Delay-Independent Design for Distributed Systems

GREGOR v. BOCHMANN, SENIOR MEMBER, IEEE

Abstract—**Methods for limiting the impact of communication delays on the logical behavior of distributed systems are considered. It is assumed that a distributed system is described in terms of a number of interconnected modules, and each module is described in terms of its possible states and the possible state transitions. Transitions may be initiated spontaneously by a module and may give rise to output messages, which will be received, after some possible time delay, by another module as an input. Otherwise, transitions may be initiated by received input. If the system has the property called regularity, its behavior is logically independent of the communication delays. A simple condition for regularity is given. This condition is the basis for the implementation of counter-based synchronization conditions in a distributed environment. Weaker forms of regularity, which make abstraction of internal operations invisible from the point of view of an outside observer, are also considered. The application of these concepts to the design of module interfaces involving "collisions" and to communication protocols including timeouts is discussed in some detail with examples.**

I. Introduction

DISTRIBUTED systems can be considered a particular class of parallel systems where several distinct physical system components operate in parallel and largely independently of one another. Certain approaches to the description of parallelism which have been developed for single and multiprocessor computer operating systems are not very suitable for distributed computer systems because they are based on the notion of shared memory, which is not readily available in a distributed system. Instead, approaches using the concept of message exchange seem to be more appropriate. Although many distributed computer systems have been built in the past, the design of such systems is still more an art than a science, and few methods which help in mastering the design complexities of distributed systems are known. This paper tries to give some answers to these problems.

A basic mode of synchronization in the case of shared resources is the enforcement of mutual exclusion between the use of the resource by different processes. Unfortunately, the realization of this simple concept in a distributed and unreliable environment may become quite complicated [17], [23]. More sophisticated sharing involves interleaved operations by different processes using the same resource. In particular, the problems of simultaneous access by several users to shared distributed databases have been studied extensively [2] (interleaved access is essential for obtaining efficiency in a distributed context). The interactions usually have to satisfy "serializability," which means that the result of any interleaved execution of any set of operations must be equivalent to some serial sequence of successive executions of these operations. A general condition sufficient for serializability is described by Eswaran *et al.* [10]. Applied to systems written in terms of processes and monitors, this approach gives rise to conditions which, when satisfied by a program consisting of a certain number of processes and monitors, prove that all possible interleaved executions by the processes in the program are serializable, i.e., equivalent to some execution sequence involving the processes, one after the other [1]. If it is known that a program satisfies these conditions, it is sufficient to consider only sequential (noninterleaved) execution sequences for the verification of the program, which represents a great simplification. Similarly, regular systems [4] (see also Section III) are relatively simple to verify because only execution sequences not involving any transmission delays need to be considered.

Cooperation between several distributed and parallel executing system modules is naturally described by the exchange of messages. Over distance, this involves transmission delays between the sending and receiving of messages. While these delays may be determined with respect to the real time supposed to be universally known, another approach makes abstraction from the real-time properties of the system. This approach concentrates on the ordering between the events in the different system modules, imposed by the fact that the sending of a message always precedes its reception [16]. Sometimes this ordering results in a distributed "logical time," which may be recorded by sequence [20] or by time stamps.

In Section II of this paper, a descriptive model for distributed systems is presented. It is based on the concepts of modules which are interconnected by channels over which the modules may exchange messages for communication between one another. The behavior of each module is described in terms of its possible states and state transitions. The remaining part of the paper concentrates on the questions related to the influence of the communication delays on the system behavior. Although these delays clearly have an influence on the performance properties of the system, it is pointed out that under certain circumstances these delays have no influence on the logical system properties, that is, on the possible execution sequences that can be realized by the system.

The concept of regularity [4] is explained in Section III, and some applications in different contexts are discussed

Manuscript received April 29, 1983; revised July 31, 1985.

The author is with the Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, Montréal, P.Q., H3C 3J7 Canada.

IEEE Log Number 8822026.

in the following sections. Section IV considers the specification of module synchronization based on counters. A straightforward implementation of these synchronization primitives in a distributed environment is given, and conditions for its applicability are discussed. Section V deals with the definition of an externally visible behavior of a distributed system and various weaker system properties, including "serializability." These concepts are demonstrated by two examples: an interface between two modules with "collisions" and a retransmission protocol. The latter example also indicates how such problems as message loss and timeout operations can be logically integrated into the system design.

II. The Descriptive Model

The language in which a system is described has a strong influence on the understandability of the description and the ease by which it can be used for design validation and implementation. In this paper, we consider a model of a system consisting of a number of modules, where each module is described as a state transition machine, similar to that in [14] and extended to allow for interactions between modules by the exchange of messages (see, for instance, [15]). The modules are interconnected by a number of channels in a static structure.

