages in our approach to problem definition. First, any reasonable theory of processes must be interested in relations between processes. Second, we avoid introducing a new object - a "problem" - into our theory. Essentially our study of synchronization problems reduces to a study of relations between processes. This is a standard technique in many theories. For example, in finite state machine theory, relations between machines are used to define problems. Third, our approach makes clear the importance of the relation R. Many relations R are possible: we make no claim that any one relation is the only important one.

We will next present definitions of several of the relations that are implicit in the synchronization literature. We will then use our model to study the second reader-writer problem. Next we will define "simulate"; this relation is one of our central concepts. Finally, several processes will be defined; these processes define the behavior of several synchronization problems.

4.2 REALIZATIONS

1. <u>Definition</u>. Suppose that $\mathcal{Q}$ and $\mathcal{P}$ are processes. A <u>realization r</u> is a function from the timings of $\mathcal{Q}$ to the timings of $\mathcal{P}$ such that: if $\alpha\beta$ is a timing in $\mathcal{Q}$, then $r(\alpha\beta) = r(\alpha)r(\beta)$; and for all actions f in $\mathcal{Q}$, r(f) is either Λ or an action in $\mathcal{P}$.

2. Definition. Suppose that r is a realization from $\mathcal{Q}$ to $\mathcal{P}$. Then say that the action f in $\mathcal{Q}$ is observable under r if $r(f) \neq \Lambda$; otherwise, say that the action f is unobservable under r. When there can be no confusion we will drop 'under r'.

Informally, an action in $\mathcal{Q}$ is unobservable provided it does a "book-keeping operation" that is not present in $\mathcal{P}$; an action is observable provided it does an "operation" that is present in $\mathcal{P}$. Note, if r is a realization from $\mathcal{Q}$ to $\mathcal{P}$, then

$\mathcal{P}$ is an "abstraction" of $\mathcal{Q}$.

Since $\mathcal{Q}$ may contain unobservable actions, "detail is lost in going from $\mathcal{Q}$ to $\mathcal{P}$".

4.3 SAFE REALIZATIONS

It is not obvious that the concept of realization can be used to define the relations used in the literature: "safe" and "deadlock free" (Dijkstra [1968]). We will now show that it can.

3. Definition. Suppose that r is a realization from $\mathcal{Q}$ to $\mathcal{P}$. Then say that r is safe provided

$$r\,\{\alpha \mid \alpha \text{ active in } \mathcal{Q}\} \subseteq \{\beta \mid \beta \text{ active in } \mathcal{P}\}.$$

Also say that r is onto safe provided

$$r\,\{\alpha \mid \alpha \text{ active in } \mathcal{Q}\} = \{\beta \mid \beta \text{ active in } \mathcal{P}\}.$$

Since the concept of safe is defined informally in the synchronization area, we cannot prove that our definition of safe is valid. However, we can present some evidence to show that our definition is a reasonable one. The definition of safe is equivalent to

(1) for observable f, if f $\in$ ready-set$_{\mathcal{Q}}(\alpha)$, then

$r(f) \in$ ready-set$_{\mathcal{P}}(r(\alpha))$.

Informally, (1) states that whenever process $\mathcal{Q}$ can "make a change", then process $\mathcal{P}$ can also "make a corresponding change". Clearly, this captures the concept of safe.

We are almost exclusively interested in realizations that are onto safe and not just safe. For the realization from $\mathcal{Q}$ to $\mathcal{P}$ that maps every timing to Λ is safe; it is onto safe only when $\mathcal{P}$ is "trival", i.e., when Λ is the only active timing in $\mathcal{P}$.

4.4 DEADLOCK FREE REALIZATIONS

4. Definition. Suppose that r is a realization from $\mathcal{Q}$ to $\mathcal{P}$ Then say that r is deadlock free provided

if ready-set$_{\mathcal{Q}}(\alpha)$ is empty, then ready-set $(r(\alpha))$ is empty.

As in the case of safe, we cannot prove that our concept of deadlock free is valid. However, we can present some evidence that our definition is a reasonable one. The definition of deadlock free is

(1) if ready-set$_{\mathcal{Q}}(\alpha)$ is empty, then ready-set$_{\mathcal{P}}(r(\alpha))$ is empty.

Informally, this means that $\mathcal{Q}$ can "halt after executing the timing α" only if $\mathcal{P}$ also "halts after executing the timing $r(\alpha)$". Clearly, this captures the concept of deadlock free as used in the synchronization area. Note, if the process $\mathcal{P}$ never "halts", then (1) is equivalent to

for all finite timings α, ready-set$_{\mathcal{Q}}(\alpha)$ is non-empty.

Also note, if the process $\mathcal{Q}$ never "halts", then any realization from $\mathcal{Q}$ to $\mathcal{P}$ is deadlock free.

There is an interesting connection between safe and deadlock free. Suppose that r is a realization from $\mathfrak{D}$ to ρ. In order to avoid unnecessary complexities, suppose that for each action g in ρ, there is an action f in ρ such that r(f) = g. As we stated earlier, r is safe iff

for f observable, if $f \in \text{ready-set}_{\mathfrak{D}}(\alpha)$, then $r(f) \in \text{ready-set}_{\rho}(r(\alpha))$.

We can also show that r is deadlock free iff

for f observable, if $\text{ready-set}_{\mathfrak{D}}(\alpha)$ is empty, then $r(f) \notin \text{ready-set}_{\rho}(r(\alpha))$.

Therefore, r is deadlock free iff

(1) for f observable, if $r(f) \in \text{ready-set}_{\rho}(r(\alpha))$, then $\text{ready-set}_{\mathfrak{D}}(\alpha)$ is non-empty.

The converse of safe is

(2) for observable f, if $r(f) \in \text{ready-set}_{\rho}(r(\alpha))$, then $f \in \text{ready-set}_{\mathfrak{D}}(\alpha)$.

Comparing (1) and (2), we see that <u>deadlock free is formed by weakening the converse of safe</u>: deadlock free replaces '$f \in \text{ready-set}_{\mathfrak{D}}(\alpha)$' by '$\text{ready-set}_{\mathfrak{D}}(\alpha)$ is non-empty'. The converse of safe is too strong to be of any practical value; on the other hand, deadlock free is very weak.

4.5 SECOND READER-WRITER PROBLEM

We will now study the second reader-writer problem as defined by Courtois, Heymans, Parnas [1971]. In this study we will use three processes: CHP, W2, H.

(1) CHP. This is essentially the PV process used in Courtois, Heymans, Parnas [1971] to "solve" the second reader-writer problem. A formal representation of CHP is in Figure 8; an informal representation is in Figure 9. For simplicity we have assumed that there is one writer and three readers; this is done purely to avoid complexities in notation. Note, the definitions of READER-1, READER-2, READER-3 have been replaced by one general definition of READER-i ($1 \leq i \leq 3$). Also observe that action (k,i) is the kth action of READER-i; thus, (4,2) is

$\underline{\text{when}}\ L_2 = 4\ \underline{\text{do}}\ L_2 \leftarrow 5;\ \text{readcount} \leftarrow \text{readcount} + 1.$

Finally, note that READER-1, READER-2, READER-3, and WRITER are "cyclic subprocesses"; for example, the last action of WRITER is

$\underline{\text{when}}\ L = 13\ \underline{\text{do}}\ L \leftarrow 1;\ \text{mutex2} \leftarrow \text{mutex2} + 1$

and the first action of WRITER is

$\underline{\text{when}}\ L = 1 \wedge \text{mutex2} \geq 1\ \underline{\text{do}}\ L \leftarrow 2;\ \text{mutex2} \leftarrow \text{mutex2} - 1.$

Thus, action (13) "resets" the program counter L to 1.

```
program counter $L_1, L_2, L_3, L$; (initial value 1)
integer readcount, writecount; (initial value 0)
semaphore mutex1, mutex2, mutex3, w,a; (initial value 1)

WRITER
(1)   when $L = 1 \wedge mutex2 \geq 1$   do $L \leftarrow 2$; $mutex2 \leftarrow mutex2 - 1$
(2)   when $L = 2$                        do $L \leftarrow 3$; $writecount \leftarrow writecount + 1$
(3)   when $L = 3$                        do $L \leftarrow$ if $writecount = 1$ then 4 else 5
(4)   when $L = 4 \wedge a \geq 1$        do $L \leftarrow 5$; $a \leftarrow a - 1$
(5)   when $L = 5$                        do $L \leftarrow 6$; $mutex2 \leftarrow mutex2 + 1$
(6)   when $L = 6 \wedge b \geq 1$        do $L \leftarrow 7$; $b \leftarrow b - 1$
(7)   when $L = 7$                        do $L \leftarrow 8$
(8)   when $L = 8$                        do $L \leftarrow 9$; $b \leftarrow b + 1$
(9)   when $L = 9 \wedge mutex2 \geq 1$   do $L \leftarrow 10$; $mutex2 \leftarrow mutex2 - 1$
(10)  when $L = 10$                       do $L \leftarrow 11$; $writecount \leftarrow writecount - 1$
(11)  when $L = 11$                       do $L \leftarrow$ if $writecount = 0$ then 12 else 13
(12)  when $L = 12$                       do $L \leftarrow 13$; $a \leftarrow a + 1$
(13)  when $L = 13$                       do $L \leftarrow 1$; $mutex2 \leftarrow mutex2 + 1$

READER-i ($1 \leq i \leq 3$)
(1,i)   when $L_i = 1 \wedge mutex3 \geq 1$   do $L_i \leftarrow 2$; $mutex3 \leftarrow mutex3 - 1$
(2,i)   when $L_i = 2 \wedge a \geq 1$        do $L_i \leftarrow 3$; $a \leftarrow a - 1$
(3,i)   when $L_i = 3 \wedge mutex1 \geq 1$   do $L_i \leftarrow 4$; $mutex1 \leftarrow mutex1 - 1$
(4,i)   when $L_i = 4$                        do $L_i \leftarrow 5$; $readcount \leftarrow readcount + 1$
(5,i)   when $L_i = 5$                        do $L_i \leftarrow$ if $readcount = 1$ then 6 else 7
(6,i)   when $L_i = 6 \wedge b \geq 1$        do $L_i \leftarrow 7$; $b \leftarrow b - 1$
(7,i)   when $L_i = 7$                        do $L_i \leftarrow 8$; $mutex1 \leftarrow mutex1 + 1$
(8,i)   when $L_i = 8$                        do $L_i \leftarrow 9$; $a \leftarrow a + 1$
(9,i)   when $L_i = 9$                        do $L_i \leftarrow 10$; $mutex3 \leftarrow mutex3 + 1$
(10,i)  when $L_i = 10$                       do $L_i \leftarrow 11$
(11,i)  when $L_i = 11 \wedge mutex1 \geq 1$  do $L_i \leftarrow 12$; $mutex1 \leftarrow mutex1 - 1$
(12,i)  when $L_i = 12$                       do $L_i \leftarrow 13$; $readcount \leftarrow readcount - 1$
(13,i)  when $L_i = 13$                       do $L_i \leftarrow$ if $readcount = 0$ then 14 else 15
(14,i)  when $L_i = 14$                       do $L_i \leftarrow 15$; $b \leftarrow b + 1$
(15,i)  when $L_i = 15$                       do $L_i \leftarrow 1$; $mutex1 \leftarrow mutex1 + 1$
```

FIGURE 8. Process CHP: formal representation

```
integer readcount, writecount; (initial value 0)
semaphore mutex1, mutex2, mutex3,a,b; (initial value 1)

READER-i (1 ≤ i ≤ 3)                        WRITER
 (1,i)  P(mutex3);                          (1)  P(mutex2);
 (2,i)    P(a);                             (2)  writecount := writecount + 1;
 (3,i)      P(mutex1)                       (3)  if writecount = 1 then
 (4,i)      readcount := readcount + 1;     (4)            P(a);
 (5,i)      if readcount = 1 then           (5)  V(mutex2);
 (6,i)              P(b);                   (6)  P(b);
 (7,i)      V(mutex1);                      (7)  writing is performed
 (8,i)    V(a);                             (8)  V(b);
 (9,i)  V(mutex3);                          (9)  P(mutex2);
(10,i)  reading is performed               (10)  writecount := writecount - 1;
(11,i)  P(mutex1);                         (11)  if writecount = 1 then
(12,i)  readcount := readcount - 1;        (12)            V(a);
(13,i)  if readcount = 0 then              (13)  V(mutex2);
(14,i)              V(b);
(15,i)  V(mutex1);
```

FIGURE 9. Process CHP: informal representation

(2) W2. This is essentially the up/down process used in Wodon [1972] to "solve" the second reader-writer problem. A formal representation of W2 is in Figure 10; an informal representation is in Figure 11. As in CHP, we have assumed that there is one writer and three readers. Again observe that action (k,i) is the kth action of READER-i, thus (3,2) is

$$\underline{\text{when}}\ L_2 = 3\ \underline{\text{do}}\ L_2 \leftarrow 1;\ a \leftarrow a + 1.$$

