A SYNCHRONIZATION CALCULUS
FOR MESSAGE ORIENTED PROGRAMMING

P.R.F. Cunha
T.S.E. Maibaum

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada
N2L 3G1

Research Report CS-80-43

September 1980

This work was supported by a grant from the Natural Sciences and Engineering Research Council of Canada and the Canadian International Development Agency.

ABSTRACT

In previous reports we have motivated and described a methodology for the development of programs for distributed environments in which the synchronization mechanism is based on message passing primitives. We outline in this paper a calculus to study in a systematic way the synchronization properties for message oriented programming. These properties include such things as absense/presence of deadlocks, unpaired primitives, use of unbounded buffers (capacity of the channels involved in the communication), etc. In order to study these questions, we introduce a technique (called the synchronization tree) which is a finite representation for the reachability set. The main result presented is that, given any message oriented program, it is decidable whether the processes involved ever enter a configuration in which some subset of the processes is deadlocked.

Keywords:

Distributed computing, message oriented programming, synchronization calculus, deadlock, boundedness, unpaired primitives, vector addition systems, synchronization tree.

1. Introduction

We outline in this paper a calculus to study the synchronization properties (i.e., the properties related to termination) for message oriented programs. The main result presented is that, given any message oriented program, it is decidable whether the processes involved ever enter a configuration in which some subset of the processes is deadlocked. This provides a basis for a static, data independent test for deadlock freeness for message passing systems. We also are able to answer such questions as: Does the communication channel between two programs always contain a bounded number of messages? If so, what is the bound? Before proceeding to further describe the contents of the paper, we provide a motivation for the study of this style of programming.

The current trend in parallel programming is programming through messages and processes. The general idea of message passing for interprocess communication was preliminarily discussed by Brinch Hansen in [2]. More recently the concept has been discussed in a more general setting, by presenting processes and messages as both a structuring tool and as a synchronization mechanism. Instances of this recent effort can be found in Zave [23], Jammel [18], Hoare [17] and in the description of multiprocessing systems such as Demos [1], Mininet [20] and Thoth [4].

Zave [23] has argued for the naturalness, usefulness and generality of programming with messages and processes. We think that a further characterization of this programming technique is necessary. It needs to be at least as well understood as the techniques for parallel programming with shared variables. In other words, design principles, specification and proof methods need to be developed for the complete characterization of this novel programming style.

The methodology we have developed [5 , 8, 9] is based on the concept of resource. This concept has its roots in such notions as monitors [16] (essentially the development of a synchronization mechanism for the operations of a data type in a situation where a shared address space is assumed), managers [18], proprietors [4] and secretaries [10]. The underlying idea in all of these is the explicit expression of synchronization mechanisms. Resources are essentially an abstract data type together with a synchronization mechanism expressed in terms of message passing primitives. These mechanisms are used to allow as much parallelism as possible in the use of the operations of the data type. This is, of course, at the heart of the development of distributed computing - to gain efficiency by the use of parallelism.

As for the sequential programs, the first step in the development of a program is the identification of the abstract data types (i.e. the bases of the resources) to be used by the program. (As for sequential programs the choice will have a great influence on the final solution which is obtained.) Once an abstract data type has been identified (and presumably specified but not yet implemented) we are left with the task of adding the synchronization mechanism. This can be done as follows: Assume for the moment that the operations of the type can be directly invoked as primitive operations by the programs performing synchronization for the resource. Thus the resource is assumed to be implemented on a single address space. (This assumption will later be removed in the stepwise refinement process.) A formula, called the asynchronous condition formula (acf), is now developed which expresses the conditions under which operations of the data type may be invoked. Thus, it is at this point that one determines the degree of parallelism which is desired for the resource (within the limits set by the nature of the underlying data type). Using the acf, one can now define the process structure (i.e. the set of programs used for synchronization) associated with the resource [7 , 8, 9].

Once the process structure for a resource has been defined, for each process in turn we establish a syncrhonization formula (sf) (in the form of a regular expression) which establishes the sequence of message passing primitives which the process will use to synchronize its activities with other processes in the resource (if there be such) and the environment of the resource. Then the process is defined by "filling in" the sequential parts of the process which "fit" between the synchronization primitives of the sf.

By following the above methodology, one obtains an abstract implementation of the resource and, based on the properties of the data type and the synchronization primitives, one can prove properties of the abstract program. At this point we use the idea of stepwise refinement and remove the assumption that the synchronization processes of a resource can invoke the operations of the underlying data type as primitive operations on a shared address space. Two situations are now possible. Firstly, we may keep the assumption of a shared address space for the processes and implement the resource's underlying data type in the usual way. This corresponds to the conventional situation and techniques for such implementations have been well developed and studied [11,13,15]. Secondly, we may develop a distributed implementation for the data type [8]. This involves the use of more primitive resources (as opposed to the data types of the conventional implementations) for the implementation of the resource in question.

The advantages to be gained from such an approach are similar to the benefits espoused for the analogous technique applied to sequential programs. That is, we gain modularity in the solution and we also modularise the proofs of properties of the system. The modularity to be gained in proofs of properties is fully outlined in [5, 6, 8, 9], but we may point out here that since these programs use a control structure quite different from sequential programs, new techniques are needed for modularizing the traditional aspects of proofs (such as the correctness of implementation of a data type) as well as the new or novel aspects of proofs (such as deadlock freeness, lack of starvation, etc.).

As indicated earlier, in this report we concentrate on these so-called novel aspects of proofs-what we call synchronization aspects. In message passing systems (whether based on our methodology or not), important questions arise which have no analogues for sequential programs. For example, a set of processes (programs) could all be blocked in their execution because they are all awaiting some form of communication from some other process in the set. This is what is normally called deadlock. Since separate processes may be executing on separate processors whose relative speeds are unpredictable and since programs may produce and consume messages at different rates, it might be necessary to have buffers of unbounded size to buffer the communication between processes.

We introduce a technique (called the synchronization tree (ST)) which will allow us to predict, for example, whether a set of programs is free of deadlock situations or whether the communication between pairs of processes is bounded or not. The ST is a finite representation of the reachability set (i.e., the set of possible configurations) of a set of processes and is defined using concepts which are drawn from vector addition systems [19,21]. We note here that the technique is purely syntactic and ignores data in the programs.

The primitives used by our programs for communication are the following:

- $\text{send}_i(j, \text{msg})$: process i sends a message msg to process j.
- $\text{receive}_i(j)$: process i receives the next message from process j. If this message is not available then the process i blocks until such a message arrives.