A. Specification of a Single Module

The externally visible behavior of a module is defined by its input and output interactions over the channels by which the module is connected with the other modules within the system and the order in which these interactions may take place. For each channel, a number of input and output interaction types are distinguished, and each type of interaction may be further characterized by parameters, the values of which must be determined by the module that initiates the interaction as an output. The order in which a given module may execute input and output interactions is specified in terms of a state transition model, as described below.

The state of the module is characterized by the values of a set of module variables. The behavior of the system is characterized by a set of operations (sometimes called "transition types" or simply "transitions"). Each operation defines a set of possible state transitions. An operation is defined by its *enabling predicate*, which is a Boolean function of the variables, and its *action*, which updates the variables and may generate output interactions. Only when the enabling predicate is true may the operation be fired; i.e., the associated action is executed, thus performing a state transition.

Two kinds of operations are distinguished: 1) spontaneous operations and 2) operations on input. Spontaneous operations may be fired when the enabling predicate is true (which depends only on the present state of the module in question). A spontaneous operation may or may not generate output interactions over the channels connected to the module. It is assumed that the output of an operation is generated after the state variables of the module have been updated by the operation.

An operation on input is associated with a particular type of input interaction and the channel over which that input may occur. The operation may be fired when an input of that type is ready at the channel in question and the enabling predicate is true. For simplicity, we assume that an operation on input does not produce output. In addition, we assume that whenever an input is ready there is at least one enabled operation on input associated with that type of input. In a given module state, several different operations may be simultaneously enabled, i.e., ready for execution, but only one of them will be selected for execution. Mutual exclusion is assumed between the firing of operations.

This descriptive model is very powerful. It may be used to express the basic control structures for sequential, conditional, and repetitive execution, including nondeterministic guarded commands [8]. As shown in the example of Fig. 1(a), some of the variables may be used to record the progress of execution. These variables are sometimes called "place" or "major state" variables because their function may be graphically represented by "places" or "states" in a state diagram, as shown in Fig. 1(b). An operation is written in the form "*provided* <enabling predicate> *begin* <action> *end*" or "*when* <input> *provided* <enabling predicate> *begin* <action> *end*", respectively, similarly to the notation of [9]. The variable declarations, as well as the enabling predicates and actions of operations, are expressed by elements of the Pascal programming languages.

B. A System of Interconnected Modules

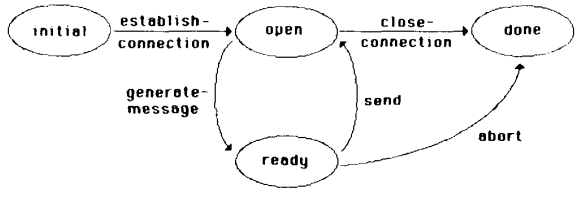
As mentioned above, a system consists of a number of modules defined as state transition machines, as described above, which are connected with one another through channels. A channel has the property that an output interaction generated by the module at one end will be presented as input to the module at the other end of the channel. Different channel properties may be assumed, such as reliable FIFO delivery, FIFO with possible losses, or channels that do not necessarily preserve the order of the interactions. Unless otherwise mentioned, we assume in the following the reliable FIFO case.

Sometimes a situation is considered where the "system" interacts through "external" channels with its "environment." Such a situation can be modeled by having the "system" connected by the "external" channels with dummy "environment modules," which absorb the outputs from the "system" and may generate arbitrary input interactions for the "system" through spontaneous operations, which may be executed in arbitrary order.

For the analysis of the behavior of a system as specified above, we use in the following the notion of a *trace*. A trace is the history of some possible execution of the system in terms of the sequences of operations that have been executed by the different modules of the system. For sim-

```
producer module
  variables
      buffer:message;
      state:(initial,open,ready,done);
      initially
        state=initial;
  operations
        provided state=initial
          begin output establish_connection; state := open end;
        provided state=open
          begin buffer := ... {produce new message} ; state := ready end;
        provided state = open
          begin output close_connection ; state := done end;
        provided state=ready
          begin output send_data(buffer); state := open end;
        provided state = ready
          begin {abort} state := done end;
```

(a)



(b)

Fig. 1. (a) Example of a module specification. (b) Corresponding finite state digram.

plicity, we assume that some fictitious observer may put the operations of all modules into a global order. (Local and global observers are considered in [6].) In practice, however, only those partial orders are relevant which determine either the order of execution of the operations of a single module (as determined by that module) or the causal/temporal relation between the generation of some output by one module and the consumption of that interaction during an operation on input by another module [16].

III. Regularity

The basic question addressed in this paper is the possible influence of message-passing delay on the behavior of a distributed system. More precisely, we are interested in whether the logical behavior of a system defined in the formalism defined above depends on the delays between the generation of output by one module and the consumption of that interaction by another module during the execution of an operation on input. For this purpose, we consider two modes of execution for a given system.

1) We consider "normal" execution, where arbitrary delays may occur between the generation of an interaction and its consumption by the inputting module; between the operation generating the interaction and the operation (by another module) consuming the interaction, other operations may occur, in an arbitrary order as far as possible according to the state of the system and the specifications of the different modules.

