What It Means for a Concurrent Program to Satisfy a Specification:

Why No One Has Specified Priority

Leslie Lamport
Computer Science Laboratory
SRI International*

2 July 1984
minor revision 27 October 1984

Abstract

The formal correspondence between an implementation and its specification is examined. It is shown that existing specifications that claim to describe priority are either vacuous or else too restrictive to be implemented in some reasonable situations. This is illustrated with a precisely formulated problem of specifying a first-come-first-served mutual exclusion algorithm, which it is claimed cannot be solved by existing methods.

pri·or′i·ty (prī·ŏr′*ĭ*·tĭ), *n.; pl.* -TIES(-tĭz). **3.** Order of preference based on urgency, importance, or merit. [1]

1 Introduction

Specification and Implementation

A formal specification method should reduce the question of whether a program satisfies its specification to a precisely formulated mathematical prob-

*This work was supported in part by the National Science Foundation under grant number MCS-8104459, and the Army Research Office under grant number DAAG29-83-K-0119.

lem. This reduction is what distinguishes a formal method from an informal one. Most researchers developing specification formalisms have concentrated upon the formal semantics of the specification language, apparently believing that such a semantics, together with a formal semantics for the programming language, provides the necessary reduction. However, formal semantics for the specification and programming languages are not enough; one must also define the correspondence between the two semantics.

As a trivial example, consider a program to compute the square of an integer. The specification might be given in terms of mathematical integers, while the program's semantics might be defined in terms of bit strings. To determine if the program meets its specification, we must define the correspondence between the implementation-level semantic concept of *bit string* and the specification-level concept of *integer*. Although this is easy to do, it is important that it be done; a program that expects input values to be in two's-complement representation may produce an incorrect answer when given an input value encoded in sign-magnitude representation.

For sequential programs, specified in terms of input and output values, the correspondence between implementation and specification concepts is, in principle, simple: it is just a mapping between two domains of values. However, this is not the case for concurrent programs, where the specification involves the program's behavior. The granularity of operations can be very different at the two levels; an atomic operation at the specification level may correspond to a large number of atomic program operations. The formal correspondence between the two semantic levels requires careful examination. In this paper, I consider the implications of this correspondence for the particular problem of specifying priority.

Priority

In concurrent systems, priority denotes the order of preference in which processes obtain service. It may be based upon the nature of the service being requested, the importance of the requesting process, or the order in which requests are issued. All popular methods for specifying concurrent systems allow one to write simple specifications that appear to describe priority. However, I will show that, depending upon how they are interpreted, these specifications are either too restrictive to be implementable in all situations or else they are vacuous, being satisfied by any program.

I will concentrate upon a particular example of priority: first-come-first-served (FCFS). There is nothing special about FCFS—the same problem arises in specifying other types of priority; FCFS just provides a simple, well-studied case. It is also an important case; the Ada language requires an FCFS queuing discipline in the implementation of the rendezvous mechanism, and problems in formally specifying this requirement may be of some interest to the Ada community.

My claim that current methods cannot specify priority is a controversial one, and provokes arguments when presented to computer scientists. I have therefore formulated a challenge to those who feel that they know how to specify priority: the specification of a precisely-defined FCFS mutual exclusion algorithm. I believe that anyone claiming to have a general method for specifying concurrent programs should be able to write the required specification. By a "general" method, I mean one that permits implementations in a reasonably broad class of programming languages. Given some particular programming language having an FCFS synchronization primitive, it is easy to specify FCFS priority for programs written in that language by requiring the implementation to use the FCFS primitive. To prevent this kind of "cheating", the challenge specifies two simple programming languages that must be handled.

The challenge is presented first, before any explanation of what makes specifying priority difficult. I urge the reader to study it and decide if it is reasonable before reading the rest of the paper. There are no tricks in the challenge; the problem that arises in trying to specify priority is a fundamental one. For a cry of "foul" to be taken seriously, it should be issued on the basis of the challenge alone, not on the ensuing discussion.

2 The Challenge

Let Blaise be a simple concurrent programming language with an atomic assignment statement, concatenation (;), **if** and **while** statements with atomic tests, a **cobegin**, and integer and boolean shared variables, but with no explicit synchronization or communication commands. The **cobegin** is assumed to be fair, meaning that a nonterminated process will eventually execute its next atomic operation, but no bound is assumed on the relative execution speeds of the different processes. All classical shared-variable multiprocess algorithms can easily be written as Blaise programs.

