# Using LOTOS for Specifying the CHORUS Distributed Operating System Kernel*

Charles PECHEUR**

## Abstract

The goal of the work presented in this paper was to test the adequacy of LOTOS for the specification of operating systems, by specifying the basic structures and functionalities of the distributed operating system CHORUS V3. The paper focusses on the discussion of the issues encountered during the development of the specification, rather than on the specification itself. Three successive approaches based on different specification styles are presented and discussed. A difficulty related to the specification of processing systems is analysed.

**Index Terms :** distributed operating systems, formal description techniques, specification languages, LOTOS, CHORUS, processing systems.

## 1. Introduction

The FDT LOTOS was developed for the purpose of specifying communication systems, in the framework of the Open Systems Interconnection model. It has now become an International Standard (ISO IS 8807) [9], and has already been used to specify numerous communication protocols and services. We refer the reader to the literature for a tutorial presentation of LOTOS (e.g. [3]).

LOTOS is based on two complementary mathematical theories : the theory of process algebras (CCS [11], CSP [7]) and the theory of abstract data types (ACT ONE [5]). The semantics of a LOTOS specification is expressed in terms of a few general abstract concepts. This allows using LOTOS not only in the field of communications, but for the specification of more general dynamic systems. However, up to now, few experiences have been engaged in other directions. In particular, we would like to evaluate the adequacy of LOTOS for the specification of computer operating systems (or OS, for short).

* This work was supported by the CEC under ESPRIT Project 1535 'Aphrodite'.

**Ch. Pecheur is a Research Assistant of the Belgian National Fund for Scientific Research, in the department of Systèmes et Automatique of the Institut Montefiore, B28, University of Liège, B-4000 Liège, Belgium. He is a member of IEEE.

This paper relates a real-case experience : we tried to use LOTOS to develop a specification of CHORUS. CHORUS is a distributed OS (or DOS) kernel, whose development started at INRIA, a French public research institute, in 1980 [6], [2]. In 1986, the *CHORUS Systèmes* company was created with the aim of turning CHORUS into an industrial product [14]. The specification presented hereafter focuses on the third version of CHORUS, and was a contribution to the ESPRIT Project 1535 'Aphrodite' [13][20].

The specification discussed in the present paper was preceeded by a specification of the previous version of CHORUS, CHORUS V2 [10]. That first work allowed us to get a first experience in applying LOTOS to CHORUS, and to get a deeper understanding of CHORUS concepts. It also showed the potential benefits of using a formal approach, by enlightening some weaknesses in the coherency of the CHORUS architecture and allowing us to discuss two alternative solutions on the basis of their formal description [12].

The initial objective was to update the specification of CHORUS V2 to include the new CHORUS V3 features. However, the important changes that were introduced in the new version lead us to restart the specification from scratch, while keeping the first specification as a reference. Moreover, the resulting increase of complexity has complicated the specification task.

It is important to notice that this work differs from a usual FDT-based design process, under several aspects. First of all, our ambition was to point out the difficulties associated with the expression in LOTOS of the architecture and functionality of CHORUS, rather than provide the specification itself — validation of the specification with respect to the real system was not essential, though desirable. The results of this work are in the experience gained, not in the written LOTOS code. Accordingly, this paper will put the emphasis on the discussion of the different difficulties encountered, rather than on the precise contents of the specification. Only a few (simplified) excerpts of the specification will be given here, for illustration purposes.

Secondly, due to the innovative nature of the approach, the goal itself was much less precisely pre-defined than in a more conventional specification. Typically, a usual specification covers a well delimited entity with a well understood interface (e.g. a protocol entity with two service access points) and respects some pre-defined specification style, according to its intended use. In our case however, the object to be specified is a complex system resulting of the combination of various kinds of entities and interfaces, without a priori instructions about which are to be specified and how. Moreover, formal specification of operating systems is a new, hardly explored domain, so that few clues were available about how to answer these questions. This explains the somewhat empiric character of the presented work, starting with a naïve, intuitive specification and then tuning it up towards a more acceptable one.

The following sections are organized as follows : the next section gives an overview of the main features of CHORUS V3, section 3 discusses the main issues encountered during the

development of the specification, and the final sections draw some conclusions from the experience.

We will use the following editorial conventions : *italics* will be used to put emphasis; *oblique helvetica* will be used for LOTOS code and specification identifiers within plain text, with LOTOS keywords in ***bold***. Fig. 1 summarizes conventions used in graphical representation of LOTOS process structures.

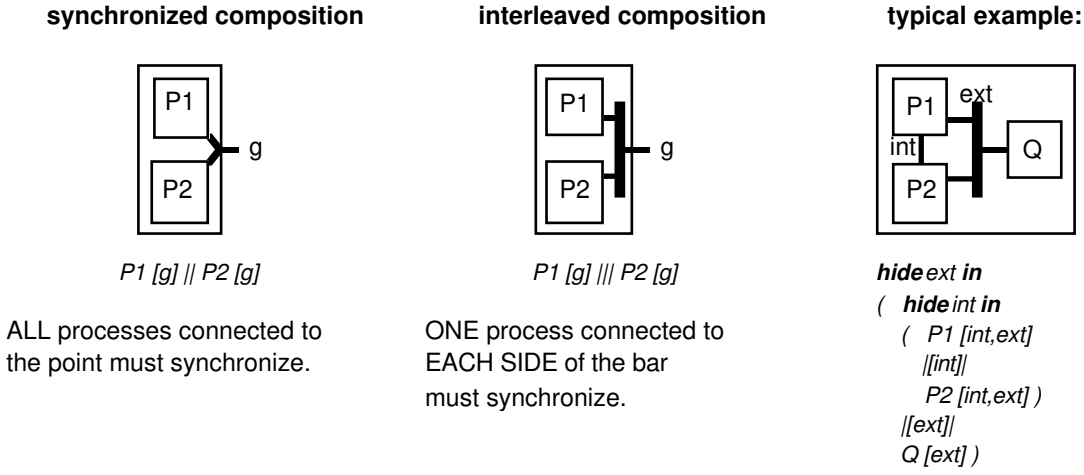


*Fig. 1. Representation of LOTOS process structures*

## 2. The CHORUS Distributed Operating System

### 2.1. CHORUS main abstractions

The architecture of CHORUS is defined in terms of a small collection of simple abstract elements (Fig. 2). This is an important feature in our point of view, as it eases the formal specification of the system.

A CHORUS *system* is a collection of computers interconnected through some communication medium. In CHORUS terminology, a machine together with its attached peripherals is named a *site*.