2) We consider *delayless* execution, where no other operation may occur (within the entire system) between a spontaneous operation generating one or more output interactions and the operations on input consuming the generated interactions. This mode of execution is considered in [21] and is similar to the rendezvous interactions defined in [12] and [22]. It is also related to serial schedules, as considered in the analysis of protocols for distributed database updates [2], [10].

A. Definitions

We use the following notation. If S is a state of the system, characterized by the state of each module and the interactions generated and not yet consumed in the different channels, and T is a trace, i.e., a sequence of operations, then we say $(S)T$ is *defined* when T is a possible execution sequence for the system starting in the state S. We say that "T' is the *delayless* trace corresponding to T" if T' is a delayless trace and contains the same spontaneous operations as T and they occur in T' in the same relative order, as seen by each module, as in T. We say that a (general) trace is *complete* if all the input interactions consumed by operations on input within T are exactly those interactions generated as output (previously) by the spontaneous operations of T. A complete trace should therefore start and end in a system state with no interactions in the channels.

Definition: We say that two system states S_1 and S_2 are *equivalent* if they allow for the same execution traces, i.e., if $(S_1)T$ is defined iff $(S_2)T$ is defined, for any T.

We assume in the following that the action of each operation (taken individually) is deterministic. Then "$(S)T$" represents a new system state which is attained by the system from the state S after the execution of the operation sequence T.

Definition: We say that a system with initial state S_0 is *regular* if, for any *complete* trace T such that $(S_0)T$ is defined, the following conditions hold:

1) $(S_0)T'$ is also defined and

2) $(S_0)T'$ is a state *equivalent* to $(S_0)T$

where T' is the *delayless* trace corresponding to T.

The condition 1) implies that each sequence of spontaneous operations which is possible in the presence of arbitrary, but finite, message-passing delays is also possible in the absence of delays. Therefore, delays cannot introduce any "new" system behavior. On the other hand, condition 2) implies that all sequences of spontaneous operations that are possible in the absence of delays are also possible in the presence of delays. Therefore, the deadlock and liveness properties of the system are independent of the delays.

We can conclude that the verification and analysis of a distributed system are simplified if it is known that the system is regular. For studying its possible behaviors, including deadlock and liveness properties, one may ignore

C. Regularity

We say that a counter relation, used in the authorization condition for a procedure of a given module, has *regular coefficients* if for each involved (copy) counter variable x_i of a procedure associated with a different module, the corresponding constant c_i is positive. Using the condition for regularity discussed in Section III-B, it is then easy to show that the following proposition holds.

Proposition: If all counter relations of the authorization conditions of all procedures of a subsystem have *regular coefficients*, then the (synchronization aspect of the) subsystem is *regular*.

We note that this result [5] was used by Herman [11], who considers a ring communication structure between the different modules and describes a method for reinitialization after the failure of a module. Very similar constraints on authorization conditions are also considered by Schmid [26] for analyzing the mutual influence of different conditional critical regions and their efficient (nondistributed) implementation.

The above proposition may be used to determine in which way the procedures of a subsystem may be distributed over several physical components, without changing the authorization conditions of the procedures, such that the logical behavior of the subsystem is not affected. For example, the authorization conditions for a system as defined in Fig. 2 may be written as follows:

- condition for the procedure **produce**:

$$\mathbf{terminated}_{\mathbf{consume}} - \mathbf{authorized}_{\mathbf{produce}} > -N$$

- condition for the procedure **remove**:

$$\mathbf{terminated}_{\mathbf{produce}} - \mathbf{authorized}_{\mathbf{consume}} > O$$

If the two procedures are distributed over two different modules as explained above, the system remains regular since the first counter variable of each condition will be a *copy* variable, whereas the second will be an *original* one.

V. Weaker Forms of Regularity

As discussed in Section III, regularity implies strong constraints on the system behavior. As shown by the examples discussed below, many useful systems do not satisfy the regularity contraints. However, it is possible to consider weaker constraints, in the following called *external regularity*, which still imply a certain delay independence of the system behavior, at least as far as the "externally" visible system behavior is concerned. An even weaker form of system property is serializability, as defined for the analysis of distributed database query management [2].

In order to make the notion of "external visibility" more precise, we use the concept of projections as defined in [21]. We suppose that a certain subset P of the operations defined within the system are "externally visible," i.e., only these operations are considered to be relevant as far as the behavior of the system is concerned, as seen by its environment. In the case of a distributed database, these externally visible operations are the **read** and **write** requests, as well as the returned **read** results. In the case of the communication protocol considered in Section V-C and the distributed queue of Fig. 2, the externally visible operations are **produce** and **consume**. In general, considering a subsystem representing an abstract data type [19], the externally visible operations correspond to the "operations" provided to its users by the abstract data type.