READER-1, READER-2, READER-3, and WRITER are "cyclic subprocesses" as in CHP.

```
program counter $L_1, L_2, L_3, L$; (initial value 1)
semaphore a,b,s; (initial value 0)

WRITER
(1)   when $L = 1$                      do $L \leftarrow 2$; $s \leftarrow s - 1$
(2)   when $L = 2 \wedge a + b \geq 0$  do $L \leftarrow 3$; $b \leftarrow b - 1$
(3)   when $L = 3$                      do $L \leftarrow 4$
(4)   when $L = 4$                      do $L \leftarrow 5$; $b \leftarrow b + 1$
(5)   when $L = 5$                      do $L \leftarrow 1$; $s \leftarrow s + 1$

READER-i ($1 \leq i \leq 3$)
(1,i)   when $L_i = 1 \wedge s \geq 0$  do $L_i \leftarrow 2$; $a \leftarrow a - 1$
(2,i)   when $L_i = 2$                  do $L_i \leftarrow 3$
(3,i)   when $L_i = 3$                  do $L_i \leftarrow 1$; $a \leftarrow a + 1$
```

FIGURE 10. Process W2: formal representation

```
semaphore a,b,s; (initial value 0)

WRITER                                  READER-i ($1 \leq i \leq 3$)
(1)   { }: down (s);                    (1,i)   {s}: down (a);
(2)   {a,b}: down (b);                  (2,i)   reading is performed
(3)   writing is performed              (3,i)   { }: up (a);
(4)   { }: up (b);
(5)   { }: up (s);
                                        [{ } is the empty set]
```

FIGURE 11. Process W2: informal representation

(3) H. This is essentially the PV process used in Brinch Hansen [1972a] to "solve" the second reader-writer problem. A formal representation of H is in Figure 12; an informal representation is in Figure 13. Originally, H was defined in a "structured notation"; we have expanded this notation into its PV definition. In addition, we have followed the footnote correction in Brinch Hansen [1972a]: we have inserted actions (1,i) and (11,i) ($1 \leq i \leq 3$). As in CHP and W2, we have assumed that there is one writer and three readers; also, each of these subprocesses is "cyclic".

In all three processes, we have assumed that in READER-i ($1 \leq i \leq 3$)

reading is performed

and in WRITER

writing is performed

is one action. This assumption is made purely for convenience; it does not affect our discussion. Note, we will refer to process W2 in later chapters.

As stated in Section 4.1 - in order to define the second reader-writer problem - we must supply two objects.

(1) a process $\mathcal{P}$ that defines the behavior of the readers and writers,

(2) a relation R between processes.

```
program counter $L_1, L_2, L_3, L$; (initial value 1)
integer readcount, writecount, c; (initial value 0)
semaphore a,b,d,e:  (initial value 1)

READER-1 $(1 \le i \le 3)$
 (1,i)  when $L_i = 1 \wedge b \ge 1$  do $L_i \leftarrow 2$; $b \leftarrow b - 1$
 (2,i)  when $L_i = 2 \wedge a \ge 1$  do $L_i \leftarrow 3$; $a \leftarrow a - 1$
 (3,i)  when $L_i = 3$                 do $L_i \leftarrow$ if writecount = 0 then 9 else 4
 (4,i)  when $L_i = 4$                 do $L_i \leftarrow 5$; $c \leftarrow c + 1$
 (5,i)  when $L_i = 5$                 do $L_i \leftarrow 6$; $a \leftarrow a + 1$
 (6,i)  when $L_i = 6 \wedge e \ge 1$  do $L_i \leftarrow 7$; $e \leftarrow e - 1$
 (7,i)  when $L_i = 7 \wedge a \ge 1$  do $L_i \leftarrow 8$; $a \leftarrow a - 1$
 (8,i)  when $L_i = 8$                 do $L_i \leftarrow 3$
 (9,i)  when $L_i = 9$                 do $L_i \leftarrow 10$; readcount $\leftarrow$ readcount + 1
(10,i)  when $L_i = 10$                do $L_i \leftarrow 11$; $a \leftarrow a + 1$
(11,i)  when $L_i = 11$                do $L_i \leftarrow 12$; $b \leftarrow b + 1$
(12,i)  when $L_i = 12$                do $L_i \leftarrow 13$
(13,i)  when $L_i = 13 \wedge a \ge 1$ do $L_i \leftarrow 14$; $a \leftarrow a - 1$
(14,i)  when $L_i = 14$                do $L_i \leftarrow 15$; readcount $\leftarrow$ readcount - 1
(15,i)  when $L_i = 15$                do $L_i \leftarrow$ if c = 0 then 19 else 16
(16,i)  when $L_i = 16$                do $L_i \leftarrow 17$; $c \leftarrow c - 1$
(17,i)  when $L_i = 17$                do $L_i \leftarrow 18$; $e \leftarrow e + 1$
(18,i)  when $L_i = 18$                do $L_i \leftarrow 15$
(19,i)  when $L_i = 19$                do $L_i \leftarrow 1$; $a \leftarrow a + 1$

WRITER
 (1)  when $L = 1 \wedge a \ge 1$      do $L \leftarrow 2$; $a \leftarrow a - 1$
 (2)  when $L = 2$                     do $L \leftarrow 3$; writecount $\leftarrow$ writecount + 1
 (3)  when $L = 3$                     do $L \leftarrow$ if readcount = 0 then 9 else 4
 (4)  when $L = 4$                     do $L \leftarrow 5$; $c \leftarrow c + 1$
 (5)  when $L = 5$                     do $L \leftarrow 6$; $a \leftarrow a + 1$
 (6)  when $L = 6 \wedge e \ge 1$      do $L \leftarrow 7$; $e \leftarrow e - 1$
 (7)  when $L = 7 \wedge a \ge 1$      do $L \leftarrow 8$; $a \leftarrow a - 1$
 (8)  when $L = 8$                     do $L \leftarrow 3$
 (9)  when $L = 9$                     do $L \leftarrow 10$; $a \leftarrow a + 1$
```

FIGURE 12. Process H: formal representation

(continued on next page)

```
(10)  when L = 10 ∧ d ≥ 1 do L ← 11; d ← d - 1
(11)  when L = 11         do L ← 12
(12)  when L = 12         do L ← 13; d ← d + 1
(13)  when L = 13 ∧ a ≥ 1 do L ← 14; a ← a - 1
(14)  when L = 14         do L ← 15; writecount ← writecount - 1
(15)  when L = 15         do L ← if c = 0 then 19 else 16
(16)  when L = 16         do L ← 17; c ← c - 1
(17)  when L = 17         do L ← 18; e ← e + 1
(18)  when L = 18         do L ← 15
(19)  when L = 19         do L ← 1; a ← a + 1
```

FIGURE 12. Process H: formal representation

```
integer readcount, writecount, c; (initial value 0)
semaphore a,b,w,e; (initial value 1)

READER-i (1 ≤ i ≤ 3)
(1,i)   P(b);
(2,i)   P(a);
(3,i)   A_i:  if writecount ≠ 0 then
(4,i)           begin c := c + 1;
(5,i)                 V(a);
(6,i)                 P(e);
(7,i)                 P(a);
(8,i)                 goto A_i end;
(9,i)   readcount := readcount + 1;
(10,i)  V(a);
(11,i)  V(b);
(12,i)  reading is performed
(13,i)  P(a);
(14,i)  readcount := readcount - 1;
(15,i)  B_i:  if c ≠ 0 then
(16,i)          begin c := c - 1;
(17,i)                V(e);
(18,i)                goto B_i end;
(19,i)  V(a);

WRITER
(1)   P(a);
(2)   writecount := writecount + 1;
(3)   A:  if readcount ≠ 0 then
(4)         begin c := c + 1;
(5)               V(a);
(6)               P(e);
(7)               P(a);
(8)               goto A end;
(9)   V(a);
(10)  P(d);
(11)  writing is performed
(12)  V(d);
(13)  P(a);
(14)  writecount := writecount - 1;
(15)  B:  if c ≠ 0 then
(16)        begin c := c - 1;
(17)              V(e);
(18)              goto B end;
(19)  V(a);
```

FIGURE 13. Process H: informal representation

We assert that we can take $\underline{\mathcal{P} \text{ to be the process W2}}$. Since the original problem was defined informally, we cannot prove that W2 expresses the behavior of the readers and writers. However, Parnas [private communication] has stated that W2 is a valid choice for $\mathcal{P}$. A more difficult decision is: what should the relation R be? As we stated in Section 4.1, the controversy between Brinch Hansen and Courtois, Heymans, Parnas stems from their different choices of R. Therefore, we will consider two relations R. $R_H(\mathcal{D},\mathcal{P})$ if and only if

(1) there is an onto safe and deadlock free realization r from $\mathcal{D}$ to $\mathcal{P}$.

On the other hand, $R_{CHP}(\mathcal{D},\mathcal{P})$ if and only if

(2) there is an onto safe and deadlock free realization r from $\mathcal{D}$ to $\mathcal{P}$ such that if $S(\alpha)$, then $S(r(\alpha))$.

$S(\alpha)$ is true if and only if

(3) α is a semi-active release pointer-bounded timing such that for some k and all $m > k$, α_m is not in WRITER.

Informally, $S(\alpha)$ states that α is a "fair timing that satisfies the PV release mechanism and from some place on, no writer executes".

Our analysis of the second reader-writer problem consists of a discussion of the assertions:

(1) $R_{CHP}(H,W2)$ is false.

(2) $R_{CHP}(CHP,W2)$ is true, $R_{H}(CHP,W2)$ is true, and $R_{H}(H,W2)$ is true.

We will now sketch a proof that (1) is true, i.e., $R_{CHP}(H,W2)$ is false. Informally, in H it is possible for a "stream of readers to execute forever". In order to simplify the presentation of this fact, we will consider another PV process: H'. An informal representation of this process is in Figure 14. Process H' is obtained from process H by deleting actions that are extraneous to this discussion.

```
semaphore a,b; (initial value 1)
READER-1        READER-2        READER-3        WRITER
(1)  P(b);      (7)  P(b);      (13)  P(b);     (19)  P(a);
(2)  P(a);      (8)  P(a);      (14)  P(a);
(3)  V(a);      (9)  V(a);      (15)  V(a);
(4)  V(b);     (10)  V(b);      (16)  V(b);
(5)  P(a);     (11)  P(a);      (17)  P(a);
(6)  V(a);     (12)  V(a)       (18)  V(a);
```

FIGURE 14. Process H': informal representation

Define α to be the timing

1 2 3 4 13 14 15 16 7 5 $\textcircled{19}$ [$\textcircled{8}$ $\textcircled{17}$ 6 8 9 17 10

1 $\textcircled{2}$ $\textcircled{11}$ 18 2 3 11 4 13 $\textcircled{5}$ $\textcircled{14}$ 12 14 15 5 16 7]...

repeat the bracketed part forever.

Note, we have circled actions α_i such that $\alpha_i \not\in$ ready-set$(\alpha_1 \ldots \alpha_{i-1})$. The timing α is a semi-active release pointer-bounded timing; moreover, for $k \geq 10$, α_k is not in WRITER. Since H' is "similar" to H, we can find a timing β in H such that $S(\beta)$ is true. Now assume that $R_{CHP}(H,W2)$ is true. Then by the definition of R_{CHP}, for some realization r,

if $S(\beta)$, then $S(r(\beta))$;

hence, $S(r(\beta))$ is true. However, it is not hard to see that

(3) for each timing δ in W2, $S(\delta)$ is false.

Informally, in W2 it is not possible for readers to stream by forever. A detailed proof of (3) would be a major digression; therefore, we will not prove (3). Thus, we have a contradiction; and hence (1) is true.

We will now consider assertion (2). The realization r_{CHP}, from CHP to W2, is displayed in Figure 15. The realization r_H, from H to W2, is displayed in Figure 16. We assert that

(4) r_{CHP} is an onto safe deadlock free realization such that

if $S(\alpha)$, then $S(r_{CHP}(\alpha))$.

(5) r_H is an onto safe deadlock free realization.

A detailed proof of the assertions (4) and (5) would be a major digression; therefore, we will not prove them.

In summary, when we use the relation r_H, both CHP and H are solutions to the second reader-writer problem. On the other hand, when we use the relation r_{CHP}, CHP is a solution to the second reader-writer problem while H is <u>not</u> a solution.

f action in CHP	$r_{CHP}(f)$
5	1
6	2
7	3
8	4
13	5
(9,i)	(1,i)
(10,i)	(2,i)
(15,i)	(3,i)
all other actions	Λ

FIGURE 15. r_{CHP}

f action in H	$r_H(f)$
9	1
10	2
11	3
12	4
19	5
(10,i)	(1,i)
(9,i)	(2,i)
(19,i)	(3,i)
all other actions	Λ

FIGURE 16. r_H

Essentially our discussion of the second reader-writer problem has proved nothing: the key assertions (1) and (2) are unproved. However, one of our major contributions is that informal statements such as

(6) "H does not solve the second reader-writer problem"

can be stated as formal statements. Thus, in our model (6) is

(7) $R_{CHP}(H,W2)$ is false.

