# Protocol Design and Implementation Using Formal Methods[1]

Marten van Sinderen, Luís Ferreira Pires, Chris A. Vissers
Tele-Informatics and Open Systems Group
University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
sinderen@cs.utwente.nl
pires@cs.utwente.nl
vissers@cs.utwente.nl

**Abstract**

This paper reports on a number of formal methods that support correct protocol design and implementation. These methods are placed in the framework of a design methodology for distributed systems that was studied and developed within the ESPRIT II Lotosphere project (2304). The paper focuses on design methods for synthesizing protocols by successive application of correctness-preserving LOTOS transformations. This transformational approach is described in some detail and is illustrated with a protocol design example. The paper concludes with some suggestions for relating design methods to milestones in the protocol design and implementation processes.

## 1 Introduction

The ESPRIT II Lotosphere project (2304) focused on a methodology for the rapid and correct design and implementation of distributed systems, in particular services and protocols, that should be suitable for industrial application. Correct design and implementation prompts the use of formal languages that support unambiguous and concise representation of designs. The standard Formal Description Technique (FDT) LOTOS [ISO89a] has been adopted by the Lotosphere project because it fulfils the requirements for precision and conciseness in the representation of distributed system design concepts, such as interactions and system structure. Industrial applicability requires that tools supporting these design methods are made available. Design support tools produced in Lotosphere have been integrated in a tool environment called LITE (*Lotosphere Integrated Tool Environment*, [Eij91]). By adding design methods and tools, the original gravity of LOTOS as a specification language was gradually turned into a design language.

This paper gives an overview of some design methods that have been developed in the Lotosphere project, focusing on a transformational approach to synthesize protocols from service specifications. This paper is further structured as follows: Section 2 presents an overview of the Lotosphere design methodology and its relevance to protocol design and implementation, Section 3 presents some transformations that can be used in protocol design, and Section 4 presents a concrete protocol design example to illustrate the transformational approach. Conclusions are drawn in Section 5.

1. This work has been partly supported the CEC research programme ESPRIT II (Lotosphere, ref.: 2304).

## 2 The Lotosphere Design Methodology

The Lotosphere design methodology adopts the usual view of a *design trajectory* as a sequence of design steps in which first the user requirements are formulated and then successively refined until the design of a real system is obtained. It further adapts this design trajectory on basis of notions such as rapid cycling using key functional elements, tree search, reference implementations, and the distinction between functional and non-functional properties of the system ([FV90], [Bog89]).

The methodology strives for using LOTOS as long as possible throughout the design trajectory. This enables notions of implementation to be formalized, and design methods to be implemented in design support tools, all based on LOTOS. Figure 1 depicts a simplified view of the Lotosphere design trajectory. Initially the user requirements are formally defined in terms of the observable behaviour of the system, such that only properties which are relevant to the system users are expressed. Successive intermediate designs represent design decisions concerning for example internal structure, data and behaviour refinements, and the mapping onto software and hardware structures.

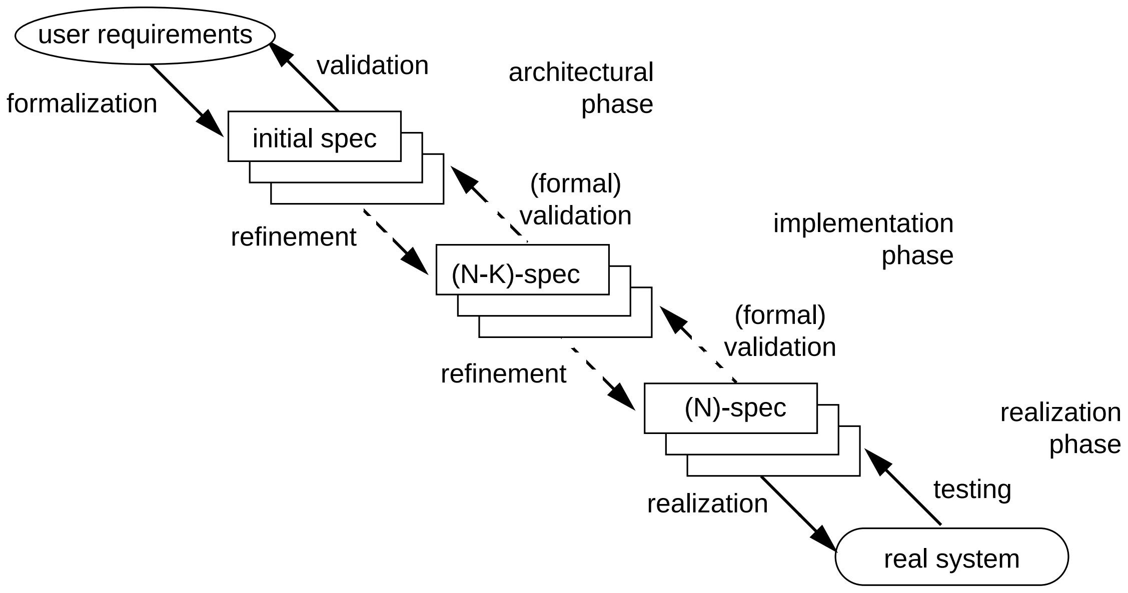


Figure 1: Design Trajectory

In protocol design, the definition of the user requirements corresponds to a *service* specification, which is an integrated representation of the observable behaviour of a distributed system as can be experienced by the service users at *service access points*. The service abstracts from all possible internal structure that can be given to the system. The service is used as the starting point for protocol design in which this internal structure is gradually decided and defined.

Protocol design is usually carried out in two phases. In the first phase the functions of the protocol are defined in an as much as possible *implementation independent* way, but such that proper interworking of peer protocol entities is guaranteed. The resulting design supports the definition of *open systems*. Therefore it is usually produced (next to service design) by a (pre-) standardization body, and published as an international standard or recommendation. The resulting design we call the *protocol design*, or simply the *protocol*. The next step in protocol design is its *implementation dependent* design. During this step a protocol design is refined until a proper composition of (hardware and software) components is obtained that can be more easily mapped onto a real system. Such a composition we call the *protocol implementation*. Protocol designs and implementations must be correct with respect to the service, i.e. they both must implement the service.

LOTOS is a broad spectrum design language allowing it to be used not only as a specification language for services and protocols (for which it was originally developed), but also as a language to represent intermediate protocol implementations. Formally defined LOTOS-to-LOTOS transformations support the design steps from the service to a protocol design and protocol implementation. These transformations can be repeated until we can compile a LOTOS specification directly or transform it by hand into code of a target implementation language (e.g. ADA or C).

The correctness of the real system can be checked by validating the user requirements and each refinement throughout the design trajectory. In addition, the real system can be tested for conformance against intermediate specifications.

The purpose of the next sub-sections is to present a taxonomy of design methods that support protocol design and implementation. Section 2.1 discusses the architectural and language considerations involved in a design step and defines the concept of design method, Section 2.2 presents a catalogue of LOTOS transformations to be used throughout a design process, Section 2.3 addresses protocol design and Section 2.4 tackles protocol implementation.

## 2.1 Architectural versus Language Considerations

*Architectural concepts* represent abstractions of aspects of technical objects of concern. In service and protocol design, for example, generic architectural concepts are service, service primitives, service access points, protocol, protocol entities, etc. A *specification language* serves the purpose of *representing* these architectural concepts. In case elements of a specification language (e.g. syntax and semantics elements) are derived from generic architectural concepts related to the technical area of concern, it may result in a general purpose specification language.

An abstraction of a technical object of concern is called a *design*. A *specification* is a representation of a design using a specification language. This representation is necessary in the design process as a vehicle for analysis, communication, and manipulation, since the technical object itself is not necessarily available. This means that a distinction must be made between a design and its specification. This distinction is depicted in Figure 2.

Design decisions taken during a design step must refine a design through the manipulation of architectural concepts. Since we can actually only manipulate specifications rather than the design itself, a design step is characterized by an *architectural transformation*, i.e. the conceptual modification of a design based on design objectives and including design decisions, and a *language transformation*, i.e. the manipulation of the specification of the design. There is not necessarily a one-to-one relationship between an architectural transformation and a language transformation. For example, in the design example presented in Section 4 we apply several language transformations to perform one architectural transformation.

We define the term *architectural semantics* as the relationship between architectural concepts and their possible representations in a specification language. More on architectural semantics for LOTOS can be found in [Tur87].

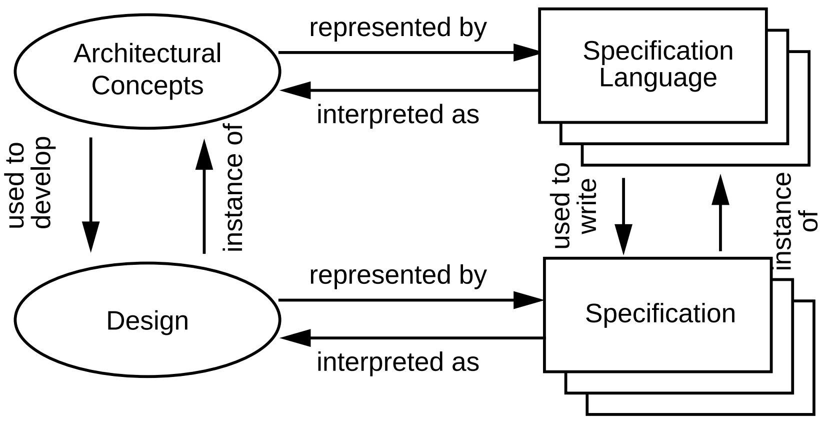


Figure 2: Relationship between Architectural Concepts and Specification Language

We define a *design method* as a combination of language transformations used to perform an architectural transformation.

