On the Use of Pre-Defined Implementation Constructs in Distributed Systems Design

Luís Ferreira Pires, Marten van Sinderen, Chris A. Vissers

University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands

Abstract

This paper introduces the concept of pre-defined implementation constructs (PDICs) as a method for distributed systems design. It discusses how PDICs should be selected and used, and shows the applicability of this method on basis of a middle size design example: the implementation of a sliding window protocol.

1 Introduction

In the design of information systems normally a large design gap has to be bridged between the initial formulation of the user requirements and the final implementation and realization of the system. Whereas the user requirements are expressed in high level definitions of system functions and facilities, the implementation and realization are expressed in terms of elementary software or hardware components. In open distributed systems, user requirements have to be expressed at a very high level of abstraction and in an as much as possible implementation independent way in order to cope with the complexity of these systems and to fulfil the requirement of openness. In this case the bridging of the gap between user requirements and implementation is even more complex, requires a broad range of skills, is time consuming, error prone and costly.

Over the years we can observe several approaches that aim at controlling the complexity of bridging this design gap. Examples of these approaches are the introduction of higher level programming languages, the introduction of design methodologies, the definition of reference models, and the introduction of special (design) languages such as formal description techniques.

This paper focuses on one particular approach which is based on the use of so-called, *pre-defined implementation constructs* (PDICs). A PDIC is a general purpose specification construct with a well defined, possibly parameterized, high level functionality whose correct implementation and realization in one or more target environments (e.g. programming language and operating system facilities) is available. In a design specification a PDIC is represented by its (e.g. formal) specification construct. This specification construct is not further refined in successive design steps, but is kept as a final implementation component and directly replaced by its realization in the target environment during the realization of the system.

The objective of using PDICs in distributed systems design is to transform as soon as possible after requirements capturing, i.e. in the early phase of the design trajectory, an architectural specification into an implementation specification that is composed completely or largely from combinations of these constructs. Once such a composition is obtained, the remaining part except the last step of the design trajectory can be skipped, since the composition of PDICs can be simply replaced by the composition of their realizations. Obviously this requires that a sufficiently rich library of PDICs is made available to the designer.

The use of PDICs potentially entail important benefits for open distributed systems design since it may narrow the design gap as well as reduce its complexity:

- It separates design disciplines. The system designer needs to focus only on the first phase of the design trajectory, i.e. the architectural phase. This phase requires the discipline to consider a system as a composition of abstractly defined PDICs. The implementation and realization of PDICs is done separately. This work requires the discipline to consider the more concrete aspects of PDICs such as their programming, their use of operating system commands, and their choice of supporting hardware. It can, by the way, again be based on using PDICs, but of lower abstraction levels.
- It reduces verification, validation, and test efforts. Since PDICs can be correctly implemented once and

0-8186-2755-7/92 $03.00 © 1992 IEEE

for all, the effort of preserving correctness can be limited to verifying, validating or testing the composition of PDICs against the architectural specification.

- It shortens the design life cycle. The design trajectory can be bridged much faster and products can be brought out much earlier than when traditional design methods are used.

Similar approaches can be observed in a number of large scale European (ESPRIT and RACE) projects that are developing a set of distributed high level capabilities, often called "implementation platforms". Such a set of capabilities can be considered as a framework for the development of PDICs. Representative projects are ANSA, ISA, RIBA, HARNESS, COMMANDOS and ROSA. The way in which our approach differs from these projects is in its use of formal methods and in its embedding in a design methodology.

This paper is further structured as follows: section 2 discusses the consequences of using PDICs in distributed systems design, section 3 presents a small design example, in which PDICs are identified and used in the implementation of a protocol entity of the Sliding Window Protocol, and section 4 draws some conclusions.

2 PDICs in the Implementation Process

Before PDICs can be used routinely in the design process a number of difficult problems have to be surmounted. We will elaborate on a few of them.

Definition of PDICs. The first major problem is how to identify and define those high level system functions that are candidates for becoming PDICs. This requires designers that have sufficient experience with the design of large systems and thus are capable of identifying those functions that are frequently used. Although one could state that all systems differ in functionality, it is also obvious that at the end their implementation must be based on general purpose implementation elements. This implies that somewhere along the trajectory from architecture to final implementation, PDICs can be identified. If not, systems cannot be built economically. Furthermore many distributed systems show large similarities in the type of problems that have to be solved, in the type of facilities that have to be provided, and in the internal structure that is given to them. In fact the OSI reference model, which is based on the centralisation of invariants, is an example that demonstrates that common solutions, in the form of service definitions and protocol specifications, can be provided for a large variety of applications. Rather than discussing a recipe how PDICs can be identified and defined, we discuss some criteria that PDICs must satisfy.

