Deriving Protocol Specifications from Service Specifications Written in LOTOS+

Christian Kant*, Teruo Higashino** and Gregor v. Bochmann

Département d'IRO, Université de Montréal,
C.P. 6128, Succursale A, Montréal, Québec, H3C 3J7, Canada

ABSTRACT

A complete communication system is broken down into a number of protocol layers each of which provides services to the layer above it and uses services provided by its underlying layer. A service specification defines a particular ordering of the operations that a given layer provides to the layer above it. The active elements in each layer are called entities and they use a protocol in order to implement their service definition. On the basis of this relation between the service and protocol concepts we have developed algorithms for deriving protocol entity specifications from a formal service specification. The derived protocol entities ensure the correct ordering of the service primitives by exchanging synchronization messages through an underlying communication medium. This paper presents a new version of our derivation algorithms; it is an extension of the method to a more comprehensive specification language. This version of the algorithm can handle now all operators and unrestricted process invocation and recursion as defined by basic LOTOS. The correctness of the derivation algorithm is formally proved.

1. Introduction

Communication protocols within a distributed system are usually organized in a hierarchy of layers [OSI 84]. In this context two concepts are of prime importance: the communication service provided by a particular layer and the protocol to be followed by the protocol entities of that layer [Boch 90]. The relation between these two concepts is illustrated by the following architectural model (Fig. 1). A service is realized by a service provider, which is a distributed system. At this level of abstraction the system is seen as a black box offering some specified communication service to some service users. The service is available through a certain number of Service Access Points (SAP's), in the following also called places (Fig. 1-a).The service specification is a description of the temporal ordering of service primitives occurring at the SAP's.

On the protocol level, several entities PE, in general one per service access point, may cooperate to provide the required service (Fig. 1b); they exchange synchronization messages through a communication medium, using a lower level service. In the following we assume that in the communication medium there is a communication channel from each entity i to any other entity j; each communication channel is assumed to be a FIFO queue whose capacity is infinite. The channel does not lose, duplicate or insert messages; each of the messages is delivered after an arbitrary delay.

In the area of communication protocols, analysis techniques have been developed to detect design errors, such as deadlocks, unspecified receptions and non-executable interactions, and to determine whether a given protocol satisfies a given service specification. In this paper we follow another design paradigm: protocol synthesis. Since the protocol is seen as the logical implementation of the service [Viss 85], one may ask whether it is possible to formally derive, from a given service specification, a correct protocol specification which provides that service.

+ Reseach supported in part by the IDACOM-NSERC-CWARC Industrial Research Chair on Communication Protocols

* Christian Kant is with the Université de Moncton, Canada.

** Teruo Higashino is with Osaka University, Japan; he was on leave at Université de Montréal, Canada, during 1990.

Based on the above architectural model, we can phrase the question in more precise terms as follows: given a service specification e_s (see Fig. 1-a) is it possible to formally derive the specifications $PE_i(e_s)$ (see Fig. 1-b) for all protocol entities ($1 \le i \le n$) ?

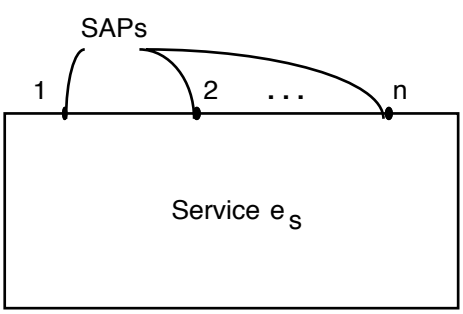

Figure 1-a : Service architecture

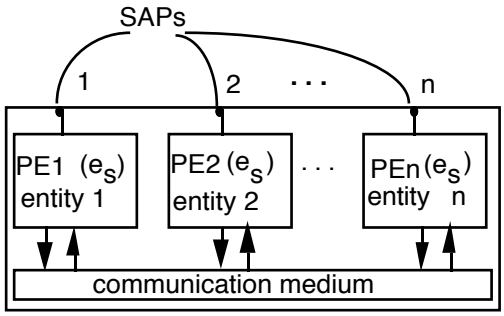

Figure 1-b : Protocol architecture

Various approaches to protocol synthesis have been described in the literature (for a survey, see [Prob 91]). Our approach, introduced in [Boch 86] and extended in [Khen 89], is the first to consider the derivation of a protocol from a given service specification without any further information. This paper shows how this approach can be extended to handle service specifications which are written in a language close to LOTOS [Lotos 89].