## 2.2 Catalogue of LOTOS Transformations

Considering protocol design and implementation using LOTOS, several architectural transformations address problems that can be generalized and formalized in terms of general purpose LOTOS-to-LOTOS transformations. The Lotosphere project produced a catalogue of these transformations; Table 1 presents a selection of the most prominent ones ([Bol91]). Some of these transformations have been implemented in tools of LITE. These are indicated with an * in Table 1. An overview of these tools can be found in [Eij91].

Table 1: Some LOTOS-to-LOTOS Transformations

| **name** | **solution** | **tool support** |
|---|---|---|
| resolution of non-determinism | no | no |
| bi-partition of functionality | yes | yes* |
| making states explicit | yes | yes* |
| making parallelism explicit | partial | no |
| rearrangement of interaction points | yes | yes* |
| regrouping of parallel processes | yes | yes* |
| from full to basic LOTOS | yes | yes* |
| explicit dynamic process generation | yes | no |
| from LOTOS contexts to transducers | yes | yes |
| multi-way to two-way synchronization | yes | no |

In each transformation, some properties of an input specification are preserved and some others are modified in an output specification. In case the relationship between the input and output specifications can be formalized, we call it a *correctness preserving transformation* (CPT).

Since some or all semantics of the input specification is maintained in the output specification, designers must usually manipulate syntax elements in order to achieve a language transformation. An example is the replacement of one process in an input specification by multiple processes in the output specification in order to represent a structural refinement. Depending on the preserved properties of a transformation, different formally defined notions of correctness can be used. Some of these notions and associated laws can be found in [ISO89a] and [Mil89]. In this paper we make use of two of these notions: *observation equivalence* (≈), also called *weak bisimulation equivalence,* and *strong bisimulation equivalence* (~).

Some language transformations have solutions in terms of algorithms that, when applied to an input specification, produce a correct output specification. These algorithms can be fully or partly automated. Sections 3.1 and 3.2 present examples of algorithms that can be fully automated. Algorithms that can be only partly automated work interactively with the designer, by alternating between computation and request for more (design) information. Section 3.3 presents an example of an interactive algorithm.

## 2.3 Protocol Design

In protocol design we assume that a formally defined initial service specification is available as an input specification and that we can focus on transforming this into a protocol specification. The iterative design strategy, based on repeated decomposition, can be used to achieve a final protocol specification. The decomposition process starts from an (N)-service specification and produces an (N)-protocol specification that is structured in terms of (N)-protocol entities and an underlying (N-1)-service. A protocol is a specific decomposition of the service, i.e. its specification includes specific design choices in the functions of the protocol entities and the underlying service. The underlying (N-1)-service is again decomposed in the same way as the (N)-service. This process is repeated until an underlying service is obtained that can be directly realized by an implementation component which matches the required behaviour. Figure 3 depicts (layered) protocol design.

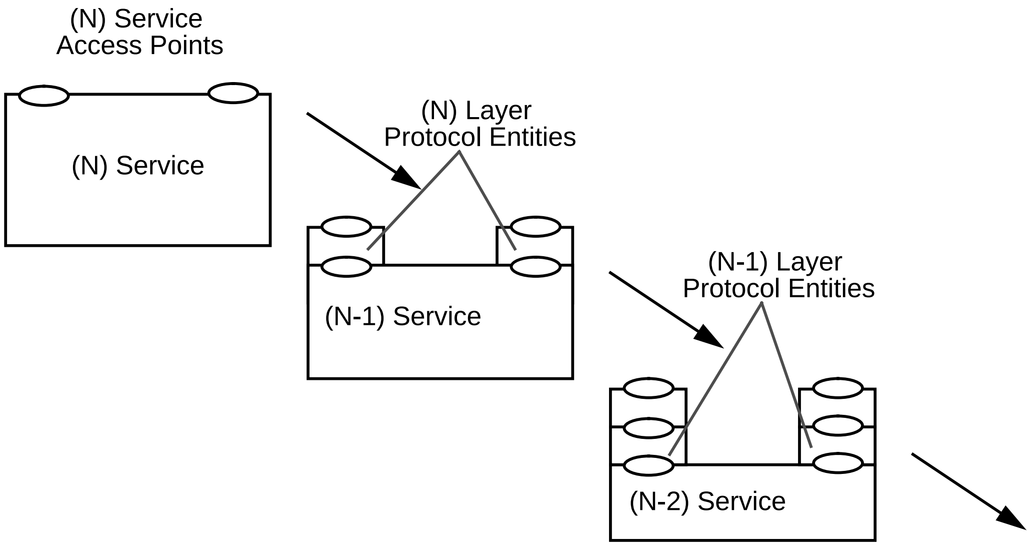


Figure 3: Layered Protocol Design

A service and a corresponding protocol are therefore alternative representations of the same distributed system behaviour, viewed from different perspectives. The service is an integrated representation of the system and is best specified in terms of a set of separate constraints on the occurrence of observable interactions (in OSI terminology *service primitives*). These constraints can be structured in terms of *local constraints*, i.e. constraints related to behaviour local to service access points, and *remote* or *end-to-end constraints*, i.e. constraints related to behaviour between different service access points. This form of representation is called *constraint-oriented style*.

A protocol is a distributed representation of a system, i.e. it represents the system in terms of a composition of objects or resources. Here a *resource-oriented style* is the most appropriate form for its specification. In a resource-oriented specification, the structure of the design is represented by a composition of processes, where a process represents an object or resource. The internal communication between resources is hidden from the environment of the system.

The protocol entities implement the distribution of the service. Therefore the local constraints of the service can reappear in the definition of the protocol entities, allowing re-use of earlier defined elements of specification and facilitating validation. The remote constraints of the service can be decomposed over protocol entities and an underlying service. Specification styles are discussed in detail in [VSS88]. Figure 4 depicts the use of specification styles in the formulation of service and protocol specifications.

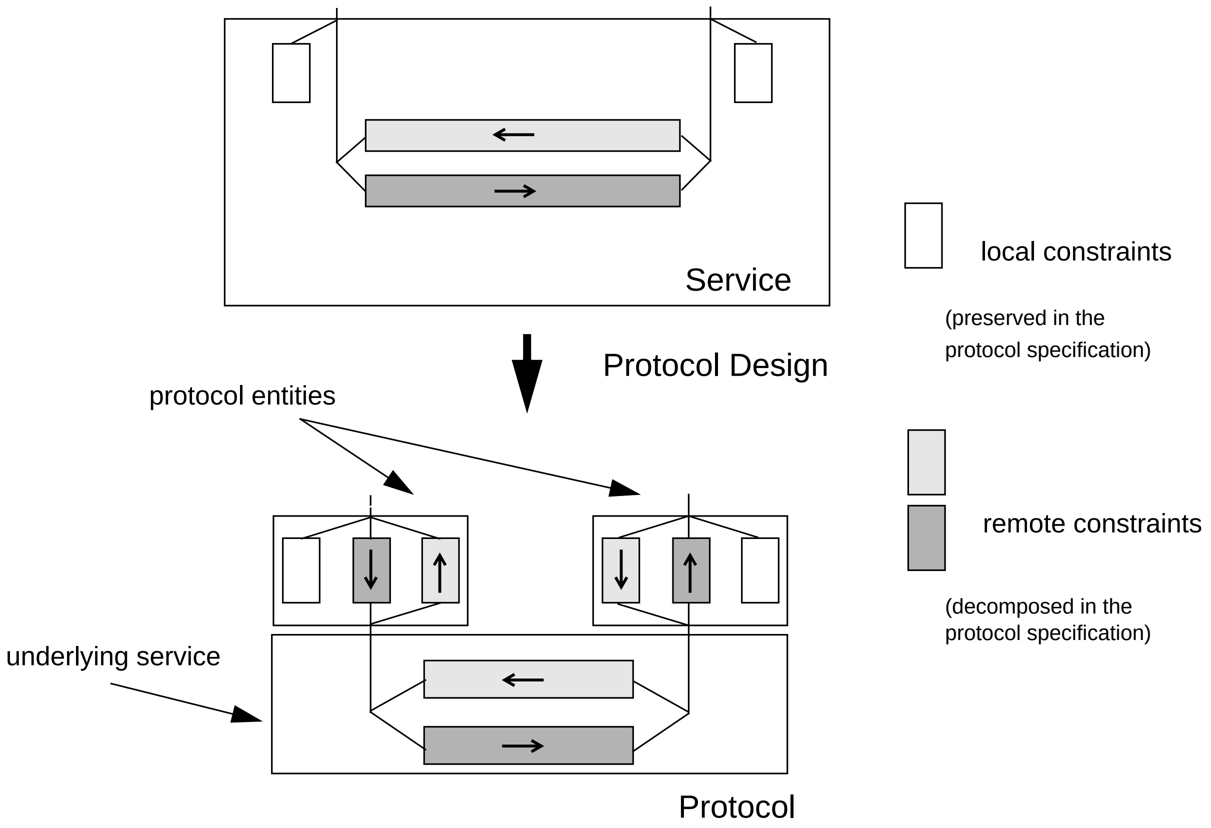


Figure 4: Decomposition of a Service into a Protocol using Specification Styles

### 2.3.1 Validation

One approach towards protocol design consists of designing a protocol using heuristics and using the service specification as a functional requirement, and validating the protocol afterwards against the service by analytical verification or by testing.

