Distributed Synchronization and Regularity

Gregor V. Bochmann
Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, Canada

The communication delays between the different components of a distributed system often create problems for the logical consistency of the overall system behaviour. The ideas presented in this paper suggest to eliminate these problems by observing certain regularity constraints during the system design, which guarantee that the logical behaviour of the system is independent of the communication delays. The paper presents a descriptive model for the specification of distributed systems, and defined system properties which imply regular system behaviour. A sufficient condition for checking the regularity of a given system is given. The application of the concepts presented is illustrated by several examples.

Keywords: Distributed synchronization, communication delays, deadlocks, mutual exclusion, distributed system modules, distributed system design, regularity constraints, synchronization primitives, distributed system verification.

Gregor V. Bochmann received the Diplom in physics from the University of Munich, Munich, Germany, in 1968, and the Ph.D. degree from McGill University, Montreal, P.Q., Canada, in 1971. He has worked in the areas of programming languages and compiler design, communication protocols, and software engineering. He is currently Associate Professor in the Département d'Informatique, University of Montreal. His present work is aimed at design methods for communication protocols and distributed systems. In 1977-1978 he was a Visiting Professor at the Ecole Polytechnique Fédérale at Lausanne, Switzerland.

[1] This work was performed while the author was on leave at the Département de Mathématiques de l'Ecole Polytechnique Fédérale de Lausanne, Suisse. An earlier version of this paper has been presented at the Berkeley Workshop, San Fransisco, August 1978, under the title "Synchronization in distributed system modules".

© North-Holland Publishing Company
Computer Networks 3 (1979) 36–43

1. Introduction

It is common practice to subdivide complex systems into a number of modules. Usually, a module is characterized by a given data structure and a set of operations that may be executed on this data. Such a module may be considered an instance of an abstract data type. Seen from the outside, a data type is characterized by the operations that may be invoked on the module by other modules, i.e. the order in which these operations may be executed and the parameter values exchanged. The other point of view considers the implementation of these operations inside the module, for which it is usually necessary to specify the data structures of the module and the procedures that implement the externally available operations. There may also be some internal bookkeeping operations invoked inside the module.

Most systems contain some kind of parallel activities. Thus, in principle, it is possible for the operations of a given module to be invoked by several other modules, in an arbitrary order. However, many module types only function correctly if the operations are invoked in a specific order. Therefore, the correct synchronization of the module operations must be enforced in some appropriate way, perhaps by delaying the calling modules. Many different tools have been developed for the specification of synchronization mechanisms, for example semaphores, monitors, conditional critical regions, conditions on history counter variables [1], etc. In this paper, we use a kind of conditional critical region (see section 2).

1.1. The problem considered

In this paper we consider the problems that arise from the distribution of a module over several physical components, which may be located at different places. Because of the communication delay between the different components, the sharing of information between the different components of a module becomes more complicated. For example, reading a variable in a distant component generally involves a delay implying that the variable may already have been updated when the value read is obtained. Another example is the potential deadlock

resulting from call collision, which occurs when two communication stations call one another at the same instant and, due to the transmission delay, each finds the other one busy. In a local context, these problems may be resolved by the introduction of critical regions for the access of shared variables, but this approach is difficult in a distributed context.

The situation is complicated by the fact that the different components may proceed with their local processing at a varying speed independent of one another. To obtain a meaningful system behaviour, it is necessary to introduce some kind of synchronization between the different components. A given communication protocol between two communicating components usually describes a specific synchronization behaviour. This paper considers the more general problems of describing an arbitrary synchronization scheme between several (possibly more than two) asynchronous components, and analysing its behaviour in the presence of varying communication delays.

1.2. The approach

We propose the concept of "regularity" for the analysis of distributed system modules. Broadly speaking, a distributed module is regular if its logical behaviour is independant of the internal communication delays. The analysis of a regular module is simplified. A proof of the correct behavior of a module which is known to be regular may be made by assuming negligeable communication delays, which is equivalent to assuming a non-distributed implementation. Therefore the analysis is reduced to the non-distributed case. However, the problem remains how to decide whether a distributed module is regular or not. A sufficient condition for regularity is given below, which applies in many cases.

In order to clearly specify and analyse a distributed system, it seems necessary to describe it in some convenient formalism. For this purpose we have adopted a particular descriptive model which avoids the use of variables shared at a distance, but uses instead actions, executed in a given component, which may be initiated by remote components, as explained in section 2. In section 3 the concept of regularity is formally defined in terms of this model, and a sufficient condition for regularity is given. These ideas are applied to several simple examples in section 4.