A. Definitions

We give in the following some definitions leading to the notion of "external regularity" which are used in the discussion of the examples in the Sections V-B and V-C. We assume in the following that a system is specified using the model described in Section II and that a certain subset P of the operations of that system are considered to be relevant for the externally visible behavior of that system.

Notation: We write $P(T)$ for the projection of a trace T on the subset P of relevant operations; that is, $P(T)$ is the subsequence of T of those operations of T that are included in P.

Definition: We say that two *traces* T_1 and T_2 are *equivalent with respect to* P if their projections on P are identical; i.e., $P(T_1) = P(T_2)$.

Definition: We say that two system *states* S_1 and S_2 are *equivalent with respect* to P if, for any trace T_1 possible from state S_1 (that is, $(S_1)T_1$ is defined), there is a trace T_2 possible from state S_2 (that is, $(S_2)T_2$ is defined) such that the two traces are *equivalent with respect* to P.

Definition: We say that a system is *externally regular*, or more precisely, *regular with respect to* P, if, for each *complete* trace T_1 which is possible in the initial state S of the system (that is, $(S)T_1$ is *defined*), there is a *delayless* trace T_2 such that

1) T_2 is *equivalent* to T_1 *with respect* to P,
2) T_2 is possible in the initial state S of the system (that is, $(S)T_2$ is *defined*), and
3) the final states $(S)T_1$ and $(S)T_2$ are *equivalent with respect* to P.

Definition: We say that a system is serializable with respect to P if, for each T_1 as above, there is a *delayless* trace T_2 such that

1) the projections of T_1 and T_2 on P contain the same operations of P, but not necessarily in the same order: only the relative order between operations in the same module must be preserved;
2) as above; and
3) as above.

The regularity condition of Section III-B can be generalized to the case of external regularity as follows.

Sufficient Condition for External Regularity: If for any trace T such that $(S)T$ is *defined*, the conditions

1) $(S)T'$ is also *defined* and

teracting modules must be designed to operate in such an environment. (An example is given in Section V.)

2) Implementing Channel Flow Control: A spontaneous operation generating output may be delayed due to the slow processing speed of the receiving module.

Channel flow control may be introduced into a given system by requiring that each spontaneous operation be subject to an additional (flow control) enabling predicate for each output generated. This predicate is true when the corresponding channel is ready to receive the output interaction. In the extreme case of channels with zero-length queues, this predicate is true when the receiving module is not in the process of executing any operation (and therefore ready to execute an operation taking the generated output as an input).

An important property of a regular system is the following. The logical behavior of such a system is not influenced by the introduction of channel flow control mechanisms. This may be shown as follows.

The introduction of a flow control mechanism has no impact on the possible *delayless* traces since for the execution of these traces no message need be stored in the channels. The only impact of the flow control mechanism on the system is that certain traces, which involve "too many" messages in transit, become impossible to realize. However, such traces lead to system states which are *equivalent* to states attained by the corresponding *delayless* trace. Therefore, that logical system behavior, as discussed in Section III-A, is not affected by the flow control mechanism.

IV. Distributed Synchronization Based on Counters

A. Counter-Based Synchronization

We assume in this section that the enabling predicates, which determine when certain operations may be executed, depend only on variables which, essentially, count the number of times an operation is executed. We adopt in the following the approach of Robert and Verjus [25], but our conclusions may also be applied to other similar approaches [24], [28].

Following the approach of [25], we suppose that the synchronization aspect of a subsystem is described separately from the processing aspect. The processing aspect consists of a set of procedures which may be called by other parts of the system. However, the execution of a called procedure may be delayed until a certain authorization condition is satisfied. This is the synchronization aspect, which is handled by what we call the "control part" of the subsystem. In the nondistributed case [25], the control part contains the following three counter variables associated with each procedure:

- *requested:* the number of requests for an execution of the procedure since the subsystem initialization;
- *authorized:* the number of executions authorized; and
- *terminated:* the number of recorded terminations and procedure executions.

A requested execution of a procedure may be authorized by the control part when the authorization condition of the procedure is satisfied. This condition depends on the counter variables. We assume that the condition is a Boolean expression built out of counter relations which are tied by the logical operators AND and OR. Each counter relation has the form

$$c_1x_1 + c_2x_2 + \cdots + c_nx_n > c_{n+1}$$

where the $c_i (i = , \cdots , n + 1)$ are constants and the $x_i (i = 1, \cdots , n)$ are particular counter variables. More details and many examples may be found in the references mentioned above.

B. Distributed Implementation

A distributed implementation of the synchronization rules described above may be obtained in the model described in Section II. We suppose that the subsystem consists of several modules, and each procedure of the subsystem is associated with one of the modules. In this module, the requests are generated (possibly based on the information in messages received from other submodules), the decisions for procedure executions are made, and the execution (i.e., the processing part) of the procedure is performed. The module also contains the *original* counter variables, **requested**, **authorized**, and **terminated** of the procedure.