The fact that we do not prove (7) may be less important than the fact that we can formalize (6) and similar statements. However, we are currently preparing a paper that presents a general method of showing that a realization is onto safe and deadlock free. This paper includes proofs that r_H and r_{CHP} are onto safe and deadlock free.

4.6 IRREGULARITIES IN THE SOLUTIONS: CHP,H

As we stated in Section 4.5, r_{CHP} and r_H are both onto safe and deadlock free realizations. However, both these realizations have "irregularities": these irregularities have been noticed by Parnas [private communication] and Wodon [1972]. For example, in CHP, action (1,1) is not in the ready-set((1,2)): since mutex3[value((1,2))] = 0. Informally, READER-1 is "stopped" and yet "WRITER" is not "writing".

These irregularities can be attributed to the fact that deadlock free is very weak. We will now define a stronger condition than deadlock free; this new condition will allow us to explain the irregularities of r_{CHP} and r_H.

5. Definition. Suppose that r is a realization from $\mathcal{D}$ to $\mathcal{P}$. Then say that r is deadlock free on subprocesses provided, for each SUBPROCESS-i,

if $\text{ready-set}_{\mathcal{D}}(\alpha) \cap$ SUBPROCESS - i is empty, then $\text{ready-set}_{\mathcal{P}}(r(\alpha)) \cap$ r (SUBPROCESS - i) is empty.

Suppose that r is a realization from $\mathcal{D}$ to $\mathcal{P}$. Informally, if r is deadlock free on subprocesses, then "if no action in SUBPROCESS-i can run after executing α, then no action in r(SUBPROCESS-i) can run after executing $r(\alpha)$".

Of course, no action in SUBPROCESS-i may be able to run after executing α while some action in SUBPROCESS-i may be able to run after executing $\alpha\beta$. Essentially deadlock free on subprocesses is motivated by the requirement stated in Dijkstra [1968]: "stopping a process in its 'remainder of cycle' has no effect upon the others".

We will now compare the two concepts of deadlock free. Suppose that r is a realization from $\mathcal{Q}$ to $\mathcal{P}$. In order to avoid unnecessary complexities, suppose that for each action f in $\mathcal{P}$, there is an action g in $\mathcal{Q}$ such that $r(g)=f$. Now r is deadlock free if and only if

(1) for observable f, if $r(f) \in \text{ready-set}_{\mathcal{P}}(r(\alpha))$, then $\text{ready-set}_{\mathcal{Q}}(\alpha)$ is non-empty.

We can show that r is deadlock free on subprocesses if and only if

(2) for each SUBPROCESS-i and f an observable action in SUBPROCESS-i, if $r(f) \in \text{ready-set}_{\mathcal{P}}(r(\alpha))$, then $\text{ready-set}_{\mathcal{Q}}(\alpha) \cap$ SUBPROCESS-i is non-empty.

Therefore, deadlock free on subprocesses has substituted '$\text{ready-set}_{\mathcal{Q}}(\alpha) \cap$ SUBPROCESS-i is non-empty' for 'ready-set $(\alpha)_{\mathcal{Q}}$ is non-empty'. Note, deadlock free on subprocesses is still weaker than the converse of safe.

The irregularities of r_{CHP} and r_H can be explained in terms of the concept of deadlock free on subprocesses: neither r_{CHP} nor r_H is deadlock free on subprocesses. For example, in r_{CHP},

$\text{ready-set}_{CHP}((1,2)) \cap$ READER-1 is empty

and

$$\text{ready-set}_{w2}(r_{CHP}((1,2))) \cap r_{CHP}(\text{READER-1}) \text{ is non-empty.}$$

Note, (1,1) is in $\text{ready-set}_{w2}(r_{CHP}((1,2))) \cap r_{CHP}(\text{READER-1})$, for $r_{CHP}((1,2)) = \Lambda$ and (1,1) is in $r_{CHP}(\text{READER-1})$.

An immediate question is: can we "repair" CHP and r_{CHP} to avoid these irregularities? In fact, Wodon [1972] on page 10 states that this is possible.

4.7 THE RELATION SIMULATE

6. Definition. Suppose that r is a realization from $\mathcal{D}$ to $\mathcal{P}$. Say that r is faithful provided, for observable action f and g,

$$\text{if } r(f) = r(g), \text{ then subprocess}_{\mathcal{D}}(f,g).$$

All the realizations used implicitly in the literature are faithful. In fact, most of them satisfy,

$$\text{if } r(f) = r(g) \text{ and } f,g \text{ are observable, then } f = g.$$

7. Definition. The process $\mathcal{D}$ simulates process $\mathcal{P}$ with respect to the realization r provided,

(1) r is faithful,

(2) r is onto safe,

(3) r is deadlock free on subprocesses.

Moreover, say $\mathcal{D}$ simulates $\mathcal{P}$ provided, for some realization r, $\mathcal{D}$ simulates $\mathcal{P}$ with respect to r.

The assertion of Wodon [1972] that the irregularities of CHP and r_{CHP}

could be repaired is not correct, provided his assertion is interpreted as:

(1) there is a PV process that simulates W2.

In Section 7 we prove that (1) is false.

The stronger concept of "solve" that was mentioned in the introduction is exactly the relation simulate. <u>The rest of this paper is a detailed analysis of this relation</u>. A reasonable question is: what is the practical importance of the relation simulate? Informally, CHP does not simulate W2, because in CHP subprocesses are sometimes "stopped" when they "really should not be stopped". If CHP were used in an operating system, then it is possible that these irregularities could lead to a poor utilization of system resources. We do not claim that this is true; however, in our current state of knowledge it does seem possible.

We can also use simulate to state a relationship between PV processes and up/down. Although the set of PV processes is not a subset of the set of up/down processes, every PV process can be simulated by an up/down process.

4.8 OTHER SYNCHRONIZATION PROBLEMS

We will now define three processes, W1, WS, and BRW. Each of these processes is the process that defines the behavior of some synchronization problem. We will reference these processes in Section 7.

We will now define the process W1. This process is an up/down process; it is essentially the process used by Wodon [1972] to "solve" the "first reader-writer problem" of Courtois, Heymans, Parnas [1971]. A formal representation of W1 is in Figure 17; an informal representation of W1 is in Figure 18. Parnas [private communication] has stated that W1 is a valid choice for the process that defines the behavior of the readers and writers.

```
program counter L_1,L_2,L_3,L; (initial value 1)
semaphore a,b; (initial value 0)
WRITER
when L = 1 ∧ a + b ≥ 0  do L ← 2; b ← b - 1
when L = 2              do L ← 3
when L = 3              do L ← 1; b ← b + 1
READER-i (1 ≤ i ≤ 3)
when L_i = 1 ∧ b ≥ 0    do L_i ← 2; a ← a - 1
when L_i = 2            do L_i ← 3
when L_i = 3            do L_i ← 1; a ← a + 1
```

FIGURE 17. Process W1: formal representation

```
semaphore a,b; (initial value)
WRITER                          READER-i (1 ≤ i ≤ 3)
{a,b}: down (b);                {b}: down (a);
writing is performed            reading is performed
{ }: up (b);                    { }: up (a);
```

FIGURE 18. Process W1: informal representation

We will now define process WS. This process is an up/down process; it is essentially the process used by Wodon [1972a] to "solve" the "Five Dining Philosophers Problem" of Dijkstra [1971]. A formal representation of WS is in Figure 19; an informal representation of WS is in Figure 20. We will take WS as the process that defines the behavior of the philosophers. Since the original problem is defined informally, we cannot prove that this is true. However, we claim that WS is a reasonable choice.

```
program counter $L_0, L_1, L_2, L_3, L_4$; (initial value 1)
semaphore $S_0, S_1, S_2, S_3, S_4$; (initial value 0)
PHILOSOPHER-i $(0 \le i \le 4)$
(1,i)  when $L_i = 1 \wedge S_{i-1} + S_{i+1} \ge 0$  do $L_i \leftarrow 2$; $S_i \leftarrow S_i - 1$
(2,i)  when $L_i = 2$                                do $L_i \leftarrow 3$
(3,i)  when $L_i = 3$                                do $L_i \leftarrow 1$; $S_i \leftarrow S_i + 1$
[By convention, $S_{-1} = S_4$ and $S_5 = S_0$]
```

FIGURE 19. Process WS: formal representation

```
semaphore $S_0, S_1, S_2, S_3, S_4$; (initial value 0)
PHILOSOPHER-i $(0 \le i \le 4)$
(1,i)  $\{S_{i-1}, S_{i+1}\}$: down $(S_i)$;
(2,i)                  eat
(3,i)                  up $(S_i)$;
[By convention, $S_{-1} = S_4$ and $S_5 = S_0$]
```

FIGURE 20. Process WS: informal representation

Finally we will define the process BRW. This process is a PVmultiple process. A formal representation of BRW is in Figure 21; an informal representation is in Figure 22.

```
program counter $L_1, L_2, L_3, L$; (initial value 1)
semaphore $a_1, a_2, a_3$; (initial value 1)
semaphore $d$; (initial value 2)
READER-i $(1 \le i \le 3)$
(1,i)  when $L_i = 1 \wedge a_i \ge 1 \wedge d \ge 1$  do $L_i \leftarrow 2;\ a_i \leftarrow a_i - 1;\ d \leftarrow d - 1$
(2,i)  when $L_i = 2$                                   do $L_i \leftarrow 3$
(3,i)  when $L_i = 3$                                   do $L_i \leftarrow 1;\ a_i \leftarrow a_i + 1;\ d \leftarrow d - 1$
WRITER
(1)  when $L = 1 \wedge a_1 \ge 1 \wedge a_2 \ge 1 \wedge a_3 \ge 1$ do $L \leftarrow 2;\ a_1 \leftarrow a_1 - 1;\ a_2 \leftarrow a_2 - 1;$
                                                                     $a_3 \leftarrow a_3 - 1$
(2)  when $L = 2$                                           do $L \leftarrow 3$
(3)  when $L = 3$                                           do $L \leftarrow 1;\ a_1 \leftarrow a_1 + 1;\ a_2 \leftarrow a_2 + 1;$
                                                                     $a_3 \leftarrow a_3 + 1$
```

FIGURE 21. Process BRW: formal representation

```
semaphore $a_1, a_2, a_3$; (initial value 1)
semaphore $d$; (initial value 2)
READER-i $(1 \le i \le 3)$              WRITER
(1,i)  P({$a_i$,d});                    (1)  P({$a_1, a_2, a_3$});
(2,i)  reading is performed             (2)  writing is performed
(3,i)  V({$a_i$,d});                    (3)  V({$a_1, a_2, a_3$});
```

FIGURE 22. Process BRW: informal representation

Informally, this process defines the "behavior" of the first reader-writer problem with the additional requirement that

"at most 2 of the 3 readers can be reading at once".

We have presented this "bounded first reader-writer problem" for three reasons. First, it appears to have some practical interest. In an operating system we may wish to restrict the number of readers that can be reading at once. For instance, this restriction might be due to buffer limitations. Second, this process is used in Section 7 to show that there are differences between the predicate systems: PVchunk, PVmultiple, and up/down. Third, this process shows that small changes in the way an informal problem is translated into a formal process can make a great deal of difference. Consider the process BRW' defined in Figure 23.

```
semaphore a1,a2,a3; (initial value 1)
semaphore d; (initial value 2)
READER-i (1 ≤ i ≤ 3)           WRITER
P({d});                        P({a1,a2,a3});
P({a1,a2});                    writing performed
reading is performed           V({a1,a2,a3});
V({a1,a2});
V({d});
```

FIGURE 23. BRW': informal representation

In process BRW', "entering the reading section" is divided into two "steps", while in BRW entering the reading section is one step. We can show that

(1) no up/down process simulates BRW and

(2) an up/down process simulates BRW'.

(1) is proved in Section 7; (2) follows since the up/down process represented in Figure 24 simulates BRW'. In an operating system whether we use BRW or BRW' may be immaterial - we just do not know.

```
semaphore d; (initial value 1)
semaphore $a_1$,$a_2$,$a_3$,b; (initial value 0)
READER-i ($1 \le i \le 3$)           WRITER
{d}: down (d);                       {$a_1$,$a_2$,$a_3$}: down (b);
  b: down ($a_i$);                       writing is performed
  reading is performed                       up (b);
         up ($a_i$);
         up (d);
```

FIGURE 24. An up/down process: informal representation

5. INVARIANCE OF LOCAL BEHAVIOR OF A PROCESS

We are interested in the relation 'simulate'. We will now state a necessary condition for $\mathfrak{Q}$ to simulate $\mathfrak{P}$. In order to state this condition, we will first formalize the concept of the "local behavior" of a process. We will then state the invariance theorem:

if $\mathfrak{Q}$ simulates $\mathfrak{P}$ and $\mathfrak{Q}$ is a C process, then any local behavior of $\mathfrak{P}$ is the local behavior of some C process.