2. The descriptive model

In the following we give only a brief explanation of the descriptive model used for the specification of distributed modules since it is similar to models described elsewhere [2,3].

2.1. The basic model

In the basic model, the distribution aspect is ignored. A module is characterized by a set of variables (declared within the module) and a set of operations. The (internal) state of the module is characterized by the values of these variables. Each operation defines a set of possible state transitions. They are characterized by the "enabling predicate" of the operation, which is a boolean function of the variables, and the "action" of the operation, which updates the variables and is usually written in some high-level programming language. Only when the enabling predicate is true may the operation be fired,

```
Resource Module (the resource manager)
Variables
   free: boolean;
Operations
   request;
      when free
      do    free := false;
   release;
      free := true;
Initially
   free := true;

User i Module (the i-th user module of the resource)
Variables
   state: (start, reserved, done);
   ...
Operations
   request;
      when      state = start
      do begin  Resource.request;   state := reserved end;
   usage;
      when      state = reserved
      do begin...{use resource};   state := done end;
   release;
      when      state = done
      do begin  Resource.release;   state := start end;
   ...
Initially
   state := start; ...
```

Fig. 1. Example of mutual exclusion between *N User* modules accessing a shared resource.

i.e. the associated action is executed, thus performing a state transition. We assume mutual exclusion between the firing of different operations.

To illustrate this basic model, we show in Fig. 1 an example of mutual exclusion between N *User* modules accessing a shared resource. We use a notation similar to Pascal [4] to specify the variables, enabling predicates, and actions. An operation is written in the form "*when* ⟨enabling predicate⟩ *do* ⟨action⟩". We write *name. operation* to indicate the initiation of an operation in another module or component. To control the resource access, the example uses the customary *request* and *release* operations which must be called in a consistent order by the user modules. A cyclic order of execution is enforced for each *User* module by the *state* variable.

2.2. The distributed model

We will now consider a module to be distributed over several physical "components". A typical situation is shown in fig. 2, where the module A is distributed over the physical components, X, Y, Z (possibly placed in different geographical locations). Some operations of the module are invoked by the modules B and D in components X and Z, respectively. A, in turn, invokes some operations of the module C, which is completely inplemented in component Y. It also invokes operations of submodules located in components X and Z.

For the description of such distributed modules, we propose in the following a distributed model related to the basic model above. The model is a kind of language containing simple primitives for specifying synchronization and interactions between actions executed in the different components of a module. To deal with the distribution aspect we introduce the following conventions (in addition to those of the basic model):

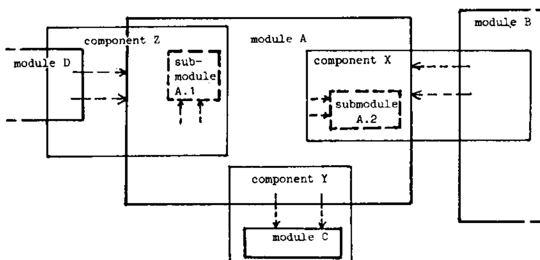
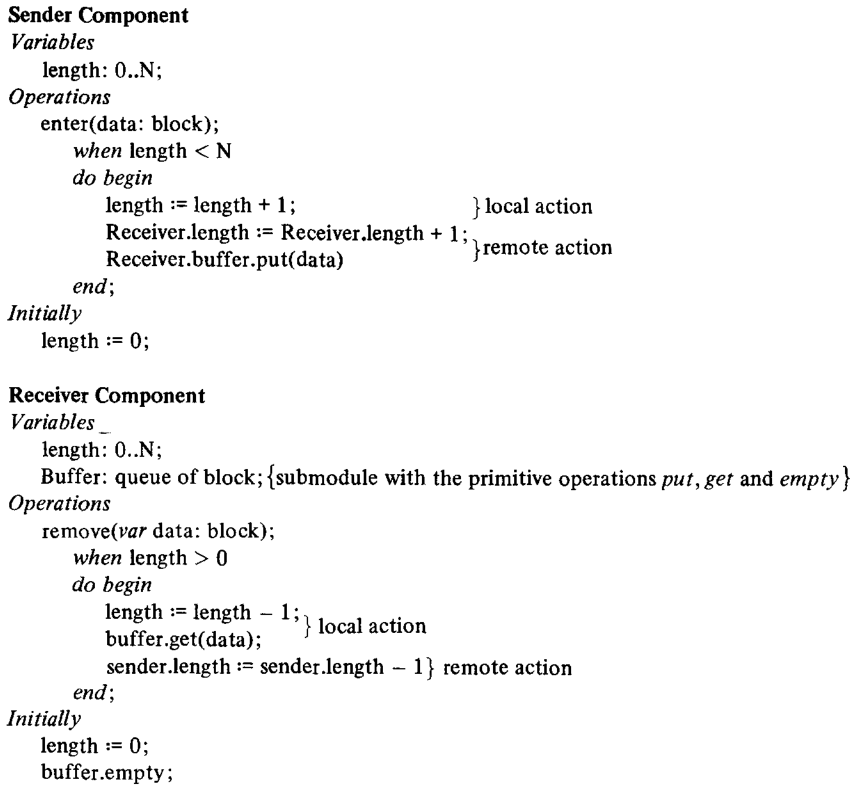