- *High level functionality*: PDICs, generally, should provide a high level of functionality and not be too simple. High level PDICs narrow the design gap, simple PDICs complicate the design process since they do not really guide us where to aim at during intermediate design steps. On the other hand one can expect that not every intermediate design can be composed fully from high level PDICs. This implies that we need PDICs ranging from high to low level functionality that can be used at various abstraction levels in the design process.
- *Generality*: PDICs must be general purpose so that they can be used in many implementations. The approach in the definition and use of PDICs should be to avoid unnecessary divergencies and favour general purpose building bricks over slightly more optimal special purpose building bricks. It is evident that we cannot define a PDIC for any implementation requirement, even if we would restrict ourselves to specific systems. This problem can be solved if we allow PDICs to be parameterized such that each PDIC represents a collection of non-parameterized constructs. Designers can then use parameter settings to tune a PDIC to a certain behaviour.
- *Mutuality*: the argumentation used above also demands that PDICs should be (functionally) complementary to each other in order to be able to build different compositions of PDICs. This implies that there should be a sufficiently rich library of PDICs available to compose any implementation specification from PDICs.
- *Composition*: PDICs must be "easy to combine". It is important to consider how these constructs can be combined. During the early phase of the design process the PDICs are used as abstractly defined functions. This implies that care has to given to the definition of the abstract interaction facilities of PDICs. In the realization these interaction facilities are replaced by real interfaces. This may imply that each PDIC must be equipped with a variety of real interfaces to allow effective interconnection with other PDICs. A complication that we can encounter, for example, is the use of multi-way synchronous interaction at the architectural level and the use of binary asynchronous communication at realization level.
- *Correctness*: PDICs should be correctly interpreted so that their use in the design process should be free of ambiguity, and their implementation should provide only intended behaviour. This implies that PDICs should be unambiguously specified, preferably by using a formal design language.

Application of PDICs. The second major problem, directly related to the previous one, is how to transform an architectural specification into an implementation specification that is composed from general purpose PDICs. This question indeed may be difficult to answer since an architecture is user-oriented, can be quite specific, can be implemented in an arbitrary number of different ways, and may not show any indication of how PDICs can be applied. The only indication we can use is in principle the externally observable behaviour given by the architectural specification. The approaches we can use here are in fact offered by modern design methods. We will mention two of them:

- *Specification styles:* specification styles allow us to express design constraints in the structure of a specification [Vis88]. A constraint oriented style, for example, allows us to structure the externally observable behaviour in a composition of constraints. Each constraint may model a different system function. This composition can suggest an internal stucture which may be obtained by transforming the constraint oriented specification into a resource oriented specification. This process may be repeated iteratively since resources themselves can be specified in a constraint oriented style. Resources can be chosen as PDICs or compositions of PDICs.
- *Separation of concerns:* in separation of concerns a system with a complex functional behaviour is decomposed into a (set of) relatively simple component(s) that cope(s) with a specific design concern and a relatively complex component that embodies the yet unresolved design problems. The latter is again decomposed by repeating the previous decomposition step until sufficiently simple components are found. Obviously this approach can be used in combination with PDICs by trying PDICs as candidates for the relatively simple components. Care should be taken that the components and their interfaces, that result from separation of concerns, are defined as much as possible in a general purpose way. This in fact gives us a way how to find and define PDICs and what design concerns should be embodied in a PDIC.

3 Illustration of the PDICs approach: the Sliding Window Protocol

This section shortly discusses a design example of a *Sliding Window Protocol* (SWP) using the PDICs approach. We introduce the architectural specification of the SWP, without giving much attention to the initial design steps that led to it. A part of this specification, representing the transmitter side, is analysed for the purpose of identifying the type of PDICs that could be used in an implementation specification. Based on this analysis, a small library of PDICs is presented, and a final implementation specification in terms of PDICs is constructed.

The formal description technique LOTOS [IS8807] is used as the specification language. The realization environments for the PDICs consisted of Ada and C-Ex[1] on a Unix workstation and is not further discussed here.

3.1 Architectural Specification

We begin the design process by defining a Data Transfer (DT) Service, which is "reliable" by preserving the correctness and order of transmitted data units. We simplify this example by considering only two *Service Access Points* (SAPs), one for sending data and another for receiving data.