CHORUS applications are subdivided into collections of *actors*. An actor corresponds essentially to a private virtual memory space for program execution. The actor is the basic unit of distribution : an actor is always local to a site. A single CHORUS actor may support multiple concurrent program executions; a single sequential flow of execution within an actor is named a *thread*. Resources allocated to an actor are shared by all its threads. The memory space of an actor is divided into *regions*, each region allowing access to a portion of a so-called

*segment*. A segment is some representation of information; typically, it is some file or some swap space on secondary storage.

Actors interact by means of exchanges of asynchronous *messages* via message queues named *ports*. This mechanism is called *Inter-Process Communication* (IPC for short). Ports are always owned by one and only one actor, but can be created, moved between actors and destroyed dynamically. The messages are addressed to the ports, independently of the actor to which these ports are attached. *Groups* of ports can also be defined and used as addresses. Actors, ports and groups are assigned system-wide *unique identifiers*, which allows them to be uniformly designated from any site in the system, thereby offering a basic mechanism for transparent distribution.

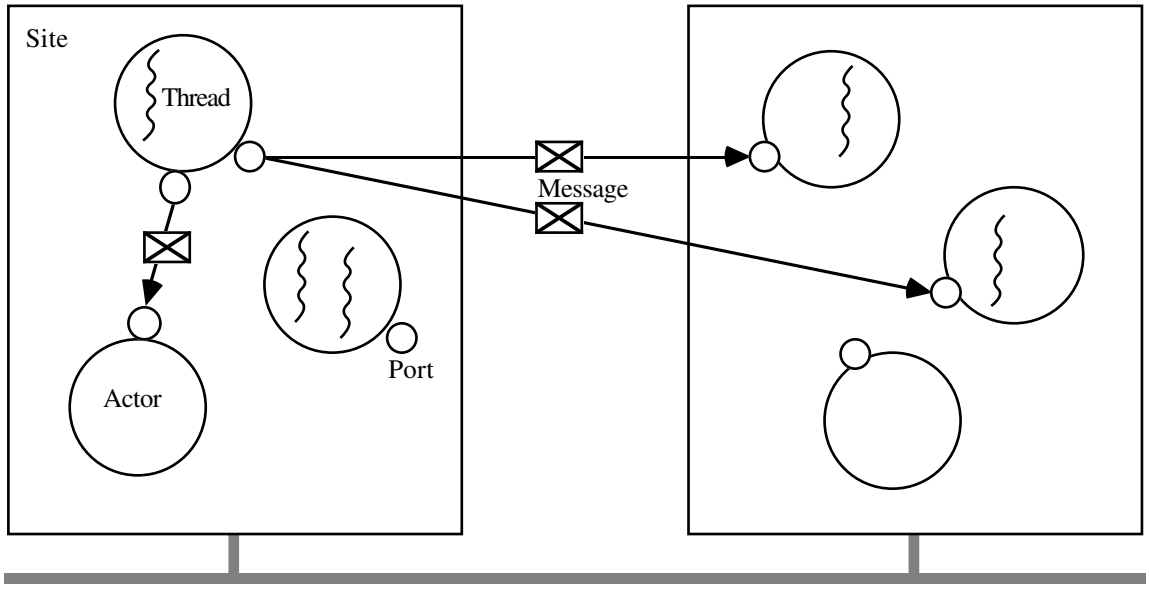


*Fig. 2. Main CHORUS abstractions*

## 2.2. Internal organization

A main choice in CHORUS is to have a two level logical structure, where specific subsystems are built by adding, on each site, *system server* actors on top of a generic small-sized *kernel*. The kernel handles the local CHORUS resources and offers a collection of primitives to local threads. In turn, the servers make use of these basic tools to provide extended, subsystem-specific functionalities (for example, a distributed UNIX has been implemented on top of CHORUS by adding actors for managing processes, files, pipes and devices). Our specification will concentrate mainly on the kernel itself.

The functions handled by the CHORUS kernel may be divided into a few basic categories (Fig. 3) :

- A *supervisor* is in charge of dispatching low level exceptions (interrupts, traps, errors…).
- A *real-time executive* controls the allocation of processors to local threads and provides simple synchronization mechanisms.
- The *VM manager* controls the allocation of memory space and structures virtual memory address spaces. It is only in charge of managing main memory (CHORUS *regions*) : swap files or any other piece of information residing on external devices (CHORUS *segments*) are controlled by specific actors named *segment servers* or *mappers.* The VM manager interacts with these mappers by means of messages to provide the overall VM facilities.
- The *IPC manager* is responsible for the local part of the IPC facility, while relying on so-called *network manager* actors for managing network-specific protocols and devices in order to exchange messages with remote sites.

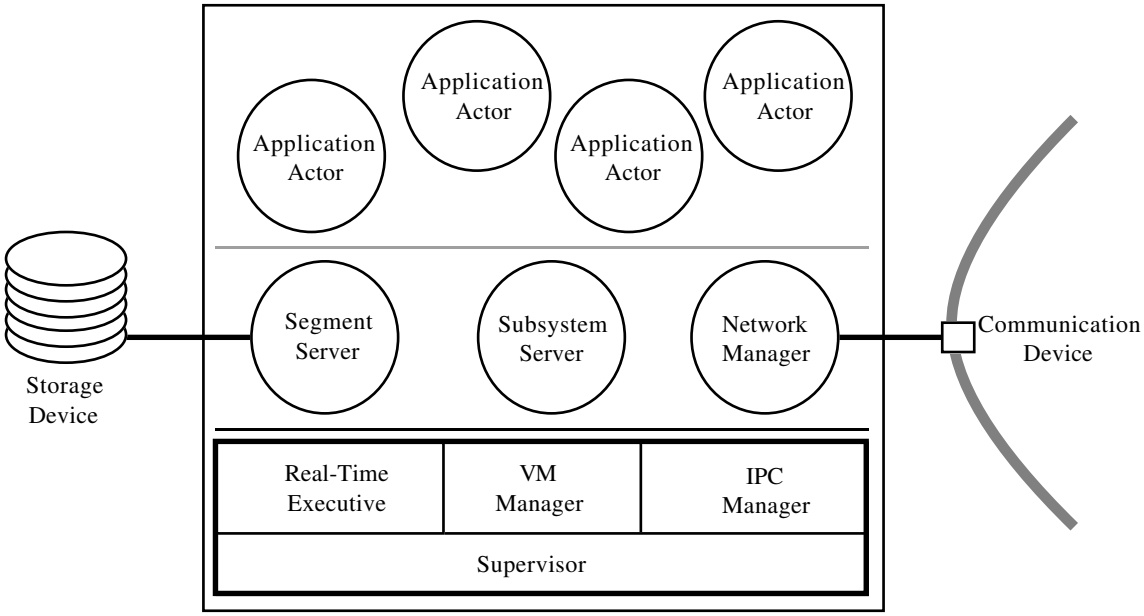


*Fig. 3. CHORUS internal organization*

## 3. Specification issues

As mentioned in the introduction, this work differs from a classical specification job (e.g. of a communication service or protocol), mainly in that much less specification options were initially fixed, demanding more choices to be made. These choices range in the following three main categories, which will be developed in the next sections :

- *Scope* : A CHORUS system is a composite entity involving different kinds of activities (physical or virtual I/O devices, actors, kernels,…), more or less implied in its

functionality, and which have to be accordingly taken into account or not in the specification.