A verification method described in [V+91] assumes that the service and protocol specifications are structured in constraint-oriented and resource-oriented specification styles respectively. This choice of styles allows verification in a modular way. The method is exemplified by the verification of a protocol that provides a kind of question-answer service. If the underlying service comprises two independent directions of transfer, and the send and receive actions of each of the protocol entities are not related except by local constraints at the required service boundary, it turns out that modularization is possible and that verification is relatively straightforward. Experience has shown, however, that in more realistic designs verification using analytical methods is not feasible.

Testing a protocol against the service, according to the formal notion of *test equivalence* ([Abr87]), consists of defining test sequences derived from the service specification and applying them to the protocol specification. Test methods can be *engineered*, i.e. the ratio between the amount of computation necessary to achieve a certain coverage and the amount of computation to achieve complete coverage can be chosen. In most realistic designs, however, the use of testing does not constitute a formal proof of correctness.

### 2.3.2 Transformation

Another approach towards protocol design is derivation of a the protocol specification from the service specification with the use of correctness preserving transformations. A design method that supports this approach is described in [ES91]. The starting point of this method is a constraint-oriented service specification, structured in terms of local and remote constraints. It assumes that the service is point-to-point (there are only two service users), and that its specification has two local constraints (one for each of the service access points). The resulting specification contains components which can be considered as preliminary forms of the two desired protocol entities. These components interact directly with each other via synchronous interaction, which implies that the synchronous interaction must be refined, in terms of an underlying service.

In Section 4 we show design methods that allow the design of protocol entities that communicate using a reliable or an unreliable medium as underlying communication service, and thus go a step further then the design method described in [ES91].

## 2.4 Protocol Implementation

Some methods used for protocol design can be also used for protocol implementation. In addition, the Lotosphere project has addressed a couple of specific implementation approaches, of which the use of pre-defined implementation constructs ([FSV92]) and compilation represent two quite different alternatives.

### 2.4.1 Use of Pre-defined Implementation Constructs

This approach assumes the availability of a set of high level general-purpose implementation components, called *pre-defined implementation constructs* (PDICs). Each PDIC is represented by its formal specification in LOTOS, whereas a correct implementation (and realization) of this formal specification in the target implementation environment is available. The objective of using PDICs is to transform the protocol specification as early as possible in the design trajectory into a composition PDIC specifications. Since the correctness of each PDIC implementation has been already established by the time of its construction, the correctness of a protocol implementation becomes only dependent on the correctness of the composition of PDIC specifications.

This can be established early in the design trajectory on basis of a formal relation between LOTOS specifications, i.e. the formal specification of the protocol and its corresponding composition of PDIC specifications.

Using this approach, designers are encouraged to develop a library of general purpose PDICs, such that many different protocols can be implemented using this library. Figure 5 depicts this approach.

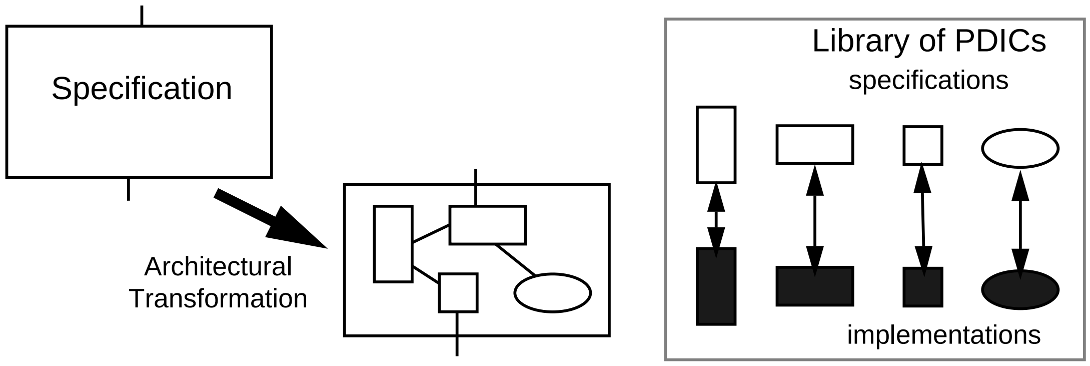


Figure 5: The PDICs Approach

The use of PDICs has the advantage of bringing a clear distinction between structural (architectural) design and the technicalities inherent to different implementation environments. It also has the potential of reducing validation efforts and shortening the design life cycle of individual protocols. The approach has the disadvantage that designers have to cope with the development and maintenance of a library of PDICs. Furthermore automated methods to transform protocol specifications into compositions of PDICs are not (yet) available.

### 2.4.2 Compilation

Two compilers have been developed in Lotosphere and are available in LITE: TOPO and COLOS.

TOPO is a tool that generates C code from a LOTOS specification, by transforming the specification into a model called the Λβ *model* [MS91]. The LOTOS specification is then implemented as a composition of behaviour units (BUTs) which communicate via a synchronization kernel. An algorithm has been defined in [MS91] that corresponds to a set of rules for the evolution of BUTs and generation of new BUTs, in order to mimic the behaviour of the original LOTOS specification. Abstract data type (ADT) specifications are in principle implemented as rewrite rules.

TOPO allows the use of annotations in specifications in order to represent aspects which cannot be formally represented in LOTOS, such as time delays. More efficient ADT implementations hand-coded in C, default values or random choice for value negotiation events, mapping of events on occurrences of concrete interfaces, implementation side effects (e.g. print-out for implementation debugging) and busy waiting for events can be also incorporated in the final implementation by using annotations.

COLOS ([Dub89]) approaches the implementation problem in a slightly different way than TOPO. COLOS is targeted to the implementation of the behaviour part of annotated LOTOS specifications which are rather near to the implementation, i.e. which contain already the implementation decisions which can be represented in LOTOS. This means that the restrictions on the LOTOS specifications that can be implemented with COLOS are more severe than in TOPO. For example, in contrast to TOPO, COLOS does not support the direct implementation of internal events. Furthermore COLOS assumes that the ADT implementations are produced outside COLOS, imported as C code, and integrated during the implementation.

COLOS transforms a LOTOS specification into a set of extended finite state automata that co-operate using a synchronization kernel. The synchronization kernel handles synchronization requests that are deposited in ports, while sorting out the possible events and scheduling pseudo-concurrence of the automata. Time-out's must be explicitly represented as events with timer components, and not as annotations such as in TOPO.

## 3 Some Algorithms for Protocol Design

This section presents three algorithms that have been tailored to protocol design. Section 3.1 presents a modification of the bi-partition of functionality algorithm of [Lan90], Section 3.2 presents the regrouping of parallel processes of [Bol91] and Section 3.3 presents a modification of the tableau method of [Par89].

### 3.1 Bi-partition of Functionality

Structuring of functionality can be performed in LOTOS by using a transformation called *functionality decomposition*. A special case of functionality decomposition in which one LOTOS process is replaced by two cooperating processes has been fully formalized in [Lan90]. Its solution for the case of cooperation through direct (synchronous) communication has been implemented in LITE. This transformation is called *bi-partition of functionality*.

We have modified the asynchronous algorithm of [Lan90] in order to be able to replace (parts of) a service by two (partial) protocol entities and an underlying reliable service. The original algorithm assumes that the process whose functionality is to be bi-partitioned is defined in Basic LOTOS, i.e. without the use of structured events and data types. Our modification supports some limited forms of data values.

We assume that the input specification contains two gates and that it is represented in LOTOS by *B[a, b]*. We also assume that there is no internal event in the input specification and that there is no choice between event offers in which both gates *a* and *b* appear. In order to handle various different sorts of data values, we assume that the input specification and its derivatives have the following form:

$$B\,[a, b] := \textstyle\sum_k \{a\ ?d{:}\ Data_k\ !req\ ;b\ !d\ !ind\ ;\ B_k \mid k \in K\}$$

$$\text{or } \textstyle\sum_k \{b\ ?d{:}\ Data_k\ !req\ ;\ a\ !d\ !ind\ ;\ B_k \mid k \in K\}$$

This means that for some finite index set $K = \{1,..., n\}$, each $Data_k$ represents a distinct data sort and each $B_k$ is a process of the same form as *B*. Parameters *req* and *ind* represent that the direction of the data flow is from the environment to the process being decomposed and vice-versa, respectively. In case $K = \emptyset$, then *B [a, b] := stop*.

Bi-partition will be performed in the gate set, such that there will be one new process per gate. A bi-directional reliable medium capable of transferring one message or acknowledgement per direction of communication at a time can be represented by:

$$M[p, q] := M_1[p, q]\ |||\ M_1[q, p]$$

where
$M_1[p, q] := \sum_k p\ ?m{:}\ Message_k\ !req;\ q\ !m\ !ind;\ M_1[p,q]$
$\qquad [] \sum_k p\ ?ack{:}\ Ack_k\ !req;\ q\ !ack\ !ind;\ M_1[p, q]$

This means that we want to build processes $T_1[a, p]$ and $T_2[b, q]$, such that the structure of the output specification will be:

$P[a, b] := hide\ p, q\ in\ T_1[a, p]\ |||\ T_2[b, q]\ |[p, q]|\ M[p, q]$

We also assume that after data has been "used" it can be deleted. Data is considered to be "used" when it is forwarded to another component. Under these conditions we can define the algorithm for building $T_1[a, p]$ and $T_2[b, q]$ as:

- in case $B\ [a, b] := \sum_k \{a\ ?d{:}\ Data_k\ !req\ ;b\ !d\ !ind\ ;\ B_k \mid k \in K\}$ then

  $T_1(B) := \sum_k \{\ a\ ?d{:}\ Data_k\ !req;\ p!Message_k(d)\ !req;\ p!Ack_k(Message_k(d))\ !ind;\ T_1(B_k) \mid k \in K\}$

  $T_2(B) := \sum_k \{q\ ?m{:}\ Message_k\ !ind;\ b\ !Use_k(m)\ !ind\ ;\ q!Ack_k(m)\ !req;\ T_2(B_k) \mid k \in K\}$

- in case $B\ [a, b] := \sum_k \{b\ ?d{:}\ Data_k\ !req\ ;\ a\ !d\ !ind\ ;\ B_k \mid k \in K\}$ then

  $T_1(B) := \sum_k \{p\ ?m{:}\ Message_k\ !ind;\ a\ !Use_k(m)\ !ind\ ;\ p\ !Ack_k(m)\ !req;\ T_1(B_k) \mid k \in K\}$

  $T_2(B) := \sum_k \{b\ ?d{:}\ Data_k\ !req;\ q\ !Message_k(d)\ !req;\ q!Ack_k(Message_k(d))\ !ind;\ T_2(B_k) \mid k \in K\}$

The following signature defines the relationships between data and the corresponding messages exchanged through the medium:

$Message_k{:}\ Data_k \rightarrow Message_k$
$Use_k{:}\ Message_k \rightarrow Data_k$
$Ack_k{:}\ Message_k \rightarrow Ack_k$

The condition $Use_k(Message_k(d)) = d$, for all $d$ of sort $Data_k$ follows from the form of $B[a, b]$.