Let the language Tony be the same as Blaise except with no shared variables, instead using CSP-style communication primitives. Moreover, assume an appropriate fairness requirement on communication so that a

Blaise program can be simulated in the obvious way by a Tony program in which shared variables are replaced by extra processes.[1] Reading the value of a Blaise shared variable is simulated by a "?" operation in the Tony program, and writing its value is simulated by a "!".

Mutual exclusion algorithms that can be written in Blaise have been studied for years [3], and there is a general agreement that certain algorithms are FCFS and others are not. It is less clear what it means for a Tony program to be FCFS, but it is easy to write Tony programs that are obviously not FCFS—for example, an algorithm with a central scheduling process that does not always grant requests in the order it receives them.

The challenge is to specify program statements $entry_p$ and $exit_p$, for $p = 1, \ldots, 17$, such that if the statement

$$entry_p\text{''};\ \ \text{critical section};\ \ exit_p \tag{1}$$

is embedded in the sequential process π_p, then

$$\textbf{cobegin}\ \ \pi_1 \parallel \ldots \parallel \pi_{17}\ \ \textbf{coend}$$

is an FCFS mutual exclusion algorithm in which π_p requests entry to its critical section by initiating the execution of (1). (There may also be declarations of the shared variables in a Blaise program and extra processes in a Tony program.) The specification, and the specification method, must have the following properties.

1. For any Blaise or Tony implementations of the $entry_p$ and $exit_p$ statements, the method defines a mathematical formula $\mathcal{C}$ and a formal system $\mathbf{L}$ such that the implementation satisfies the specification if and only if $\mathcal{C}$ is a valid formula of $\mathbf{L}$.

   [*This is a precise statement of the requirement that a formal method reduce the question of whether a particular program satisfies the specification to a well-defined mathematical problem.*]

2. (a) Any Blaise implementation that is generally regarded to be an FCFS mutual exclusion algorithm must satisfy the specification.

   (b) A Tony simulation of such a Blaise program must also satisfy the specification.

[1] Without some fairness constraint on communication, a Tony program cannot guarantee the fairness condition for a Blaise process that accesses a shared variable.

3. Any Blaise or Tony program that is generally regarded not to be an FCFS mutual exclusion algorithm must not satisfy the specification.

I will attempt to answer all serious responses to this challenge. To meet the challenge, you must provide the specification and indicate how one constructs the $\mathcal{C}$ and $\mathbf{L}$ of condition 1 for any Blaise or Tony program. I will then attempt to present one or more programs that violate condition 2 or 3, in which case you must show that these conditions are not violated. The construction of $\mathcal{C}$ and $\mathbf{L}$ and the refutation of my counterexamples need not be given in full mathematical detail, but they must be rigorous enough to convince a competent computer scientist that a completely formal exposition is, in principle, possible.

3 What's So Hard About The Challenge?

Why Current Methods Don't Work

Let us now consider how one might specify the FCFS condition of the challenge. Intuitively, FCFS means that requests to enter the critical section are serviced in the order in which they are issued. To specify this more precisely, one must recognize two kinds of operations—a *request* operation and a *critical_section* operation. To each *critical_section* operation there corresponds a *request* operation, issued before it by the same process. We identify operations by subscripts, letting $request_i$ and $critical\_section_i$ denote corresponding operations. The FCFS priority condition is usually expressed as follows:

(∗) For any distinct operations $critical\_section_i$ and $critical\_section_j$, if $request_i$ precedes $request_j$ then $critical\_section_i$ must precede $critical\_section_j$.

The operations $request_i$ and $request_j$ need not be atomic actions. The condition "$request_i$ precedes $request_j$" means that $request_i$ finishes before $request_j$ begins. If these operations are nonatomic, then they can be concurrent, meaning that neither precedes the other. In this case, condition (∗) does not specify the order of the operations $critical\_section_i$ and $critical\_section_j$. Allowing the *request* operations to be nonatomic means that the order of service does not matter (is not specified) if the requests are issued "too close together".

All the formal specification methods I know of—including [4], [5], [8], [11], [13], [14], [15], [16], [17], and [18]—specify FCFS with condition (∗), although the formal expression of this condition differs with the different methods. These differences are irrelevant to the fundamental problem with condition (∗).