- *Architectural semantics* : This concept denotes the relation between the architectural elements of the specified system and the abstract FDT constructs used to represent them [18]. CHORUS implies various kinds of entities and interactions, for which an adequate representation has to be chosen within the framework of LOTOS. In particular, the representation of executive[1] functionalities will be discussed.

- *Styles* : A given system can be specified in LOTOS in many equivalent ways. It is now well accepted that the specifier should adopt a coherent style, more or less extensional[2], depending on the intended use of the specification. Under this point, the description of CHORUS asks for a mix of structural and behavioural aspects, for which we have to adopt an adequate style.

Before further discussing about the specification of CHORUS, we also want to make a few general comments over the distinction between the *style* and the *interpretation* of a LOTOS specification.

The *style* of a specification is a syntactical concept : it concerns the way the specification is *written*. A style may be more or less extensional, depending on how much internal structure the specification text presents. On the other hand, the *interpretation* of a specification has to do with the architectural semantics associated with that syntactical structure, that is, how the specification is to be *understood*. As LOTOS concepts (processes, parallel and hiding operators…) correspond to distributed system structuring concepts (entities, concurrency, communications…), we can give them a meaning in terms of the internal structure of the specified system. However this need not necessarily be so : indeed, according to the operational semantics of LOTOS, a specification only defines a resulting external behaviour.

The “structure of a specification” can thus be considered or not as a “specification of a structure”. If the standard external, extensional interpretation is appropriate, then the structure of the specification should only help understand its content, but not dictate the structure of implementations (though this is hardly avoidable in practice — see [19]). If an architectural significance is to be associated with some part of the specification structure, then the interpretation goes beyond the strict semantics of the language : the specifier is responsible for stating explicitly the expected “depth of interpretation”, i.e. which sub-expressions have a meaningful structure and which should be taken only for their resulting behaviour.

---

[1] “Executive” here means “which controls the execution of computer programs”, i.e. process creation and scheduling.

[2] A specification pertains to some style if its structure conforms to some corresponding syntactic restrictions (e.g. absence of hiding operator). Extensionality (vs. intensionality) means independence from any internal structuring [Vissers88].

## 3.1. General structure

When the work on the specification of CHORUS was engaged, the first steps were intuitive tries to map CHORUS concepts into LOTOS structures, without any concern about methodological issues. The CHORUS V2 specification did not explicitly follow any style, but its main structure was an image of the real structure of CHORUS V2 : It was in fact a *resource-oriented* specification [19], with a mix of intensional and extensional meanings. This was also the case in the first steps of the V3 specification. While gaining experience, and in order to be able to manage the complexity of CHORUS V3, we took a more careful attention to style and extensionality problems.

A first step towards a more rigorous specification was to clearly distinguish the general structural description from the behavioural specification of CHORUS functionality. Taken to its largest extent, a CHORUS system includes a set of computing machines, together with their attached devices and the network interconnecting them. We want our specification to contain some description of that overall *structure*, but the *behaviour* of most of those entities (e.g. I/O devices or user applications) is not within the scope of our specification. We emphasize this distinction by splitting the overall description into two separate specifications :

- The *system specification*, which describes the overall architecture of a CHORUS system, uses a resource-oriented style and requires a *structural*, *intensional* interpretation;
- The *kernel specification,* which defines the core behaviour of CHORUS, requires a *behavioural*, *extensional* interpretation, although its highest level also adopts a resource-oriented style.

As CHORUS communications are based (on some implementations) on the ISO transport service, we also used a published specification of that service [16] as a basis for our specification.

The system specification is not complete, in the formal sense : it describes, by means of LOTOS constructs, how the basic components of the system interact with each other; but not the behaviour of these components : they are represented as “empty” process instantiations having no corresponding process definitions (Fig. 4).

```
specification CHORUS_system [dev,init] : noexit :=

...     (* data type defs *)

behaviour

hide TSAP in
(       all_sites [dev,init,TSAP]          (* a dynamic collection of sites *)
        |[TSAP]|
        TransportService [TSAP] )          (* connected to a communication medium *)

where
```