Thus we use "unblocked send" and "blocked receive". However the technique we describe actually works (with slight modifications) for all the combinations of "blocked" and "unblocked" primitives. Also note that receives are specific - i.e., messages are received from specific processes (and not, as in some systems, from any processes which happens to have sent a message).

2. The Synchronization Structure

We motivated and described a methodology for the development of programs for distributed environments in [8, 9] where the synchronization mechanism is based on message passing primitives. The fundamental point of our methodology is the concept of resource. Resources are essentially an abstract data type plus a synchronization mechanism expressed in terms of message passing primitives. The synchronization mechanism is used to allow as much parallelism as possible in the use of the operations of the data type. Informally, the methodology decomposes a system into a set of resources which must interact to solve the problem at hand. In this report, our interest is the second major component of a resource-namely, the synchronization part. We will develop a calculus that will enable us to treat synchronization properties such as absence/presence of deadlock, unpaired primitives (defined later), or use of unbounded buffers in a systematic way.

In the development (modelling) of message oriented programs we have been using two programming languages. The first, called programming language (PL), uses a combination of Algol and Hoare's CSP[17] notations. An example of a program in PL is the following (which appears in a solution to the bounded buffer problem [17]):

```
process p-avpl ;
  { avpl ≠ 0 : msg := receive(prod) ;
               avpl := avpl -1 ;
               place item msg
  or avpl ≠ n : receive(cons) ;
                avpl := avpl + 1 ;
                get item into msg ;
                send(cons, msg)     }
  }
```

The second language, called synchronization language (SL), is used to specify the synchronization mechanism for each process (a resource is managed by a finite number of processes). The primitives of the language are s_{ij} to indicate the sending of a message from process i to process j and r_{ij} to indicate the reception of a message by process i from process j. These primitives can be preceded by boolean valued expressions (i.e., $b{:}s_{ij}$ or $b{:}r_{ij}$). Calling these primitives S and letting SL1, . . . , SLn be elements of SL we define SL recursively as follows:

(i) $S \subseteq SL$;

(ii) $SL1; SL2 \subseteq SL$;

(iii) $\{SL1\} \subseteq SL$;

(iv) $*\{SL1\} \subseteq SL$;

(v) $\{SL1 \text{ or } SL2 \text{ or } . . . \text{ or } SLn\} \subseteq SL$ (for $n \geq 1$).

```
process p-avpl ;
   * {avpl ≠ 0   :  receive(prod)
      or avpl ≠ n  :  receive(cons) ;
                      send(cons, msg) }
```

The two languages SL and PL differ essentially in the amount of detail included in the "programs" of the respective languages in the stepwise development of programs. In the design process, the first language which is used to express synchronization properties of programs gives a first approximation to the final program in PL. Here in the analysis process, the program in SL can be thought of as a more abstract version of the program in PL where the sequential part was removed. The communication skeleton (or the program in SL) only captures the information that will be necessary to answer questions related to the synchronization part of the program in PL.

Verification of message oriented programs involves questions related to the synchronization part (or to the termination) of the programs. Absence of deadlock, unpaired primitives and use of unbounded buffers are examples of such questions. These questions related to the synchronization part refer to the possible sequences of communication primitives (send's and receive's) and we can say that they are "data independent". (The sequential part which fills in the "gaps" of the communication skeleton shows how the values of the local variables are changed in the process.) In order to deal with such questions we are going to associate with each process a formula - the synchronization formula (sf) written as $f_1, f_2 \ldots f_n$ where each f_i is some primitive communications activity like send or receive and each f_i identifies the process(es) involved in the communication action. This formula establishes the sequence of message passing primitives that the process will use to synchronize its activities with the other processes. The synchronization formula is essentially the same program specified in the syncrhonization language SL except that the boolean valued expressions (predicates) and messages are removed. This action makes the branching decisions be taken arbitrarily (nondeterministically) instead of being "data dependent" and makes the sf still more abstract than the corresponding program SL. The synchronization formula for the process p-avpl presented above in the language SL is given below. Let us use s and r to denote, respectively, the primitives send and receive in the sf's. The symbol ";" denotes sequentiality of actions, "*" an indefinite number of repetitions of the enclosed communication sequence and "or" that the left and right expressions are disjoint in the code used for the process. (As we can see an sf is a kind of regular expression.)

[r(prod) or (r(cons); s(cons))]* (sf for process p-avpl)

(If we label the processes p-avpl, producer (prod) and consumer (cons) as processes 1, 2 and 3, then we can write the sf as $[r_1(2) \text{ or } (r_1(3); s_1(3))]^*$. Note that the subscripts in the sf's can be dropped when such values of the subscripts are obvious from the context.)

3. Some Analysis Questions

We have used sf expressions to specify the synchronization part of message oriented programs. The types of errors treated here are called potential errors because we are not considering the data part of the program. In the synchronization model, it is assumed that messages are transmitted in a finite amount of time and that messages are received at some destination site in the same order that they were sent from the source site. (In other words, we are assuming that a reliable transmission protocol underlies the model.) When a sending or a receiving operation is executed the control goes to one of the next possible communication pimitivies in the formula as in the usual sequential program (i.e., following the same rules for sequentiality, alternation and iteration). A sending operation transfers the associated message from the source process to the input buffer of the destination process. A receiving operation waits until its input buffer is not empty and then removes the first message of the input buffer (the one with the longest waiting time). Taking into account the previous considerations, we describe the following potential errors that can be present in the synchronization formulas. Note that these errors (properties) are defined for the communication between any number of processes.

3.1 Deadlock

A deadlock is characterized by two or more sf's (representing the synchronization part of the associated processes) reaching a state where there exists a circular chain of receiving operations (one in each sf) in which each process (sf) is blocked and waiting for a message from the next process (sf) in the chain. This state represents a situation where no further transition is possible for the processes that are deadly embraced in the chain.

Example 3.1:

```
Process 1:  (r(2); s(2))*
Process 2:  (r(1); s(1))*
```

Process 1 and process 2 trying to receive a message from each other is an example of deadlock. The two processes after the activation of the primitives r(2) in the first expression and r(1) in the second expression form a circular chain of blocked processes (deadlock). Note that in this case we have an example of an unavoidable deadlock (not potential) because there is no other processing alternative for the processes involved.