With each procedure we associate three spontaneous operations (in the sense of Section II) located at the same module. They represent a procedure request, an authorization for execution, and a termination of an execution, respectively. The authorization operation has an enabling predicate, which is the authorization condition of the procedure and which is evaluated based on local counter variables. The variables involved may include *original* counter variables of procedures (including the procedure requested) associated with the same module and local *copy* counter variables (see below) of procedures associated with other modules. The actions of the operations consist of an update of the corresponding *original* counter variable (increased by one). After the authorization action, the execution of the procedure is performed, followed by the execution of the termination operation.

In addition to the *original* counter variables of local procedures, each module also maintains so-called *copy* variables, which are counter variables for procedures associated with other modules. The copy variables are updated by operations on input which are activated by messages that are generated as output by the corresponding spontaneous operations in the module with which the procedure is associated. Therefore, the value of a *copy* counter variable is always smaller than or equal to the value of the *original*.

C. Regularity

We say that a counter relation, used in the authorization condition for a procedure of a given module, has *regular coefficients* if for each involved (copy) counter variable x_i of a procedure associated with a different module, the corresponding constant c_i is positive. Using the condition for regularity discussed in Section III-B, it is then easy to show that the following proposition holds.

Proposition: If all counter relations of the authorization conditions of all procedures of a subsystem have *regular coefficients*, then the (synchronization aspect of the) subsystem is *regular*.

We note that this result [5] was used by Herman [11], who considers a ring communication structure between the different modules and describes a method for reinitialization after the failure of a module. Very similar constraints on authorization conditions are also considered by Schmid [26] for analyzing the mutual influence of different conditional critical regions and their efficient (nondistributed) implementation.

The above proposition may be used to determine in which way the procedures of a subsystem may be distributed over several physical components, without changing the authorization conditions of the procedures, such that the logical behavior of the subsystem is not affected. For example, the authorization conditions for a system as defined in Fig. 2 may be written as follows:

- condition for the procedure **produce**:

$$\mathbf{terminated}_{\mathbf{consume}} - \mathbf{authorized}_{\mathbf{produce}} > -N$$

- condition for the procedure **remove**:

$$\mathbf{terminated}_{\mathbf{produce}} - \mathbf{authorized}_{\mathbf{consume}} > O$$

If the two procedures are distributed over two different modules as explained above, the system remains regular since the first counter variable of each condition will be a *copy* variable, whereas the second will be an *original* one.

V. Weaker Forms of Regularity

As discussed in Section III, regularity implies strong constraints on the system behavior. As shown by the examples discussed below, many useful systems do not satisfy the regularity contraints. However, it is possible to consider weaker constraints, in the following called *external regularity*, which still imply a certain delay independence of the system behavior, at least as far as the "externally" visible system behavior is concerned. An even weaker form of system property is serializability, as defined for the analysis of distributed database query management [2].

In order to make the notion of "external visibility" more precise, we use the concept of projections as defined in [21]. We suppose that a certain subset P of the operations defined within the system are "externally visible," i.e., only these operations are considered to be relevant as far as the behavior of the system is concerned, as seen by its environment. In the case of a distributed database, these externally visible operations are the **read** and **write** requests, as well as the returned **read** results. In the case of the communication protocol considered in Section V-C and the distributed queue of Fig. 2, the externally visible operations are **produce** and **consume**. In general, considering a subsystem representing an abstract data type [19], the externally visible operations correspond to the "operations" provided to its users by the abstract data type.

A. Definitions

We give in the following some definitions leading to the notion of "external regularity" which are used in the discussion of the examples in the Sections V-B and V-C. We assume in the following that a system is specified using the model described in Section II and that a certain subset P of the operations of that system are considered to be relevant for the externally visible behavior of that system.

Notation: We write $P(T)$ for the projection of a trace T on the subset P of relevant operations; that is, $P(T)$ is the subsequence of T of those operations of T that are included in P.

Definition: We say that two *traces* T_1 and T_2 are *equivalent with respect to* P if their projections on P are identical; i.e., $P(T_1) = P(T_2)$.

Definition: We say that two system *states* S_1 and S_2 are *equivalent with respect* to P if, for any trace T_1 possible from state S_1 (that is, $(S_1)T_1$ is defined), there is a trace T_2 possible from state S_2 (that is, $(S_2)T_2$ is defined) such that the two traces are *equivalent with respect* to P.

Definition: We say that a system is *externally regular*, or more precisely, *regular with respect to P*, if, for each *complete* trace T_1 which is possible in the initial state S of the system (that is, $(S)T_1$ is *defined*), there is a *delayless* trace T_2 such that

1) T_2 is *equivalent* to T_1 *with respect* to P,
2) T_2 is possible in the initial state S of the system (that is, $(S)T_2$ is *defined*), and
3) the final states $(S)T_1$ and $(S)T_2$ are *equivalent with respect* to P.

Definition: We say that a system is serializable with respect to P if, for each T_1 as above, there is a *delayless* trace T_2 such that

1) the projections of T_1 and T_2 on P contain the same operations of P, but not necessarily in the same order: only the relative order between operations in the same module must be preserved;
2) as above; and
3) as above.