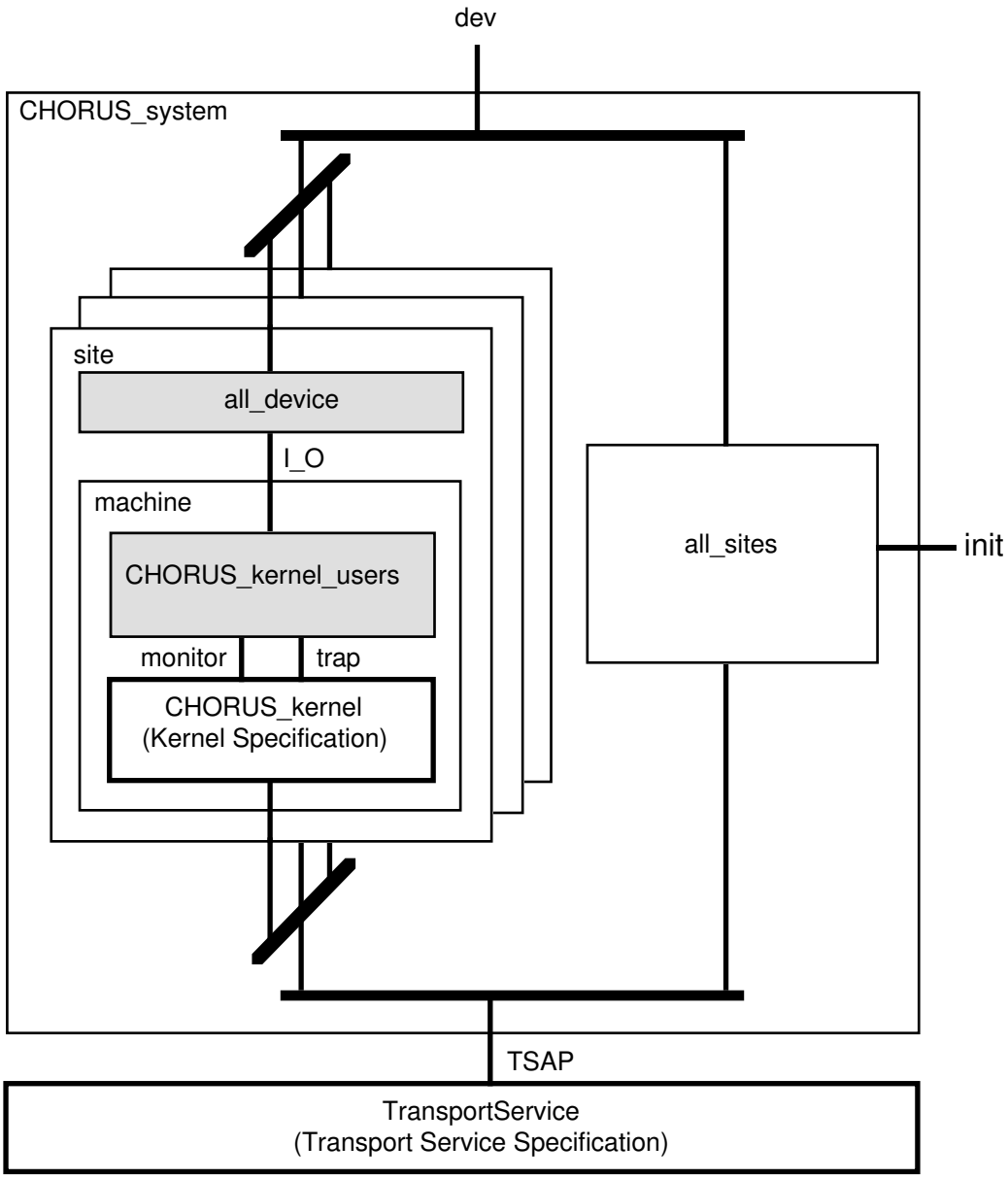


*Fig. 4. Structure of the CHORUS system specification*

```
process all_sites [dev,init,TSAP] : noexit :=

all_sites [dev,init,TSAP]
|||
site [dev,TSAP]

endproc

process site [dev,TSAP] : noexit :=

hide I_O in
init ?site: site_name !site_on;                  (* switch on a new site *)
(       (       machine [TSAP,I_O] (site, …)          (* a computer *)
                |[I_O]|
                all_device [dev,I_O] (site) )    (* its unspecified devices *)
        [> init !site !site_off; stop )          (* until it is switched off *)

endproc
```

```
process machine [TSAP,I_O] (site : site_name, …) : noexit :=

hide monitor,trap in
(       CHORUS_kernel [TSAP,monitor,trap]  (…)        (* the specified kernel *)
        |[monitor,trap]|
        CHORUS_kernel_users [I_O,monitor,trap] )      (* other unspecified parts, including actors *)

endproc
```

The *trap* gate models the system call interface between the CHORUS kernel and its local actors. The meaning of the *monitor* gate is the object of section 3.2.

The kernel specification, on the other hand, is aimed at being self-contained and semantically correct (although this was not formally validated). It corresponds to one of the processes of the system specification. The thrust behind this specification was to produce a kind of CHORUS kernel service definition; however this idea had to be altered somewhat, as will be explained in section 3.2.

Although intended as a behavioural specification, the kernel specification is written in a resource-oriented style. This internal structuring is originally inherited from the V2 specification. However, its persistence can be justified by the existence of a clear partition within CHORUS functionalities : the internal gates only partially correspond to real CHORUS internal interfaces, which are not documented anyway. A similar approach is followed and commented for the specification of the OSI session protocol in [1]. The top level structure is illustrated hereafter and in Fig. 5[3].

```
specification CHORUS_kernel [TSAP,monitor,trap] (…) : noexit :=

…       (* data type defs *)

behaviour

hide transport,IPC,local in
(       (       (       (       local_call_dispatcher [trap,IPC,local] (…)
                                                (* receives local calls, calls local kernel or
                                                sends messages to other kernels *)
                                |||
                                remote_call_dispatcher [IPC,local] (…) )
                                                (* receives messages from remote kernels,
                                                calls local kernel if appropriate *)
                        |[local]|
                        local_kernel [monitor,IPC,local] (…) )
                                                (* handles local CHORUS objects *)
                |[IPC]|
                IPC_manager [transport,IPC] (…) )
                                                (* handles local communications, forwards
                                                messages to network manager if destination
                                                is not local *)
        |[transport]|
        network_manager [TSAP,transport] (…) )
                                                (* handles communications with other sites *)
```

[3]One auxiliary process representing global kernel information, and its associated gate, have been suppressed to clarify the presentation.

***where***

*...  (* process defs *)*

***endspec***

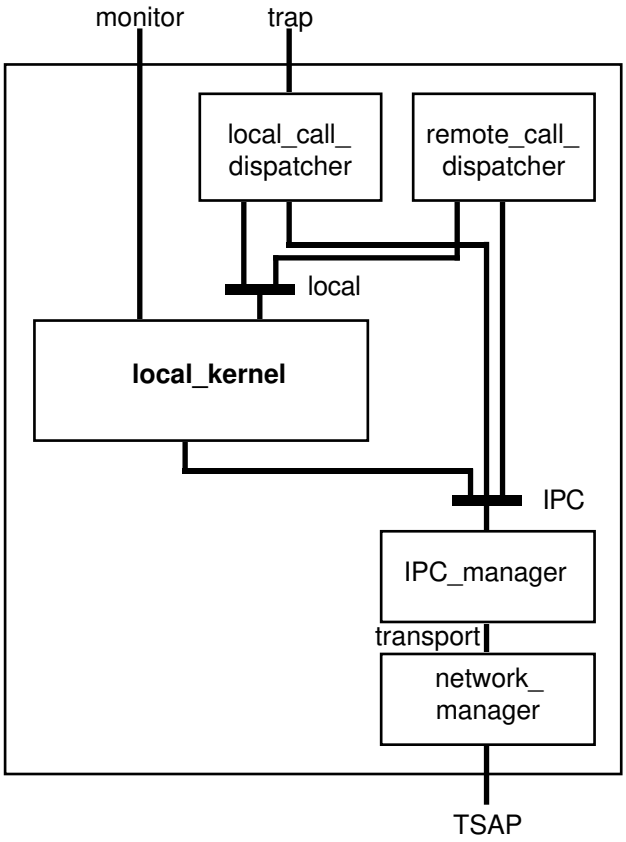


*Fig. 5. Structure of the kernel specification*

Most of the complexity of the CHORUS kernel resides in the management of local objects, causing some problems in the definition of the *local_kernel* process. These difficulties are the object of section 3.3.

## 3.2. Specification of executive systems

This section deals with the specification of the executive part of CHORUS. It appears to cause specification difficulties, which lead to the introduction of the *monitor* gate.