To verify that a Blaise program satisfies (∗), one must state what Blaise operations correspond to the operations $request_i$ and $critical\_section_i$. The $critical\_section_i$ operation clearly corresponds to an execution of the critical section, but what about the $request_i$ operation? Let $enter_i$ denote the operation by which a process initiates execution of its $entry_p$ statement, so $enter_i$ is an execution of the atomic action that puts the process's control at the beginning of that statement—in other words, the last atomic action before the process executes $entry_p$. What is the relation between the atomic action $enter_i$ and the operation $request_i$? There are two possibilities:

1. $enter_i$ equals $request_i$, making $request_i$ an atomic action.

2. $enter_i$ is the first action of the nonatomic operation $request_i$.

I will examine each of them in turn.

In the first case, where $request_i$ is the atomic action putting the process at its $entry_p$ statement, condition (∗) cannot be satisfied by any implementation. There is no way for two *entry* statements to determine in which order they were entered. Hence, no algorithm can ensure that the *critical_section* operations occur in the order required by condition (∗).

It might seem unfair not to make $enter_i$ part of the the $entry_p$ statement, and one might define $enter_i$ to be the first atomic action of $entry_p$. However, this does not solve the problem because an atomic action of a Blaise program can either read or write a shared variable, but cannot do both. Thus, if $request_i$ is the atomic action $enter_i$, then the $request_i$ operation cannot both announce the process's desire to enter its critical section and check for the presence of other processes waiting to enter their critical sections.[2] If the two operations $request_i$ and $request_j$ occur too close together, no algorithm can determine which one happened first, even though the semantics of Blaise specifies that they occur in some definite order. Hence, a Blaise implementation still cannot satisfy condition (∗).

Now consider the second case. If $enter_i$ is only the first action of the $request_i$ operation, when does the operation end? This question is not an-

[2]This appears to be a folk theorem, having been known to a number of people but never published.

swered by the specification. Since the end of the $request_i$ operation happens while executing the $entry_p$ statement, which is provided by implementor, he must be the one who decides where the $request_i$ operation ends. In order to prove that his implementation meets condition $(*)$, the implementor may define the end of the $request_i$ operation to be anywhere he wishes. In particular, he can define $request_i$ to include the entire execution of the statement $entry_p$. With this definition, any algorithm that enforces mutual exclusion of *critical_section* operations trivially satisfies condition $(*)$. Thus, in this case, the condition is vacuous.

What Does Priority Really Mean?

What do we mean when we say that something is an FCFS algorithm? FCFS was defined in [10] by condition $(*)$, under the interpretation in which $request_i$ is a nonatomic operation, with the following additional constraint:

($\dagger$) The $request_i$ operation does not involve any waiting for other processes.

To prove that his program is FCFS, the implementor is free to define $request_i$ any way he likes so long as condition ($\dagger$) is satisfied. For a Blaise program, in which "busy waiting" is the only kind of waiting possible, ($\dagger$) is satisfied if the execution of $request_i$ takes a bounded number of program steps.[3]

There is no obvious way to define absence of waiting for a Tony program, which can make it difficult to decide whether or not a Tony program is an FCFS algorithm. This is the reason the challenge requires that a Tony simulation of an FCFS Blaise program be regarded as FCFS. The simulation can be viewed as a "compiled version" of the Blaise program, and it is reasonable to expect the compiled version of an FCFS program also to be an FCFS program.

To specify FCFS, one needs both $(*)$ and ($\dagger$), so one must be able to define what waiting means. Moreover, the definition of waiting should be independent of the programming language, since the specification of the $request_i$ operation should not depend upon how $entry_p$ and $exit_p$ are implemented. For example, $entry_p$ and $exit_p$ might call subroutines that invoke special-purpose hardware to perform the necessary synchronization.

Specifying other kinds of priority poses exactly the same difficulty as specifying FCFS. Consider the classical readers/writers problem with writer

[3]Note that this does not mean that the execution takes a bounded length of time; Blaise does not guarantee any bound on how long a process may wait before executing one step.

priority [2]. In this problem, a process that has issued a request to write has precedence over a reader that has not yet begun to read. Letting $start.rd_i$ denote the operation of starting to read, this is expressed as follows.

($*'$) For any operations $write_i$ and $read_j$, if $request_i$ precedes $start.rd_j$ then $write_i$ must precede $read_j$.