The regularity condition of Section III-B can be generalized to the case of external regularity as follows.

Sufficient Condition for External Regularity: If for any trace T such that $(S)T$ is *defined*, the conditions

1) $(S)T'$ is also *defined* and

2) $(S)T$ and $(S)T'$ are states which are *equivalent with respect* to P,

are satisfied then the system is *regular with respect* to P.

B. Example of an Interface with Queueing Delays

Queueing delays in the interfaces between different modules within a given system often lead to certain difficulties. An example of this is the so-called "call collisions" that occur over an X.25 interface between a host computer and a packet-switched data network node when the computer and the network node both initiate the establishment of a virtual circuit at the same time (using the same logical channel number). If there were no delays, such collisions would not occur since each side would immediately know when the other side initiated a circuit establishment. A similar situation is demonstrated by the example below, and a more complex example is discussed in [7].

Fig. 3 shows a system consisting of two modules A and B. The possible operations of the two modules are indicated in the figure by the state transition diagrams in both modules. The behavior of the system may be characterized as follows. In the **idle** state, each module may make a request for entering a joint activity X or Y, respectively. A joint activity requires both modules to be in the same state "X" or "Y," respectively. When the joint activity is terminated, both modules go back to their **idle** states. Whether activity X or Y will be chosen depends on which request is made first.

The behavior may be characterized by the following CCS expression [22]:

$$\begin{aligned} X \textbf{ or } Y = \; & (\ (\underline{\text{enter} - X} \mid\mid \text{enter} - X)\ ;\ (\underline{\text{leave}} \mid\mid \text{leave})\ ) \\ & \mid (\underline{\text{enter} - Y} \mid\mid \text{enter} - Y)\ ;\ (\underline{\text{leave}} \mid\mid \text{leave})\ ) \\ & ;\ X \textbf{ or } Y \end{aligned}$$

where "| |" indicates "parallel" execution, that is, in arbitrary order. (The message nature of communication clearly implies that receiving operation **enter** $-$ X will be executed after the sending operation **enter** $-$ **X,** etc.) The path expression includes only the operations **enter** $-$ **X, enter** $-$ **Y, enter** $-$ **X, enter** $-$ **Y, leave,** and **leave.** These are the operations considered relevant for the "externally visible" behavior of the system; that is, they form the subset P.

In the absence of communication delays, there is no difficulty in determining which request was made first. Only the transitions drawn as continuous arrows in Fig. 3 will be executed. In the presence of delays, however, both modules may make the requesting transitions to their respective states **wait,** and it would not be clear which module made the first request. The system specification of Fig. 3 gives priority to the module A; that is, in the case of "simultaneous" requests, the module B will follow the request made by module A (see transitions drawn as dotted arrows).

An analysis of the system of Fig. 3 shows that the system is regular with respect to the set of operations P defined above. This means that the possible sequences of executions of the externally visible operations P are the same in the case of communication delays and in the case of no delays. It is important to note that one could define other systems, equivalent to the one of Fig. 3 as far as the externally visible operations are concerned, which handle the case of request collision in a different way, for instance, giving module B priority or letting both modules abandon their requests.

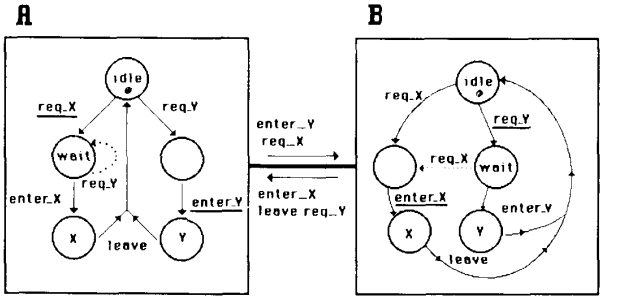


Fig. 3. Example of synchronization over an interface with queueing (Underlined operation names represent output, and the nonunderlined represent input.)

C. A Retransmission Protocol

The system of Fig. 2 may be considered a realization of the abstract data type of a **queue.** The subset of externally visible operations P would consist of **produce** and **consume.** The system of Fig. 2 assumes that the channel between the two modules realizes a reliable FIFO queue. Without this assumption, the same externally visible behavior of the system may be obtained by introducing some retransmission protocol between the two modules of the system. Such a system is shown in Fig. 4. The similarity between the systems in Figs. 2 and 4 is indicated by using as much as possible the same names for variables and transitions in both systems. However, the following important differences are noted.

1) The retransmission system contains operations to recover from the loss of messages transmitted between the two modules of the system. To keep the correct sequence of data blocks, it is necessary to number the data blocks included in the messages. (Note that similar protocols are analyzed in [3] and [27]. In contrast to the systems of Fig. 2 and [3], sequential message delivery through the communication channel is *not* assumed here.)