To achieve geographical distribution and separation of concerns we decompose th e DT Service into two relatively simple protocol entities, a *Transmitter Protocol Entity* (TPE) and a *Receiver Protocol Entity* (RPE), and a connectionless communication network. TPE and RPE constitute the SWP that implements "enhancement of reliability" as its major design concern. It includes error detection, data acknowledgment, data retransmission and reordering mechanisms. The communication network implements the remaining design concerns, namely the unreliable bridging of distance. It may corrupt or loose data, or deliver it out-of-order. Figure 1 depicts the decomposition, showing the components and the gates that appear in the formal specifications.

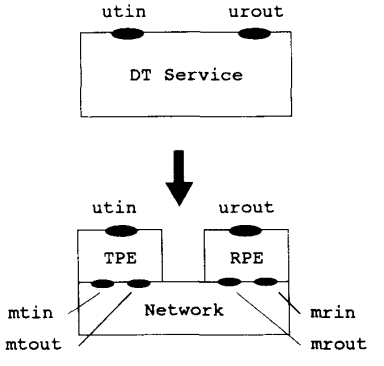

Figure 1 : Decomposition of the DT Service

For the sake of saving space we limit ourselves further to the design of the TPE.

[1] *C-Ex* (or *C-Extended*) is a C based process synchronization kernel developed at the Ecole Polytechnique Federale de Lausanne (EPFL). See [Con90] for the Modula-2 version (*M-Ex*)

The TPE interacts with a DT service user through *DT_Data_Request* primitives at gate `utin`, and with the medium through *M_Data_Request* and *M_Data_Indication* primitives at gates `mtout` and `mtin` respectively.

When a DT_Data_Request containing a *Service Data Unit* (SDU) occurs, the TPE generates a *Data Protocol Data Unit* (D-PDU) and assigns the next available PDU identifier to it. We assume that the TPE may have a buffer in which SDUs can be stored before the mapping on a D-PDU takes place. In case this buffer is full, backpressure will be exerted by the TPE on the transmitter service user. Each PDU gets a checksum to allow error detection by the RPE.

Every time a D-PDU is sent, a timeout is started and the D-PDU is retained for possible retransmission. This D-PDU is discarded in case the TPE receives an *Acknowledgement PDU* (A-PDU) for this D-PDU. In case no acknowledgement arrives in time, the D-PDU is retransmitted and a new timeout is started.

The initial TPE LOTOS specification has been written using the constraint-oriented style as defined in [Vis88]. The structure of this specification as far as it concerns the handling of correct PDUs is depicted in Figure 2. The specification also includes an (interleaved) process for handling corrupted PDUs, which shall not be discussed here.

Figure 2 indicates for each option of behaviour (`AcceptData`, `SendData`, `RecFromM` and internal event `i`, which models the occurrence of a timeout for retransmission) which parameters of the global process are used and modified. Variable `tq` represents a transmission queue, `rtq` represents a retransmission queue, `lu` represents the last unacknowledged PDU, `sid` represents the next PDU identifier to be used, and `ws` represents the transmission window size. We will analyse each option of behaviour separately below.

Process `AcceptData`. This process represents the acceptance of SDUs at the DT service boundary (gate `utin`) and their storage in `tq`. An implementation construct that can be used for this is a queue that accepts data as long as it is not full.

Process `SendData`. This process represents the transmission of D-PDUs made from SDUs taken from `tq`. This process uses `ws` and `rtq` to determine whether a PDU can be sent through the communication network (gate `mtout`). In case a PDU is sent, it is stored in `rtq`, the corresponding SDU is removed from `tq`, and `sid` is updated.

Making PDUs and updating `sid` can be implemented as the application of a function on input data. PDUs that are sent require temporary storage in `rtq`, which can again be provided by a queue. Relating the production of a PDU, storing this PDU in `rtq` and sending it through the network can be implemented with a construct that provides a scheduling function.

Process `RecFromM`. This process represents the receipt of uncorrupted A-PDUs from the communication network (gate `mtin`). This process uses `rtq` and `lu` to determine whether the received PDU is a correct acknowledgement. Acknowledged PDUs are removed from `rtq` and `lu` is updated by this process.

From the above we conclude that mutual exclusive access to `rtq` is necessary to ensure consistency, and therefore should be provided by the construct that is used for its implementation. The verification of A-PDUs against `rtq` can be performed using a choice construct that can decide between two alternatives (correct acknowledgement or not). Relating the receipt of correct A-PDUs and the updatings of `rtq` can again be done by a scheduler component.