In a first approach, CHORUS is globally organized in a similar way as communication systems (and, more generally, most modern software packages) : it follows a modular design, where each module provides some self-contained service, while using services offered by other modules. A large part of the functionalities of a CHORUS system can be described in classical terms of “protocol” layers and services, and specified in LOTOS correspondingly : this is true

for all I/O facilities in general (file systems, user terminals…), and for communications in particular.

Under these aspects, CHORUS could thus be seen as a layer which adds functionality to the low level computer resources in order to provide some service to its actors. That service is accessed through a collection of primitives, which take the form of system calls and/or function libraries. This is specified in LOTOS as a process with a "lower" gate modelling access to the machine resources and an "upper" gate whose events correspond to CHORUS primitives. Alternatively, the CHORUS layer could be joined with underlying resources and specified altogether as a CHORUS service, offering only the "upper" gate. Note that the "users" of the CHORUS service are software applications, not human users. The latter interact with the computer through I/O devices, and are thus behind I/O servers, on the point of view of the kernel.

As we concentrate on the kernel of CHORUS, which does not handle general I/O (remember that these are managed by special actors named servers), those underlying resources essentially resume to a communication service, namely the ISO transport service offered at the "lower" gate *TSAP*. Hence a first external structure for the kernel specification could be (Fig. 6) :

```
specification CHORUS_kernel [TSAP,trap] (…) : noexit :=

…

endspec
```

where *TSAP* models the access to the transport service and *trap* the system call interface, with the following event structures :

```
TSAP !ta: TAddress !tcei: TCEI !tsp: TSP        (      * local Transport address and connection
endpoint,
                                                Transport service primitive *)
trap !tid: threadid !prim: CHORUS_primitive     (* calling thread identification,
                                                CHORUS call or return from call *)
```

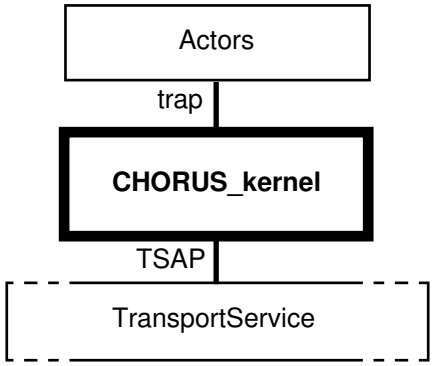


*Fig. 6. CHORUS kernel as a functional layer*

However this simple solution is not fully adequate when the specification of those parts of CHORUS which handle computing resources is taken into account (i.e. thread scheduling and memory management, mainly). The new difficulty stems from the fact that these functionalities imply that CHORUS is responsible for the existence of actors and threads, which are its own users : on their behalf, it monitors memory and processors to spawn and control new actors, memory regions and threads, which in turn become new users. This is by no means an irrelevant side effect, but a fundamental part of its semantics, which cannot simply be ignored at some abstraction level. The classical notion of OS service is not sufficient in this case, since there is a dependency between the lower level layers (i.e. the computing resources) and the service users (i.e. the applications), so that the former cannot be isolated from the latter.

If we keep the above idea of describing CHORUS system call interface as an external gate of the specification, then the actors are considered to be outside the specified system, and the way they are controlled by the OS constitutes a second means of interaction, which will be specified as a new category of events. We have emphasized this aspect by offering them on a new external gate *monitor*. This gate can be equivalently considered as a second part of the CHORUS service interface, supporting “feedback” primitives to the actors, or as an access to the lower service provided by the processing resources. The resulting external structure is the one already given in section 3.1 (Fig. 7).

```
specification CHORUS_kernel [TSAP,monitor,trap] (...) : noexit :=
...
endspec
```

The new *monitor* gate supports controlling interactions from the kernel to its local actors.

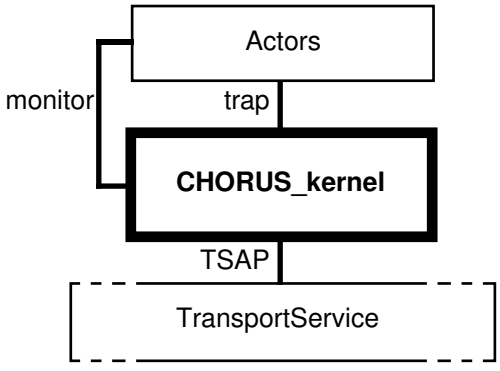


*Fig. 7. CHORUS kernel with execution control*

To avoid excessive complexity, we decided to limit our CHORUS specification to the management of CHORUS objects, excluding low level memory management (page swapping, including access to storage resources through dialog with segment servers) and execution scheduling (time slicing, interrupt handling). We assumed that a kind of virtual machine interface was offered at gate *monitor* , allowing dynamic creation of virtual memory spaces, allocation of regions and execution of virtually concurrent threads in these spaces. This is a

realistic but arbitrary assumption, as such interfaces are internal to the kernel and thus are not described in its documentation.

Things can get even more confusing if we want to take into account the dynamic reconfiguration mechanisms offered by CHORUS, possibly leading to undetermined changes in its own behaviour. However these reconfigurations can be classified into two different categories : either they are part of the modeled behaviour of the system, in which case they are implicit in the global behaviour of the specification, or they correspond to a modification of the described system (i.e., the kernel will not exhibit its correct behaviour afterwards), in which case they are irrelevant to the specification and are not taken into account.

We are aware that this approach is not completely satisfying : putting side by side such mechanisms as system calls and processor scheduling tends to introduce some confusion in the abstraction level of the specification. We believe that this problem is central to the formal specification of operating systems, and deserves further investigations.

## 3.3. Three specification approaches

The most complex part of CHORUS kernel is the management of local objects, which is covered by the *local_kernel* process. The development of that part of the specification raised overspecification difficulties : in particular, our first approach led us to specify intra-kernel synchronization mechanisms, which should normally only be considered at the implementation level. The main origin of such problems is probably to be found in the lack of initial experience of the specifier, leading to the adoption of an inadequate specification style; but also in the procedural descriptions given in the source documents (one system call = one sequence of actions), for which the most obvious specification are simple sequences of LOTOS events (especially for someone who is used to thinking in terms of sequential executions).

These difficulties led us to consider three successive approaches for specifying that part of the kernel, to which we will refer as the *procedure-based*, *object-based* and *constraint-based* approaches, respectively. The following sections will give a short description and discussion of each of these approaches.