The algorithm is proved correct if $B\ [a, b] \approx P[a, b]$, i.e. the output specification is observation equivalent to the input specification. The proof consists of showing that $<B\ [a, b], P[a, b]>$ together with the identity relation define a weak bisimulation relation.

## 3.2 Regrouping of Parallel Processes

During the design process it may occur that processes have to be regrouped, e.g. as an intermediate step in functionality decomposition. The LOTOS transformation that supports this is called *regrouping of parallel processes*. This transformation consists of regrouping processes composed in parallel in such a way that a given composition structure is matched. The behaviour of the resulting specification must remain the same as the input specification, therefore the input and output specifications must be strong bisimulation equivalent in this case.

A formalization of this transformation and a solution has been proposed in [BFO91] and has been implemented in LITE. Particularly useful instances of this transformation (i.e. with fixed composition patterns) have been proved and applied in [SF91] and [V+91].

We present here a restricted solution to this transformation, which is reported in [Bol91]. This solution makes use of the so called *Maximal Coordination Condition* (MCC) in relation to the input and output specifications. A specification satisfies this condition if and only if every occurrence of the parallel operator in the specification has a set of synchronization gates which is exactly the intersection of the set of observable gates of the argument behaviour expressions of the operator.

The algorithm requires that the input specification satisfies the MMC. The output specification is then simply derived from the given composition pattern by instantiating the parallel operators in the pattern in the unique way in which the MCC is satisfied.

For example, consider the following specification:

$B[a, b] := hide\ p\ in\ (L_1[a]\ |||\ L_2[b])\ |[a,b]|\ (E_1[a, p]\ |[p]|\ E_2[b, p])$

which satisfies the MCC, and a desired composition pattern $(L_1\ |[?]|\ E_1)\ |[?]|\ (L_2\ |[?]|\ E_2)$.

The input specification can be transformed according to the regrouping of parallel processes algorithm into:

$B'[a, b] := hide\ p\ in\ (L_1[a]\ |[a]|\ E_1[a,p])\ |[p]|\ (L_2[b]\ |[b]|\ E_2[b,p])$

The algorithm then guarantees that $B[a, b] \sim B'[a, b]$.

## 3.3 Component Construction

It would be desirable to have algorithms that accomplish functionality decomposition in a more general way than indicated in Section 3.1, i.e. even when a certain pre-defined structure of processes can not be identified. Interactive algorithms can be defined in this case, in order to allow designers to fill in behaviours as a consequence of design decisions, such that by the end of this process a correct design is obtained.

The component construction algorithm can take as input any equation with unknown components to be designed, and supports the derivation of correct results in case these are possible. We illustrate the algorithm with the following equation form:

$B[a, b] \approx hide\ p, q\ in\ ((X[a, p]\ |||\ Y[b, q])\ |[p, q]|\ M[p, q])$

where $M[p, q]$ is a given specification of e.g. a communication service, and $X[a, p]$ and $Y[b, q]$ are the unknown components to be designed. The designer builds the unknown components of the output specification interactively, while at the same time (partial) verification is performed. A similar equation solving procedure has been once proposed in [Par89] for CCS; our algorithm is a modification of this procedure for LOTOS.

The algorithm is based on transformations on a “tableau” structure. A tableau consists of a goal and an environment. The goal indicates which equations have to be proved true, and the environment records the intermediate results of the unknown components. The environment is formally defined as a mathematical relation between identifiers and behaviours. Identifiers that are not assigned to behaviours (do not belong to the domain of the environment) are called *free identifiers*.

The following operations on tableaux can be performed:

1. *Instantiation*: free identifiers are instantiated (assigned to behaviours) using heuristics, so that the right hand-side of the equation becomes guarded. This corresponds to replacing:

   $X[a, p]$ by $\sum_{i=0}^{n} x_i;X_i$ and $Y[b, q]$ by $\sum_{j=0}^{m} y_j;Y_j$

This operation is done by the designer, as an expression of the design decisions concerning the behaviour of the unknown components.

2. *Splitting*: the original equation $B \approx E$, where $E$ represents the right-hand side of the equation (side with the unknown components), is replaced by a set of equations $B_i \approx E_i$, such that

$$\text{for all } i \in \{1, \ldots, n\} : E \overset{e_i}{\Rightarrow} E_i \text{ and } B \overset{e_i}{\rightarrow} B_i, \text{ and } B \approx \sum_{i=1}^{n} e_i;B_i$$

3. *Removal*: equations similar to other equations which have already been unfolded using splitting are removed from the goal;

4. *Identification*: free identifiers can be assigned to already instantiated identifiers, under the condition that there is a matching on the set of equations for both identifiers.

Operations 1 and 4 cannot be automated, since they expect the designer to convey design information, which allows the algorithm to proceed. Operations 2 and 3 can be automated, since they correspond to mechanical unfolding and checking of the left and right-hand sides of the equations. Splitting may not be possible in some cases, which shall mean that an incorrect instantiation had taken place previously. In this case, the process has to return to the point of the incorrect instantiation (*backtracking*)

The process stops when all equations are removed, turning the goal into *true*.

The basic equation solving procedure, augmented with some heuristics to assist instantiation, has been used in [Par89] to synthesize the receiver entity of a version of the alternating bit protocol, given the required service, the sender entity and the underlying service. The additional rules permit the construction of a structured protocol specification. Some examples of their use are presented in [SF91]. Currently no tool support is available for this method in LITE. The method also lacks sufficient heuristics that capture design experience to assist instantiation in the application to medium and large scale design problems. Nevertheless we believe that this algorithm is promising, since most of its time consuming operations (splitting and removal) can be automated.

## 4 An Example of Protocol Design

This section illustrates a transformational approach to protocol design, using the algorithms presented in Section 3, with the design of a relatively simple, but non-trivial, protocol: a protocol that provides a question-answer service on top of an unreliable data transfer service. We also illustrate the application of the layering strategy: two successive decompositions are presented, resulting in two protocol layers. Each decomposition step is concerned with a distinct design problem, the nature of which determines what design methods are best suited for the design of the protocol during this step.

The first decomposition step deals with application concerns, i.e. it focuses on the provision of the application-oriented question-answer service (QAS) and assumes the availability of a reliable data transfer service (RTS). The procedure for the exchange of protocol information is very simple in this case, although the formatting and coding of the protocol information is potentially quite complex. Because of this, we use the bi-partition of functionality algorithm introduced in Section 3.1 in order to design the protocol, abstracting away from the structuring and representation of protocol information. The protocol designed during this step is called the question-answer protocol (QAP).

The second decomposition step deals with data transfer concerns, i.e. it aims at the provision of the reliable data transfer service assumed in the previous step on top of the unreliable data transfer service (UTS). Since this step takes into account possible loss of data, the procedure for the exchange of protocol information is relatively complex when compared to the first step. Due to this, we make use of the component construction algorithm introduced in Section 3.3 in order to design the protocol. A simple structuring of protocol information is considered, but not the representation of this information. The protocol designed during this step is called the reliable transfer protocol (RTP). Figure 6 depicts these two decomposition steps.

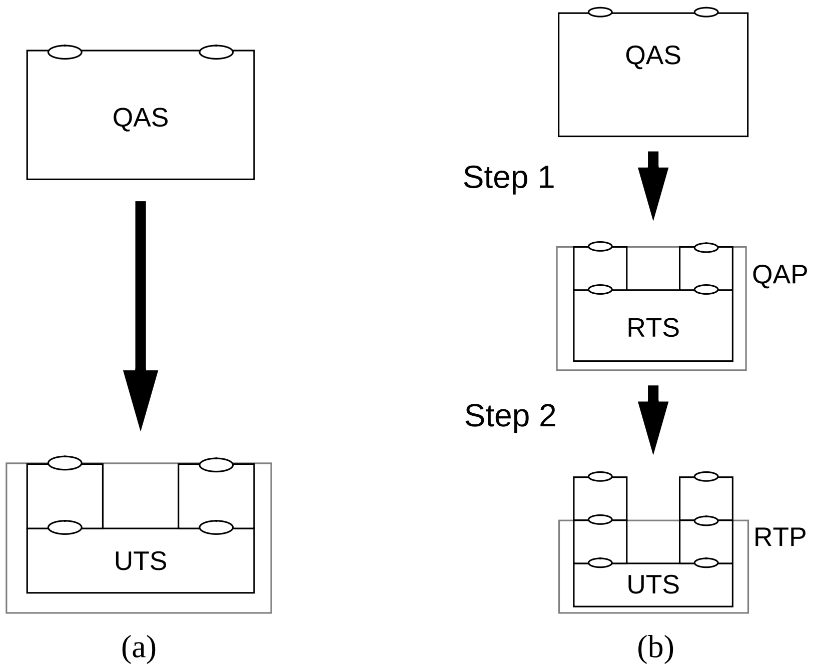


Figure 6: Protocol Design Example - (a) design objective, (b) design approach

This section is further structured as follows: Section 4.1 presents the architectural semantics of the concepts used in this example, Section 4.2 presents the specification of QAS, RTS and UTS, and the two design steps are subsequently elaborated in Sections 4.3 and 4.4.