2) The **start-retransmission** and **send-ack** operations are always enabled. For efficiency reasons, the **start-retransmission** operation should be executed only if, after the execution of a **transmit** operation, the number of **outstanding** messages is not reduced to zero (by the execution of an **acknowledge** operation) after a certain time period ("timeout").

An analysis of this system shows that its operations satisfy the regularity condition of Section III-B, except for the presence of the operation **start-retransmission.** In fact, the operation **acknowledge** does not move *left over*

```
Variables
   length : 0..N ;
   buffer : array [0..N-1] of data;
   next   : 0..N-1 ; {number of next data block to be received from
                      the environment}
   VS     : 0..N-1 ; {number of next data block to be sent}
   outstanding : 0..N-1 ; {number of next data block to be acknowledged}

Operation
   Produce:: provided length < N
             begin
                length := length + 1 ;
                buffer [next] := ... {produce new unit};
                next := (next + 1) mod N
             end ;

   transmit::
      provided (VS - outstanding) mod N < windowsize
            and VS ≠ next

             begin
              VS := (VS + 1) mod N;
              output transfer (VS,buffer[VS])
             end
   Start-retransmission::
      {provided time-out} begin VS := outstanding end;
   Receive_ack : when acknowledge (NR : 0..N-1)
             begin
              length := length - [(NR - outstanding) mod N] ;
              outstanding := NR
             end;
Initially
   length := VS := next := outstanding := 0 ;
```

(a)

```
Variables
   length : 0..N ;
   VR  :  0..N-1 ; {number of next data block to be received}
   buffer : queue of data ;
   unit : data ;

Operations

   Consume ::
      provided length > 0
      begin
         length : = length - 1 ;
         buffer.get (unit);
         ...{consume data unit}
      end;

   Receive:: when transfer (NS : 0..N-1, received-unit : data)
      begin if NS = VR
         then begin
           length := length + 1 ;
           buffer.put (received-unit);
           VR := (VR + 1) mod N
         end;
   Send-ack :: provided true

      begin output acknowledge (VR) end;

Initially

      length := VR := 0;
      buffer.empty;
```

(b)

Fig. 4. Producer and consumer modules for a retransmission protocol. (a) Specification of producer module. (b) Specification of consumer dule.

start-retransmission since the traces ***T* start-retransmission acknowledge** and ***T* acknowledge start-retransmission** do not lead, in general, to *equivalent* module states. (For instance, a new data block may be acknowledged which is taken into account by the **start-retransmission** operation in the case of the latter trace, but not in the case of the former. This results in different values for the variable ***VS***.) However, the module is *regular with respect* to the subset of operation $P = \{$ **produce, consume**$\}$. In fact, the external regularity condition of Section V-A is satisfied by the system. To prove this, it is sufficient to show that the two traces above lead to module states *equivalent with respect* to P. In fact, the state reached by the second trace is reachable from the state reached by the first trace through the execution of a sufficient number of **transmit** and **receive** operations.

This example also demonstrates the nature of "timeout" operations, which are usually initiated some time delay after the execution of an operation for which some form of acknowledgment is expected, but not received. This class of operations often introduces nonregularity, but by the nature of its recovery action, it should keep the system externally regular. The operation is usually introduced into a system in order to prevent the deadlock which could result from a "lost message" or a failure of a subsystem. Problems of logical consistency often arise if the original acknowledgment arrives when the timeout operation has already been started. Such problems may be avoided, as in the example above, if the system design is such that the timeout operation may be enabled at all times. However, for efficiency considerations, it is usually necessary to restrict the frequency of execution for these operations.

VI. Conclusions

It has been demonstrated that for a regular system the communication delays have no influence on the logical behavior of the system. Therefore, the analysis of the behavior of a system is simplified if it is known to be regular. In that case, it is sufficient to analyze the system assuming no communication delays (which usually makes the analysis much simpler). Unfortunately, it is not always easy to determine whether or not a system is regular. However, an easily verified sufficient condition for regularity is described in Section III-B. This condition was applied in Section IV to derive a distributed implementation of counter-based synchronization conditions.

Although many practical systems are not regular, some weaker form of regularity can still be applied. The so-called "external regularity" is based on the externally visible behavior of the considered subsystem and constitutes an abstraction from certain internal operations of the subsystem. These considerations give a framework for the design of module interfaces (including delays) in distributed systems and for the handling of timeout situations, as discussed in Section V. Although some sufficient condition for external regularity is given in Section V-A, it does not always apply. It would be interesting to find more general regularity conditions and to clarify the relation with the weaker property of "serializability," which is often used in the analysis of distributed database systems.

References

[1] E. A. Akkoyunlu, A. J. Bernstein, F. B. Schneider, and A. Silberschatz, "Conditions for the equivalence of synchronous and asynchronous systems," *IEEE Trans Software Eng.*, vol. SE-4, pp. 507-516, Nov. 1978.