The invariance theorem (proved in section 6) will be used later, among other things, to prove the results stated in the introduction, i.e., Figure 1.

5.1 LOCAL BEHAVIOR OF A PROCESS

1. Definition. Suppose that Σ is a finite set. The set Π is a Σ-slice provided

(1) each element of Π is a finite sequence of distinct elements from Σ;

(2) each element of Σ is in Π;

(3) if $\alpha\beta$ is in Π, then α is in Π.

2. Definition. The process $\mathfrak{P} = \langle A, \mathfrak{J}, w\rangle$ defines the Σ-slice Π provided there is a one to one correspondence d from A to Σ such that

$$d\{\alpha \mid \alpha \text{ is an active timing in } \mathfrak{P}\} = \Pi.$$

Note, we extend d to finite timings of ρ by defining $d(\alpha_1 \dots \alpha_k)$ to be $d(\alpha_1) \dots d(\alpha_k)$.

Consider the up/down process M that is informally represented by

<u>semaphore</u> a,s; (initial value 0)

SUBPROCESS-1	SUBPROCESS-2
(1) {s}: down (a);	(2) { }: down (s);

The set of active timings of M is $\{\Lambda,1,2,12\}$: 21 is not an active timing because s[value(2)] = -1. Thus, M defines the $\{x,y\}$-slice $\{\Lambda,x,y,xy\}$. Informally, the slice $\{\Lambda,x,y,xy\}$ represents the "behavior" where

y "stops" x and x does not stop y.

Thus, action 2 stops action 1 and action 1 does not stop action 2.

3. <u>Definition</u>. The process $\langle A,\mathfrak{Y},w\rangle$ <u>implicitly defines</u> the Σ-slice Π provided there is a subset A' of A and a finite active timing α in $\langle A,\mathfrak{Y},w\rangle$ such that $\langle A',\mathfrak{Y},\text{value}(\alpha)\rangle$ defines Π.

This relation, implicitly defines, allows us to do two things. First, we can focus our attention on a part of the process, i.e., we can use A' instead of A. Second, we can focus our attention on the process after some timing α has "executed", i.e., we can use value(α) instead of w.

Consider the up/down process W2 - introduced earlier - that is informally represented by

```
semaphore a,b,s; (initial value 0)
READER-i  (1 ≤ i ≤ 3)                 WRITER
                                      (1) { }: down (s);
(1,i) {s}: down (a);                  (2) {a,b}: down (b);
(2,i) reading is performed            (3) writing is performed
(3,i) { }: up (a);                    (4) { }: up (b);
                                      (5) { }: up (s);
```

Let W2 = $<A,\mathfrak{D},w>$. Since $<\{(1,1),1\},\mathfrak{D},w>$ defines $\{\Lambda,x,y,xy\}$, W2 implicitly defines $\{\Lambda,x,y,xy\}$. We can therefore say that the "local behavior" of w2 contains the "behavior" where

y stops x and x does not stop y.

Note, in contrast to process M introduced earlier, process W2 does not define $\{\Lambda,x,y,xy\}$.

Different slices represent different kinds of local behavior. For example, the slice $\{\Lambda,x,y,z,xy,yx\}$ represents the behavior where

x and y stop z, z stops x and y, but x and y do not stop each other.

5.2 INVARIANCE THEOREM

4. Definition. Say that the predicate system C defines the Σ-slice Π provided there is a C process $\mathfrak{D}$ such that $\mathfrak{D}$ defines Π.

5. Definition. Suppose that C and C' are predicate systems. Say that C → C' provided there is a C process that cannot be simulated by a C' process.

6. Theorem [Invariance Theorem]. If $\mathfrak{Q}$ simulates $\mathfrak{P}$, $\mathfrak{Q}$ is a C process, and $\mathfrak{P}$ implicitly defines the Σ-slice Π, then C defines Π.

Proof. This theorem is proved in Section 6. □

7. Corollary. If $\mathfrak{P}$ implicitly defines Π and C does not define Π, then no C process can simulate $\mathfrak{P}$.

Proof. Immediate from the invariance theorem. □

Informally, Corollary 7 states that if no C process can "handle the local behavior of $\mathfrak{P}$", then no C process can simulate $\mathfrak{P}$.

8. Corollary. If C defines Π and C' does not define Π, then $C \rightarrow C'$.

Proof. Suppose that C defines Π and C' does not define Π. Let $\mathfrak{P}$ be a C process that defines Π. Assume that $C \rightarrow C'$ is false; let $\mathfrak{Q}$ be a C' process that simulates $\mathfrak{P}$. Since $\mathfrak{P}$ implicitly defines Π, by the invariance theorem, C' defines Π. This is a contradiction; hence, $C \rightarrow C'$. □

Corollary 8 is the principle tool we use when we compare different predicate systems.

We will now present an informal application of the invariance theorem. Consider the up/down process W2 introduced earlier. We will now informally show that no PV process can define the slice $\{\Lambda,x,y,xy\}$. Since W2 implicitly defines $\{\Lambda,x,y,xy\}$, by Corollary 7, no PV process can simulate W2. This is a non-trivial result, for we allow unobservable actions and conditionals.

Suppose that $\mathfrak{Q}$ is a PV process, and suppose that $\mathfrak{Q}$ defines $\{\Lambda,x,y,xy\}$.

Let f and g be the two actions of 𝔇. Then we can suppose that

(1) ready-set(Λ) = $\{f,g\}$,

(2) ready-set(f) = $\{\ \}$,

(3) ready-set(g) = $\{f\}$.

By (1) and (2), g must be a P on some semaphore S. By (2), f must also be a P on the semaphore S; otherwise, the result of "executing" f could not move g out of the ready-set. Since both f and g are P's on the semaphore S, f is not in ready-set(g). However, this contradicts (3). Thus, no PV process can define the slice $\{\Lambda,x,y,xy\}$. A formal proof of this result will be supplied in Section 7.

An interesting facet of this argument is that we have reduced the *a priori* hard question, of whether or not a process exists that simulates another process, to a simple "combinatorial question". In effect, all the work has been done in proving the invariance theorem.

6. A PROOF OF THE INVARIANCE THEOREM

We now present a proof of the invariance theorem. The proof uses only the properties I-IV of a C process.

1. Definition. Suppose that r is a realization from the process $\mathfrak{D}$ to the process $\mathfrak{P}$. Then the finite timing α in $\mathfrak{D}$ is in canonical form under r provided, either $\alpha = \Lambda$ or

$$\alpha = \beta^1 f^1 \ldots \beta^k f^k$$

where

(1) for each i, f^i is observable,

(2) for each i, each action in β^i is unobservable,

(3) for each i, if g is an action in β^i, then subprocess (f^i, g).

When there can be no confusion we will delete 'under r'.

2. Theorem. Suppose that r is a realization from the process $\mathfrak{D}$ to the process $\mathfrak{P}$. Then

(1) if $\alpha\beta$ is in canonical form, then β is in canonical form

(2) if αf is in canonical form, then f is an observable action.

Proof. Immediate from the definition of canonical form. □

Suppose that $\mathfrak{D}$ simulates $\mathfrak{P}$ with respect to the realization r. By the definition of simulates, if α is a finite active timing in $\mathfrak{P}$, then the set

$$\{\beta \mid \beta \text{ is a finite active timing in } \mathfrak{D} \text{ and } r(\beta) = \alpha\}$$

is non-empty. Many of the timings in this set are complex. Our first goal is to prove that this set contains a unique timing in canonical form.

For the rest of this section we assume that

(1) $\mathfrak{D}$ is a C process and

(2) $\mathfrak{D}$ simulates $\mathcal{P}$ with respect to the realization r.

3. Theorem. Suppose that $r(\alpha\beta)$ is active, β is in canonical form, and β_1 is in pointer-set$_{\mathfrak{D}}(\alpha)$. Then β_1 is in ready-set$_{\mathfrak{D}}(\alpha)$.

Proof. Assume that β_1 is not in ready-set$_{\mathfrak{D}}(\alpha)$. Suppose that β_1 is in SUBPROCESS-i. By properties I and II,

(1) ready-set$_{\mathfrak{D}}(\alpha)$ $\cap$ SUBPROCESS-i is empty.

Define β_k to the first observable action in β. By the definition of canonical form, β_k is in SUBPROCESS-i. Now $r(\alpha\beta_1 \ldots \beta_k) = r(\alpha)r(\beta_k)$. Since $r(\alpha\beta)$ is active, $r(\beta_k)$ is in ready-set$(r(\alpha))$. Thus,

(2) ready-set$_{\mathcal{P}}(r(\alpha))$ $\cap$ r(SUBPROCESS-i) is non-empty.

By the definition of simulate, r is deadlock free on subprocesses. This is a contradiction with (1) and (2). Therefore, β_1 is in ready-set$_{\mathfrak{D}}(\alpha)$. □

4. Theorem. Suppose that $\alpha f\beta$ is an active timing in $\mathfrak{D}$; f is unobservable; β is in canonical form; and for each action g in β, not subprocess (f,g). Then $\alpha\beta$ is active.

Proof. Assume that $\alpha\beta$ is not active. Suppose that $\beta = \lambda\mu$ where $\alpha\lambda$ is active and $\alpha\lambda\mu_1$ is not active. By property I, μ_1 is in pointer-set$_{\mathfrak{D}}(\alpha f\lambda)$. By property III, μ_1 is in pointer-set$_{\mathfrak{D}}(\alpha\lambda)$. By the definition of simulate, $r(\alpha f\beta)$ is active. Since $r(\alpha f\beta) = r(\alpha\beta)$, $r(\alpha\beta)$ is active. Also μ is in canonical form. Thus, by Theorem 3, μ_1 is in ready-set$_{\mathfrak{D}}(\alpha\lambda)$. This is a contradiction, and hence $\alpha\beta$ is active. □

5. Theorem. Suppose that $\alpha f\beta\delta$ is an active timing in $\mathfrak{D}$; f is unobservable; $\beta\delta$ is in canonical form; subprocess (f,δ_1); and for each action g in β, not subprocess (f,g). Then $\alpha\beta f\delta$ is active.

Proof. Clearly, β and $f\delta$ are in canonical form. By Theorem 4 $\alpha\beta$ is active. First, we will show that $\alpha\beta f$ is active. By property I, f is in pointer-set$_{\mathfrak{D}}(\alpha)$. By property III, f is in pointer-set$_{\mathfrak{D}}(\alpha\beta)$. By the definition of simulate, $r(\alpha f\beta\delta)$ is active. Since $r(\alpha f\beta\delta) = r(\alpha\beta f\delta)$, $r(\alpha\beta f\delta)$ is active. Also $f\delta$ is in canonical form. Thus, by Theorem 3 f is in ready-set$_{\mathfrak{D}}(\alpha\beta)$; and hence $\alpha\beta f$ is active. Second, we will show that $\alpha\beta f\delta$ is active. Assume that $\alpha\beta f\delta$ is not active. Suppose that $\delta = \lambda\mu$ where $\alpha\beta f\lambda$ is active and $\alpha\beta f\lambda\mu_1$ is not active. By property I, μ_1 is in pointer-set$_{\mathfrak{D}}(\alpha f\beta\lambda)$. By property IV, μ_1 is in pointer-set$_{\mathfrak{D}}(\alpha\beta f\lambda)$. Also μ is in canoncial form. Thus, by Theorem 3 μ_1 is in ready-set$(\alpha\beta f\lambda)$. This is a contradiction, and hence $\alpha\beta f\delta$ is active. □

6. Theorem. If λ is a finite active timing in $\mathfrak{P}$, then there is a finite active timing μ in $\mathfrak{D}$ such that $r(\mu) = \lambda$ and μ is in canonical form.

Proof. Suppose that λ is a finite active timing in $\mathfrak{P}$. Since r is onto safe, there is a finite active timing μ in $\mathfrak{D}$ such that $r(\mu) = \lambda$. If μ

is not in canonical form, then either

(1) $\mu = \alpha f \beta$ where the hypothesis of Theorem 4 is true, or

(2) $\mu = \alpha f \beta \delta$ where the hypothesis of Theorem 5 is true.

In case (1), apply Theorem 4 to μ; in case (2) apply Theorem 5 to μ. As long as the resulting timing is not in canonical form, continue to apply either Theorem 4 or Theorem 5 We cannot apply these theorems forever, for

(3) Theorem 4 removes an action and

(4) Theorem 5 moves an action and never moves it again.

Call the timing obtained this way μ'. Now μ' is active and in canonical form. Since Theorem 4 and Theorem 5 only delete or move unobservable actions, $r(\mu') = r(\mu) = \lambda$. □

7. <u>Theorem</u>. Suppose that $\alpha\lambda$ and $\alpha\mu$ are active in $\mathfrak{D}$, each action in λ is in SUBPROCESS-i, and each action in μ is in SUBPROCESS-i. Then $\lambda \leq \mu$ or $\mu \leq \lambda$.