### 3.3.1. The procedure-based approach

This first approach is directly issued from the one followed in the V2 specification. It is the closest to a potential real implementation of the kernel : basically, it is a collection of concurrent sequential processes modelling system call executions. Each of these “procedures” can interact independently with a common process managing all kernel information in global tables, so that all those “data” are available to all those “procedures”. This leads to a “low level” resource-oriented specification, where the resources are kernel processes, on one side, and kernel variables, on the other (Fig. 8).

```
process local_kernel [monitor,IPC,local] : noexit :=

hide tables in
(       kernel_procs [monitor,IPC,local,tables]   (* execution of system calls *)
        |[tables]|
        kernel_tables [tables] )                          (* handles all kernel data *)

endproc

process kernel_procs [monitor,IPC,local,tables] : noexit :=

kernel_proc [monitor,IPC,local,tables]            (* handles one call *)
|||
kernel_procs [monitor,IPC,local,tables]   (* handles other calls *)

endproc

process kernel_proc [monitor,IPC,local,tables] : noexit :=

accept_system_call [local]                                    (* get a request *)
>> accept req : CHORUS_req in
handle_system_call [monitor,IPC,tables] (req)    (* serve the request *)
>> accept conf : CHORUS_conf in
return_system_call_result [local] (conf)                  (* return the result *)

endproc
```

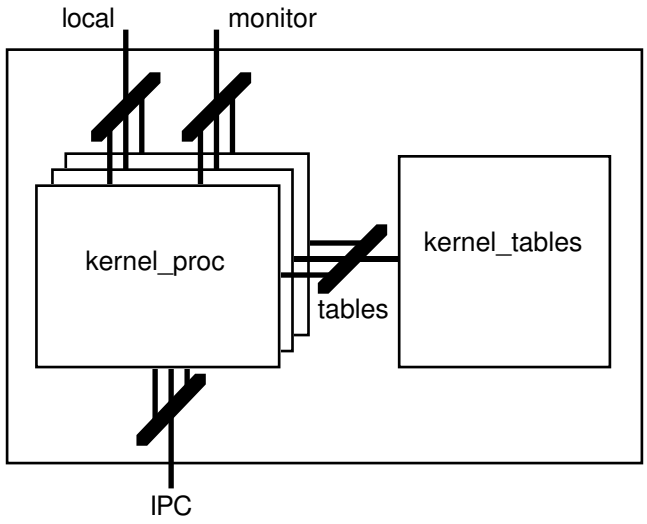


*Fig. 8. The procedure-based approach*

This first approach had the merit (which appeared to be its weakness as well) of being close to the structure of a real implementation. It was simple and easily understandable. However it failed to take into account the ability of the system to handle safely concurrent system calls referring to a common object. To deal with such problems, explicit synchronization structures such as semaphores had to be introduced into the specification. Obviously, this first try was too close to an implementation structure : such synchronization properties had to be a consequence of the structure of our specification, instead of appearing explicitly.

### 3.3.2. The object-based approach

This second approach tried to ease the specification of synchronization properties by associating one LOTOS process with each local CHORUS object handled by the kernel (actors, threads, ports, groups, etc.). With this approach, synchronization of concurrent accesses to one object are naturally specified as restrictions on the events offered by the process handling that object. A dispatching process accepts and forwards each system call to the process handling the concerned object; moreover, objects associated with an actor (threads, ports, regions, messages) are handled by subprocesses of the process handling that actor. This approach corresponds to a "high level" resource-oriented style; where the resources are subdivided according to the object to which they correspond. It also has some flavour of object-oriented programming (Fig. 9).

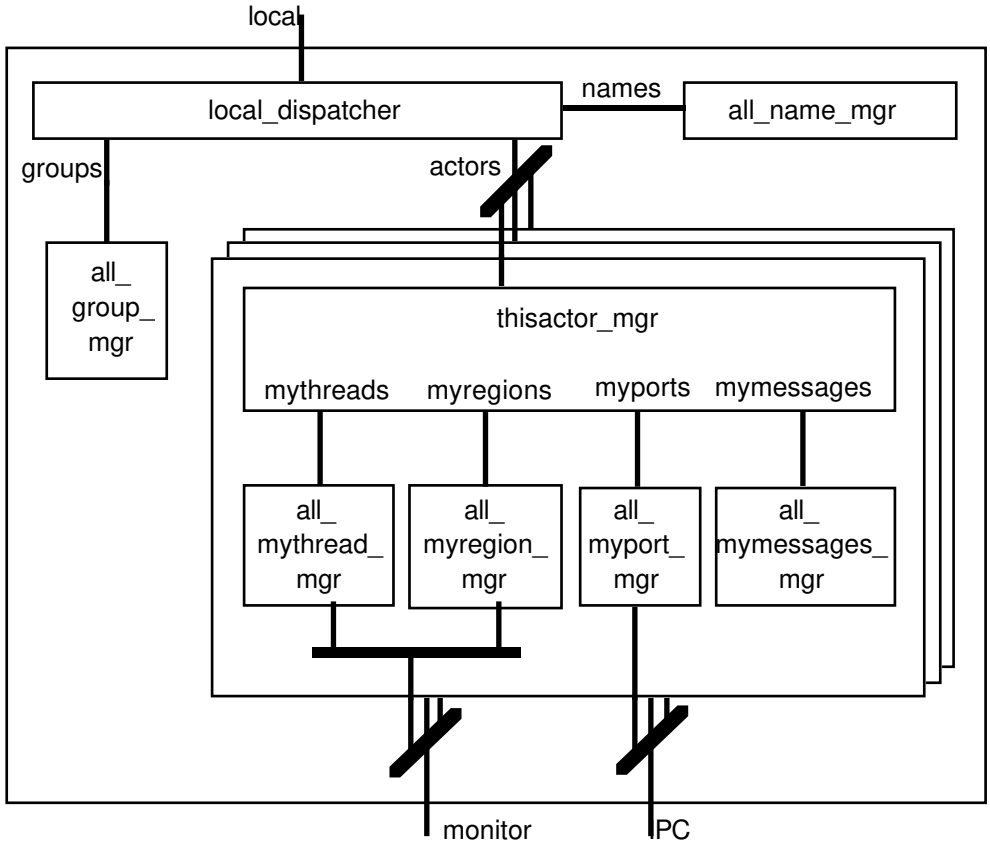


*Fig. 9. The object-based approach*

```
process local_kernel [monitor,IPC,local] : noexit :=

hide actors,groups,names in
(       local_dispatcher [local,actors,groups,names]    (* gets and dispatches system calls *)
        |[actors,groups,names]|
        (       all_actor_mgr [monitor,IPC,actors]      (* management of actors *)
                |||
                all_group_mgr [groups]                  (* management of groups *)
                |||
                all_name_mgr [names] ))                 (* management of names *)
```

```
endproc

process all_actor_mgr [monitor,IPC,actors] : noexit :=

choice act: UI []                                   (* for any actor identifier *)
(       one_actor_mgr [monitor,IPC,actors] (act)    (* management of that actor *)
        |||
        all_actor_mgr [monitor,IPC,actors] )        (* management of other actors *)

endproc

process one_actor_mgr [monitor,IPC,actors] (act : UI) : noexit :=

hide myports,mythreads,myregions,mymessages in
(       thisactor_mgr [actors,myports,mythreads,myregions,mymessages] (act)    (* the actor itself *)
        |[myports,mythreads,myregions,mymessages]|
        (       all_myport_mgr [IPC,myports]                (* management of ports of the actor *)
                |||
                all_mythread_mgr [monitor,mythreads]        (* management of threads of the actor *)
                |||
                all_myregion_mgr [monitor,myregions]        (* management of regions of the actor *)
                |||
                all_mymessage_mgr [mymessages] ))           (* management of messages of the actor *)

endproc
```