Condition ($*'$) has the same trouble as condition ($*$). If $request_i$ is the operation $enter_i$ that begins the request, then no Blaise program can meet this specification. On the other hand, if the implementor is allowed to define the endpoint of this operation, then condition ($*'$) is vacuous because $request_i$ can be defined to extend until the beginning of $write_i$.

4 A Closer Look at Specification

An Informal Look

The difficulty in specifying priority should convince the reader that we need to examine more closely what it means for a program to implement a specification. To write a specification, there must be an object to be specified and a well-defined interface between the object and its environment. We can ask for a specification of a telephone exchange because we know both the object to be specified (the exchange) and its interface with the environment (the wires leading to the telephones on the exchange and to other exchanges). It is meaningless to ask for a specification of the solar system because we have no idea what the interface is between the solar system and its environment.

I will call the object being specified a "module". A complete specification of a module must contain all the information needed to determine if a particular implementation is correct, where correctness means that the module interacts properly with its environment. An examination of the specifications presented to illustrate most methods—for example, the specification of a queue (bounded buffer) in [8], [17], or [18]—reveals that they are incomplete. From these specifications, one cannot tell whether the operations are initiated by calling a subroutine or by raising a voltage on a wire. A program and a piece of hardware cannot both interact properly with the same environment. Only in [11] is the interface specified, being defined as a simple subroutine-calling mechanism, but there was no explanation of why this implementation-level detail was introduced into a paper on specification.

A complete specification must have two parts: a specification of the module's interface and a specification of its internal behavior. The internal behavior can be specified in terms of high-level abstractions like queues and *write* operations. However, since the interface determines whether an operation is initiated by calling a Pascal subroutine named `put` or by raising the voltage on line number 7 to 4.5 ± 1 volts, it must be specified in terms of implementation-level concepts like subroutine names and voltages. We want to make the interface specification as small as possible, specifying as much as we can in terms of the internal behavior, which can be described with nice, high-level concepts; but the interface specification is necessary. Most specification methods ignore the interface and consider only the internal behavior.

The implementor should have complete freedom in implementing the objects and operations that describe the internal behavior. If the specification contains an internal operation that puts an object on a queue, then the implementor can define that operation to be the act of storing an item in an array, of adding it to a linked list, or of setting the voltage levels on the wires leading to some special device. On the other hand, the interface must be completely specified at the implementation level. The need to partition a specification into an internal part, which is implementation-independent, and an interface specification, which depends upon the implementation, was recognized in [6], and is embedded in the Larch system [7].

In the challenge, the interface is described by requiring the implementation to consist of Blaise or Tony $entry_p$ and $exit_p$ statements that are used in a particular way. Because I ignored the problem of how shared variables and extra processes are declared, I could pretend that the Blaise and Tony implementations had the same interface specification. In a more formal approach, the interface would have to be specified somewhat differently for the two languages.

A Formal View

As described in condition 1 of the challenge, a formal method for proving that an implementation meets its specification must convert the specification and its implementation into a mathematical formula $\mathcal{C}$ in some formal system $\mathbf{L}$ such that the implementation is correct if and only if $\mathcal{C}$ is a valid formula of $\mathbf{L}$. I now give a very vague, high-level discussion of how this is done.

A formal specification is written in a language having a formal semantics, which means that the specification can be translated to a mathematical object $\mathcal{S}$ in some formal system $\mathbf{S}$. Similarly, a formal semantics for the implementation language describes the implementation as a semantic object

$\mathcal{I}$ in a formal system **I**. To be able to speak formally about the correctness of the implementation, there must be a mapping $\mathcal{F}$ from objects in the system **S** to objects in the system **I**, so that $\mathcal{F}(\mathcal{S})$ is an object of **I**. The object $\mathcal{F}(\mathcal{S})$ is the formal representation of the specification in the semantic domain of the implementation.

The formal system **L** of the challenge is the system **I**, and the formula $\mathcal{M}$ that expresses the correctness of the implementation is the formula of **I** that means "$\mathcal{I}$ satisfies $\mathcal{F}(\mathcal{S})$". Exactly how $\mathcal{M}$ is constructed from $\mathcal{F}(\mathcal{S})$ and $\mathcal{I}$ depends upon the specification method. I will illustrate with two examples: a pure axiomatic approach and a pure behavioral approach.