Example 3.2:

```
Process 1: ((s(3); r(3)) or (s(2); r(2)))*
Process 2: ((r(1); s(1)) or r(3))*
Process 3: ((r(1); s(1)) or s(2))*
```

Consider the following situation in the execution of the processes: process 1 sends a message to process 2 and then blocks itself awaiting a response from process 2, process 2 wants to receive a message from process 3, and process 3 wants to receive a message from process 1. This situation is clearly a case of deadlock. However, it is called a potential deadlock because it may not necessarily represent an error. Remember that we have removed the predicates in the sf's and consequently, by evaluating them, we may conclude that the sub-expressions (s(2); r(2)) in process 1 and (r(3)) in process 2 cannot be activated at the same time. Therefore, the final evaluation of a deadlock detected by the synchronization calculus would involve an examination of the "data dependent" part of the program.

3.2 Unpaired Primitives

Unpaired primitives occur when an sf contains a sending or receiving operation for which there is no corresponding receiving or sending operation in the other invoked sf. There are two distinct cases of unpaired primitives. The first refers to a complete omission of the corresponding primitive and the second to a possible inbalance between the number of paired primitives in the sf's (e.g., just one of them appears inside an iteration). An unpaired sending primitive results in a message (or messages) being transmitted and not received (equivalent to the loss of the message), and an unpaired receiving primitive results in a process being blocked forever. (Remember that we are considering a blocking receive and a nonblocking send operation). Unpaired primitives indicate an incomplete specification of the program (except for some intentional bizarre design).

Example 3.3:

Process 1: ((s(2))* or (s(3)*))

Process 2: (s(3); r(3))*

Process 3: ((r(2))* or r(1))

The two distinct cases of unpaired primitives are present in this example. There is an omission of a receiving operation in process 2 (the primitive s(2) in process 1 is unpaired) and of sending operation in process 3 (the primitive r(3) in process 2 is also unpaired). Although we have the sending operation s(3) in process 1 and the corresponding receiving operation r(1) in process 3, there is a possible case of unpaired primitives since the first primitive s(3) appears inside an iteration.

Example 3.4:

Process 1: ((s(2); r(2))* or s(2))

Process 2: ((r(1); s(1))* or r(1))

In this situation, we have a case of potential unpaired primitives. For example, if process 1 uses the second alternative (s(2)) to send a message to process 2 which in turn makes use of r(1) in the first alternative ((r(1); s(1))*) to receive the message, then the next transmitted message from process 2 will not be received by process 1. However, it may be derived from the program that the two subexpressions s(2) and (r(1); s(1))* are never activated at the same time. (In this example, it looks like the first and the second alternatives in process 1 are related with the first and the second ones in process 2, respectively. In general, a well structured program should be able to reach a final state (a state at the end of some complete activation of the sf's) where all the messages transmitted were also received. We will refer to this state later as a normal termination state.

3.3 Boundedness

Bounded communication means that there is a least upper bound for the number of slots needed in the input buffer of any process. (Otherwise, if the communication requires an infinite buffer then it is referred to as unbounded.) The unbounded buffer problem arises when there are no synchronization constraints in preventing one process from sending messages uninterruptedly. The guarantee of finite input buffers for the several processes involved in a message oriented program is a very important analysis question.

Example 3.5:

Process 1: (s(2); r(2))*

Process 2: (r(1); s(1))*

This is an example of bounded communication because the positions of the two receiving operations (r(2) and r(1)) in the processes 1 and 2 force an interleaving of the sending of a message and its corresponding receiving. In this special case, we can see that the least upper bound is equal to one. If we had considered the sf's for processes 1 and 2 as being just (s(2))* and (r(1))*, respectively, then the communication would be unbounded. (We are not making any assumption about the relative speed of the processors.)

4. The Synchronization Calculus

As we know, the synchronization formulas represent the possible sequences of sending and receiving operations for each of the processes that form a message oriented system. During the execution of the system what happens is an interleaving of these formulas due to the interaction of the processes. (Of course, this interleaving is subject to the definition of the primitives send and receive.) In this section, we present a synchronization calculus to detect (potential) design errors in these interactions. Our technique is based on some of the ideas underlying vector addition systems [19,21]. The description of our technique, called the synchronization tree, follows but before that some definitions are needed.

4.1 The Synchronization Tree

Let us assume that in a certain configuration of a message oriented system we have n processes represented by the corresponding sf's. To each of the sf's, we associate a formula pointer and a set of at most (n-1) different input counters (one for each possible sending process). The formula pointer specifies the next communication primitive to be activated in the sf and each input counter the number of messages sent from some given process that were not yet consumed.

Def. 4.1:

An sf-state of a synchronization formula is a vector where the first component belongs to the set $\{p0, p1, \ldots, pi\} \cup \{pf\}$ where i is the length of the sf (considering only the number of send's and receive's) and the other components belong to N (the set of non-negative integers).

The first component of an sf-vector is the formula pointer and the other components are the input counters. The two values p0 and pf of the pointer denote an initial position before the execution of the sf and a final position after its execution, respectively. The definition 4.1 above can be extended easily to a system state in the following way:

Def. 4.2:

Let $\{(p_1, c_{11}, \ldots, c_{1n_1}), (p_2, c_{21}, \ldots, c_{2n_2}), \ldots, (p_m, c_{m1}, \ldots, c_{mn_m})\}$ be the set of sf-states of the system (each sf-state representing the state of a process). The system state is a new $(m + n_1 + n_2 + \ldots + n_m)$ length vector formed by the join of the sf-states in the format $SS = (p_1, \ldots, p_m, c_{11}, \ldots, c_{1n_1}, c_{21}, \ldots, c_{2n_2}, \ldots, c_{m1}, \ldots, c_{mn_m})$. We refer to the component of SS corresponding to $c_{ij}(p_k)$ by $SS(c_{ij})(SS(p_k))$.

In the initial state of a system, all pointers have value p0 and all input counters have value zero. (From now on, we will use the terms "state" and "system state" in the same sense.) When a process sends or receives a message, the system moves from one state to another. If a message is sent then the corresponding input counter is incremented by one. Similarly, if a message is received then we decrement the corresponding input counter by one. (A receiving operation can only be performed if the value of the related input counter is greater than zero.) In both cases, the associated pointer moves to one of the next possible positions in the sf. We introduce the notation $S_1 \xrightarrow{cp_i} S_2$ to mean that the system moved from the state S_1 to state S_2 by execution of the communication primitive cp_i in sf_i (a sending or a receiving operation).

Def. 4.3:

A move $S_1 \xrightarrow{cp_i} S_2$ is said to be valid if and only if it satisfies the following conditions:

i) The pointer associated to sf_i in S_1 refers to cp_i and moves to a next allowable position in S_2,

ii) If cp = $s_i(j)$ then the value of the input counter for process i related to process j (sf-state for sf_j) in S_1 is incremented by 1 in S_2. Otherwise, if cp = $r_i(j)$ then the value of the input for process j related to process i in S_1 is greater than zero and is decremented by one in S_2, and

iii) All the other components of S_1 remain the same in S_2.