This approach leads to a better match between the system functionalities and the specification structure : most synchronization problems are reduced to restrictions on the events offered by one process. Its main drawback is the introduction of many internal gates into the specification, splitting it into a network of many loosely coupled processes, for which it is difficult to certify a correct global behaviour. In particular, it complicates the specification of a few system calls implying synchronization between different, not directly interacting processes (e.g. the reception of a message, which implies a waiting thread and a reception port). Moreover, the specification of all interaction primitives associated with each new internal gate increases the size of the specification considerably.

### 3.3.3. The constraint-based approach

As a consequence of the shortcomings encountered in our second approach, we felt the need for a more extensional style of specification, for which the *constraint-oriented style* [17] was the obvious candidate. Our approach was to build the specification as a conjunction of constraining processes on the gate *local* (i.e. the gate supporting system calls), with one compound process for each managed CHORUS object, so that any of these processes could “see” and possibly set constraints on any system call. On the other gates of the specification however, the processes were kept interleaved, because no multiple constraint seemed necessary (Fig. 10).

In fact, this kind of total synchronization was much too strong for our purpose; and each process had to be specified as one effective constraining process accepting only system calls concerning its object, interleaved with a “filter” process accepting and ignoring anything else.

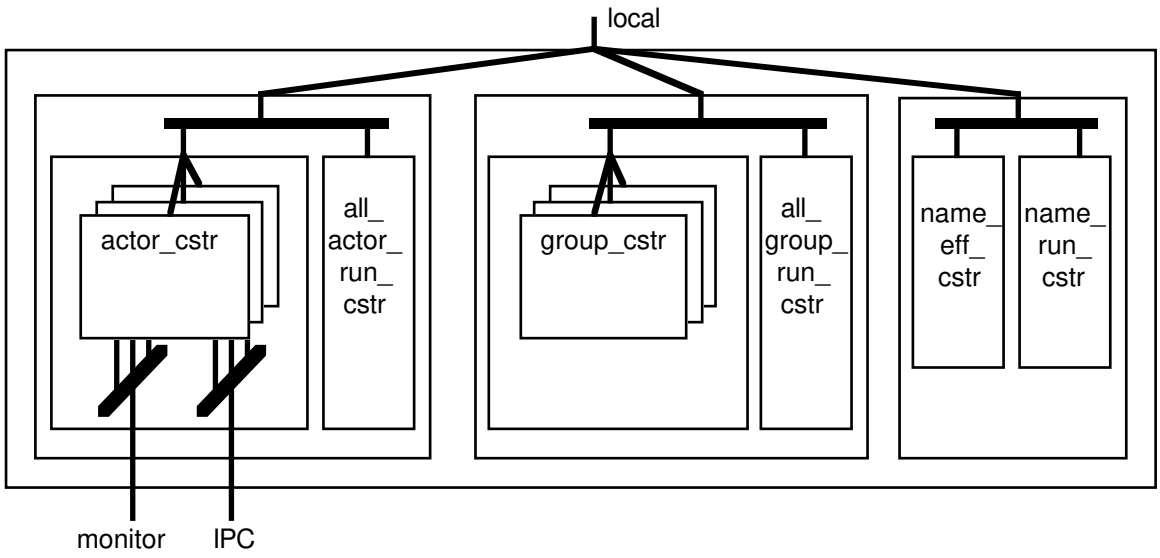


*Fig. 10. The constraint-oriented approach*

```
process local_kernel [monitor,IPC,local] : noexit :=

all_actor_cstr [monitor,IPC,local]  (* also includes constraints concerning threads, ports, etc. *)
|[local]|
all_group_cstr [local]
|[local]|
name_cstr [local]

endproc

process all_actor_cstr [monitor,IPC,local] : noexit :=

all_actor_eff_cstr [monitor,IPC,local]      (* all constraints concerning actors *)
|||
all_actor_run_cstr [local]              (* accepts and ignores anything not concerning an actor *)

endproc


process all_actor_run_cstr [local] : noexit :=

local ?p : CHORUS_primitive [not(concerns_actors(p))];          (* accept and ignore one event *)
all_actor_run_cstr [local]

endproc
```

The constraint-oriented style offers two major characteristics: (1) the specification is structured in terms of requirements instead of system components, and (2) the process structure is monolithic (i.e. there are no internal gates), which eases the inference of the overall behaviour from the behaviour of each component. However, this rigid structure is not well fitted for all kinds of descriptions. Moreover, it appears to push LOTOS to its limits : inadequacies and limitations in LOTOS expressivity make the specification of some constraints quite complex, heavy and counter-intuitive. Here is a list of the main difficulties encountered in our particular case :

- *Lack of internal communications* : the information kept by the system is sparsed among all constraining processes. As no internal gate is allowed, a problem appears when one constraint needs to know about information it does not hold. Our solution was to add 'auxiliary' parameters to external interactions, which serve only for information transfer between constraints.

- *Value negotiation* : Some tricky value and constraint manipulations are needed when an event must be offered whose value depends on a variable number of constraints. Consider, for example, a response value which must be *ERROR* if and only if at least one constraint detects an error, and *OK* otherwise : in case of success, none of the concerned constraints may force the *OK* value, because it knows nothing about the other constraints[4].

- *Dynamic aspects* : Our specification needed dynamic generation and cancelling of constraints. Creation brings no special difficulty, but cancelling a constraint means that the corresponding process must begin to accept any event, any number of times. This kind of process (the neutral element of the // operator, i.e. the ***RUN*** process of CSP [7]) corresponds to no basic LOTOS construct, and furthermore needs a new specification for each different kind of value offer scheme. Moreover (due to the "atomicity" of LOTOS gates w.r.t. synchronization), there is no easy way to add a constraint concerning only a subset of the events occuring at a given gate (cf the "filter" processes mentioned above).

All these drawbacks led to a heavy specification, where the interesting parts were lost in a maze of tricky specification detours. For this reason, we eventually decided to get back to the object-based approach, which allowed a reasonably simple and clear specification, although its underlying style was less appealing than the constraint-oriented style, at least on a theoretical basis.