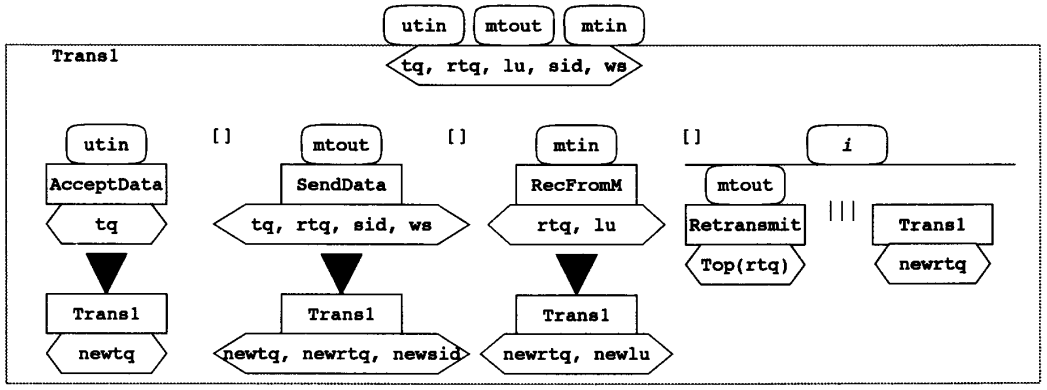

Figure 2 : Constraint Oriented TPE Specification Structure

Retransmission Timeout (*i*). This option of behaviour represents the occurrence of a timeout for acknowledgement, which shall cause a D-PDU from `rtq` to be retransmitted through the communication network (gate `mtout`) in parallel with an instance of `Transl` in which `rtq` is updated.

An implementation construct that can be identified from this functionality is a pool of timers, one timer for each unacknowledged PDU. Furthermore, it follows that `rtq` can be implemented by a circular queue if the timeout values are expected to be the same for all PDUs. Relating timeout handling, manipulation of `rtq` and retransmission of PDUs can be performed by a scheduler component. The implementation of the internal event *i* as a timer affects the implementation of process `SendData` and `RecFromM` since it must take care of setting and resetting the timer respectively.

This analysis shows that we can identify a number of implementation components. These components could be related either to parts of the options of behaviour or constraints in the architectural specification, or to variables that have to be shared between these parts of behaviour (all of which correspond with a parameter of the specification). The properties of the latter components are determined by the parts of behaviour that refer to the variables embodied by the components. For example, the properties of the queue that embodies `rtq` is determined in part by `SendData`, `RecFromM` and `Retransmit` (following a timeout represented by *i*).

In order to be useful as PDICs, all of these components should be defined such that the criteria mentioned in section 2 are satisfied.

3.2 A Small Example Library of PDICs

This section introduces entries for a library of PDICs, based on the analysis of the TPE's constraint oriented specification. PDICs defined in this section are of a quite general nature; similar constructs can be found in other work, such as in [Mil80]. All the PDICs presented in this section have been specified in LOTOS and implemented in Ada ([Juf91]) and C-Ex ([Wes91]).

The `Do_f` Construct. The `Do_f` receives as an input a data value and delivers the result of the application of a specific function on this data value as an output. We consider here a `Do_f` which models the processing delay explicitly, operates continuously and accepts any input values, reacting only to values that are in the domain of `f`. This construct can be used to implement the checksum verification that acts as a filter for correct A-PDUs and to implement the production of D-PDUs. Figure 3 depicts the `Do_f` LOTOS specification and its symbol.

```
process Do_f [inp, out] :noexit :=
inp ?x :Element ;
( [x IsIn Domain_f] -> out !f (x) ; Do_f [inp, out]
 [] [x NotIn Domain_f] -> Do_f [inp, out] )
endproc (* Do_f *)
```

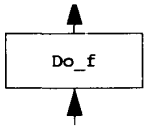

Figure 3 : `Do_f` Specification and Symbol

The `Choice_p` Construct. This construct uses input data and a predicate on this data to decide between two alternative next interactions. We consider here a construct that executes choices based on a boolean predicate applied to the input data recursively. An instance of this construct can be used in our example to evaluate whether a received A-PDU acknowledges previously sent D-PDUs. Figure 4 presents the LOTOS specification and symbol of this construct.

```
process Choice_p [inp, out1, out2] :noexit :=
inp ?x :Element;
( [p(x)] -> out1 ; Choice_p [inp, out1, out2]
 []
 [not(p(x))] -> out2 ; Choice_p [inp, out1, out2] )
endproc (* Choice_p *)
```