In a pure axiomatic approach, an axiomatic semantics is given for both the specification and the implementation. An axiomatic semantics defines $\mathcal{S}$ to be a formula of the logical system **S**—the conjunction of all the "axioms" comprising the specification—and $\mathcal{I}$ to be a formula of **I**. The mapping $\mathcal{F}$ is a function from the formulas of **S** to those of **I**. For example, suppose the specification is in terms of the value of a queue **q**, which is implemented with an array **a**. To talk about the correctness of the implementation, for every possible value a of the array **a** we must know the value $Q(a)$ of **q** that it represents.[4] For any formula $\mathcal{R}$ of **S**, the formula $\mathcal{F}(\mathcal{R})$ of **I** is obtained by substituting $Q(\mathbf{a})$ for **q** in $\mathcal{R}$. Thus $\mathcal{F}(\mathcal{R})$ is obtained by translating the statement $\mathcal{R}$, which is an assertion about the specification-level object **q**, into an assertion about the implementation-level object **a**.

In this approach, $\mathcal{M}$ is the formula $\mathcal{I} \supset \mathcal{F}(\mathcal{S})$. In other words, the implementation is correct if and only if the axioms comprising the semantics of the implementation imply the axioms of the specification, after the latter are translated by $\mathcal{F}$ into assertions about the implementation. This is discussed at greater length in [12] for one particular axiomatic method.

In a pure behavioral approach, the formal semantics of the implementation and specification are sets of behaviors: $\mathcal{S}$ is the set of all behaviors allowed by the specification, $\mathcal{I}$ is the set of all behaviors that could be produced by the implementation, and **S** and **I** are formal systems for reasoning about sets of behaviors. For a behavior b in the specification domain, $\mathcal{F}(b)$ is the corresponding behavior in the implementation domain. In the mutual exclusion example, the operation $\mathit{critical\_section}_i$ is a single action in the behavior b; it corresponds to a set of actions in $\mathcal{F}(b)$—namely, the set of all the Blaise program steps in a single execution of the critical section.

[4] The mapping Q may be partial, since $Q(a)$ need only be defined for values a of **a** that can arise during the program's execution.

One can define the formula $\mathcal{M}$ to be $\mathcal{I} \subset \mathcal{F}(\mathcal{S})$, where $\mathcal{F}(\mathcal{S}) = \{\mathcal{F}(b) : b \in \mathcal{S}\}$. In other words, the implementation is correct if and only if every possible behavior of the implementation is allowed by the specification. Some specification methods define $\mathcal{M}$ to be $\mathcal{I} = \mathcal{F}(\mathcal{S})$, requiring that the implementation be able to produce all the behaviors described by the specification.

There are other possibilities—for example, an axiomatic semantics for the specification and a behavioral semantics for the implementation. In any case, the definition of correctness of the implementation involves the mapping $\mathcal{F}$. For the specification of FCFS, the mapping $\mathcal{F}$ is what determines which operations at the implementation level correspond to the specification's $request_i$ operation. A complete specification must include not only $\mathcal{S}$, but also the part of $\mathcal{F}$ that determines the interface. For the queue example, it is this part of $\mathcal{F}$ that specifies whether one puts an element in the queue by calling a subroutine or raising a voltage level. Correctness means that there exists some $\mathcal{F}$, part of which is determined by the specification, such that $\mathcal{F}(\mathcal{S})$ satisfies $\mathcal{I}$. The implementor is free to define the rest of $\mathcal{F}$, specifying the correspondence between the implementation and the internal part of the specification any way he wishes in order to prove the correctness of his implementation. The specification places no constraint on any part of the implementation other than the interface.

5 Conclusion

Having described the difficulty in specifying priority, it would be nice if I could either explain how it can be done or else prove that it is impossible. Unfortunately, I can do neither. I believe that one cannot write a satisfactory general specification of priority—one that works for a variety of implementation domains. The difficulty in expressing priority arises from the requirement that the *request* operation should involve no waiting for other processes. Waiting is an implementation-level concept that I feel cannot be expressed in a general way. However, this conjecture, like Church's thesis, is not susceptible to formal proof. At best, one can prove only that some particular formalism cannot express priority.

If priority is not expressible by current formal specification techniques, how should we specify concurrent systems? Priority is generally regarded to be a fundamental concept that must be specified. Must we add new primitives to express it? My tentative answer is no. I believe that priority cannot be expressed precisely in those situations when it is not a fundamental property.