Def. 4.4:

A state S_n is reachable from another S_1 if there is a sequence of valid moves from S_1 to S_n (i.e., $S_1 \xrightarrow{cp_1} S_2 \xrightarrow{cp_2} S_3 \xrightarrow{cp_3} . . . \xrightarrow{cp_{n-1}} S_n$).

Def. 4.5:

The reachability set for a set of sf's (message oriented system) consists of all states reachable from the initial state. (i.e., the state in which all pointers have value p0 and all input counters have value zero.)

Informally, the reachability set of a message oriented system is the set of all configurations which the system can enter by any possible execution of its communication primitives. Most of the synchronization questions such as the ones presented in the last section can be stated in terms of the reachability set. However, the reachability set is often infinite. (When the reachability set is finite or there are definite bounds for the input counters as in many practical systems, all these analysis questions can be easily answered.) We are going to develop a technique called the synchronization tree to give a finite representation for the reachability set. This synchronization tree is conceptually similar to the idea of reachability tree used in vector addition systems [19,21].

In order to construct a finite representation of an infinite set (as in the case of the reachability set), we have to map many states into the same state (node) of the synchronization tree. The solution will be the introduction of a special symbol "w" with semantic value "arbitrarily large" (or infinity) to represent values in the input counters that can be made arbitrarily large. The interpretation of this special symbol w is such that if n is any non-negative integer number then it follows that $n < w$, $w + n = w$ and $w - n = w$. We can now define the algorithm which constructs the synchronization tree (ST) for a given set of sf's.

Algorithm 4.1:

Step 1:

Take the initial state (p0, . . . , p0, 0, . . . , 0) as the root of the tree. Move each of the pointers to any possible starting configuration in the sf's. (That is, the pointer is moved to any of the first primitives that can be activated in the sf's.) Make each of these starting configurations a direct descendent of the initial state and label the corresponding arcs with the null symbol ε.

Step 2:

For each level in the tree in turn, while there is a node v at that level which has not been processed do:

(a) Make each state s, which is a valid move from v, a son of v in the tree (referred as s(v)) and label the arc with the primitive that was activated (s_{ij} or r_{ij}). (Mark v as having been processed.)

(b) For each s(v) generated above do:

(i) If there is any p in the tree such that p = s(v) then make s(v) a leaf and mark it with "rl" (repetition-leaf).

(ii) If there is a predecessor p of s(v) in the tree (s(v) is reachable from p) such that $p \neq s(v)$, the pointer components in p and s(v) are the same, and the counter components in p are less than or equal to the corresponding ones in s(v) (i.e., for all i, j $p(c_{ij}) \leq s(v)(c_{ij})$) then change the (i,j)th counter component of s(v) to w if $p(c_{ij}) < s(v)(c_{ij})$

(iii) In node s(v) look for one or more chains of processes blocked on each other. (Process i is blocked on process j if it is waiting to receive a message from process j.) If such chains (or just one) exist and they involve all processes then make s(v) a leaf and mark it with "dl" (deadlock-leaf). Otherwise, if only a subset of the processes is involved then mark s(v) with "pdn" (partial deadlock node).

(iv) If all pointer components in s(v) are equal to pf or the ones that are different involve processes that are blocked on terminated processes (value pf in the corresponding component) or unexistent processes, then make s(v) a leaf. If s(v) = (pf, . . . , pf, 0, . . . , 0) then mark it with "nt" (normal termination); otherwise, mark it with "at" (abnormal termination).

Step 3:

Stop.

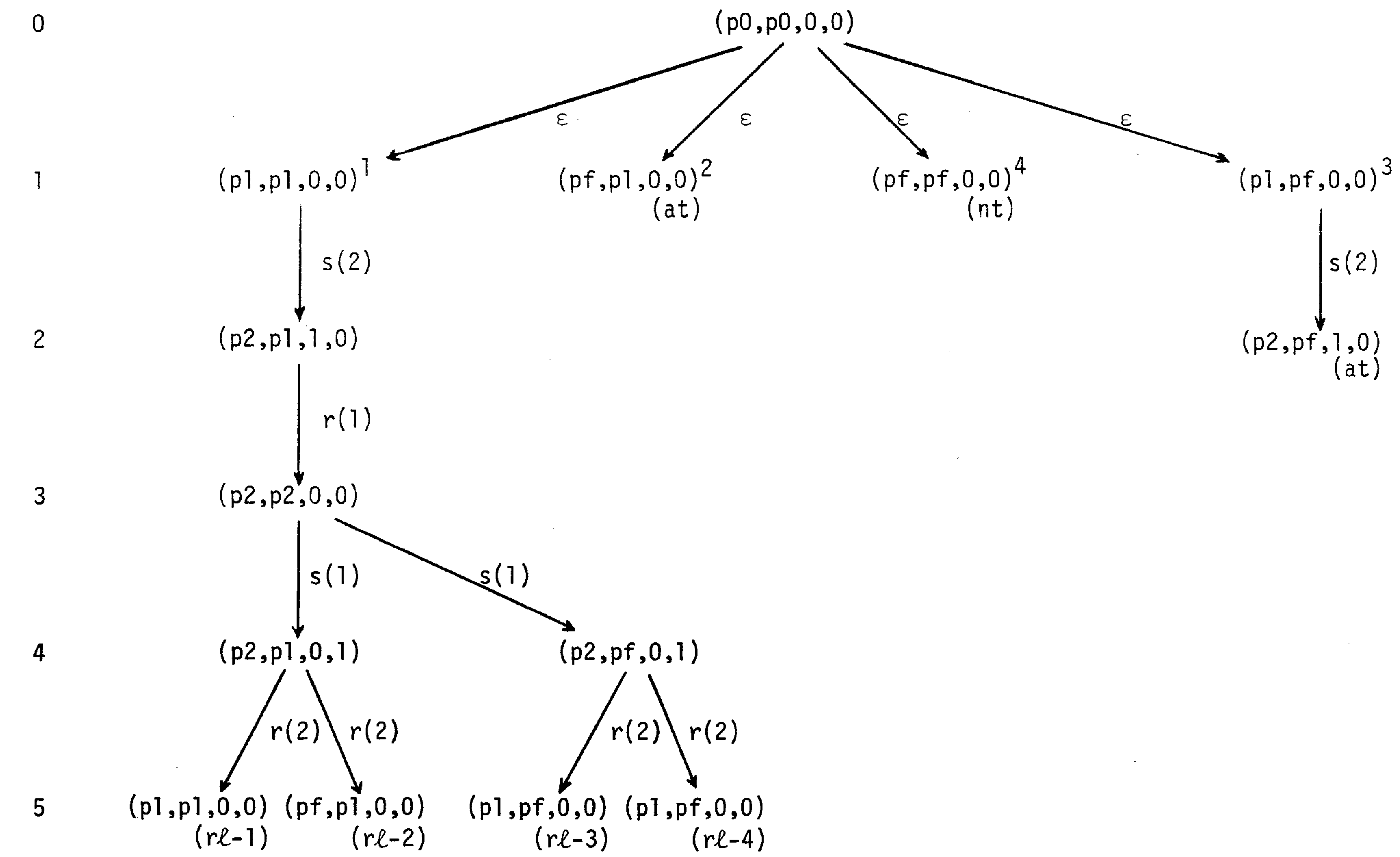
The last part of the algorithm (section 2.b.iv) which refers to normal and abnormal termination is the test for unpaired primitives. Although the second (at) is a potential error (the possible occurence of which will depend on the data part of the program), the first (nt) should appear in any synchronization tree because it expresses the complete match of the primitives. Note also that we have abstracted from the sf's the type and consequently the sequence of messages transmitted. If necessary, we could label the arcs in the ST with the type of the message that was sent and then by looking at these labels in the path from the root to some specific node figure out the associated sequence of messages transmitted. (However, for our applications this is not needed.) The example below helps to clarify the description of the algorithm.