LOTOS is a language which has been developed within ISO for the formal specification of standard OSI protocols and services. A system is described in LOTOS as a collection of processes. A process consists by definition of a sequence of actions. A process definition will specify its behavior, i.e. the sequence of observable actions that may be executed by the process in question. LOTOS provides a set of operators which are used to specify process behaviors as expressions. The operators define in the simplest case the relative order between two actions, but they may be used also to build up complex expressions out of elementary actions and/or other expressions. A sub-language ignoring the aspects of interaction parameters and data type, sometimes called "Basic LOTOS" [Bolo 87], contains such operators as sequential execution of simple interactions ";" , choice of alternatives "[]", independent parallelism "|||", dependent parallelism with rendezvous interactions "||", sequential composition ">>" of complex expressions including process invocations, and the disabling operator "[>" which expresses the interruption of a particular sequence of actions by a disabling event, providing such an event occurs.

In the previous works we considered only a subset of the above language operators (";", "[]" and "|||"). Then in [Khen 89] we introduced the process invocation but for simplicity we restricted it to tail recursion. In this paper, important extensions are proposed which handle new operators such as "||", ">>" and "[>"; arbitrary process invocations are also allowed. It is possible now, in particular, to describe interaction sequences of the form "$(a)^n$; $(b)^n$", for $n > 0$, which are not regular.

Section 2 describes in more detail the language used for service and protocol specification. Section 3 explains the basic ideas of the protocol derivation algorithm which is described formally in Section 4. The algorithm proceeds in several steps. First, based on the syntax analysis of the service specification, a certain number of attributes are evaluated for the different nodes of the syntax tree (see Section 4.1). Then a recursive algorithm is applied which traverses the syntax tree from the top down and evaluates the protocol specification for each of the given SAP's, as explained in Section 4.2. An outline of a formal proof of the correctness of the algorithm is provided in Section 5. Section 6 contains the conclusions.

2. Specification language

The specification language supported by our derivation algorithm is functionally close to Basic LOTOS, except that the hiding of interactions is not supported. The language is used for the specification of the communication services, as well as for the derived protocol specifications. As in LOTOS, a distributed system is described in terms of processes. The whole system is a

process which is build-up of some cooperating sub-processes. Each one of these sub-processes is also build-up of some sub-sub-processes, so the system is described as a structured set of processes.

A process may execute two kinds of actions;
- internal actions, unobservable by its environment defined here by the symbol "i", and
- interactions, which are actions executed in synchronization with the environment.
A process is observable only trough its interactions, so its behavior will be formally described by an expression defining the temporal order of possible interactions. Even though we are interested in the observable behavior of a process, sometimes the use of the internal actions in a process specification might be necessary [Bolo 87].
Table 1 gives the syntax overview of the language. The format of a specification is given by rule 1, where "Def_block" defines the system behavior. Process and interaction parameters are not supported.

Table 1: Syntax of Specification language

Nr.	Syntactic rule		
(1)	Spec	-->	SPEC Def_block ENDSPEC
(2)	Def_block	-->	e WHERE Process_block
(3)	Def_block	-->	e
(4)	Process_block	-->	Process_def Process_block
(5)	Process_block	-->	Process_def
(6)	Process_def	-->	PROC Proc_Id = Def_block END
(7)	e	-->	Dis >> e
(8)	e	-->	Dis
(9)	Dis	-->	Par [> Dis
(10)	Dis	-->	Par
(11)	Par	-->	Choice |[event_subset]| Par
(12)	Par	-->	Choice ||| Par
(13)	Par	-->	Choice
(14)	Choice	-->	Seq [] Choice
(15)	Choice	-->	Seq
(16)	Seq	-->	Event_Id ; Seq
(17)	Seq	-->	Event_Id ; exit
(18)	Seq	-->	Proc_Id
(19)	Seq	-->	(e)

In Table 1 "Proc_Id" and "Event_Id" are defined using some terminal symbols; " SPEC", "ENDSPEC", "PROC", "END", "WHERE", ">>", "[>", "|[", "]|", "|||", "[]", "(", ")", ";" and "exit" are terminal symbols and "event_subset" is a set of "Event_Id".