<u>Proof</u>. This follows immediately from Theorem 14 of Section 2.10. □

8. <u>Theorem</u>. Suppose that $r(\alpha) = r(\beta)$ where α and β are active and in canonical form. Then $\alpha = \beta$.

<u>Proof</u>. We will use induction on the length of α. If $\alpha = \Lambda$, then $\beta = \Lambda$; hence, in this case $\alpha = \beta$. Now suppose that $\alpha \neq \Lambda$. As in Definition 1, let $\alpha = \alpha^1 f^1 \ldots \alpha^k f^k$ and $\beta = \beta^1 g^1 \ldots \beta^k g^k$ where $k \geq 1$. Since r is <u>faithful</u>, for all i, subprocess (f^i, g^i). By induction hypothesis,

(1) $\alpha^1 f^1 \ldots \alpha^{k-1} f^{k-1} = \beta^1 g^1 \ldots \beta^{k-1} g^{k-1}$.

By Theorem 7 and symmetry, we can assume that $\alpha^k f^k \leq \beta^k g^k$. Since f^k is observable and each action in β^k is unobservable, $\alpha^k f^k = \beta^k g^k$. Thus, $\alpha = \beta$. □

9. Theorem. Suppose that α is a finite active timing in $\mathcal{P}$. Then there is a unique finite active timing β in canonical form, in $\mathcal{D}$, such that $r(\beta) = \alpha$.

Proof. The existence of β is Theorem 6; the uniqueness of β is Theorem 8. □

We have, so far, achieved our first goal: we have shown that if α is a finite active timing in $\mathcal{P}$, then the set

$$\{\beta \mid \beta \text{ is a finite active timing in } \mathcal{D} \text{ and } r(\beta) = \alpha\}$$

contains exactly one timing in canonical form. We will now prove a theorem that allows us to "piece together" different active timings to form one active timing.

10. Theorem. Suppose that $\alpha\beta^i f^i$ $(1 \leq i \leq n)$ is a finite active timing in $\mathcal{D}$ which is in canonical form. Also suppose that

(1) for each i, each action in β^i is unobservable,

(2) for $i \neq j$, $f^i \neq f^j$.

Then $\alpha\beta^1 \ldots \beta^n f^i$ is active $(1 \leq i \leq n)$.

Proof. We assert that for $i \neq j$, not subprocess (f^i, f^j). For suppose that subprocess (f^i, f^j). Then by the definition of canonical form and Theorem 7, we can assume without any loss of generality that $\beta^i f^i \leq \beta^j f^j$.

By (1), $\beta^i f^i = \beta^j f^j$; and hence, $i = j$.

We will now prove the following lemma.

<u>Lemma</u>. If $\delta^i \leq \beta^i$ $(1 \leq i \leq n)$, then $\alpha\delta^1 \ldots \delta^n$ is active.

<u>Proof of Lemma</u>. Proof by induction on the length of $\alpha\delta^1 \ldots \delta^n$. Clearly, α is active. Now there are two cases. First, $\alpha\delta^1 \ldots \delta^n = \alpha\mu$ where each action in μ is in SUBPROCESS-i, for some i. Then $\mu \leq \beta^j$ for some j. Thus, $\alpha\mu$ is active. Second,

$$\alpha\delta^1 \ldots \delta^n = \alpha\lambda h\mu g$$

where each action in μg is in SUBPROCESS-i, for some i, and h is not in SUBPROCESS-i. By induction hypothesis, $\alpha\lambda h\mu$ and $\alpha\lambda\mu g$ are active. By properties I and III, g is in pointer-set$_{\mathfrak{I}}(\alpha\lambda h\mu)$. Let $\mu g \leq \beta^k$. Since r is safe, $r(\alpha\beta^k f^k)$ is active. Since $r(\alpha\beta^k f^k) = r(\alpha\lambda h\mu g f^k)$, $r(\alpha\lambda h\mu g f^k)$ is active. Also, gf^k is in canonical form. Thus, by Theorem 3, g is in ready-set$_{\mathfrak{I}}(\alpha\lambda h\mu)$. Therefore, $\alpha\delta^1 \ldots \delta^n$ is active. □

By the lemma, $\alpha\beta^1 \ldots \beta^n$ is active. By property I, f^i is in pointer-set$_{\mathfrak{I}}(\alpha\beta^i)$. By the lemma, there is a list of active timings

$$\begin{array}{l} \alpha\beta^i \\ \vdots \\ \alpha\beta^1 \ldots \beta^n \end{array}$$

such that each timing is obtained from the one above by the insertion of one action. By repeated applications of property III, f^i is in pointer-set$_{\mathfrak{I}}(\alpha\beta^1 \ldots \beta^n)$. Since $r(\alpha\beta^1 \ldots \beta^n f^i) = r(\alpha\beta^i f^i)$, $r(\alpha\beta^1 \ldots \beta^n f^i)$ is active. Thus, by Theorem 3, f^i is in ready-set$_{\mathfrak{I}}(\alpha\beta^1 \ldots \beta^n)$. Therefore, $\alpha\beta^1 \ldots \beta^n f^i$ is active. □

11. <u>Theorem</u>. Suppose that A is a subset of ready-set(δ) with δ active. Then there is a B and a μ such that B is a subset of ready-set$_{\mathcal{D}}(\mu)$, μ is active, $r(\mu) = \delta$, and r is a one to one correspondence from B to A. Note, each action in B is observable.

<u>Proof</u>. Let $A = \{g^1,\ldots,g^n\}$. By Theorem 9, there exists $\alpha,\lambda^1,\ldots,\lambda^n$ such that for $1 \le i \le n$,

(1) $r(\alpha\lambda^i) = \delta g^i$,

(2) $\alpha\lambda^i$ is active and in canonical form, and

(3) $r(\lambda^i) = g^i$.

The existence of these timings depends essentially on the uniqueness part of Theorem 9. Define $\beta^i f^i = \lambda^i$, for $1 \le i \le n$. If $f^i = f^j$, then $r(\lambda^i) = r(\lambda^j)$; hence, $i = j$. Thus, if $i \neq j$, then $f^i \neq f^j$. Define $\mu = \alpha\beta^1\ldots\beta^n$, and define $B = \{f^1,\ldots,f^n\}$. Clearly, $r(\mu) = \delta$, and r is a one to one correspondence from B to A. By Theorem 10, for $1 \le i \le n$, $\alpha\beta^1\ldots\beta^n f^i$ is active. Therefore, B is a subset of ready-set$_{\mathcal{D}}(\mu)$. $\square$

Theorem 11 has an intuitive interpretation. Suppose that g^1 and g^2 are actions in ready-set$_\rho(\delta)$. Informally, we can say that g^1 and g^2 are "parallel at δ". Then Theorem 11 shows that there exist actions f^1 and f^2 and a timing μ such that

(1) f^1 and f^2 are both in ready-set$_{\mathcal{D}}(\mu)$,

(2) $r(\mu) = \delta$,

(3) $r(f^1) = g^1$ and $r(f^2) = g^2$.

Informally, f^1 and f^2 are parallel at μ. Since $r(\mu) = \delta$, $r(f^1) = g^1$, and

$r(f^2) = g^2$, we can say that the "parallel structure of process $\mathcal{P}$ is reflected in the parallel structure of process $\mathcal{Q}$".

We are now in a position to prove the invariance theorem. Recall that $\mathcal{Q}$ is a C process. The invariance theorem states that

if $\mathcal{P}$ implicitly defines the Σ-slice Π, then C defines Π.

Informally, since $\mathcal{Q}$ simulates $\mathcal{P}$, the predicate system C must be able to define all the local behavior of the process $\mathcal{P}$.

12. Theorem [Invariance Theorem] If $\mathcal{P}$ implicitly defines Π a Σ-slice, then C defines Π.

Proof. Let $\mathcal{P} = \langle A,\mathfrak{J},w\rangle$ implicitly define Π a Σ-slice; let $\mathcal{Q} = \langle B,\mathfrak{J}',x\rangle$. Since $\mathcal{P}$ implicitly defines Π, there exists a A_0 a subset of A and an active finite timing α such that $\langle A_0,\mathfrak{J},\text{value}(\alpha)\rangle$ defines Π. Also let d be the one to one correspondence between A_0 and Σ.

Suppose that g is in A_0. Then by the definition of a slice, d(g) is in Π. Thus, g is active in $\langle A_0,\mathfrak{J},\text{value}(\alpha)\rangle$; and, hence g is in ready-set$_{\mathcal{P}}(\alpha)$. By Theorem 11, there exists a finite active timing β in $\mathcal{Q}$ and a subset B_0 of ready-set$_{\mathcal{Q}}(\beta)$ such that $r(\beta) = \alpha$ and r is one to one correspondence from B_0 to A_0. We now assert that $\langle B_0,\mathfrak{J}',\text{value}(\beta)\rangle$ defines Π. The one to one correspondence from B_0 to Σ is the composition of d and r.

Suppose that δ is active in $\langle B_0,\mathfrak{J}',\text{value}(\beta)\rangle$. Then $\beta\delta$ is active in $\mathcal{Q}$. By the definition of simulates, $r(\beta\delta)$ is active; and hence $r(\delta)$ is active in $\langle A_0,\mathfrak{J},\text{value}(\alpha)\rangle$. Thus, $d(r(\delta))$ is in Π.

On the other hand, suppose that δ is a finite timing in $\langle B_0,\mathfrak{D}',\text{value}(\beta)\rangle$ and $d(r(\delta))$ is in Π. Since $\langle A_0,\mathfrak{D},\text{value}(\alpha)\rangle$ defines Π, $r(\delta)$ is active in $\langle A_0,\mathfrak{D},\text{value}(\alpha)\rangle$; and hence $\alpha r(\delta)$ is active in $\mathcal{P}$. We now assert that

(1) if $i \neq j$, then not subprocess (δ_i,δ_j).

For suppose that subprocess (δ_i,δ_j). Since δ_i and δ_j are in ready-set$_{\mathfrak{D}}(\beta)$, $\delta_i = \delta_j$ by property II. Since $d(r(\delta))$ is in Π, $i = j$ by the definition of slice; hence, (1) is true. We now assert that $\beta\delta$ is active. Suppose that $\delta = \lambda\mu$ and $\beta\lambda$ is active and $\beta\lambda\mu_1$ is not active. Now $r(\beta\delta)$ is active, and δ is in canonical form. By property I, μ_1 is in pointer-set$_{\mathfrak{D}}(\beta)$. By property III and (1), μ_1 is in pointer-set$_{\mathfrak{D}}(\beta\lambda)$. Thus, by Theorem 3, $\beta\lambda\mu_1$ is active. This is a contradiction, and hence $\beta\delta$ is active. Therefore, δ is active in $\langle B_0,\mathfrak{D}',\text{value}(\beta)\rangle$.

We have, therefore, shown that $\langle B_0,\mathfrak{D}',\text{value}(\beta)\rangle$ defines Π. Clearly, this is a C process; and thus, C defines Π. □

7. ANALYSIS OF SLICES

We now study questions of the form, does the predicate system C define the slice Π? In general, these questions are very complex. Therefore, in order to get interesting results we actually study not all slices but restricted classes of slices. However, these classes of slices help us to show that there are differences between the predicates systems: PV, PVchunk, PVmultiple, and up/down. In particular, we are able to prove the results stated in the introduction.

7.1 A PROPERTY OF PV, PVCHUNK, PVMULTIPLE

1. Theorem. Suppose that ρ is a PV (respectively PVchunk or PVmultiple) process. Also suppose that f is an action in ρ that is not a P. Then for any finite timing α,

f is in ready-set(α) iff f is in pointer-set(α).

Proof. Immediate from the definition of a C process and the definitions of the predicate systems: PV, PVchunk, PVmultiple. □

2. Theorem. Suppose that ρ is a PV (respectively PVchunk or PVmultiple) process. Also suppose that not subprocess (f,g). Then

(1) If fg is active and g is a V, then gf is active.

(2) If fg is active and f is a P, then gf is active.

Proof. We will prove the theorem for the case where ρ is a PV process; the other cases follow in a similar manner. Suppose that $\rho = \langle A,\mathfrak{M},w\rangle$.

Let '$(L_1,\ldots,L_n,D,(S_1,\ldots,S_m))$' be a typical element of $\mathfrak{D}$.

Suppose that fg is active, g is a V, and not subprocess (f,g). By property I and the fact that g is in ready-set(f), g is in pointer-set(f). By property III, g is in pointer-set(Λ). By Theorem 1, g is active. Now assume that gf is not active. Then f is not in ready-set(g). Again by properties I and III, f is in pointer-set(g). Therefore, by Theorem 1, f is a P. We can therefore assume that

(3) $f = \underline{\text{when}}\ L_i = a \wedge S_j > 0\ \underline{\text{do}}\ L_i \leftarrow a';\ S_j \leftarrow S_j - 1.$

Since f is in ready-set(Λ), $S_j[w] > 0$. Since g is a V, $S_j[g(w)] \geq S_j[w]$. This is a contradiction, and hence (1) is true.