Fig. 2

(a) Each variable of a distributed module is located in a single component.
(b) Each operation is assigned to a single component, where it is initiated (either internally by the module, or in interaction with another module), and its enabling predicate depends only on variables located within that component.
(c) The action of an operation is partitioned into a certain number of localized actions, each updating the local variables of a particular component, such that the new values depend only on the previous values of the local variables of that component and (possibly) on the variable values of the initiating component. We call "local action" an action localized in the initiating component, and "remote action" an action localized in another component.
(d) Mutual exclusion between local and remote actions of different operations is ensured separately for each component.
(e) When an operation is fired, its local action is executed immediately, and each corresponding remote action is executed some finite time later (possibly using the values of the variables in the initiating component as updated by the execution of the local action). During the intermediate time intervals, other actions may be executed on the different components.

It is clear that an implementation of this model may be obtained by using an inter-component communication subsystem for the transfer of messages. When an operation is fired in the initiating component, a message, containing the identification of the operation and the necessary variable values of the initiating component, is sent to each component where a remote action must be executed. Depending on the message transmission service provided by the communication subsystem, different varieties of point (e) above may be considered: (e1) as above (message delivery is guaranteed); (e2) as above, but some remote actions may never be executed (messages may be lost). In addition sequentiality of the message delivery and execution of remote actions may or may not be guaranteed for certain pairs of system components. In general, these characteristics of the communication subsystem have a strong impact on the behavior of the distributed system module.

In the above discussion, we have considered only a single system module, such as for example module A of the figure above. We have ignored the fact that a given physical system component will, in general,



```
Sender Component
Variables
   length: 0..N;
Operations
   enter(data: block);
      when length < N
      do begin
         length := length + 1;                     } local action
         Receiver.length := Receiver.length + 1;   }
         Receiver.buffer.put(data)                 } remote action
      end;
Initially
   length := 0;

Receiver Component
Variables
   length: 0..N;
   Buffer: queue of block; {submodule with the primitive operations put, get and empty}
Operations
   remove(var data: block);
      when length > 0
      do begin
         length := length - 1;  }
         buffer.get(data);      } local action
         sender.length := sender.length - 1} remote action
      end;
Initially
   length := 0;
   buffer.empty;
```

Fig. 3. Implementation of a queuing module distributed over two components.

contain components of several modules which may interact locally through external module operations.

As in example, we show in Fig. 3 an implementation of a queuing module distributed over two components. It may be used to transfer data from the *sender* component to the *receiver* component. It is derived from a straight-forward non-distributed queuing module by duplicating the variable *length* in both components, and assigning the *buffer* variable to the *receiver* component. This corresponds to the design strategy of assigning a variable to the component where it is used, possibly creating several copies of the same "logical variable" in different components. (We note that the module assumes sequential message transmission from the sender to the receiver component, but not necessarily in the opposite direction).

3. Regular systems

In this section we consider a certain class of constraints which we call regularity, which may be satisfied by distributed systems, described in the model outined above, and which, when satisfied, allow a logical system validation ignoring the possible message transmission delays between the components. The following subsection explains some useful concepts and notations needed for the following discussion.

3.1. Traces of operation execution

We consider a system module with a certain set of possible operations. The (local and remote) actions of these operations are identified by action symbols. A "trace" is a string of action symbols, and indicates a possible execution sequence for these actions. A trace contains all local and remote actions of the operations in the order in which they are executed, as seen by a hypothetical outside observer. Given that the module is in a certain internal state, which is characterized by the values of its local variables and the set of "outstanding" remote actions, only certain actions may be executed next. A local action, initiating a new operation, may only be executed when the

enabling predicate of the operation is satisfied. A remote action may only be executed when it is outstanding, i.e. when the corresponding local action has been executed previously. This limits the set of possible traces for a given module.