## 4.1 Architectural Semantics

Two architectural concepts, viz. service primitive and protocol data unit, and their representations in LOTOS are discussed below.

### 4.1.1 Service Primitive

Services are defined in terms of service primitives. A *service primitive* is an elementary interaction between a *service user* (user for short) and the *service provider* (provider for short). We consider a simplified model of service primitives, which is suitable for our example:

- service primitives are formally represented by (structured) events. The conditions imposed by a user or the provider to participate in a service primitive are represented by an event offer;
- service primitives have a location, which is called a *service access point*. A service access point is represented by a gate identifier;

- service primitives effect the exchange of information. We call the information exchanged in a service primitive a *service information unit* (SIU)[1]. An SIU exchanged in a service primitive is represented by an event parameter;
- service primitives have a direction associated with the exchange of their SIU. A direction is represented by a second event parameter which can have one of two values: *req* (request), if the direction is from a user to the provider, and *ind* (indication), if the direction is from the provider to a user.

### 4.1.2 Protocol Data Unit

Protocol entities communicate through the exchange of *protocol data units* (PDUs). The exchange of PDUs is necessary to support the exchange of SIUs. Therefore PDUs convey (parts of) SIUs, and/or internally generated protocol control information that guarantee error-free exchange of SIUs.

Since there is no direct exchange of PDUs between protocol entities in our example, a PDU exchange is not represented by a single event. Instead, PDUs must be mapped onto data of service primitives of the underlying service in order to be exchanged. Mapping functions to relate PDUs to service primitives of the required service, and to service primitives of the underlying service, have to be defined.

## 4.2 Specification of the Services

This section presents the simplified specifications of the services which are used in the protocol design example.

### 4.2.1 Question-Answer Service (QAS)

The question-answer service accepts a question from a calling user, delivers it to a called user, accepts an answer from the called user, and returns the answer to the calling user. This service bears some resemblance to the Remote Operations Service Element standardized by ISO ([ISO89b]), which is used in many distributed applications.

An instance of QAS with two users, each user with a fixed role (either calling or called), can be specified as follows, using a monolithic style:

*QAS[a,b] := a ?q:Question !req ; b !q !ind ; b ?a:Answer !req ; a !a !ind ; stop*

The specification of this service can alternatively be structured in terms of local and remote constraints, i.e. using a constraint-oriented style:

$$QAS[a, b] := (L_1[a] \; ||| \; L_2[b]) \; || \; (R_1[a, b] \; ||| \; R_2[a, b]) \qquad (1)$$

where

$L_1[a]$ *:= a ?q:Question !req; a ?a:Answer !ind; stop*

$L_2[b]$ *:= b ?q:Question !ind; b ?a:Answer !req; stop*

$R_1[a, b]$ *:= a ?q:Question !req; b !q !ind; stop*

1. The concept of SIU defined here does not necessarily correspond to the OSI concept of SDU. An SIU actually corresponds to the collection of parameters associated with a service primitive. We use SIU instead of SDU because it allows a more convenient notation in the example.

*$R_2$[a, b] := b ?a:Answer !req; a !a !ind; stop*

These two specifications are equivalent on basis of the operational semantics of LOTOS. Figure 7 depicts the behaviour of QAS.

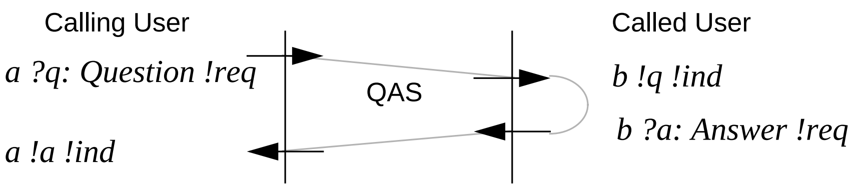


Figure 7: Behaviour of QAS

### 4.2.2 Reliable Data Transfer Service (RTS)

The reliable data transfer service accepts data from a sending user and delivers it to a receiving user. This means that the correct delivery is guaranteed. Communication networks which perform error detection and error correction provide this type of service.

For the specification of RTS we again assume two users, but in this case the users do not have a fixed role, i.e. either user may be a sending or a receiving user. Another restriction we impose is that the transfer capacity of the service for each direction of transfer is one data unit, i.e. at most one data unit can be “under way” in a certain direction.

A specification which represents the two directions of transfer as independent constraints is:

*RTS[a, b] := $RR_1$[a,b] ||| $RR_2$[a,b]* (2)

where
*$RR_1$[a, b] := a ?d:Data !req; b !d !ind; $RR_1$[a,b]*

*$RR_2$[a, b] := $RR_1$[b, a]*

Figure 8 depicts some possible primitive sequences of this service:

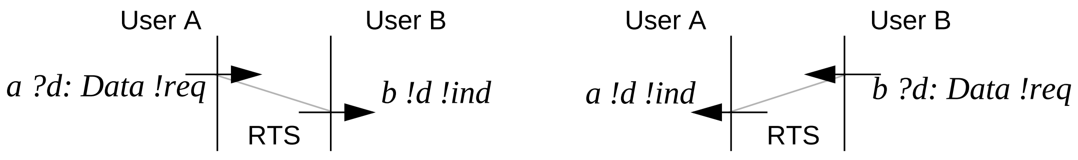


Figure 8: Possible Primitive Sequences of RTS

### 4.2.3 Unreliable Data Transfer Service (UTS)

The unreliable data transfer service accepts data from a sending user and attempts to deliver it to a receiving user. The service is unreliable in the sense that data is either delivered unchanged (not corrupted) to the intended user or it is lost (discarded). Communication networks which perform error detection but no (or not sufficient) error correction provide this type of service.

We adopt the same restrictions for UTS as for RTS, i.e. the service is provided to two users and the transfer capacity per direction of transfer is one data unit.

Possible loss of data must be represented in the UTS specification. Whether data is lost or not depends exclusively on internal behaviour of the service provider, and therefore cannot be influenced by the users. The choice between delivery and loss is modelled by internal events (instances of *i*) in the following specification:

*UTS[a, b] := UR$_1$[a, b] ||| UR$_2$[a, b]* (3)

where
*UR$_1$[a, b] := a !d:Data !req; (i (* delivery *); b !d !ind; UR$_1$[a, b] [] i (* loss *); UR$_1$[a, b])*

*UR$_2$[a, b] := UR$_1$[b, a]*

Figure 8 depicts some possible primitive sequences of this service:

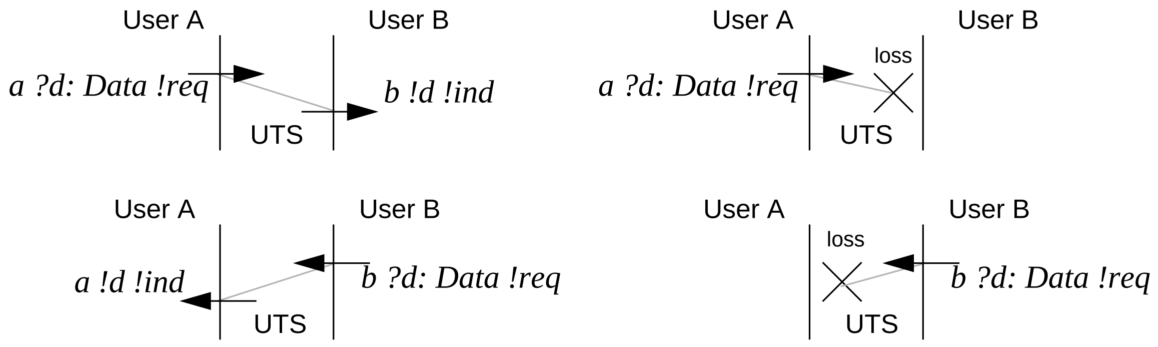


Figure 9: Possible Primitive Sequences of UTS

## 4.3 Step 1: Design of the Question-Answer Protocol

Bi-partitioning the remote constraints which are defined by $R_1$ and $R_2$ in (1) using the algorithm described in Section 3.1 yields:

*R$_1$[a, b] ≈ hide p, q in (R$_{11}$[a, p] ||| R$_{12}$[b, q]) |[p, q]| M$_1$[p, q]* (4)

where
*R$_{11}$[a, p] := a ?q:Question !req; p !MessageQ(q) !req; p !AckQ(MessageQ(q)) !ind; stop*

*R$_{12}$[b, q] := q ?p:MessageQ !ind; b !UseQ(p) !ind; q !AckQ(p) !req; stop*

and

*R$_2$[a, b] ≈ hide p',q' in ( R$_{21}$[a, p'] ||| R$_{22}$[b, q'] ) |[p',q']| M$_2$[p', q']* (5)

where
*R$_{21}$[a, p′] := p' ?p:MessageA !ind; a !UseA(p) !ind; p' !AckA(p) !req; stop*

*R$_{22}$[b, q′] := b ?a:Answer !req; q' !MessageA(a) !req; q' !AckA(MessageA(a)) !ind; stop*

Suffixes *Q* and *A* are used above instead of indexes from an index set *K*; processes $M_1$ and $M_2$ represent different instances of a reliable medium (see Section 3.1). $M_1$ has gate set *{p, q}* and handles communication of type *Q*, while $M_2$ has gate set *{p′, q'}* and handles communication of type *A*.