An "Event_Id" may denote either:

- a service primitive interaction, and is written in this case as "Identifier^{place}", where "Identifier" represents the service primitive itself, and "place" defines the service access point at which the interaction takes place; for example the event "a^2" represents the service primitive "a" at the service access point 2, or
- a send_a_message interaction, written in this case as "$s_i(m)$", which means the sending of the message "m" to the place "i", or
- a receive_a_message interaction, written as "$r_j(m)$" which means the receiving of the message "m" from the place "j".

A "Proc_Id" denotes a process, and is written as "Identifier"; to distinguish between process identifiers and service primitive identifiers, the former will be written using capital letters.

The sequential composition operator ";" (rule (16)) defines a new expression by prefixing another one with an "Event_Id". For instance, the expression "a^i ; Seq" specifies a process that

executes interaction "a" at place "i", then behaves as defined by "Seq". "exit" in rule (17) specifies the successful termination of a sequence of actions.

The expression "Seq [] Choice" (see rule (14)) specifies a process that behaves either as defined by expression "Seq" or as defined by expression "Choice". Depending on the initial actions of each alternative, the choice may be resolved either by:
- the process, as in the following expression
"i ; Event_Id...[] i ; Event_Id..."
where the initial action of each alternative is an internal, unobservable action "i", or by
- the process environment, as in the expression
"Event_Id...[] Event_Id..."
where the initial action of each alternative is an interaction.

For instance, in the expression
"a^1 ; ... [] i ; b^1 ; ..."
the choice to execute either "a^1" or "b^1" is not made by the user in place 1, but by the process itself; deciding to execute in an unobservable manner the internal action "i", the process constrains afterwards the execution of "b^1". Obviously, this situation differs from that described by the expression:
"a^1 ; ... [] b^1 ; ..."
where the process offers both "a^1" and "b^1"; here it is the user ability to execute either "a^1" or "b^1" who decides the choice between those interactions.

The general parallel operator, "|[event_subset]|", as defined in rule (11), specifies that the actions of "Choice" and "Par" in the expression "Choice |[event_subset]| Par" are executed in parallel, either with synchronization, if some of their interactions are in "event_subset", or independently (without synchronization) otherwise. When the "event_subset" is empty, the parallel operator "|[event_subset]|" may be written as "|||" and expresses in this case an arbitrary interleaving of the actions of "Choice" with the actions of "Par" (see rule (12)). When "event_subset" is the set of all events in the given expression then the parallel operator "|[event_subset]|" is written "||".

In the expression "Dis >> e", the operator ">>" is called "enabling operator" (rule (7)). It is used to specify that if the process defined by "Dis" terminates successfully, then the execution of the process defined by "e" is enabled. This operator is conveniently used in conjunction with explicit process invocation: parts of a specification may be defined as separate processes (see rule(6)) and then invoked in a desired sequence (rules (16) and (18)). For instance, we can write:

```
SPEC   ( a1 ; b2 ; B ) >> ( d3 ; exit )        WHERE                    (Example 1)
PROC  B = ....  END
ENDSPEC
```

which defines the sequential execution of the actions "a^1", "b^2", followed by the execution of the actions defined for process B (process instantiation); when B terminates, the initial (calling) process resumes with the action "d^3". A process may invoke itself or be involved in mutual recursion. In this case we say that the process is recursive. An example is:

```
SPEC  A        WHERE
PROC  A = ( ai ; A >> bk ; exit ) []  ( ai ; bk ; exit )  END                (Example 2)
ENDSPEC
```

Here any instance of A may execute either "a^i ; A >> b^k" or "a^i ; b^k". If the first alternative is chosen then a new instance of A is created by the invocation of A after "a^i". If the second alternative is chosen the current instance is eventually ended and the previous invocation of A, if any, will be resumed. We conclude that this specification defines a sequence of "n" actions "a^i ; a^i ; a^i .... a^i " followed by a sequence of "n" actions "b^k ;b^k ;b^k ; ..... b^k" for some n > 0.