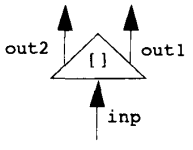

Figure 4 : `Choice_p` Specification and Symbol

The `Scheduler` Construct. This construct represents a component that receives an input data and distributes it to other processes. In the TPE example, this construct has been used to distribute information or synchronize components, since the realization environments we have used do not support multi-way synchronization directly. Figure 5 presents the LOTOS specification and symbol of this construct.

```
process Scheduler [inp, out1, out2] :noexit :=
inp ?x :Element; out1 !x; out2 !x ;
Scheduler [inp, out1, out2]
endproc (* Scheduler *)
```

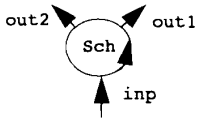

Figure 5 : `Scheduler` Specification and Symbol

The `Timer` Construct. The timer PDIC is very important since it allows to isolate timing from control flow. Since timers cannot be completely represented in LOTOS, we represent the elapsed time value as a formal parameter of the timer and keep it as such until the timer is implemented. This construct can be used in the TPE example to signal a retransmission timeout corresponding to a previously sent D-PDU. A timer specification and its symbol are depicted in Figure 5 below.

```
process Timer [set, expired, reset, value] :noexit :=
set ; ( i; expired ; exit [] reset ; exit )
 >> Timer [set, expired, reset]
endproc (* Timer *)
```

Figure 6 : `Timer` *Specification and Symbol*

The `FIFOQueue` Construct. The `FIFOQueue` is used to allow buffering and orderly delivery of data. In the example we can notice that data to be (re-)transmitted is put in FIFO queues in order to be delivered in proper order. Figure 7 depicts a FIFO queue LOTOS specification and its symbol.

```
type FIFOQueue is Element
sorts
 FIFOQueue
opns
 EmptyQueue : -> FIFOQueue
 Enqueue : Element, FIFOQueue -> FIFOQueue
 Dequeue : FIFOQueue -> FIFOQueue
 Top : FIFOQueue -> Element
 NotEmpty : FIFOQueue -> FBool
eqns
 (* equations are omitted *)
endtype (* FIFOQueue *)

process FIFOQueue[inp, out] (q: FIFOQueue) :noexit :=
inp ?x: Element ; FIFOQueue[inp, out] (Enqueue(x, q))
[]
[NotEmpty(q)] ->
 out !Top(q) ; FIFOQueue[inp, out] (Dequeue(q))
endproc (* FIFOQueue *)
```

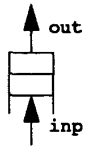

Figure 7 : `FIFOQueue` *Specification and Symbol*

3.3 Implementation Specification

We can now use specific instances of the PDICs library to transform the architectural specification of the TPE to a final implementation specification. These instances correspond to the implementation components suggested in section 3.1. The mapping between the behaviour defined in the architectural specification and the PDICs in the final implementation specification is as follows:

- Process `AcceptData` is mapped onto an instance of the `FIFOQueue` construct (to queue accepted SDUs).
- Process `SendData` is mapped onto an instance of the `Do_f` construct (to produce D-PDUs), part of the `FIFOQueue` construct (to queue sent D-PDUs which await acknowledgement) and the `Scheduler` construct (to relate the production, queueing and sending of D-PDUs and the setting of acknowledgement timers).
- Process `RecFromM` is mapped onto an instance of the `Choice_p` construct (to determine the validity of acknowledgements indicated by received A-PDUs), part of the `FIFOQueue` construct (to update queued D-PDUs which await acknowledgement) and the `Scheduler` construct (to relate the checking and updating of queued D-PDUs and the resetting of acknowledgement timers).
- The retransmission timeout (`i`) option is mapped onto a number of instances of the `Timer` construct (to signal acknowledgement timeouts), part of the `FIFOQueue` construct (to update, after each timeout, the order of queued D-PDUs which await acknowledgement) and an instance of the `Scheduler` construct (to relate expiration of timers, retransmission of D-PDUs, setting of timers and updating of queued D-PDUs). As mentioned in section 3.1, reordering of queued D-PDUs can be in this case easily implemented with a circular queue. The `FIFOQueue` construct should be modified to support this.