Applying regrouping of parallel processes in (4) and (5), and integrating $M_1$ and $M_2$ in a single communication medium *M*, we transform (1) into the following protocol specification:

*QAP[a, b] := hide p,q in*
*((*$L_1$*[a] |[a]| (*$R_{11}$*[a, p] ||| *$R_{21}$*[a ,p])) ||| (*$L_2$*[b] |[b]| (*$R_{12}$*[b, q] ||| *$R_{22}$*[b, q])))*
*|[p,q]| M[p, q]* *(6)*

where *M* is defined as in Section 3.1 such that it handles communication of both types *Q* and *A*.

Process *M* represents a dedicated channel for the exchange of PDUs. It is more cost effective however if the protocol could use a general-purpose data transfer service, such as RTS defined in (2), since then the same service can be shared by many application protocols. In order to achieve this, we apply the operations *Wrap*, *UnwrapM*$_k$ and *UnwrapA*$_k$, with $k \in \{Q,A\}$, and modify the protocol specification of (6) as follows:

*QAP[a, b] := hide p,q in (QAPE*$_1$*[a, p] ||| QAPE*$_2$*[b, q]) |[p,q]| RTS[p, q]* *(7)*

where
*QAPE*$_1$*[a, p] := *$L_1$*[a] |[a]| (*$R_{11}$*'[a, p] ||| *$R_{21}$*'[a, p])*

$R_{11}$*'[a, p] := a ?q:Question !req; p !Wrap(MessageQ(q)) !req;*
*p !Wrap(AckQ(MessageQ(q))) !ind; stop*

$R_{21}$*'[a, p] := p ?d:Data !ind; a !UseA(UnwrapMA(d)) !ind;*
*p !Wrap(AckA(UnwrapMA(d))) !req; stop*

$L_1$ as defined in (1)

and
*QAPE*$_2$*[b, q] := *$L_2$*[b] |[b]| (*$R_{21}$*'[b, q] ||| *$R_{22}$*'[b, q])*

$R_{12}$*'[b, q] := q ?d:Data !ind; b !UseQ(UnwrapMQ(d)) !ind;*
*q !Wrap(AckQ(UnwrapMQ(d))) !req; stop*

$R_{22}$*'[a, p] := b ?a:Answer !req; q !Wrap(MessageA(a)) !req;*
*q !Wrap(AckA(MessageA(a))) !ind; stop*

$L_2$ as defined in (1)

The requirement that *QAS[a, b] ≈ QAP[a, b]*, under the condition that the wrapping and unwrapping operations are inverse, is guaranteed by the use of transformations. *QAP*, as represented in (7), is a question-answer protocol that fulfils the functional requirements of this design step.

## 4.4 Step 2: Design of the Reliable Transfer Protocol

The design of a protocol that implements RTS corresponds to solving the following equation:

*RTS[a, b] ≈ hide p,q in (RTPE*$_1$*[a, p] ||| RTPE*$_2$*[b, q]) |[p, q]| UTS[p, q]* *(8)* where $RTPE_1$ and $RTPE_2$ are the unknown protocol entities to be designed, *RTS* is a reliable data transfer service as defined in (2), and *UTS* is a unreliable data transfer service as defined in (3).

Since the RTS comprises two independent directions of transfer, we can assume that each protocol entity also consists of two independent parts, each part related to one of the directions of data transfer of RTS. Therefore $RTPE_1$ and $RTPE_2$ can be structured as follows:

$$RTPE_1[a,p] := SE_1[a,p] \mid\mid\mid RE_1[a,p] \qquad (9)$$

$$RTPE_2[b,q] := SE_2[b,q] \mid\mid\mid RE_2[b,q]$$

where $SE_i$ is a “sender” process and $RE_i$ is a “receiver” process. We further assume that $SE_1$ and $RE_1$ (also $SE_2$ and $RE_2$), have mutual exclusive conditions on service primitives of the underlying service, i.e. they handle different service primitives.

Using the instantiations of $RTPE_1$ and $RTPE_2$ in (9), applying regrouping of parallel processes, and replacing *UTS* by two different instances of UTS as defined in (3), we can transform (8) into:

$$\begin{aligned} RTS[a, b] \approx\ & (hide\ p,q\ in\ (SE_1[a, p] \mid\mid\mid RE_2[b, q]) \mid[p,q]\mid UTS[p, q]) \\ & \mid\mid\mid (hide\ p',q'\ in\ (RE_1[a, p'] \mid\mid\mid SE_2[b, q']) \mid[p',q']\mid UTS[p', q']) \end{aligned}$$

Under the given assumptions, we can split the equation into two simpler equations ([SF91]):

$$R_1[a, b] \approx hide\ p,q\ in\ (SE_1[a, p] \mid\mid\mid RE_2[b, q]) \mid[p,q]\mid UTS[p, q] \qquad (10)$$

$$R_2[a, b] \approx hide\ p,q\ in\ (RE_1[a, p] \mid\mid\mid SE_2[b, q]) \mid[p,q]\mid UTS[p, q]$$

From (2) and (3) we can conclude that a solution for $SE_1$ ($RE_1$) is also a solution for $SE_2$ ($RE_2$). Hence it is enough to solve one of the equations in (10).

### 4.4.1 Design Decisions

The solution of (10) cannot be obtained automatically, since there are a number of design decisions which cannot be automated and have to be taken in order to reach a correct result. Therefore some design decisions to be applied together with the component construction algorithm are considered here.

The provision of a reliable data transfer service implies that lost data should be re-sent by the sender until it is successfully delivered to the receiver. Hence the sender must be able to decide when it has to re-sent data. We adopt the following solution:

- the receiver returns the sender an acknowledgement to indicate successful delivery of data;
- after it has (re-) sent data, the sender waits a time-out period for an acknowledgement. If the acknowledgement is received within this period, it assumes that the data is successfully delivered and stops (re-) sending data. Otherwise it assumes that the data is lost and sends it again.

Since an acknowledgement may also be lost by the underlying service, the sender may wrongly decide that data is lost. Consequently, the receiver may receive the same data more than once. The receiver must be able to decide whether data is duplicate or not. We adopt the following solution:

- an identifier (sequence number) is assigned to each sent data, such that successive data units have different identifiers. In this case it suffices to have two different identifier values, e.g. *0* and *1*, which are alternately used;
- duplicate data are acknowledged, but not delivered to the user of the required service.

In case the time-out period is too short, it may occur that the sender re-sends data while in fact neither the data, nor the acknowledgement is lost. Consequently, the sender may receive more than one acknowledgement related to the same data. The sender must be able to decide whether an acknowledgement is duplicate or not. We adopt the following solution:

- an identifier (sequence number) is assigned to each acknowledgement, such that this identifier corresponds to the identifier of the data it acknowledges;
- duplicate acknowledgements are discarded.

### 4.4.2 Heuristics

The equations to be considered by the components construction algorithm when it is applied to (10) are always of the form:

$$B[a, b] \approx \text{hide } p, q \text{ in } ( X[a, p] \;|||\; Y[b, q] ) \;|[p, q]|\; M[p, q] \qquad (11)$$

where $B$ is a derivative of the remote constraint $R_1$ in (9), $X$ and $Y$ are unknown parts of the sender process and receiver process respectively, and $M$ is a derivative of the unreliable data transfer service *UTS* in (10). Considering also the design decisions discussed above, we are able to identify some heuristics which can help taking instantiation decisions when solving (10).

We assume that instantiations of $X$ and $Y$ have the form:

$$X := \sum \{x_i; X_i \mid i \in I\}$$

$$Y := \sum \{y_j ; Y_j \mid j \in J\}$$

where each $x_i$ ($y_j$) is either:

- an initial event offer of $B$, or
- an event offer which allows a synchronization of $X$ ($Y$) and $M$, such that the matching event of $M$ is the first in a sequence of event offers which occurs at the common gate of $X$ ($Y$) and $M$.

An event offer that satisfies this condition is called a *useful event offer*. The designer must decide which subset of the set of useful event offers will be used in an instantiation.

We further assume that:

- if $M$ has an initial event offer of type indication, then the same event offer is only used in an instantiation of $Y$ ($X$) if the information expected in this event offer can be derived from a preceding event of $X$ ($Y$) and $M$;
- an event offer of type request which allows a synchronization of $X$ ($Y$) and $M$ is only used in an instantiation of $X$ ($Y$) if the (wrapped) information exchanged in this event can be derived from a preceding event of $X$ ($Y$) and $M$, or from the parameters (stored status) of $X$ ($Y$);

The latter heuristics can be used during instantiation to select properly from the set of useful event offers.

The following heuristics can be used to instantiate *X* (*Y*) by identification:

- *X* (*Y*) can be identified with a bound identifier *V*, i.e. *X* := *V* (*Y* := *V*), if the equations containing *X* (*Y*) are a subset of the equations containing *V*, and *X* (*Y*) would be instantiated such that its initial events are the same as those of *V*. In this case substituting *X* by *V* adds no new equations to the goal;
- *X* and *Y* can be simultaneously identified with the bound identifier *V* and *W* respectively, i.e. *X* := *V* and *Y* := *W*, if the equations containing *X* and *Y* constitute a subset of the equations containing *V* and *W*, and *X* and *Y* would be instantiated such that its initial events are the same as those of *V* and *W*, respectively.