[2] P. A. Bernstein *et al.*, "Concurrency control in a system for distributed data bases," *ACM Trans. Database Syst.*, vol. 5, no. 1, Mar. 1980.

[3] G. v. Bochmann, "Logical verification and implementation of protocols," in *Proc. 4th Data Commun. Symp. (ACM-IEEE)*, Quebec City, P.Q., Canada, Oct. 1975, pp. 7-15-7-20; reprinted in *Communication Protocol Modeling*, C. Sunshine, Ed. Artech, 1981.

[4] ——, "Distributed synchronization and regularity," *Comput. Networks*, vol. 3, pp. 36-43, 1979.

[5] ——, "Towards an understanding of distributed and parallel systems," Dep. d'IRO, Univ. Montréal, Montréal, P.Q., Canada, Pub. #317, 1980.

[6] G. v. Bochmann, R. Dssouli, and J. R. Zhao, "Trace analysis for conformance and arbitration testing," *IEEE Trans. Software Eng.*, to be published.

[7] G. v. Bochmann and A. Finkel, "Impact of queued interaction on protocol specification and verification," Dep. Inform. Récherche Opérationnelle, Univ. Montréal, Montréal, P.Q., Canada, Tech. Rep., 1988.

[8] E. W. Dijkstra, "Guarded commands, nondeterminancy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453-457, Aug. 1975.

[9] "Estelle: A formal description technique based on an extended state transition model," ISO Int. Standard 9074, 1987.

[10] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624-633, Nov. 1976.

[11] D. Herman, "Controle reparti des synchronisations entre processus," IRISA, Rennes, France, 1981, to be published.

[12] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666-677, Aug. 1978.

[13] R. M. Keller, "A fundamental theorem of asynchronous parallel computation," in *Parallel Processing*, T. Y. Feng, Ed. New York: Springer-Verlag, 1974, pp. 102-112.

[14] ——, "Formal verification of parallel programs," *Commun. ACM*, vol. 19, no. 7, pp. 371-384, July 1976.

[15] S. S. Lam and A. U. Shankar, "Protocol verification via projections," *IEEE Trans. Software Eng.*, vol. SE-10, July 1984.

[16] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

[17] G. LeLann, "Distributed systems—Towards a formal approach," in *Proc. IFIP Congr. 1977.* Amsterdam, The Netherlands: North-Holland, 1977, pp. 155-160.

[18] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, pp. 717-721, Dec. 1975.

[19] B. Liskov and S. Zilles, "Specification techniques for data abstractions," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 7-18, Mar. 1975.

[20] P. M. Merlin and A. Segall, "A failsafe distributed routing protocol," *IEEE Trans. Commun.*, vol. COM-27, pp. 1280-1287, Sept. 1979.

[21] P. H. Merlin and G. v. Bochmann, "On the construction of submodule and communication protocols," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 1, pp. 1-25, Jan. 1983.

[22] R. Milner, "A calculus of communicating systems," in *Lecture Notes in Computer Science, No. 92.* New York: Springer-Verlag, 1980.

[23] M. Raynal, "Algorithmique du parallèlisme: Le problème de l'exclusion mutuelle," *DUNOD* (in French), 1984, 164 p.

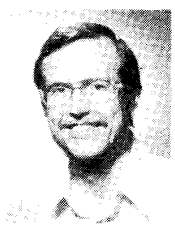
[24] D. P. Reed and R. K. Kanodia, "Synchronization with event counts and sequencers," in *Proc. Sixth ACM Symp. Operating Syst. Principles*, Nov. 1977.

[25] P. Robert and J. P. Verjus, "Toward autonomous descriptions of synchronization modules," in *Proc. IFIP Congr. 1977.* Amsterdam, The Netherlands: North-Holland, 1977, pp. 981-986.

[26] H. A. Schmid, "On the efficient implementation of conditional critical regions and the construction of monitors," *Acta Inform.*, vol. 6, pp. 227-249, 1976.

[27] N. V. Stenning, "A data transfer protocol," *Comput. Networks*, vol. 1, pp. 99-110, 1976.

[28] A. J. Gerber, "Process synchronization by counter variables," *ACM Opt. Syst. Rev.*, vol. 11, no. 4, pp. 6-17, Oct. 1977.



Gregor v. Bochmann (M'82-SM'85) received the Diploma degree in physics from the University of Munich, Munich, West 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, compiler design, communication protocols, and software engineering and has published many papers in these areas. He is currently a Professor in the Département d'Informatique et de Recherche Opérationelle, Université de Montréal, Montréal. His present work is aimed at design models for communication protocols and distributed systems. He has been actively involved in the standardization of formal description techniques for OSI. From 1977 to 1978 he was a Visiting Professor at the Ecole Polytechnique Federale, Lausanne, Switzerland. From 1979 to 1980 he was a Visiting Professor in the Computer Systems Laboratory, Stanford University, Stanford, CA. From 1986 to 1987 he was a Visiting Researcher at Siemens, Munich.