In describing a system, it is often necessary to specify the case of the interruption of the normal course of a process execution, by means of events signaling an abnormal situation. For this

purpose, the operator "[>" (disabling) can be used. In the expression "Par [> Dis" (see rule(9)) the process defined by "Par" may be at any time interrupted by the occurrence of the first executable event of "Dis", and thereafter, only the actions of "Dis" are executed. On the other hand, if "Par" terminates without any interruption by "Dis", then "Dis" will never be executed. We assume that the expression "Dis" in "Par [> Dis" may be written in an action prefix form as:

$$Dis = \mathop{[\,]}_{i=1..n} (Event\_Id_i ; Seq_i)$$

where "$Event\_Id_i$" are event identifiers used in "Dis". Using expansion theorems [Lotos 89] every finitely branching expression can be written in action prefix form, so with this assumption we restrict "Dis" to finitely branching expressions. Even more, we will consider in the following that, if a service specification contains disabling expressions, they are transformed in action prefix forms, before any processing by our algorithm. This allows us to formally define the above restriction on disabling expressions by the syntax of specification language: rule(9) in Table 1 is replaced by the following rules:

Table 1 (extension)

(9^1)	Dis	-->	Par [> Mc
(9^2)	Mc	-->	Pref [] Mc
(9^3)	Mc	-->	Pref
(9^4)	Pref	-->	Event_Id ; Seq

Let us consider an example. Suppose we have three users at three different places, connected through a communication service. The user at place 1 will read sequentially a file, using **read^1**, a primitive who gives access to the next record of the file, and passes it to the communication service. Actually the record should be defined as a parameter for the **read^1** operation, but as our language does not support parameters we will consider that to each interaction **read^1** it is associated a distinct record. The same observation applies to all interactions of this example. The reading of the file continues up to the end, when the user at place 1 executes the operation **eof^1**. The behavior of the user at place 1 may be described as follows:

SPEC A WHERE
PROC A = read^1 ; A [] eof^1 ; exit END
ENDSPEC

Another user at place 3 can write a file. The initial empty file is created with the primitive **make^3** and the records are written with the primitive **write^3**, which takes the last received record and writes it to the local file. The user at place 3 may also request the end of all the operations, issuing at any time the **interrupt^3** primitive. Its behavior may be described by the following specification:

SPEC make^3 ; C WHERE
PROC C = write^3 ; C [> interrupt^3 ; exit END
ENDSPEC

Now suppose that these users want to copy a file from place 1 into a file in place 3, but in reverse order. The users may use for that a third user in place 2, which can execute either **push^2**, to insert the last received record in a local stack or **pop^2** to extract the top of the stack and send it to the communication medium. The specification of this user is:
SPEC B WHERE
PROC B = push^2 ; B [] pop^2 ; B END
ENDSPEC

It is important to note that all the above primitives are executed as interaction between the communication service and the users (see Fig. 2). For instance, if the user at place 1 is ready to

execute **read**^{1}, the action won't be executed until the communication service is also ready to execute it.
In order to accomplish the proposed task, the users' operations must be defined such as:
- every record in the place 1 be read, carried to the place 2 and inserted here in the local stack;
- at the end, when all the records were pushed in the stack at place 2, the file at place 3 be created, and
- every record popped from the stack at place 2, be carried to the place 3 and inserted sequentially in the local file.