### 4.4.3 Application of the Components Construction Algorithm

We start the application of the component construction algorithm with equation (10) to be solved and an empty environment. Table 2 lists the initial results of the algorithm, while indicating the operations which were performed to obtain these results. The initial value of the sequence number is taken to be *0*.

For the sake of readability of Table 2, some additional process identifiers have been introduced. These are:

$RR_1[a, b] := a\ ?d{:}Data\ !req;\ RR_{11}[a, b](d)$
where
$RR_{11}[a,b]\ (d{:}\ Data) := b\ !d\ !ind;\ RR_1[a, b]$

$UTS[p, q] := UR_1[p, q]\ |||\ UR_2[p, q]$
where
$UR_1[p, q] := p\ ?d{:}Data\ !req\ ;\ UR_{11}[p, q](d)$
$UR_{11}[p, q](d{:}\ Data) := i;\ UR_{12}[p, q](d)\ []\ i;\ UR_1[p, q]$
$UR_{12}[p, q](d{:}\ Data) := q\ !d\ !ind\ ;\ UR_1[p, q]$

A number of operations to handle abstract data types have also been introduced. They are represented in the following signatures:

*MessageD: Data, Number -> MessageD*
*UseD: MessageD -> Data*
*AckD: Number -> AckD*
*Number: MessageD | MessageA -> Number (* extract sequence number parameter *)*
*Wrap: MessageD | AckD -> Data*
*UnwrapMD: Data -> MessageD*
*UnwrapAD: Data -> AckD*
*Next: Number -> Number (* generate next sequence number *)*

Equations involving terms generated by these signatures are omitted for the sake of brevity.

A number of shorthand notations are used in Table 2 for representing concatenation of operations. These shorthand notations are:

*NA:= Number * UnwrapAD*
*NM:= Number * UnwrapMD*
*M:= Wrap * MessageD*
*D:= UseD * UnwrapMD*

Furthermore, process instantiations are abbreviated in the table by omitting gates and parameters (e.g. we use $UR_{11}$ for $UR_{11}[p, q](d)$).

Repeated application of the operations of the components construction algorithm eventually leads to a tableau without free variables in the equations of the goal. A solution for the sender and receiver process can then be constructed. In order to ensure that this solution is a correct one, the remaining equations in the goal should be proved true. This can be done by repeatedly applying splitting and removal operations until all equations are removed in the goal of the final tableau.

Table 2: Application of the Component Construction Algorithm

| **operation** | **goal (equations)** | **environment (instantiations)** |
|---|---|---|
| | $RR_1 \approx \text{hide } p,q \text{ in } (SE_1 \vert\vert\vert RE_2) \vert[p,q]\vert (UR_1 \vert\vert\vert UR_2)$ | |
| instantiation | | $SE_1 :=$ a ?d:Data !req; $SE_{11}$ |
| | | [] p ?d:Data !ind [NA(d) = 1]; $SE_{12}$ |
| | | $RE_2 :=$ q ?d !ind [NM(d) = 0]; $RE_{21}$ |
| | | [] q ?d !ind [NM(d) = 1]; $RE_{22}$ |
| splitting | $RR_{11} \approx \text{hide } p,q \text{ in } (SE_{11} \vert\vert\vert RE_2) \vert[p,q]\vert (UR_1 \vert\vert\vert UR_2)$ | |
| instantiation | | $SE_{11} :=$ p !M(d,0) !req; $SE_{13}$ |
| | | [] p ?d !ind [NA(d) = 1]; $SE_{14}$ |
| splitting | $RR_{11} \approx \text{hide } p,q \text{ in } (SE_{13} \vert\vert\vert RE_2) \vert[p,q]\vert (UR_{11} \vert\vert\vert UR_2)$ | |
| instantiation | | $SE_{13} :=$ p ?d:Data !ind [NA(d) = 0]; $SE_{15}$ |
| | | [] p ?d:Data !ind [NA(d) = 1]; $SE_{16}$ |
| | | [] p! M(d,0) !req; $SE_{17}$ |
| splitting | $RR_{11} \approx \text{hide } p,q \text{ in } (SE_{13} \vert\vert\vert RE_2) \vert[p,q]\vert (UR_{12} \vert\vert\vert UR_2)$ | |
| | $RR_{11} \approx \text{hide } p,q \text{ in } (SE_{13} \vert\vert\vert RE_2) \vert[p,q]\vert (UR_1 \vert\vert\vert UR_2)$ | |
| splitting | $RR_{11} \approx \text{hide } p,q \text{ in } (SE_{13} \vert\vert\vert RE_{21}) \vert[p,q]\vert (UR_1 \vert\vert\vert UR_2)$ | |
| | $RR_{11} \approx \text{hide } p,q \text{ in } (SE_{17} \vert\vert\vert RE_2) \vert[p,q]\vert (UR_{11} \vert\vert\vert UR_2)$ | |
| identification | | $SE_{17} := SE_{13}$ |
| removal | $RR_{11} \approx \text{hide } p,q \text{ in } (SE_{13} \vert\vert\vert RE_{21}) \vert[p,q]\vert (UR_1 \vert\vert\vert UR_2)$ | |
| instantiation | | $RE_{21} :=$ b !D(d) !ind; $RE_{23}$ |
| etc. | | |

The solutions found for $SE_1$ and $RE_2$ are then:

*$SE_1$[a, p](n:Number) :=*
*a ?d:Data !req ; $SE_{11}$[a, p] (n,d)*
*[] p ?d:Data !ind [Number(UnwrapAD(d)) ≠ n]; $SE_1$[a, p](n)*

where
*$SE_{11}$[a, p] (n:Number, d:Data) :=*

$\quad\quad p\ !Wrap(MessageD(d,n))\ !req\ ;\ SE_{13}[a, p](n,d)$
$[]\ p\ ?d{:}Data\ !ind\ [Number(UnwrapAD(d)) \neq n];\ SE_{11}[a, p](n,d)$

$SE_{13}[a, p](n{:}Number, d{:}Data) :=$
$\quad\quad p\ ?d{:}Data\ !ind\ [Number(UnwrapAD(d)) = n];\ SE_1[a, p]\ (Next(n))$
$[]\ p\ ?d{:}Data\ !ind\ [Number(UnwrapAD(d)) \neq n];\ SE_{13}[a, p](n,d)$
$[]\ p\ !Wrap(MessageD(d,n))\ !req\ ;\ SE_{13}[a, p](n,d)$

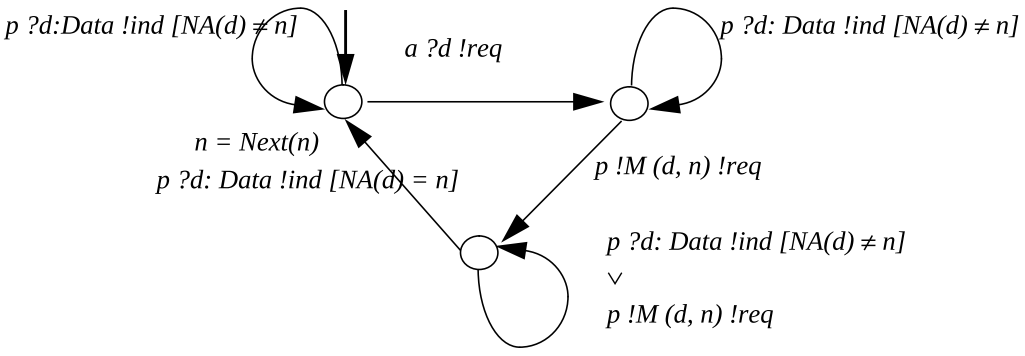
$RE_2[b, q](n{:}Number) :=$
$\quad\quad q\ ?d{:}Data\ !ind\ [Number(UnwrapMD(d)) = n];\ RE_{21}[b, q]\ (n,d)$
$[]\ q\ ?d{:}Data\ !ind\ [Number(UnwrapMD(d)) \neq n];\ RE_{22}[b, q](n)$

where
$RE_{22}[b, q]\ (n{:}Number) :=$
$\quad\quad q\ !Wrap(AckD(Next(n)))\ !req\ ;\ RE_2[b, q](n)$

$RE_{21}[b, q]\ (n{:}Number, d{:}Data) :=$
$\quad\quad b\ !UseD(UnwrapMD(d))\ !ind\ ;\ RE_{23}[b, q](n)$

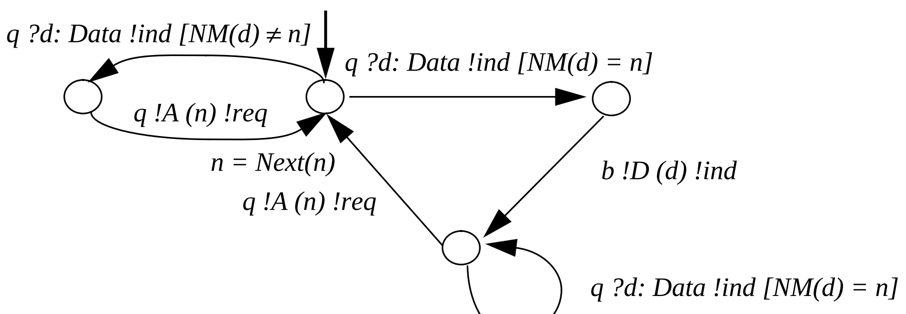
$RE_{23}[b, q](n{:}Number) :=$
$\quad\quad q\ !Wrap(AckD(n))\ !req\ ;\ RE_2[b, q](Next(n))$
$[]\ q\ ?d{:}Data!ind\ [Number(UnwrapMD(d)) = n];\ RE_{23}[b, q](n)$