In the graphical specification of Figure 8 we show the specification of TPE as a composition of PDICs. This resource oriented TPE specification is expected to be weak bisimulation equivalent to the initial constraint oriented specification. The implementation specification has been validated against the initial specification using simulation tools. In addition, the interworking of the realizations of the TPE and RPE, based on their PDIC realizations in Ada and C-Ex on a Unix workstation, has been tested. It should be noted, however, that neither of these approaches constitute a rigorous proof of correctness.

4 Conclusions

This paper discussed several aspects of the definition and use of PDICs in the design of distributed systems. It seems that this use has the potential of becoming a powerful design method since it allows to produce quality products in shorter timescales. To use PDICs effectively, a number of difficult problems have to be surmounted. One of them is the development of a sufficiently rich library of PDICs.

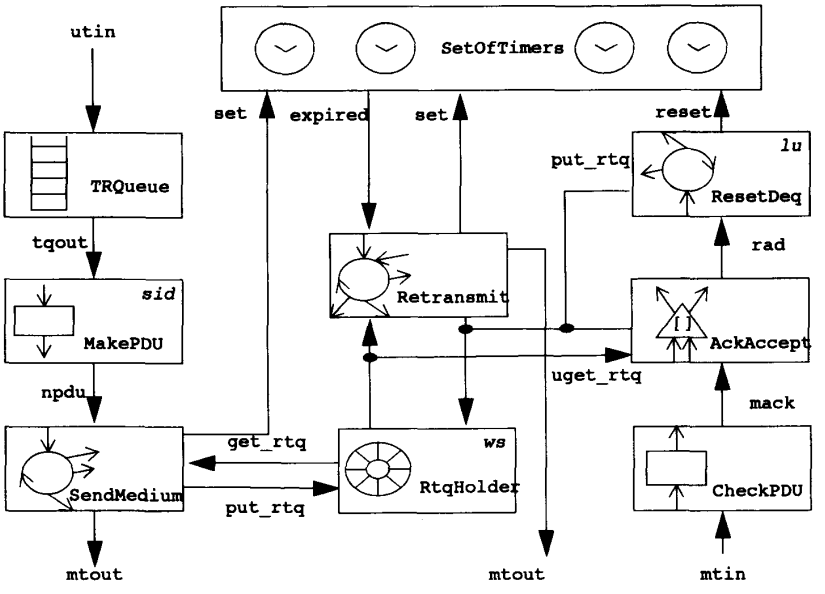

Figure 8 : Resource Oriented TPE Specification Structure (in Terms of PDICs)

Another related problem is the development of methods to quickly transform an architecture into a composition of PDICs. Specification styles appear useful methods in this respect. Further research in this area is recommended.

The PDICs approach has been illustrated with the design of a sliding window protocol. This example includes the transformation of a constraint oriented (architectural) specification to a resource oriented (implementation) specification in terms of PDICs, both expressed in LOTOS. The PDICs used in the example have been realized in Ada and C-Ex on a Unix workstation, and two realizations of the sliding window protocol could thus be automatically derived from the final implementation specification.

5 Acknowledgements

We would like to thank Marcel Juffermans and Sicke Westerdijk for their contributions to the realization of the sliding window protocol .

This work has been partly funded by the Commission of the European Communities in the ESPRIT II Lotosphere project (project 2304).

6 References

[ANSA89] Architecture Projects Management Limited. *ANSA - Advanced Networked Systems Architecture Reference Manual (Volumes A, B and C)*. Release 01.01. July, 1989.

[Wes91] Westerdijk, S. *Implementation of LOTOS specifications in C++-ex. Master's Thesis.* University of Twente, Enschede, the Netherlands. August, 1991

[Juf91] Juffermans, M.J. *Implementation of LOTOS specifications in Ada. Master's Thesis.* University of Twente, Enschede, the Netherlands. June, 1991

[Mil80] Milner, R. *A Calculus for Communicating Systems - LNCS 92.* Springer-Verlag. 1980.

[Vis88] C.A. Vissers, G. Scollo, M. van Sinderen. Architecture and Specification Style in Formal Descriptions of Distributed Systems. In: S. Aggarwal and K. Sabnani (eds.). *Protocol Specification, Testing, and Verification, VIII.* North-Holland, June, 1988. pages 189-204.

[IS8807] ISO/IEC. *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* IS 8807. ISO/IEC. 1988.

[Con90] Conti. G. *Methodologie d'Implantation des Protocoles de Communication. PhD Thesis* Lausanne. Lausanne. 1990.