Please note however that we do not pretend to have been exhaustive : There is an infinity of different specifications for a given system, even within a given style. Other approaches or even variations of the proposed approaches could possibly have led to a better specification.

[4]Let $P_1,\ldots,P_n$ be the constraints. Each $P_i$ holds a result $R_i$ (*OK* or *ERROR*). $P_1$ // … // $P_n$ must offer a value $R_{eq}$ = *OK* iff $R_1 = \ldots = R_n$ = *OK*, and *ERROR* otherwise. The solution uses an auxiliary "associative table" parameter *t* in the same offer, and a complementary constraint *P** which "knows" all $P_i$ (whereas the $P_i$ know nothing about each other). Each $P_i$ forces *t* to contain its partial result $R_i$, and P* restricts *t* to containing entries for the $P_i$ only and fixes the value of $R_{eq}$ from the value of *t* :

*$P_i$ = g ?R: result ?t: table [ t contains (i,$R_i$) ] ; …*
*P* = g ?R: result ?t: table [ domain(t) eq {1,…,n} and R eq result(t) ] ; …*

# 4. Conclusions

This paper has presented an application of the FDT LOTOS to the specification of the DOS kernel CHORUS V3. The objective was to evaluate the adequacy of LOTOS for specifying operating systems, by using it to describe the main structures of CHORUS. Several encountered specification difficulties tied to the operating system nature of CHORUS were discussed.

The final result of our work on CHORUS is a specification of more than 8000 lines, including 363 (standard and shortcut) data types and 75 processes. The behavioural description covers the higher part of the CHORUS kernel (not including scheduling and memory paging) and the network manager, down to its transport service interface.

Globally, the obtained results can be considered as positive. As anticipated, LOTOS can be used for such kind of specifications, even if some problems still have to be more carefully analyzed. The main difficulty found resides in the executive functionalities of CHORUS, which are immediate consequences of its OS nature.

This study shows the need for some clearly defined operating system architectural concepts, with an associated specification methodology. In a well-structured architecture like the OSI Reference Model [8], the limits of each functional entity and the interaction channels between them are clearly defined; the specifier only needs to formalize these interactions and then try to express relations between them using LOTOS constructs. In OS world however, those architectural concepts are to be defined and agreed upon, before any subsequent behaviour specification can begin. This is the direction that our future work should follow.

Other obstacles, more specific to LOTOS, were also encountered. Besides the stylistic issues and constraint-oriented limitations discussed in section 3.3.3, we will also mention the lack of conciseness of abstract data types and the weakness of specification structuring mechanisms. The latter are well known and already heavily debated questions (see for example [15], [4]), which we will not comment any further. Note however that the “verbosity” of ADT led us to use non-standard shortcuts (as proposed in [15]), thereby precluding the application of software tools to the obtained specification : only the syntax could be checked and debugged.

# 5. Acknowledgements

The presented specification is a follow-up of previous work done at the University of Liège by Guy Leduc, André Danthine and Michel Ghys. A lot of information and some pictures concerning CHORUS come from documents from CHORUS Systèmes, France. Special thanks go to CHORUS developers, for providing me with all this information and for some constructive discussions.

# References


[1] Ajubi I. et al
"Formal Description of the OSI Session layer : Introduction", in
"The Formal Description Technique LOTOS : results of the ESPRIT/SEDOS project"
*North Holland, 1989, pp. 89-96.*

[2] Banino J.-S.
"Architecture of the CHORUS distributed system"
*Distributed Computing Systems, Academic Press London, 1983, pp. 251-264.*

[3] Bolognesi T., Brinksma E.
“Introduction to the ISO specification language LOTOS”
*Computer Networks and ISDN Systems,n° 14, 1987, pp25-59.*

[4] Brinksma E.
"On the design of Extended LOTOS"
*PhD Thesis, Universiteit Twente, Nov 88, 237 p.*

[5] Ehrig H., Mahr B.
"Fundamentals of Algebraic Specification 1"
*EATCS Monographs on Computer Science, Vol. 6, Springer Verlag, 1985, 321 p.*

[6] Guillemont M.
"The CHORUS distributed operating system: design and implementation"
*International symposium on local computer networks, Florence, Italy, April 82, pp. 207-233.*

[7] Hoare C.A.R.
"Communicating Sequential Processes"
*Prentice-Hall International Series in Computer Science, 1985, 255 p.*

[8] International Standardization Organization
"Information Processing Systems - Open Systems Interconnection - Basic Reference Model"
*ISO/TC 97/SC 21 IS 8072, 1984.*

[9] International Organization for Standardization
"Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour"
*ISO/TC 97/SC 21 IS 8807, 1989.*

[10] Leduc G., Danthine A., Ghys M.
"An overview of the LOTOS specification of CHORUS V2"
*SART 88/20/13, University of Liège, Systèmes et Automatique, July 88, 39 p.*

[11] Milner R.
"Communication and Concurrency"
*Prentice-Hall International Series in Computer Science, 1989, 260 p.*

[12] Pecheur Ch., Danthine A., Leduc G.
"Modeling of the Transport Station of CHORUS V2"
*SART 88/22/13, University of Liège, Systèmes et Automatique, July 88, 24 p.*

[13] Pecheur Ch.
"An overview of the LOTOS specification of CHORUS V3"
*SART 89/03/13, University of Liège, Systèmes et Automatique, May 89, 76 p.*

[14] Rozier M. et al
"Overview of the CHORUS distributed operating system"
*CHORUS Technical Report TR-88-7.8, February 89, 50 p.*

[15] Scollo G.
"Some Facilities for Concise Data Types in LOTOS"
*Esprit P410 SEDOS/C1/WP/13/T, 17 March 86, 52 p.*

[16] Scollo G.
"Formal Description of the OSI Session layer : the Transport Service", in
"The Formal Description Technique LOTOS : results of the ESPRIT/SEDOS project"
*North Holland, 1989, pp. 97-116.*

[17] Scollo G., Van Sinderen M.
"On the architectural design of the formal specification of the session standards in LOTOS"
*6th Symposium on Protocol Specification, Testing and Implementation, Montreal, North Holland, 1986, 11 p.*

[18] Turner K.
"An architectural semantics for LOTOS"

*7th Symposium on Protocol Specification, Testing and Implementation, Zurich, North Holland, 1987, 14 p.*

[19] Vissers A. et al
"On the use of Specification Styles in the Design of Distributed Systems"
*new version of "Architecture and Specification Style in Formal Descriptions of Distributed Systems", Protocol Specification, Testing and Verification VIII, Atlantic City, June 1988, 32 p.*

[20] Pecheur Ch.
"A LOTOS specification of CHORUS V3"
*SART 89/04/13, University of Liège, Systèmes et Automatique, May 89, 156 p.*