Now suppose that fg is active, f is a P, and not subprocess (f,g). By (1), we can assume that g is a P. We can therefore assume without loss of generality that

(4) $f = \underline{\text{when}}\ L_1 = a \wedge S_i > 0\ \underline{\text{do}}\ L_1 \leftarrow a';\ S_i \leftarrow S_i - 1$

(5) $g = \underline{\text{when}}\ L_2 = b \wedge S_j > 0\ \underline{\text{do}}\ L_2 \leftarrow b';\ S_j \leftarrow S_j - 1.$

By properties III and I, g is in pointer-set(Λ). Since g is in ready-set(f), $S_j[f(w)] > 0$. Since f is a P, $S_j[w] \geq S_j[f(w)]$. Hence, g is active. By properties III and I again, f is in pointer-set(g). Since f is in ready-set(Λ), $S_i[w] > 0$. Assume that gf is not active; then $S_i[g(w)] \leq 0$. Hence, i = j; and $S_i[w] = 1$. Therefore, $S_j[f(w)] = 0$. But this is a contradiction, for g is in ready-set(f). Thus, (2) is true. □

We can extend the proof of Theorem 2 to prove the following. Suppose $\mathcal{P}$ is a PV (respectively PVchunk or PVmultiple) process. Also suppose that not subprocess (f,g). Then

(1) if $\alpha fg\beta$ is active and g is a V, then $\alpha gf\beta$ is active

(2) if $\alpha fg\beta$ is active and f is a P, then $\alpha gf\beta$ is active.

This is a fundamental property of what we could call "PV like" predicate systems.

7.2 EXCLUSION SLICES

3. Definition. Suppose that R is a reflexive relation on a finite set Σ. Define α in exclusion (R) iff

(1) α is a finite sequence of elements from Σ and

(2) for $1 \le i < j \le \text{length}(\alpha)$, not $\alpha_i R \alpha_j$.

Informally, we think of aRb as meaning: "a stops b" or "a excludes b".

4. Theorem. Suppose that R is a reflexive relation on a finite set Σ. Then exclusion (R) is a Σ-slice. A slice that can be defined in this way will be called an exclusion slice.

Proof. Immediate from the definition of Σ-slice. $\Box$

Not all slices are exclusion slices. Consider the {a,b,c}-slice {Λ,a,b,c,ac,acb}. This is not an exclusion slice. For suppose that it is an exclusion slice. Since acb is in the slice, by the definition of exclusion slice, ab must also be in the slice. This is a contradiction, and hence not all slices are exclusion slices.

Let $\Sigma = \{r_1, r_2, w\}$, and let $R = \{(r_1,w),(w,r_1),(r_2,w),(w,r_2), (w,w), (r_1,r_1),(r_2,r_2)\}$. Then exclusion $(R) = \{\Lambda, w, r_1, r_2, r_1r_2, r_2r_1\}$. Informally, we can consider w as a "writer", and we can consider r_1 and r_2 as "readers". Then writer excludes readers, and readers exclude writer. However, readers do not exclude each other. Note, the relation R is non-transitive.

We can represent exclusion (R) as a directed graph. The nodes are the elements of Σ. An arrow goes from node a to node b iff aRb. Since R is always reflexive, we will drop all the arrows from a node to itself. Thus, the exclusion slice defined above is

$$\begin{tikzcd}
r_1 \arrow[r, leftrightarrow] & w \arrow[d, leftrightarrow] \\
 & r_2
\end{tikzcd}$$

We will now consider some of the exclusion slices implicitly defined by the processes W1, W2, and WS.

(1) W1. This process implicitly defines the exclusion slice represented by the directed graph

$$\begin{tikzcd}
r_1 \arrow[r, leftrightarrow] & w \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & r_3 \\
 & r_2 &
\end{tikzcd}$$

Note, this corresponds to a non-transitive symmetric relation. Informally, w is a "writer"; and r_1, r_2, r_3 are "readers". The relation is non-transitive because readers do not exclude each other.

(2) W2. This process implicitly defines the exclusion slice represented by the directed graph

$$\begin{tikzcd} w \arrow[r] & r \end{tikzcd}$$

Note, this corresponds to a non-symmetric relation. Informally, w is a "writer" and r is a "reader". The relation is non-symmetric because a writer can exclude a reader, while a reader cannot exclude a writer.

(3) WS. This process implicitly defines the exclusion slice represented by the directed graph

$$\begin{tikzcd}
 & 1 \arrow[rr, leftrightarrow] \arrow[dl, leftrightarrow] & & 2 \arrow[dr, leftrightarrow] & \\
3 \arrow[drr, leftrightarrow] & & & & 4 \arrow[dll, leftrightarrow] \\
 & & 5 & &
\end{tikzcd}$$

Note, this corresponds to a non-transitive symmetric relation. Informally, each node is a "philosopher". The relation is non-transitive because each philosopher only excludes his adjacent neighbors.

7.3 RESTRICTIVE RESULTS

5. Theorem. Suppose that a PV (respectively a PVchunk or PVmultiple) process defines exclusion (R). Then R is symmetric.

Proof. Suppose that R is a reflexive, non-symmetric relation on Σ. Also let d be the one to one correspondence from the actions of the process to Σ. Suppose that aRb and not bRa. Let $f = d^{-1}(a)$ and $g = d^{-1}(b)$ where d^{-1} is the inverse of d. By the definition of defines, gf is active and fg is not active. By Theorem 2, g is a V. By property II and the fact that f and g are active timings, not subprocess (f,g). Thus, by properties III and I, g is in pointer-set(f). Therefore, by Theorem 1, fg is active. This is a contradiction. □

6. <u>Theorem</u>. Suppose that a PV process $\wp$ defines exclusion (R). Then R is an equivalence relation.

<u>Proof</u>. Let R be a reflexive relation on Σ, and let $\wp = \langle A, \mathfrak{J}, w\rangle$. Let '$(L_1,\ldots,L_n,D,(S_1,\ldots,S_m))$' be a typical element of $\mathfrak{J}$. Let d be the one to one correspondence from A to Σ. Also let d^{-1} be the inverse of d.

By Theorem 5, R is symmetric. Now assume that R is not transitive, i.e., suppose that a_1Ra_2, a_2Ra_3, and not a_3Ra_1. Let $f = d^{-1}(a_1)$, $g = d^{-1}(a_2)$, and $h = d^{-1}(a_3)$. By the definition of defines and exclusion (R),

(1) f,g,h,fh,hf are active; fg,hg,gf,gh are not active.

By property II, not subprocess (f,g). By properties III and I, g is in pointer-set(f). Therefore by Theorem 1 and the fact that fg is not active, g is a P. In a similar manner, we can show that f and h are P's. Therefore, we can assume without loss of generality that

$$f = \underline{\text{when}}\ L_1 = a_1 \wedge S_i > 0\ \underline{\text{do}}\ L_1 \leftarrow a_1'; S_i \leftarrow S_i - 1$$
$$g = \underline{\text{when}}\ L_2 = a_2 \wedge S_j > 0\ \underline{\text{do}}\ L_2 \leftarrow a_2'; S_j \leftarrow S_j - 1$$
$$h = \underline{\text{when}}\ L_3 = a_3 \wedge S_k > 0\ \underline{\text{do}}\ L_3 \leftarrow a_3'; S_k \leftarrow S_k - 1.$$

Since gf is not active, $S_i[g(w)] \le 0$. Since f is active, $S_i[w] > 0$. Thus, $i = j$. By the same reasoning, $j = k$, and hence $i = j = k$. Since fh is active, $S_i[w] \ge 2$. Thus, $S_i[g(w)] \ge 1$, and hence gf is active.

This is a contradiction; therefore, R is transitive. Finally, a reflexive, symmetric, transitive relation is an equivalence relation. $\square$

7. Definition. Suppose that f is an action in a PVchunk process. Also suppose that synchronizer(f) and that the pair of f is P(S with amount m). Then define amount(f) to be m.

8. Theorem. Suppose that a PVchunk process $\mathcal{P}$ defines exclusion (R). Also suppose that d is the one to one correspondence between the actions of $\mathcal{P}$ and Σ, where R is a realtion on Σ. Suppose further that a_1Ra_2, a_2Ra_3, and not a_3Ra_1. Then (d^{-1} is the inverse of d)

$$\text{amount}(d^{-1}(a_2)) > \text{amount}(d^{-1}(a_1)).$$

Proof. Let $\mathcal{P} = \langle A,\mathfrak{M},w\rangle$. Let '$(L_1,\ldots,L_n,D,(S_1,\ldots,S_m))$' be a typical element of $\mathfrak{M}$. Let $f = d^{-1}(a_1)$, $g = d^{-1}(a_2)$, and $h = d^{-1}(a_3)$. By the same reasoning in Theorem 6, we can assume that

$$f = \underline{\text{when}}\ L_1 = a_1 \wedge S_i \geq b_1\ \underline{\text{do}}\ L_1 \leftarrow a_1'; S_i \leftarrow S_i - b_1$$
$$g = \underline{\text{when}}\ L_2 = a_2 \wedge S_j \geq b_2\ \underline{\text{do}}\ L_2 \leftarrow a_2'; S_j \leftarrow S_j - b_2$$
$$h = \underline{\text{when}}\ L_3 = a_3 \wedge S_k \geq b_3\ \underline{\text{do}}\ L_3 \leftarrow a_3'; S_k \leftarrow S_k - b_3$$

where $b_1 > 0$, $b_2 > 0$, and $b_3 > 0$.

Since gf is not active, $S_i[g(w)] < b_1$. Now since f is active, $S_i[w] \geq b_1$. Hence, $i = j$. By the same reasoning, $j = k$; and hence, $i = j = k$. Since fh is active, $S_i[w] \geq b_1 + b_3$. Since gh is not active, $S_i[w] < b_2 + b_3$. Therefore, $b_2 > b_1$. Since $\text{amount}(d^{-1}(a_2)) = b_2$ and $\text{amount}(d^{-1}(a_1)) = b_1$, the theorem is proved. $\square$

9. <u>Theorem</u>. There exists a non-transitive symmetric reflexive relation R on a finite set such that no PVchunk process defines exclusion (R).

<u>Proof</u>. Define R to be the non-transitive symmetric reflexive relation on {1,2,3,4,5} represented by the directed graph

$$\begin{tikzcd}
& 1 \arrow[rr, leftrightarrow] \arrow[dl, leftrightarrow] & & 2 \arrow[dr, leftrightarrow] & \\
3 \arrow[drr, leftrightarrow] & & & & 4 \arrow[dll, leftrightarrow] \\
& & 5 & &
\end{tikzcd}$$

Suppose that ρ is a PVchunk process that defines exclusion (R). Let d be the one to one correspondence from the actions of ρ to Σ. By Theorem 8 applied to 1,2,4; amount$(d^{-1}(2)) >$ amount$(d^{-1}(1))$. Now by Theorem 8 applied to 2,1,3; amount$(d^{-1}(1)) >$ amount$(d^{-1}(2))$. This is a contradiction, and hence no PVchunk process can define exclusion (R). □

Let us summarize what we have, thus far, proved about exclusion slices.

(1) If a PV process defines exclusion (R), then R is an equivalence relation.

(2) If a PVmultiple process defines exclusion (R), then R is a symmetric relation.

(3) If a PVchunk process defines exclusion (R), then R is a symmetric relation. Also there is a symmetric reflexive relation R' such that no PVchunk process can define exclusion (R').

These results are restrictive in nature. We will next prove the following existence results.

(4) If R is an equivalence relation, then some PV process defines exclusion (R).

(5) If R is a symmetric reflexive relation, then some PVmultiple process defines exclusion (R).

(6) There are non-transitive symmetric reflexive relations R such that some PVchunk process defines exclusion (R).

(7) If R is a reflexive relation, then some up/down process defines exclusion (R).

7.4 EXISTENCE RESULTS

10. Theorem. Suppose that R is an equivalence relation on a finite set $\{x_1,\dots,x_n\}$. Then some PV process defines exclusion (R).

Proof. Let $A_1,\dots,A_m$ be the equivalence class of R. Define the integers N_i $(1 \le i \le n)$ by: N_i is the index of the equivalence class of x_i, i.e., $x_i \in A_{N_i}$. Consider the PV process ρ represented by

```
semaphore $S_1,\dots,S_m$; (initial value 1)
SUBPROCESS-i (1 ≤ i ≤ n)
P($S_{N_i}$);
```

By the definition of the process, α is an active timing in ρ iff

(1) for $1 \le i < j \le \text{length}(\alpha)$, not $S_{N_i} = S_{N_j}$.

Therefore, α is an active timing in ρ iff

(2) for $1 \le i < j \le \text{length}(\alpha)$, not $x_i R x_j$.