Remember that condition ($*$) does express FCFS priority if the $request_i$ operation is interpreted to be the interface operation $enter_i$. The atomicity of $enter_i$ is irrelevant; what matters is that $request_i$ be the interface operation. Priority is a basic system requirement only when its effect is directly visible to the user, which is the case only when the *request* operation is externally visible—that is, when it is part of the interface. For example, suppose we want transactions issued by certain users to receive higher priority. The *request* operation can then be defined as the entire sequence of actions performed by the user in issuing the request, from the first keystroke to his notification that the request has been accepted by the system.

When the *request* operation is not externally visible, then priority is a mechanism, not an end in itself. In the internal specification, we give writers priority not because correctness requires them to have precedence, but rather to ensure that they receive adequate service. For example, a common use of writer priority is to guarantee absence of starvation—a waiting writer eventually writes despite a continual stream of readers. What we need to specify is the required service, not the mechanism used to achieve it. The absence of starvation belongs to a fundamental class of properties, known as liveness properties, that are easily expressed in temporal-logic based methods like the ones of [8], [11], [13], [16], [17], and [18]. It is the basic requirement that should be specified, not the priority mechanism used to achieve it.

Acknowledgement

The difficulty in specifying priority was discovered during a discussion with Richard Schwartz and Michael Melliar-Smith, in which they kept poking holes in my attempts to specify FCFS until we all finally recognized the fundamental problem. In writing this paper, I was aided by the helpful comments of Brent Hailpern and Fred Schneider and the enthusiastic objections of the members of IFIP Working Group 2.2.

References

[1] John Bethel, ed. *Webster's New Collegiate Dictionary.* G. & C. Meriam Co., Springfield, Mass. (1956).

[2] P. J. Courtois, F. Heymans and D. L. Parnas. Concurrent Control with "Readers" and "Writers". *Comm. ACM 14*, 10 (October 1971), 667-668.

[3] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Comm. ACM 17*, 11 (November 1974), 643-644.

[4] Susan Gerhart et al. An Overview of AFFIRM: A Specification and Verification System, *IFIP Congress 80*, (Oct. 1980).

[5] Irene Greif. A Language for Formal Problem Specification, *Comm. ACM 20*, 12 (Dec. 1977), 931-935.

[6] J. V. Guttag and J. J. Horning. Formal Specification as a Design Tool. *Proc. ACM Symposium on Principles of Programming Languages*, Las Vegas, (January 1980), 251-261.

[7] J. V. Guttag and J. J. Horning. An Introduction to the Larch Shared Language. *Proc. IFIP Congress '83*, Paris, (1983).

[8] Brent Hailpern. *Verifying Concurrent Processes Using Temporal Logic*, Lecture Notes in Computer Science 129, Springer-Verlag (1982).

[9] Howard Katseff. A Solution to the Critical Section Problem with a Totally Wait-free FIFO Doorway. Technical Memorandum, Computer Science Division, University of California, Berkeley.

[10] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem, *Comm. ACM 17*, 8 (Aug. 1974), 453-455.

[11] Leslie Lamport. Specifying Concurrent Program Modules, *ACM Trans. on Prog. Lang. and Systems 5*, 2 (Apr. 1983), 190-222.

[12] Leslie Lamport. What Good is Temporal Logic? *Information Processing 83*, R. E. Mason (ed.), Elsevier Science Publishers (North-Holland), 1983, 657-668.

[13] Amy Lansky and Susan S. Owicki. GEM: A Tool for Concurrency Specification and Verification, *Proc. of the Second Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1983), 198-212.

[14] Peter E. Lauer, P. Torrigiani and M. Shields. COSY: A System Specification Language Based on Paths and Processes, *Acta Informatica 12* (1979), 109-158.

[15] Robin Milner. *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag (1980).

[16] Richard L. Schwartz and P. M. Melliar-Smith. Temporal Logic Specification of Distributed Systems, *Proc. of the IEEE Conference on Distributed Systems*, (Apr. 1979).

[17] Richard L. Schwartz, P. M. Melliar-Smith and F. H. Vogt. An Interval Logic for Higher-Level Temporal Reasoning, *Proc. of the Second Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1983), 198-212.

[18] Pierre Wolper. Specification and Synthesis of Communicating Processes Using an Extended Temporal Logic, *Proc. of the Conference on the Principles of Programming Languages*, Albuquerque (Jan. 1982).