Example 4.1:

Process 1: (s(2); r(2))*

Process 2: (r(1); s(1))*

Level:



In our example we begin with the initial state as the root of the tree which has as possible successor states the four possible starting configurations in level 1. State (p1,p1,0,0) in the tree moves to state (p2,p1,1,0) when process 1 sends a message to process 2. Similarly, the state (p2,p2,0,0) in level 3 can move to two different nodes by activating s(1) because at this point process 2 can either repeat the loop or enter the final state. This procedure continues until all nodes have been processed and thus indicating the complete construction of the synchronization tree. Note that in our example the reachability set is finite.

4.2 Analysis Using the ST

There are a number of important questions that can be answered using the synchronization tree. Let us consider some properties related to the questions raised in the previous section. Consider F as being any set of synchronization formulas (representing the corresponding set of processes). The associated synchronization tree ST(F) has the following properties stated in terms of informal lemmas.

Lemma 4.1:

The synchronization tree ST(F) is finite.

Proof:

In the construction algorithm for the synchronization tree ST(F) every time there is a valid move to a new node (state) which is a repetition of a previous node (state) generated in the tree we make this node a leaf. Since this is a repetition node all states reachable from it will also be added to the identical node in the tree.

On the other hand, we cannot have an infinite path in ST(F) (without repetition). Firstly, there is a finite number of possible configurations for the pointers since the sf's are finite. (We cannot keep the configuration of pointers changing forever.) Secondly, every time we move to a new node s(v) (an ancestor of some node v) where the pointer components are the same as in v and the counter components in s(v) are greater than or equal to the corresponding ones in v, we replace those components in s(v) which are strictly greater by the special symbol w (semantics "unbounded"). Again we cannot keep the configuration of pointers always different and the values of the input counters always decreasing because there is only a finite number of possibilities. The inclusion of an w component has the effect of decreasing the size of the system state (vector) because the value of w does not change and consequently, a repetition node is eventually reached. We have shown that there is no infinite path in the tree and therefore, the synchronization tree ST(F) is finite.

□

Lemma 4.2:

If the largest i th input-counter value of a vector in the reachability set RS(F) is finite then it also appears as the largest i th input-counter value of a node in the synchronization tree ST(F). Otherwise, the w component appears as the i th input-counter value of a node in ST(F).

Proof:

In the synchronization tree, we show all possible values of the i th input counter except when this component in the vector involves an w value. The introduction of an w value is the only way that we can lose information about the content of some input counter in the synchronization tree because it has the effect of mapping many states into the same state. Otherwise, if an w value appears in the i th input counter anywhere in the ST(F) then this component can become arbitrarily large (value infinite). As we know, when an w value is introduced in the i th input counter of a node s(v) (an ancestor of node v) then there is a node v where all the pointer components are the same as in s(v) and all counter components are less than or equal to the corresponding ones in s(v). In view of this, any sequence of valid moves from v is also possible from s(v) and, in particular, the sequence from node v to node s(v) in ST(F) can be infinitely repeated making the value of the i th input counter arbitrarily large. Therefore, the largest i th input counter in RS(F) is shown in ST(F) where we consider the value of w as being infinite. □

Corollary 4.3:

The communication between a set of synchronization formulas F is bounded if no w component appears in the synchronization tree ST(F).

In the lemmas above, we have shown that if the symbol w does not appear anywhere in the synchronization tree then the reachability set is finite and the communication between the sf's is bounded. Consequently, a practical result that can be derived easily is that we can determine the necessary bounds for each of the input buffers of the interacting processes by inspecting the ST and finding the largest value for the corresponding input counter.

Another important question raised in the last section and related to the synchronization tree is deadlock freeness. A deadlock situation always refers to a certain configuration of pointers (or specific primitives) in the sf's. In general, we may have an infinite number of deadlock states involving the same configuration of pointers. Note that a repetition leaf in the ST represents a loop and therefore the same deadlock node can be reached in several different situations. Indeed, a deadlock node in the synchronization tree is a representative of a class of deadlock states in the reachability set involving the same configuration of pointers. These ideas can be clarified in the example below.

Example 4.2:

Process 1: (s(2))*; r(2)

Process 2: (r(1))*; r(1)

Different deadlock situations involving the same configuration of pointers and using shuffled expressions: (r(2); r(1)), (s(2); r(1); r(2); r(1)), (s(2); r(1); s(2); r(1); r(2); r(1)), . . .

We can have different number of matches (and also in different orders) between the primitive s(2) in process 1 and the first primitive r(1) in process 2 before these two processors may enter a deadlock state. The first deadlock situation (r(2); r(1)) represented by (p2,p1,0,0) in the ST-notation would be the one to appear in the corresponding synchronization tree. The following theorem is an important result for the validation of the ST with respect to the deadlock-freeness property.

Def. 4.6:

A message oriented programming is deadlock-free iff there is never a set of processes blocked on receive operations such that the blocked receive operations form a circular chain.

Theorem 4.4:

A message oriented program is deadlock-free iff there is no deadlock node in the synchronization tree.