We assume that each action has a deterministic effect on the local variables of the executing component. Then each given trace which is possible from a given state s of the module leads to a particular module state, which we write $(s)\sigma$. We say $(s)\sigma$ is "defined" iff σ is a possible trace starting from s. We say that two states s and s' are "equivalent" iff for all traces σ, $s(\sigma)$ is defined iff $(s')\sigma$ is defined, i.e. the same traces are possible from s and s'.

3.2. Regularity

As mentioned above, for each possible trace the remote actions are executed sometimes *after* the corresponding local actions. If the remote actions are executed *immediately after* the corresponding local actions, one obtains traces of the form $a_L a_R b_L b_R^{(1)} b_R^{(2)} ... b_R^{(n)} c_L d_L d_R ...$ (where we distinguish local and remote actions by the indexes L and R respectively, and assume that the operation c has no remote action, the operations a and d have one, and b has n remote actions). We call such traces "delayless", since all possible traces are of this form in the case that the communication delays between the components can be ignored.

Given a module state s with no outstanding remote actions, and a possible trace σ starting from this state, we write $\hat{\sigma}$ for the "corresponding delayless" trace, defined to be the delayless trace which contains the same local actions as σ, and in the same order as σ. For example, the delayless trace above is the corresponding delayless trace of $a_L b_L c_L d_L d_R a_R b^{(1)} b_R^{(2)} ... b_R^{(n)}$, as well as of $a_L a_R b_L c_L d_L b_R^{(1)} d_R b_R^{(2)} ... b_R^{(n)}$, and many others.

As far as the logical behaviour of the module is concerned, a trace and its corresponding delayless trace are considered equivalent, since both define the same operation sequence. This is the basis for the following definition of regularity which corresponds to the approach explained in the Introduction.

Definition: A distributed module is "regular" if for any trace σ such that $(s_0)\ \sigma$ is defined, where s_0 is the initial state of the module, the following holds:
(a) $(s_0)\ \hat{\sigma}$ is defined, and
(b) $(s_0)\ \hat{\sigma}$ is a state equivalent to $(s_0)\sigma$.

The condition (a) implies that each operation sequence that is possible in the presence of arbitrary, but finite communication delays is also possible in the absence of delays. Therefore delays cannot introduce any "new" behaviour which would not be possible in the case of negligeable delays. On the other hand, condition (b) implies that all operation sequences that are possible in the case of negligeable delays are also possible in the presence of delays. Therefore the deadlock and liveness properties of the module are independent of the delays. We may conclude that the behaviour of the module, as characterized by the possible operation sequences that may occur, is independent of any (finite) communication delays between the different module components.

3.3. Commutation relations and regularity condition

We give in the following a sufficient condition for a system to be regular. This condition may be checked by considering separately each component of the module, ignoring the particular form of intercomponent communication. Therefore the analysis is relatively simple.

Definition: An action a "semi-commutes" with an action b if for all traces σ such that $(s_0)\sigma\ b\ a$ is defined, $(s_0)\sigma\ b\ a$ is a state equivalent to $(s_0)\sigma\ a\ b$. We say that a "semi-commutes strongly" with b if for all states s such that $(s)\ b\ a$ is defined, $(s)\ a\ b$ is a state equivalent $(s)\ b\ a$.

We note that any action of a given component semi-commutes strongly with any action of a different component, except for a remote action with the corresponding local action.

Definition: A remote action of a module component is "regular" if it semi-commutes with all actions of the same component.

Lemma: *A remote action a_R is regular iff for* any *possible trace of the form $\sigma_1 a_L \sigma_2 a_R$, where a_L is the local action corresponding to a_R, $(s_0)\ \sigma_1 a_L \sigma_2 a_R$ is a state equivalent to $(s_0)\ \sigma_1 a_L a_R \sigma_2$. (The proof is straightforward).*

Proposition: *If all remote actions of a module are regular then the module is regular. (The proof, using the lemma above, is straightforward).*

4. Examples

In this section we consider several examples of distributed modules, and analyse their regularity using the tools developed in the preceding section.

4.1. The queuing module

We consider the distributed queuing module of section 2.2. To check the applicability of the above Proposition, it is sufficient to investigate the commutation of $enter_R$ with $remove_L$, and $remove_R$ with $enter_L$. We find strong semi-commutation in both cases. Therefore the module is regular.