Thus, ρ defines exclusion (R). □

The construction used in Theorem 10 is essentially due to Dijkstra [1968]. For an example of this construction, let $\Sigma = \{x_1, x_2, x_3, x_4\}$; and represent R by the directed graph

$$x_1 \leftrightarrow x_3 \quad x_2 \leftrightarrow x_4$$

Then ρ is represented by

```
semaphore S_1,S_2; (initial value 1)
SUBPROCESS-1     SUBPROCESS-2     SUBPROCESS-3     SUBPROCESS-4
(1)  P(S_1);     (2)  P(S_2);     (3)  P(S_1);     (3)  P(S_2);
```

Now exclusion (R) = $\{\Lambda, x_1, x_2, x_3, x_4, x_1x_2, x_1x_4, x_2x_1, x_4x_1, x_3x_2, x_3x_4, x_2x_3, x_4x_3\}$. Also the set of active timings of ρ is

$$\{\Lambda,1,2,3,4,12,14,21,41,32,34,23,43\}.$$

Thus, ρ defines exclusion (R).

11. Theorem. There exists a non-transitive symmetric reflexive relation R such that some PVchunk process defines exclusion (R).

Proof. Represent R by the directed graph

$$a \leftrightarrow b \leftrightarrow c$$

Then exclusion (R) = $\{\Lambda, a, b, c, ac, ca\}$. Represent the PVchunk process ρ by

```
semaphore S; (initial value 2)
SUBPROCESS-1                  SUBPROCESS-2                  SUBPROCESS-3
(1) P(S with amount 1);       (2) P(S with amount 2);       (3) P(S with amount 1
```

Clearly, $\{\alpha | \alpha \text{ active in } P\} = \{\Lambda,1,2,3,13,31\}$. Therefore, P defines the exclusion slice, exclusion (R). $\sqcap$

The construction used in Theorem 11 is essentially due to Vantilborgh and van Lamsweerde [1972]. They actually show that some PVchunk process can define exclusion (R) provided:

(1) R is a relation on $\Sigma = \Sigma_w \cup \Sigma_r$ where Σ_w and Σ_r are disjoint;

(2) for all x and y in Σ_w, xRy ;

(3) for all x in Σ_w and y in Σ_r, xRy and yRx ;

(4) for all x in Σ_r, xRx;

(5) for all x and y in Σ_r with $x \neq y$, not xRy.

Informally, we can interpret Σ_w as a set of "writers" and Σ_r as a set of "readers". For example, some PVchunk process can define R where R is represented by

$$\begin{tikzcd}
 & r_1 \arrow[dr, leftrightarrow] & \\
w_1 \arrow[ur, leftrightarrow] \arrow[rr, leftrightarrow] \arrow[dr, leftrightarrow] & & w_2 \\
 & r_2 \arrow[ur, leftrightarrow] &
\end{tikzcd}$$

12. Theorem. Suppose that R is a reflexive relation on the finite set Σ. Then some up/down process defines exclusion (R).

Proof. Let $\Sigma = \{x_1,\ldots,x_m\}$. Consider the up/down process P represented by

```
semaphore $S_1,\ldots,S_n$ (initial value 0)
SUBPROCESS-1  $(1 \le i \le n)$
$\{S_k | x_k R x_i\}$: down $(S_i)$;
```

By the definition of $\mathcal{P}$, α is an active timing in $\mathcal{P}$ iff

(1) for $1 \le i < j \le \text{length}(\alpha)$, S_i is not in $\{S_k | x_k R x_j\}$.

Therefore, α is an active timing in $\mathcal{P}$ iff

(2) for $1 \le i < j \le \text{length}(\alpha)$, not $x_i R x_j$.

Thus, $\mathcal{P}$ defines exclusion (R). $\square$

The construction used in Theorem 12 is essentially due to Wodon [1972]. As an example, let us consider the relation R represented by

$$x_1 \rightarrow x_2 \leftrightarrow x_3$$

Then represent the up/down process $\mathcal{P}$ by

<u>semaphore</u> S_1, S_2, S_3; (initial value 0)

```
SUBPROCESS-1              SUBPROCESS-2              SUBPROCESS-3
(1) { }: down (S1);       (2) {S1,S3}: down (S2);   (3) {S2}: down (S3);
```

Now exclusion (R) $= \{\Lambda, x_1, x_2, x_3, x_1x_3, x_2x_1, x_3x_1\}$. Also the set of active timings of $\mathcal{P}$ is $\{\Lambda, 1, 2, 3, 13, 21, 31\}$. Thus, $\mathcal{P}$ defines exclusion (R).

We now turn our attention to the predicate system PVmultiple. We will show - as stated in Section 7.3 - that PVmultiple can define any exclusion (R), provided R is symmetric. This result does not appear to have been previously stated in the literature.

13. <u>Definition</u>. Suppose that Π is a Σ-slice. Then Π is a <u>permutation slice</u> provided,

if α is in Π and β is a permutation of α, then β is in Π

14. <u>Theorem</u>. Suppose that Π is a Σ-slice that is a permutation slice. Then some PVmultiple process defines Π.

<u>Proof</u>. Define $\Phi = \{A \mid \text{for some } a_1 \ldots a_k \text{ in } \Pi,\ A = \{a_1,\ldots,a_k\}\}$. Since Π is permutation closed, for all $a_1,\ldots,a_k$ distinct

(1) $\{a_1,\ldots,a_k\}$ is in Φ iff $a_1\ldots a_k$ is in Π.

Also by (1) and the definition of slice,

(2) if B is in Φ and $A \subseteq B$, then A is in Φ.

For each $B \subseteq \Sigma$, define k_B by

$$k_B = \begin{cases} |B| & \text{if } B \text{ is in } \Phi \\ |B|-1 & \text{if } B \text{ is not in } \Phi \end{cases}$$

where $|B|$ is the cardinality of the set B. Then

(3) A is in Φ iff for all $B \subseteq \Sigma$, $|A \cap B| \le k_B$.

Suppose that A is in Φ. Assume that $|A \cap B| > k_B$. Since $|B| \ge |A \cap B|$, $k_B = |B|-1$; and B is not in Φ. Therefore, $|B| = |A \cap B|$; and so $B \subseteq A$. By (2), B is in Φ. This is a contradiction. Conversely, suppose that for each $B \subseteq \Sigma$, $|A \cap B| \le k_B$. Assume that A is not in Φ. Then $k_A = |A|-1$. However, not $|A \cap A| \le |A|-1$; and this is a contradiction. Thus, (3) is true.

In summary, we have shown that we can test whether or not A is in Φ, by checking the conjunction of certain inequalities. We now will use this fact to construct a PVmultiple process that defines the slice Π. Let $\Sigma = \{x_1,\ldots,x_n\}$, and let $B_1,\ldots,B_m$ be the subsets of Σ. Let P be the

PVmultiple represented by

<u>semaphore</u> $S_1,\dots,S_m$ (initial value of S_i is k_{B_i}, for $1 \le i \le m$)
SUBPROCESS-i ($1 \le i \le n$)
(f_i) $P(\{S_k | x_i \text{ is in } B_k\})$;

Note, $P(\{S_k | x_i \text{ is in } B_k\})$ decreases S_k by one, provided x_i is in B_k. Also define the one to one correspondence from the actions of ρ to Σ by

for $1 \le i \le n$, $d(f_i) = x_i$.

Let '$(L_1,\dots,L_n,D,(S_1,\dots,S_m))$' be a typical element of ρ.

Suppose that $\alpha_1\dots\alpha_k$ is an active timing in ρ. Then α_i decreases the semaphore S_j by 1 iff $d(\alpha_i)$ is in B_j. Since the semaphore S_j is initially k_{B_j},

$$S_j[\text{value}(\alpha_1\dots\alpha_k)] = k_{B_j} - \sum_{v=1}^{k} |\{d(\alpha_v)\} \cap B_j|.$$

Note, $|\{d(\alpha_v)\} \cap B_j|$ is equal to

if $d(\alpha_v)$ is in B_j, then 1; otherwise 0.

By the definition of the process ρ, if $\alpha_1\dots\alpha_k$ is active, then $\alpha_1\dots\alpha_k$ are all distinct. Therefore, if $\alpha_1\dots\alpha_k$ is active, then

(4) $S_j[\text{value}(\alpha_1\dots\alpha_k)] = k_{B_j} - |\{d(\alpha_1),\dots,d(\alpha_k)\} \cap B_j|$.

We are now ready to show that ρ defines the slice Π. Suppose that $\alpha_1\dots\alpha_k$ is active. By (1), we need only show that $\{d(\alpha_1),\dots,d(\alpha_k)\}$ is in δ. Select an integer j with $1 \le j \le m$. Clearly, $S_j[\text{value}(\alpha_1\dots\alpha_k)] \ge 0$;

thus, by (4),

$$(5)\quad |\{d(\alpha_1),\ldots,d(\alpha_k)\} \cap B_j| \le k_{B_j}.$$

Since j was arbitrary, (5) is true for all j. Therefore, by (3), $\{d(\alpha_1),\ldots,d(\alpha_k)\}$ is in Φ. Conversely, suppose that $d(\alpha_1)\ldots d(\alpha_k)$ is in Π. Assume that $\alpha_1\ldots\alpha_k$ is not active. We can assume without loss of generality that

$$\alpha_1\ldots\alpha_{k-1} \text{ is active.}$$

Since α_k is not in ready-set$(\alpha_1\ldots\alpha_{k-1})$, there is an integer j such that

$$S_j[\text{value}(\alpha_1\ldots\alpha_{k-1})] = 0 \text{ and } d(\alpha_k) \text{ is in } B_j.$$

By (4),

$$k_{B_j} = |\{d(\alpha_1),\ldots,d(\alpha_{k-1})\} \cap B_j|.$$

Since $d(\alpha_1)\ldots d(\alpha_k)$ is in Π, the $d(\alpha_1),\ldots,d(\alpha_k)$ are all distinct. Therefore,

$$k_{B_j} < |\{d(\alpha_1),\ldots,d(\alpha_k)\} \cap B_j|.$$

Then by (3), $\{d(\alpha_1),\ldots,d(\alpha_k)\}$ is not in Φ. Thus, $d(\alpha_1)\ldots d(\alpha_k)$ is not in Π. This is a contradiction, and hence P defines Π. □

15. Corollary. Suppose that R is a symmetric reflexive relation on the finite set Σ. Then some PVmultiple process defines exclusion (R).

Proof. The slice, exclusion (R), is a permutation slice. □

As an example, consider the slice $\Pi = \{\Lambda,a,b,c,d,ab,ba,ac,ca,bc,cb\}$. This slice is a permutation slice; hence, some PVmultiple process defines Π. The PVmultiple process constructed by the method of Theorem 14 uses 16 semaphores. Since there are simpler such processes, we will now present one. The PVmultiple process ρ is represented by

<u>semaphore</u> S_1,S_2,S_3,S_4; ($S_1 = S_2 = S_3 = 1$ and $S_4 = 2$ initially)

SUBPROCESS-1	SUBPROCESS-2	SUBPROCESS-3	SUBPROCESS-4
(1) $P(\{S_1,S_4\})$;	(2) $P(\{S_2,S_4\})$;	(3) $P(\{S_3,S_4\})$;	(4) $P(\{S_1,S_2,S_3\})$;

The active timings of ρ are $\{\Lambda,1,2,3,4,12,21,13,31,23,32\}$. Clearly, ρ defines Π. The importance of Theorem 14 is that it is an existence theorem.

We will now summarize our results on exclusion slices:

(1) Up/down defines all exclusion slices.

(2) PVmultiple defines exclusion (R) iff R is symmetric.

(3) PVchunk defines exclusion (R) implies that R is symmetric. There is a symmetric relation such that PVchunk does not define R. In addition, there is a non-transitive symmetric relation R such that PVchunk can define exclusion (R).

(4) PV defines exclusion (R) iff R is an equivalence relation.

The predicate system up/down is "universal" - in the sense that up/down can define any exclusion slice. An immediate question is

Can up/down define every slice that PVmultiple or PVchunk can define?

We will next show that the answer to this question is no. When we consider all slices - not just exclusion slices - we find that up/down and PVmultiple (respectively PVchunk) are "incomparable", i.e., neither one can define everything the other one can. This indicates the complex nature of slices.

7.5 ANOTHER TYPE OF SLICE

16. Definition. A Σ-slice is meager if for a and b in Σ there exists an α in Π such that

αa is in Π, αb is in Π, and αab is not in Π.

17. Theorem. In a C process, if not synchronizer(f), then for any finite timing α,

f is in pointer-set(α) iff f is in ready-set(α).

Proof. Immediate from the definition of C process. □

18. Theorem. Suppose that some up/down process defines a meager Σ-slice, Π, with $|\Sigma| > 1$. Then if αa is in Π, β is in Π, length(α) = length(β), and a is not in β, then βa is in Π.

Proof. Suppose that $\mathcal{P} = \langle A, \mathfrak{Y}, w\rangle$ is an up/down process that defines Π. Let '$(L_1,\ldots,L_n,D,(S_1,\ldots,S_m))$' be a typical element in $\mathfrak{Y}$.