Proof:

a) First we will prove that if a message oriented program is deadlock-free then there is no deadlock node in ST. Assume that the message oriented program is deadlock-free and the ST(F) contains a deadlock node where F is the set of sf's associated with the program. Thus there is a path in the tree IS $\xrightarrow{\varepsilon}$ S_0 $\xrightarrow{t_0}$ . . . $\xrightarrow{t_i}$ d where d is a node which contains a deadlock and t_k are of the form r_{ij} or s_{ij} for some i, j belonging to the set of process names. We will now construct a valid sequence of moves IS $\xrightarrow{\varepsilon}$ S_0 $\xrightarrow{t_0}$ . . . $\xrightarrow{t_n}$ d' where $n \geq i$. The components of d involved in the deadlock must have value zero as must the corresponding components of d'. Also, the non-zero components of d' will equal the corresponding components of d unless this component is w. Assume d contains no w component. Then the algorithm generates the sequence of valid moves for the formulae F so as to reach that node in the tree. Thus $IS \xrightarrow{\varepsilon} S_0 \xrightarrow{t_0} \ldots \xrightarrow{t_i} d$ is a valid sequence of moves for the program (so $i = n$ and $d = d'$).

The non-trivial case occurs when d dontains w. Then the problem is that, since this component is arbitrarily large, receive operations can be "done" in the tree to construct a new node without the existence of corresponding send operations (labelling arcs in the path). If an w component is introduced in the tree then we reach in ST nodes $Sq > Sp$ (where Sp is a predecessor of Sq). In this case, we do not introduce an w component in the corresponding state of the valid sequence of moves but we insert a repetition factor for this subsequence in order to supply enough sending operations for the corresponding receiving operations. That is, we add to the sequence of valid moves the subsequence $(Sp \xrightarrow{t_p} \ldots \xrightarrow{t_{q-1}} Sq)^{\ell}$ where ℓ is equal to the length of the path in the tree. In this way, we have replaced a number that is "arbitrarily" large (infinite) by a number "sufficiently" large (finite). Thus, the generated sequence of valid moves for the formulae F leads to a deadlock situation in the program. But this contradicts the assumption that the program is deadlock-free and therefore, the first part of the theorem is proved.

b) In order to complete the proof of the theorem, we have to show that if there is no deadlock node in ST then the message oriented program is deadlock-free. Assume no deadlock node in the tree and that there is a deadlock situation in the program. Let us represent this deadlock by the following sequence of valid moves IS $\xrightarrow{\varepsilon}$ S_0 $\xrightarrow{t_0}$ S_1 $\xrightarrow{t_1}$ . . . $\xrightarrow{t_{i-1}}$ S_i $\xrightarrow{t_i}$ d where the S's are states, the t's are primitives, and "d" is the state which contains a deadlock involving a set DC of n input (deadlocked) counters. Starting from the root (which corresponds to the initial state IS) in ST we will try to build a path in the tree by applying the same sequence of communication primitives $(t_0, t_1, \ldots t_{i-1}, t_i)$. If this path does not introduce an w component in ST then obviously d is in the synchronization tree. Otherwise, if an w component is introduced, then we reach in ST nodes $r_q > r_p$ (where r_p is a predecessor of r_q in the tree). Now we have two different cases for the introduction of an w component. Firstly, there is no problem if all w components that are introduced involve input counters which do not belong to DC. This is clear from the following reasoning: There is a 1-1 correspondence between states in the above sequence of moves and nodes in the corresponding path in ST up to state S_{q-1} (node r_{q-1}). At this point, instead of r_q being exactly s_q, some components of r_q are now w. From here on, in the path, these components always equal w. Thus for each s_k, $k > q$, the corresponding components of r_k are always equal to w. Note that a node with components which are w can never lead to nodes in which the system is blocked on these components since the "meaning" of w is that as many messages as are needed are actually available. Thus no receive can be blocked on these components. Other w components (<u>not</u> in DC) may be introduced in nodes corresponding to states s_ℓ for $\ell > q$, and the above discussion holds for these components as well.

Secondly, if any of the w components that are introduced involve an input counter which belongs to DC then the value of the counter is being made arbitrarily large in this path and consequently, the deadlock node corresponding to d is not reached on any path from this node. (Note that the counter must be zero to be part of DC but the introduction of w never allows this counter to become zero again on any path leading from this node.) In this case, we will construct a new sequence of valid moves using the one given above as a model and this sequence of moves will have the following properties. The initial portion of the sequence of moves will be $IS \overset{\varepsilon}{\rightarrow} S_0 \overset{t_0}{\rightarrow} \ldots \overset{t_{p-1}}{\rightarrow} S_p$. The new sequence will be shorter (i.e., fewer moves than in the original sequence). All moves in the new sequence will have appeared in the old one but not necessarily in the same order. The new sequence will end in a deadlock state but not in general d and not necessarily involving the same set DC.

We will use primes to distinguish states of the new sequence from the corresponding states of the old sequence. The construction of the new sequence of valid moves is defined by the following algorithm.

(i) Make the initial portion of the new sequence of valid moves $IS' \xrightarrow{\varepsilon} S'_0 \xrightarrow{t'_0} \ldots \xrightarrow{t'_{p-1}} S'_p$ equal to $IS \xrightarrow{\varepsilon} S_0 \xrightarrow{t_0} \ldots \xrightarrow{t_{p-1}} S_p$.

(ii) Erase from the original sequence all the primitives (and associated states) which appear between the states S_p and S_q and belong to any of the sending processes that caused an introduction of an w component in a deadlocked counter. (This is equivalent to a removal of the iteration which caused the introduction of the w component from the execution of the corresponding process.)

(iii) Let us consider the last state reached in the new sequence of valid moves S'_i and make the set of blocked processes BP empty. Take the first unmarked valid move m in the original sequence after S_p by considering the corresponding primitive and associated process $p(m)$.

(iv) If the move m is also a valid move from S'_i then make it the next valid move in the new sequence and mark it in the original sequence. Go back to step (iii).

(v) If the move m is not a valid move from S_i' then include the process p(m) in the set of blocked processes BP. Take the next unmarked valid move in the original sequence which is not a primitive of any process in BP as the new m and go back to step (iv). If there is no such next move (i.e. we have reached d) then stop.

The removal of the primitives of process i because of the introduction of an w component in a deadlocked counter c_{ij} now makes the number of s_{ij} less than the number of r_{ji} in the original sequence. This imbalance assures the correctness of the termination condition of the algorithm (in step (v)) since we know that at least the last primitive r_{ji} cannot be a valid move and it will not be marked. By construction of the algorithm we assure that the new sequence is also a sequence of valid moves. The new sequence is shorter because some moves in the original sequence were erased in the execution of step (ii) above.