4.2. An erroneous mutual exclusion algorithm

An adaptation of the mutual exclusion algorithm of section 2.1. to a distributed environment is not easily obtained. If the strategy of variable assignment and duplication, which was successful in the case of the queuing module, is applied in this example we obtain a distributed module such as in Fig. 4. (the variable *free* is duplicated). It is easy to see that this module is not regular. For example, the trace $user_1.request_L$ $user_2.request_L$ $user_1.request_R^{(2)}$ is a possible trace, whereas the corresponding delayless trace $user_1.request_L$ $user_1.request_R^{(2)}$ $user_2.request_L$ is not, which is a contradiction to condition (a). (We note that $user_1.request_R^{(2)}$ is the remote action of the *request* operation initiated in the $user_1$ component executed in the $user_2$ component). By the way, this also results in a violation of the mutual exclusion in the case of communication delays, and is an example of a "new" behaviour introduced by non-negligeable delay.

```
User i Component
Variables
   state: (start, reserved, done);
   free: boolean;
   . . .
Operations
   request;
   when state = start and free
   do begin
         state := reserved;                    } local action
         free := false;
         for all j ≠ i do userj.free := false;  } remote actions
         end;
   usage;
      when state = reserved
      do begin ... {use resource}; state := done end;
   release;
      when state = done
      do begin
            state := start;                    } local action
            free := true;
            for all j ≠ i do userj.free:=true;  } remote actions
      end;
Initially
   state := start; free := true;
```

Fig. 4. Distributed module.

4.3. A centralized mutual exclusion algorithm

Similar to the algorithm of section 2.1, the distributed resource sharing module of Fig. 5 uses a centralized resource manager. Mutual exclusion is maintained by a rather more complex exchange of "messages" between the user components and the manager.

To prove the regularity of the algorithm we consider, as in section 4.1., the commutation of each remote action with the local actions executed in the same component. This leads to the following pairs of actions:

- $allocate_R$ commutes with $request_L$ strongly,
- $allocate_R$ semi-commutes with $usage_L$ strongly,
- $allocate_R$ semi-commutes with $release_L$ (there is no possible trace of the form "... $release_L$ $allocate_R$"),
- $request_R$ semi-commutes with $allocate_L$ strongly,
- $release_R$ semi-commutes with $allocate_L$ (there is no possible trace of the form "... $allocate_L$ $release_R$").

Therefore the condition of the Proposition is satisfied, and the module is regular.

4.4. A distributed mutual exclusion algorithm

Dijkstra has described self-stabilizing algorithms with distributed control [5], which provide mutual exclusion in the stabilized situation. His solution with K-state machines may be re-written in the form given in Fig. 6, where nothing is assumed about the initial values of the variables, and $\oplus$ stands for addition modulo N. It can be shown [6] that the algorithm is self-stabilizing, which means that for arbitrary initial values of the local variables S and L in the different components the system will enter, after a finite number of operations, a "stable" state such as for instance the state with $L = S = 0$ in all components.

To show the regularity of the algorithm, we

```
Resource Component (the resource manager)
Variables
   free: boolean;
   requests: queue of userid; {user identification i.e. 1..N}
Operation
   allocate;
      var next: userid;
      when free and not empty (requests)
      do begin
         free := false;                    }local action
         requests.get (next);
         next.state := reserved;           }remote action
      end;
Initially
   free := true;
   requests.empty;
User i Component
Variables
   state: (start, reserved, done);
   ...
Operations
   request:
      when state = start
      do begin
         Resource.requests.put(myid)  }remote action
      end;
   usage;
      when state = reserved
      do begin
         ... {use resource}
         state := done
      end;
   release;
      when state = done
      do begin                             }local action
         state := start;
         Resource.free := true;            }remote action
      end;
Initially
   state := start;
```

Fig. 5. Distributed resource sharing module.

consider the following pairs of actions:

- *send-status*$_R$ and *usage*$_L$ (for $i = 0$),
- *send-status*$_R$ and *usage*$_L$ (for $i > 0$), and
- *send-status*$_R$ and *send-status*$_L$ (the remote action is executed in the same component as the local action, but belongs to a different operation).