Figure 10 depicts an alternative representation of the sender and receiver processes by way of state transition diagrams, using the shorthand notations of Table 2.



p ?d:Data !ind [NA(d) ≠ n]
a ?d !req
p ?d: Data !ind [NA(d) ≠ n]
n = Next(n)
p ?d: Data !ind [NA(d) = n]
p !M (d, n) !req
p ?d: Data !ind [NA(d) ≠ n]
∨
p !M (d, n) !req

(a)



q ?d: Data !ind [NM(d) ≠ n]
q ?d: Data !ind [NM(d) = n]
q !A (n) !req
n = Next(n)
q !A (n) !req
b !D (d) !ind
q ?d: Data !ind [NM(d) = n]

(b)

Figure 10: Behaviour of the (a) Sender and (b) Receiver

Finally, it follows from (9) that the protocol entities are:

$RTPE_1[a, p] := SE_1[a, p](0) \,|||\, RE_1[a, p](0)$

$RTPE_2[b, q] := SE_2[b, q](0) \,|||\, RE_2[b, q]\,(0)$

where
$RE_1[a, p](n{:}Number) := RE_2[a, p](n)$

$SE_2[p,q](n{:}Number) := SE_1[p, q](n)$

Hence, the protocol is defined by:

$RTP[a, b] := hide\ p,q\ in\ (RTPE_1[a, p] \,|||\, RTPE_2[b, q])\ |[p,q]|\ UTS[p, q]$

The protocol defined above fulfils the functional requirements of this design step. The fact that *RTS[a, b] ~ RTP[a, b]*, under the condition that the operations defined on abstract data types satisfy certain requirements (to be defined by their corresponding equations), is guaranteed by the correctness of the algorithm.

## 5 Conclusion

This paper presented a collection of design methods and has shown that they can be useful in the design and implementation of protocols. Special attention has been given to a transformational approach towards protocol design, which has been illustrated by means of a simplified protocol design example.

Protocol designers are interested in design methods that altogether cover the complete design trajectory. Considering the design methods presented and discussed in this paper, we can make the following observations:

- specifications that are structured according to specification styles facilitate validation. The use of a few related specification styles, as proposed in [VSS88], contributes to the structuring of specifications along the architectural and implementation phase of the protocol design trajectory;
- the transformation approach illustrated in Section 4.3 can be used in the early phases of protocol design, when the distribution of basic protocol functions that directly support the required service is considered. This transformation is particularly suitable if the required service is application-oriented rather than data transfer oriented. The reason for this is that application-oriented services are generally defined in terms of different service elements, involving different service primitives. For each service element, separate protocol functions, involving different PDUs, can be defined that require a reliable data transfer service for the exchange of PDUs. Therefore the decomposition of functionality in application and data transfer becomes straightforward;
- the transformational approach illustrated in Section 4.4 can be used in subsequent steps of protocol design, where protocol functions are defined that are more concerned with the hiding of undesirable properties of the underlying service, rather than merely providing properties of the required service;
- it may happen that the approach illustrated in Section 4.4 is not applicable for some instances of design, since the instantiation of free variables may become unmanageable due to lack of appropriate heuristics. In this case the refined protocol has to be designed by hand and validated using testing or verification afterwards;

- the protocol implementation can be structured in terms of resources that correspond to PDICs. The PDIC approach mentioned in Section 2.4.1 can be used to derive an implementation of the protocol in a particular target environment. This covers the last part of the implementation phase of the design trajectory. An alternative to the use of PDICs is to refine the protocol specification until a specification is reached which can be mapped straightforwardly onto a realization. An example of the latter approach is given in [EKS90] where a resource-oriented specification is first transformed into a state-oriented one and subsequently mapped onto C code;
- another approach towards protocol implementation is the use of compilers. The more realistic and effective approach, however, is to combine the use of compilers with the use of PDICs. COLOS seems to be the compiler which is most suitable for the hybrid approach;
- since during the protocol design and implementation processes ad-hoc methods may be used, designers have to validate the protocol implementations against their specifications. Furthermore, protocol implementations very often have to certified, which means that organizations with a certain authority must state that the implementation complies to certain standard specification. Conformance testing methods can be used for both validation and certification of protocol implementations. Conformance testing methods using LOTOS have been discussed in [B+90].

Although a sophisticated set of design methods and support tools have been developed in the Lotosphere project, some effort is still necessary to integrate methods and to enhance tool functionality. It is out firm belief that the Lotosphere design methodology and its tools constitute already a important step forward in the direction of efficient and correct protocol design and implementation.

## 6 References

[Abr87] S. Abramsky. Observational Equivalence as a Testing Equivalence. *Theorethical Computer Science*, pp. 225-241, 1987.

[B+90] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, J. Tretmans. A Formal Approach to Conformance Testing. In: J. de Meer, L. Mackert, W. Effelsberg (eds.). *2nd International Workshop on Protocol Test Systems*, North-Holland, 1990, pp. 349-363.

[Bog89] K. Bogaards. LOTOS Supported System Development. In: K.J. Turner (ed.). *Formal Description Techniques*. North-Holland, 1989, pp. 279-294.

[Bol91] T. Bolognesi (ed.). *Catalogue of LOTOS Correctness Preserving Transformations*, Lotosphere Task 1.2 Final Deliverable. Lotosphere Project, 1992.

[BFO91] T. Bolognesi, D. de Frutos-Escrig, Y. Ortego-Mallén. Graphical Composition Theorems for Parallel and Hiding Operators. In: J. Quemada, J. Mañas, E. Vázquez (eds.). *Formal Description Techniques III*, North-Holland, 1991, pp. 459-470.

[Bri91] E. Brinksma. What is the Method in Formal Methods? In: K. Parker, G. Rose (eds.). *Formal Description Techniques IV.* North-Holland, Netherlands, 1992. pp. 33-50.

[Dub89] E. Dubuis. An Algorithm for Translating LOTOS Behaviour Expressions into Automata and Ports. In: *Second International Conference on Formal Description Techniques*, Vancouver, 1989.

[Eij91] P. van Eijk. *Tool Demonstration: The Lotosphere Integrated Tool Environment Lite.* In: K. Parker, G. Rose (eds.). *Formal Description Techniques IV.* North-Holland, Netherlands, 1992. pp. 471-474.

[ES91] P. van Eijk, J. Schot. An Exercise in Protocol Synthesis. In: K. Parker, G. Rose (eds.). *Formal Description Techniques IV.* North-Holland, Netherlands, 1992. pp. 117-131.

[EKS90] P. van Eijk, H. Kremer, M. van Sinderen. On the Use of Specification Styles for Automated Protocol Implementation from LOTOS to C. In: L. Logrippo, R.L. Probert, H. Ural (eds.). *Protocol Specification, Testing and Verification X*, North-Holland, 1990, pp. 157-168.

[Fel91] M. Feldhoffer. Communication Support for Distributed Applications. In: *International IFIP Workshop on Open Distributed Processing* - Participants Proceedings, Berlin, October 8-11, 1991.

[ISO89a] ISO. *Information Processing Systems - Open Systems Interconnection - LOTOS,* International Standard ISO/IEC 8807, 1989.

[ISO89b] ISO. *Information Technology - Text Communication - Rempote Operations, Part 1: Model, Notation and Service Definition*, International Standard ISO/IEC 9072-1, 1989.

[Lan90] R. Langerak. Decomposition of Functionality: A Correctness-Preserving LOTOS Transformation. In: L. Logrippo, R.L. Probert, H. Ural (eds.). *Protocol Specification, Testing and Verification X,* North-Holland, 1990, pp. 229-243.

[MS91] J. A. Mañas, J. Salvachúa. $\Lambda\beta$: a Virtual LOTOS Machine. In: K. Parker, G. Rose (eds.). *Formal Description Techniques IV.* North-Holland, Netherlands, 1992. pp. 441-456.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International Series in Computer Science. Prentice-Hall, Great Britain, 1989.

[Par89] J. Parrow. Submodule Construction as Equation Solving in CCS. In: *Theoretical Computer Science 68*, North-Holland, 1989, pp. 175-202.

[FSV92] L. Ferreira Pires, M. van Sinderen, C.A. Vissers. On the Use of Pre-Defined Implementation Constructs in Distributed Systems Design. In: *3rd IEEE Workshop on Future Trends in Distributed Computing Systems in the 1990's*. Taipei, April, 1992.

[FV90] L. Ferreira Pires, C.A. Vissers. Overview of the Lotosphere Design Methodology, In: CEC. *ESPRIT Conference 1990*, Kluwer Academic Publishers, 1990, pp. 371-387.

[SF91] M. van Sinderen, L. Ferreira Pires. FDT-Based Protocol Design. In: W. Komorowski (ed.). *Computer Networks '91*, Wydawnictwo Politechniki Wroclawskiej, 1991, pp. 161-168.

[Tur87] K. Turner. An Architectural Semantics for LOTOS. In: H. Rudin, C.H. West (eds.). *Protocol Specification Testing and Verification VII*, North-Holland, 1987.

[VSS88] C.A. Vissers, G. Scollo, M. van Sinderen. Architecture and Specification Style in Formal Descriptions of Distributed Systems. In: S. Aggarwal, K. Sabnani (eds.). *Protocol Specification, Testing and Verification VIII*, North-Holland, 1988, pp. 189-204.

[V+91] C.A. Vissers, G. Scollo, M. van Sinderen, E. Brinksma. Specification Styles in Distributed Systems Design and Verification. In: *Theoretical Computer Science 89*, North-Holland, 1991, pp. 179-206.