Now we have to show that the last state d' in the new sequence of valid moves is also a deadlock state. Note that all the primitives erased in the original sequence (in step (ii)) were from processes involved in the deadlock state d. (The input counters that do not belong to DC were not affected.) All the deadlocked prccesses associated with the input counters in DC continue blocked in the new sequence of valid moves. They may be blocked at some earlier point because of the possible removal of primitives in step (ii) above. There are two cases to consider for the processes associated with input counters that do not belong to DC. Firstly, if the process executes all its previous primitives from the original sequence (i.e., all its primitives in the original sequence were also included in the new sequence) then it cannot block in the state d' a process which is involved in the deadlock situation. This is because the process was not supposed to do this blocking in the original sequence and we know that all its primitives in the original sequence were also included in the new sequence (i.e., the process has performed all sending operations). Secondly, if the process does not execute all its previous primitives from the original sequence then it is blocked on some process which is related to DC by an input counter. (As explained before we know that it cannot be blocked on a process that has executed all its primitives.) In this case the input (deadlocked) counter is added to DC. Thus, we have in the state d' a set of n' (where n' ≥ n) processes, each of them blocked on another process of the group and consequently, there is a deadlock situation in state d'.

Given the new sequence of valid moves $IS \xrightarrow{\varepsilon} S'_0 \xrightarrow{t'_0} S_1 \xrightarrow{t'_1} \cdots \xrightarrow{t'_{i'-1}} S'_{i'} \xrightarrow{t'_{i'}} d'$ we repeat the same process described previously of trying to build the corresponding path in the tree. If we do not manage to build the path in the tree then we use the algorithm above to construct another new sequence of valid moves. As shown before, this new sequence is always shorter and eventually, we will be able to build the path in the tree corresponding to the sequence of valid moves. Thus, the deadlock state (the last state in the sequence) is represented in the tree and this contradicts the initial assumption that there is no deadlock node in ST. Consequently, the second part of the theorem is proved. □

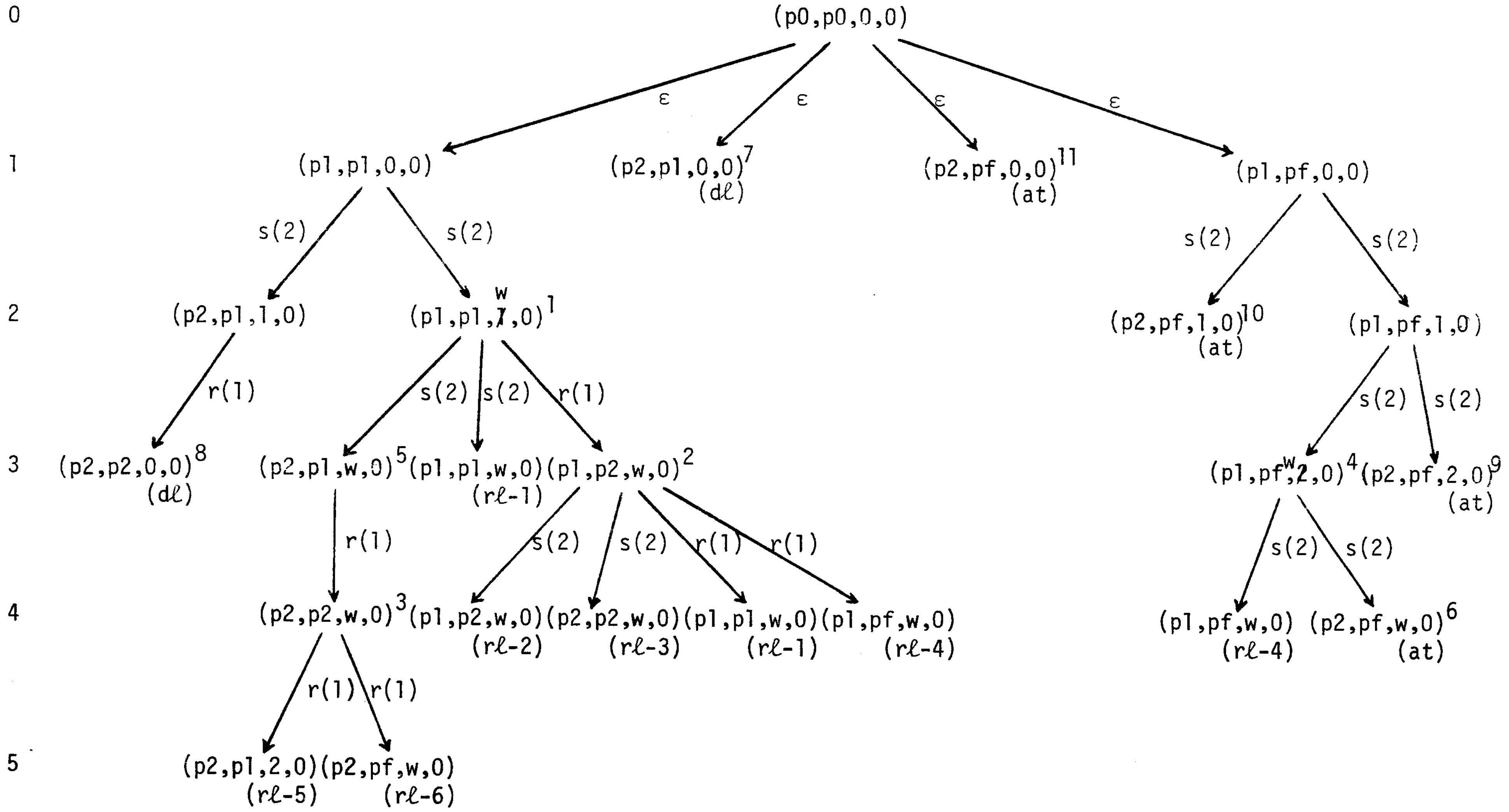
Unpaired-primitives nodes appear in the synchronization tree when we have possible imbalances between the sending and corresponding receiving operations in the sf's. It is easy to see that this verification really happens in the construction of the ST because the algorithm tries all different matches between the iterations in the sf's. (In fact the introduction of w components in ST id due to an imbalance between these iterations.) Other properties may be analyzed and solved by using the synchronization tree. For example, the reachability problem which seems to be still open for Petri nets (in spite of an unpublished reference in [19]) and was solved for symmetric nets [14] could be investigated for ST's. (The reachability problem asks for a given net if a state S_1 is reachable from another state S_0.) We will leave other questions that may have significance for message oriented programming to be studied in future works. Now we give another example of the use of the synchronization tree to answer the questions treated above.