The last pair commutes strongly. The first two pairs semi-commute (assuming as initial state $L = S = 0$ in all components), but not strongly, i.e. not during the self-stabilization phase when nothing can be assumed about the variable values. Once the system is stabilized, the semi-commutation relation ensures regularity.

```
User i Component
Variables
   S, L: 0..K - 1 {own machine state, and latest knowledge
                   about the state of the left neighbour}
Operations
   send-status;
      do user_{i⊕1}.L := S; } remote action
   usage (for i = 0);
      when L = S
      do begin
         ... {use resource};
         S := (S + 1) mod K
      end;
   usage (for i > 0);
      when L ≠ S
      do begin
         ... {use resource};
         S := L
      end;
```

Fig. 6. Dijkstra's solution with K-state machines.

5. Concluding remarks

The communication delays between the different components of a distributed system often cause problems for the logical consistency of the overall system behaviour. The ideas discussed in the paper suggest how to eliminate these problems by observing certain regularity constraints during the system design which guarantee that the logical behaviour of the system is independent of the communication delays. We believe that the concept of regularity may lead to a better understanding of the problems of distributed systems, to the design of simpler systems and more reliable implementations. The paper presents a descriptive model for the specification of distributed systems and a simple condition which guarantees the regular behaviour of the system. Several examples are given which show how these concepts apply for some simple systems. More work is required to study the applicability of the ideas for larger and more complex systems.

The concepts developed in this paper are related to many algorithms designed for distributed implementation, in particular communication protocols, routing algorithms in data networks, and synchronization algorithms for distributed data bases.

In a recent paper [7] Lamport presented some synchronization algorithms which rely on a particular property of shared multi-digit variables, namely that the value read is always smaller than or equal to the

last value written. As in the case of synchronization based on counter variables [1], he complies with the restriction that the values of the shared variables are never decreased. Assuming individual increments are always one, this leads to the situation where the value read from a shared multi-digit variable is more or less "old". The same situation holds in our distributed model where the variables of distant components contain "old" values until they are updated by remote actions. When re-written in our model, with an appropriate distribution of variables, most of Lamport's algorithms turn out to be regular, and may therefore be used in a distributed context, although not originally intended for this purpose.

While the regularity condition given in this paper is particularly simple to check, it is certainly not satisfied by all regular systems. We know of a simple data transmission protocol with message numbering and retransmission after time-outs (for recovering from lost messages) which is essentially regular, but has some non-regular remote actions. Some more powerful regularity condition may be useful for proving the regularity in more general situations.

It would be very valuable to have design methods for building distributed system modules which are known to be regular. Although we mention the strategy of duplication of variables for obtaining a distributed module from a non-distributed design, a strategy which works for the example of section 2.2, but does not for the case of section 4.2, we do not deal with this question in the paper. However, Vissers [8] described several methods to turn a system, designed for negligeable communication delays and therefore in general not regular, into what we call a regular system. He considers a "turn method" which results in a system where each component in turn does an operation, similar to the mutual exclusion algorithm of section 4.3. He also considers "freeze" and "grant" methods which result in systems where two components interact in an inherent asymmetry.

Other interesting questions not covered in this paper are systematic methods for handling message losses, non-sequential message delivery, and failures of system components.

Acknowledgements

I would like to thank Jan Gecsei and Chris Vissers for interesting discussions related to this topic, and Marcel Berthoud, André Schiper, Simon Waddell and Jim Gray for suggesting many improvements of different versions of this paper.

References

[1] P. Robert and J.P. Verjus, "Toward autonomous descriptions of synchronization modules", Proc. IFIP Congress 1977, pp. 981-986.

[2] G.V. Bochmann and J. Gecsei, "A unified method for the specification and verification of protocols", Proc. IFIP Congress 1977, pp. 229-234.

[3] E. André and P. Decitre, "On providing distributed application programmers with control over synchronization", Proc. Computer Network Protocols Symposium, Université de Liège, Febr. 1978, pp. DI-1 to DI-6.

[4] K. Jensen and N. Wirth, "Pascal user manual and report", Springer Verlag, Berlin, 1974.

[5] E.W. Dijkstra, "Self-stabilizing systems in spite of distributed control", Comm. ACM 17, 11 (Nov. 1974), pp. 643-644.

[6] J. Mossière et al., "Sur l'exclusion mutuelle dans les systèmes informatiques", Internal report No 75, IRISA, Université de Rennes, France, Oct. 1977.

[7] L. Lamport, "Concurrent reading and writing", Comm. ACM 20, 11 (Nov. 1977), pp. 806-811.

[8] C.A. Vissers, "Interface: Definition, design and description of the relation of digital system parts", Technische Hogeschool Twente, The Netherlands, 1977 (section 5.2).