Suppose that f is an action in $\mathcal{P}$. Select an action g in $\mathcal{P}$ with $f \neq g$. By the definition of meager and defines, there is an active timing α such that

αf is active, αg is active, and αgf is not active.

By property II, not subprocess (f,g). By properties III and I, f is in pointer-set(αg). Since f is not in ready-set(αg), synchronizer(f) is true by Theorem 17. Since f is arbitrary, for all actions h in ρ, synchronizer(h).

We now assert that

(1) for all actions g in ρ, the pair of g is a F: down (S_i), for some F and S_i.

Assume that the pair of f is a F: up (S_j). Select an action g such that f $\neq$ g. By the definition of meager and defines, there is an active timing α such that

αf is active, αg is active, and αfg is not active.

Suppose that the pair of g is either H: down (S_i) or H: up (S_i). As before g is in pointer-set(αf). Since g is not in ready-set(αf),

$$\sum_{S_k \in H} S_k[\text{value}(\alpha f)] < 0.$$

Since the pair of f is a F: up (S_j), for all S_k,

$$S_k[\text{value}(\alpha f)] \geq S_k[\text{value}(\alpha)].$$

Thus,

$$\sum_{S_k \in H} S_k[\text{value}(\alpha)] < 0$$

which is a contradiction: g is in ready-set(α). Therefore, (1) is true.

Let the pair of the ith action of ρ, f_i, be

$$F_i\text{: down } (S_{B_i}).$$

We now assert that

(2) for $i \neq j$, B_i is in F_j.

Suppose that for $i \neq j$, B_i is not in F_j. Again by the definition of meager and defines, there is an active timing α such that

αf_i is active, αf_j is active, and $\alpha f_i f_j$ is not active.

As before, f_j is in pointer-set(αf_i). Since f_j is not in ready-set(αf_i),

$$\sum_{S_k \in F_j} S_k[\text{value}(\alpha f_i)] < 0.$$

Also since f_j is in ready-set(α),

$$\sum_{S_k \in F_j} S_k[\text{value}(\alpha)] \geq 0.$$

This is a contradiction: for each $S_k \in F_j$, $S_k[\text{value}(\alpha)] = S_k[\text{value}(\alpha f_i)]$. Therefore, (2) is true.

Finally, suppose that αf_i is active, β is active, f_i is not in β, and length(α) = length(β). We will now show that βf_i is active. By the definition of defines, this is sufficient to prove the theorem. Since αf_i is active, f_i is not in α. By our two assertions, (1) and (2),

$$\sum_{S_k \in F_i} S_k[\text{value}(\alpha)] = \sum_{S_k \in F_i} S_k[w] - \text{length}(\alpha)$$

and

$$\sum_{S_k \in F_i} S_k[\text{value}(\beta)] = \sum_{S_k \in F_i} S_k[w] - \text{length}(\beta).$$

Thus,

(3) $$\sum_{S_k \in F_i} S_k[\text{value}(\alpha)] = \sum_{S_k \in F_i} S_k[\text{value}(\beta)].$$

By property III, f_i is in pointer-set(β). Since f is in ready-set(α),

(4) $$\sum_{S_k \in F_i} S_k[\text{value}(\alpha)] \geq 0.$$

By (3) and (4), f_i is in ready-set(β); hence, βf_i is active. $\sqcap$

19. Theorem. Let $\Pi = \{\Lambda, a, b, c, d, ab, ba, ac, ca, bc, cb\}$. Then

(1) Π is a meager Σ-slice where $\Sigma = \{a,b,c,d\}$

(2) no up/down process defines Π

(3) some PVchunk process defines Π

(4) some PVmultiple process defines Π.

Proof. (1) Clearly, Π is a Σ-slice. We will now show that Π is meager. By symmetry, there are essentially three cases that we must check. First, consider d and a. Then d is in Π, a is in Π, and da is not in Π. Second, consider a and d. Then a is in Π, d is in Π, and ad is not in Π. Third, consider a and b. Then ca is in Π, cb is in Π, and cab is not in Π. Therefore, Π is a meager Σ-slice.

(2) Now ab is in Π, d is in Π, length(a) = length(d), and $b \neq d$. Thus, if an up/down process defines Π, then by Theorem 18, db is in Π. Since db is not in Π, no up/down process defines Π.

(3) The following PV process $\mathcal{P}$ defines Π. $\mathcal{P}$ is represented by

```
semaphore S; (initial value 2)
SUBPROCESS-1 (1 ≤ i ≤ 3)          SUBPROCESS-4
(f_i) P(S with amount 1);          (f_4) P(s with amount 2);
```

Clearly, the active timings of $\wp$ are $\{\Lambda, f_1, f_2, f_3, f_4, f_1f_2, f_2f_1, f_1f_3, f_3f_1, f_2f_3, f_3f_2\}$. Thus, $\wp$ defines Π.

(4) This follows from theorem 14. □

7.6 PROOF OF RESULTS STATED IN INTRODUCTION

We are now in a position to prove the results stated in the introduction. These results are displayed in Figure 25; an arrow from x to y means that x → y.

$$\begin{tikzcd}
\text{PVchunk} \arrow[r] \arrow[dr] & \text{up/down} \arrow[l, bend right] \arrow[r] \arrow[d] & \text{PVmultiple} \arrow[l, bend right] \arrow[ll, bend right=40] \arrow[dl] \\
 & \text{PV} &
\end{tikzcd}$$

FIGURE 25. Results of the Thesis

(1) up/down → PVchunk, up/down → PVmultiple, up/down → PV. This is a consequence of Theorem 5, Theorem 12, and the invariance theorem.

(2) PVmultiple → PVchunk, PVmultiple → PV. This is a consequence of Theorem 6, Theorem 9, Corollary 15 and the invariance theorem.

(3) PVmultiple → up/down, PVchunk → up/down. This is a consequence of Theorem 19 and the invariance theorem.

(4) PVchunk → PV. This is a consequence of Theorem 6, Theorem 11, and the invariance theorem.

By Section 2.6, not PV → PVchunk and not PV → PVmultiple. However, whether or not PVchunk → PVmultiple is an open question.

7.7 OTHER APPLICATIONS OF SLICES

We can also use the results of this section - in conjunction with the invariance theorem - to analyze synchronization problems.

First Reader-Writer Problem. Recall that W1 is the process that represents this problem. Since W1 is an up/down process, there clearly is an up/down process that simulates W1. In addition, there is a PVchunk (respectively PVmultiple) process that simulates W1 (Vantilborgh and van Lamsweerde [1972] and Dijkstra [unpublished]). We can show that no PV process simulates W1. As we stated in Section 7.2, W1 implicitly defines a non-transitive exclusion slice, i.e., W1 implicitly defines the slice represented by

$$\begin{tikzcd} r_1 \arrow[r, leftrightarrow] & v \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & r_3 \\ & r_2 & \end{tikzcd}$$

By Theorem 6 and the invariance theorem, no PV process can simulate W1.

Second Reader-Writer Problem. Recall that W2 is the process that represents this problem. Since W2 is an up/down process, there clearly is an up/down process that simulates W2. We can show that no PV (respectively PVchunk or PVmultiple process) can simulate W2. As we stated in Section 7.2, W2 implicitly defines a non-symmetric exclusion slice, i.e., W2 implicitly defines the exclusion slice represented by

$$w \rightarrow r$$

By Theorem 5 and the invariance theorem, no PV (respectively PVchunk or PVmultiple) process can simulate W2.

Five Dining Philosophers Problem. Recall that WS is the process that represents this problem. Since WS is an up/down process, there trivially is an up/down process that simulates WS. In addition, there is a PVmultiple process that simulates WS. (Dijkstra [1971]) We can show that no PV (respectively PVchunk) process can simulate WS. As we stated in Section 7.2, WS implicitly defines a non-transitive exclusion slice, i.e., WS implicitly defines

$$\begin{tikzcd}
& 1 \arrow[rr, leftrightarrow] \arrow[dl, leftrightarrow] & & 2 \arrow[dr, leftrightarrow] & \\
3 \arrow[drr, leftrightarrow] & & & & 4 \arrow[dll, leftrightarrow] \\
& & 5 & &
\end{tikzcd}$$

By Theorem 6, Theorem 9, and the invariance theorem, no PV or PVchunk process can simulate WS.

Bounded First Reader-Writer. Recall that BRW is the process that corresponds to this problem. There are PVchunk and PVmultiple processes that simulate BRW. On the other hand, no PV (respectively up/down) process can simulate BRW. BRW implicitly defines the slice,

$$\{\Lambda,a,b,c,d,ab,ba,ac,ca,bc,cb\}.$$

Therefore, by Theorem 19 and the invariance theorem, no up/down process can simulate BRW. BRW also implicitly defines the exclusion slice represented by

$$a \leftrightarrow d \leftrightarrow b$$

Hence, as in the first reader-writer problem, no PV process can simulate BRW.

Clearly, we can use the invariance theorem to analyze other synchronization problems.

8. CONCLUSIONS

We have achieved our basic goal: we have shown that there are differences between the predicate systems PV, PVchunk, PVmultiple and up/down. These results are proved in two steps. First, we show that there are "local differences" between the four predicate systems. Second, we use the invariance theorem to conclude that there are "global differences" between the four predicate systems.

One of the consequences of creating a formal model in an area that is informal - but rich in ideas - is that we may have created more questions than we have answered. Some of these questions are refinements of this work while others are essentially extensions. We will now present a partial list of some of these questions.

1. One of the questions we have left unanswered is:

   does PVchunk $\rightarrow$ PV multiple?

2. An open area of research is the study of slices. Presently we have no characterization for the set of slices definable by any of the predicate systems: PV, PVchunk, PVmultiple, up/down. Predicate systems other than these are totally unexplored.

3. We have not considered whether any of the basic questions of this paper are decidable.

4. The concepts of release and pointer-bounded are important, yet they are presently without any theoretical results.

5. The concepts of "busy wait" and "restricted busy wait" (Dijkstra [1968] and Hansen [1972]) are expressible in our theory. An interesting question is: can we give a sound theoretical foundation to the folklore that states that busy wait or even restricted busy wait is "inefficient?

6. One of the key questions untouched is: what does it mean to "implement" a process?

We have attempted to formalize the basic concepts of the synchronization area. Whether or not we have been successful, we feel that the synchronization area must, in the future, become more formal and precise.

Acknowledgements

Thanks are due to Dr. A. J. Perlis for his valuable suggestions and reading of this manuscript.

References

Brinch Hansen [1972]
P. B. Hansen. A Comparison of Two Synchronizing Concepts. Acta Informatica 1:190-199.

Brinch Hansen [1972a]
P. B. Hansen. Structured Multi-programming. Communications of the ACM 15(7):574-578.

Courtois, Heymans, Parnas [1971]
P. J. Courtois, F. Heymans, D. L. Parnas. Concurrent Control with "Readers" and "Writers." Communications of the ACM 14(10):667-668.

Courtois, Heymans, Parnas [1972]
P. J. Courtois, F. Heymans, D. L. Parnas. Comments on "A Comparison of Two Synchronizing Concepts by P. B. Hansen." Acta Informatica 1:375-376.

Dennis and Van Horn [1966]
J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. Communications of the ACM 9(3):143-155.

Dijkstra [1968]
E. W. Dijkstra. Cooperating Sequential Processes, Programming Languages, edited by F. Genuys. 43-112.

Dijkstra [1968a]
E. W. Dijkstra. The Structure of the "THE" Multiprogramming System. Communications of the ACM 11(5):341-347.

Dijkstra [1971]
E. W. Dijkstra. Hierarchical Orderings of Sequential Processes. Acta Informatica 1(2):115-138.

Dijkstra [1972]
E. W. Dijkstra. Information Streams Sharing a Finite Buffer. Information Processing Letters 1:179-180.

Habermann [1972]
A. N. Habermann. Synchronization of Communicating Processes. Communications of the ACM 15(3):171-176.

Hoare [1971]
C. A. R. Hoare. Towards a Theory of Parallel Programming. International Seminar on Operating System Techniques, Belfast, Northern Ireland.

Lipton [1973]
R. J. Lipton. On Synchronization Primitive Systems, PhD thesis, Carnegie-Mellon University.

Patil [1971]
S. S. Patil. Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes. Project MAC Computational Structures Group Memo 57.

Parnas [1972]
D. L. Parnas. On a Solution to the Cigarette Smokers' Problem (Without Conditional Statements). Carnegie-Mellon University Report.

Saltzer [1966]
J. H. Saltzer. Traffic Control in a Multiplexed Computer Systems, PhD thesis, MIT (Project MAC).

Vantilborgh and van Lamsweerde [1972]
H. Vantilborgh and A. van Lamsweerde. On an Extension of Dijkstra's Semaphore Primitives. Information Processing Letters 1:181-186.

Wodon [1972]
P. Wodon. Still Another Tool for Controlling Cooperating Algorithms. Carnegie-Mellon University Report.

Wodon [1972a]
P. Wodon. The Belpaire-Wilmotte Method for Transforming Up/Down Operations into P/V Operations. Unpublished manuscript.