Example 4.3:

Process 1: (s(2))*; r(2)

Process 2: (r(1); r(1))*

Level:



Deadlocks are identified in the syncrhonization tree by nodes in which we have a set of input counters (one for each receiving process) with value equal to zero, the pointers in the corresponding processes indicate receiving operations involving these counters and the blocked processes form a cycle. The nodes 7 and 8 are examples of total deadlock. Unboundedness is characterized by the introduction of an w component in ST. In this case, we have the introduction of w components in the nodes 1 and 4. Unpaired primitives are identified by final states (leaves) in the tree where some processes are blocked forever trying to receive messages from processes that have finished or these final states contain unprocessed messages. The nodes 6, 9, 10 and 11 in the ST above show examples of unpaired primitives.

5. Conclusions

In this report we have outlined a calculus to treat in a systematic way the synchronization properties for message oriented programming. These properties include such things as absense/presence of deadlocks, unpaired primitives, use of unbounded buffers (capacity of the channels involved in the communication), etc. In order to answer these questions, we introduce a technique (called the synchronization tree (ST)) which is a finite representation for the reachability set. The correctness of the technique was supported by a number of results proved in the lemmas above.

Zafiropulo et al. [22] and Gouda [12] have studied synchronization properties of two-processes protocols. In their models, they keep the order in which the messages have arrived. This ordering forces the storage of the whole sequence of messages which have arrived in any of the input buffers. This assumption makes sense for protocols but may not be necessary for programming. Their requirement makes some of the synchronization questions (such as deadlock) undecidable (Brand [3]). In our model for message oriented programming, we have removed this constraint by introducing one input queue for each sending process and we have only considered the counting of the types of messages (not the sequence). The result in Theorem 4.4 compensated for the simplicity of our approach. In [12], Gouda also analyzed the case of n-processes communication by developing sufficient conditions under which his protocol machines (n-ary communication) could be represented by free choice Petri nets (a subclass of Petri nets). In this sense, our result can be considered as more general since we know that Petri nets can be represented as vector addition systems [19].

As mentioned earlier, we want to emphasize again that our technique with slight modifications also works for other combinations of "blocked" and "unblocked" primitives. Other directions for further research include: (i) extension of the calculus to allow the analysis of other properties, (ii) to study to what extent the sequence of messages can be introduced in our technique, and (iii) to study the usefulness of the synchronization calculus in other areas such as verification of protocols.

6. References

[1] Baskett, F., Howard, J.H., Montague, J.T.: Task Communication in DEMOS; Proceedings of the 6th ACM Symposium on O.S. Principles, 1977.

[2] Brinch Hansen, P.: The Nucleous of an Operating System; CACM, April 1970 (pp. 238-241, 250).

[3] Brand, D., Zafiropulo, P.: Synthesis of Protocols for an Unlimited Number of Processes; Proceedings of the Computer Network Protocols Conference, 1980.

[4] Cheriton, D.R., Malcolm, M.A., Melen, L.S., Sager, G.R.: Thoth, A Portable Real-Time Operating System; CACM, February 1979.

[5] Cunha, P.R.F., Lucena, C.J., Maibaum, T.S.E.: On the Design and Specification of Message Oriented Programs; Research Report CS-79-25, University of Waterloo, June 1979 (to appear in the Int. J. of Computer and Information Sciences).

[6] Cunha, P.R.F., Maibaum, T.S.E.: A Communications Data Type for Message Oriented Programming; Lecture Notes in Computer Science, Springer-Verlag, Vol. 83, 1980.

[7] Cunha, P.R.F., Lucena, C.J., Maibaum, T.S.E.: A Methodology for Message Oriented Programming; Proceedings of the 6th GI Conference on Programming Languages and Program Development, Darmstadt, March 1980.

[8] Cunha, P.R.F., Maibaum, T.S.E.: "Resource = Abstract Data Type + Synchronization" - A Methodology for Message Oriented Programming; Research Report CS-80-28, University of Waterloo, May 1980.

[9] Cunha, P.R.F., Lucena, C.J., Maibaum, T.S.E.: Message Oriented Programming - A Resource Based Methodology; Research Report CS-80-32, University of Waterloo, June 1980.

[10] Dykstra, E.W.: Hierarchical Ordering of Sequential Processes; in Operating Systems Techniques, Academic Press, New York, 1972 (pp. 72-93).

[11] Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.F.: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types; IBM Research Report RC 6487, 1976.

[12] Gouda, M.: Protocol Machines, Towards a Logical Theory of Communication Protocols; Ph.D. Thesis, University of Waterloo, 1977, also CCNG TR-T-74, University of Waterloo, Canada, 1978.

6. References - Cont'd.

[13] Guttag, J.V., Horowitz, E., Musser, D.R.: Abstract Data Types and Software Validation; CACM, Vol. 21, No. 12, 1978 (pp. 1048-1064).

[14] Hack, M.: Decidability Questions for Petri Nets; Ph.D. Thesis, MIT, Cambridge, Mass., Dec. 1975, also TR-161, Laboratory of Computer Science, MIT, June 1976.

[15] Hoare, C.A.R.: Proof of Correctness of Data Representations; Acta Informatica, Vol. 1, No. 1, 1972, (pp. 271-281).

[16] Hoare, C.A.R.: Monitors, an Operating System Structuring Concept; CACM, October 1974 (pp. 549, 557).

[17] Hoare, C.A.R.: Communicating Sequential Processes; CACM August 1978 (pp. 666-677).

[18] Jammel, A.J., Stiegler, H.G.: Managers Versus Monitors; Proceedings of the IFIP 1977 (pp. 827-830).

[19] Karp, R.M., Miller, R.E.: Parallel Program Schemata; J. Computer and Systems Science, Vol. 3, No. 4, 1969 (pp. 167-195).

[20] Manning, E.G., Peebles, R.W.: A Homogenous Network for Data-Sharing Communications; Computer Networks 1, 1977 (pp. 211-224).

[21] Peterson, J.L.: Petri Nets; Computing Surveys, Vol. 9, No. 3, Sept. 1977 (pp. 223-251).

[22] Zafiropulo, P., West, C.H., Rudin, H., Cowan, D.D., Brand, D.: Toward Analyzing and Synthesizing Protocols; IEEE Transactions on Communications, April 1980.

[23] Zave, P.: On the Formal Definition of Processes; Conf. on Parallel Processing, Wayne State University, IEEE Computer Society, 1976.