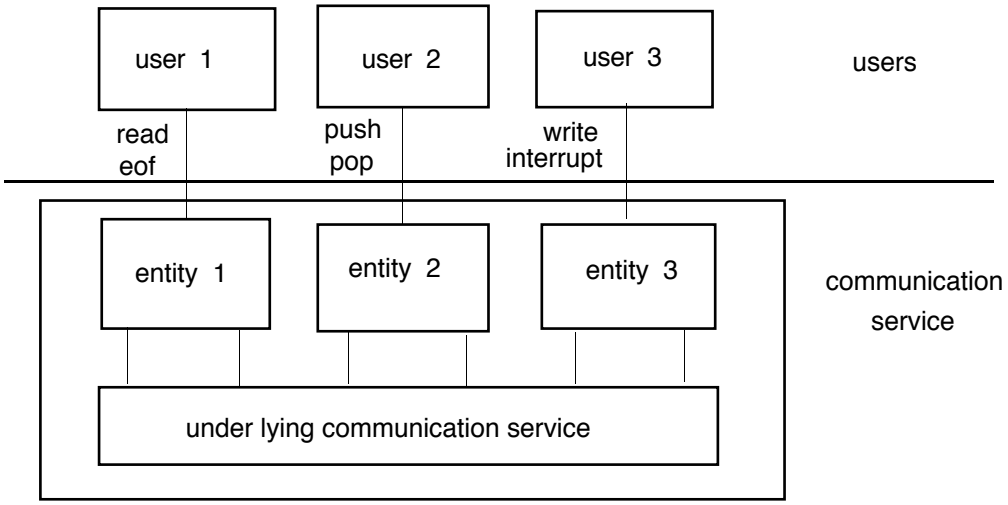

Figure 2: Example of Service

This sequence of actions might be implemented by a specific communication service, which formal specification may be defined as follows:

```
SPEC  S  [> interrupt3 ; exit                    WHERE
PROC  S =      (read1; push2; S >> pop2; write3; exit)          (Example 3)
             [] (eof1; make3; exit)              END
ENDSPEC
```

3. The principles of protocol derivation

We have studied the protocol derivation problem with different specification languages. The first results [Boch 86] were obtained for a language using only a few operators (";", "[]" and "|||"). Then a more powerful language was considered in [Khen 89], which mainly introduced the process instantiation. We present in this paper a solution for the protocol derivation problem based on the language described in Section 2, which allows the use of all operators and unrestricted process invocation and recursion as defined by basic LOTOS.

If we assume the existence of a central controller (a server PE), we can derive a trivial solution where only one PE (the server PE) has a copy of the given service specification and it informs all other PE's (client PE's) when each action should be executed by exchanging messages, and where all the client PE's execute their actions after they receive the messages from the server PE and they return a message to the server PE after each action is executed. Although this solution is simple, such a centralized control method requires many synchronization messages and the load for the server PE becomes large. In order to solve these drawbacks, we adopt the following distributed control method.

The basic idea of the protocol derivation algorithm is to get protocol entity specifications by the "projection" of the given service specification onto the service places, which means roughly:

- to select from the service specification, and assign to the protocol entity "i", only those actions defined to be local to "i", and

- to add all the necessary synchronization messages that must be exchanged between places such that the temporal order of actions at different places satisfies the order defined by the service specification,

for all places $1 \leq i \leq n$.

The algorithm attempts to preserve the syntactic structure of the service specification in every protocol entity specification, in terms of:
- processes (every protocol entity specification will consist of an equal number of process definitions, with the same names and with the same structure as in the service specification);
- parallel, choice, enable and disable operators in the process expressions.

Selecting the proper actions for each place, within a global service expression, without taking into account the need of synchronization would be a trivial task. It is clear that the central problem of the protocol derivation is to find out **when** a synchronization between the cooperating entities is necessary and **what** are the places involved in every particular synchronization. In the following we will deal intuitively with this problem.

The information concerning the need of synchronization between places is implicitly defined in the service specification by the operators linking sub-expressions, events and/or processes. For instance, in the expression "a^1 ; Seq" which means, execute first "a" at place "1" then execute the actions of "Seq", a synchronization is needed between place "1" and all places where "Seq" is initiated. We say that the action prefix operator ";" requires, or implicitly defines, a synchronization.

Not all operators require synchronization. For instance, the operator "|||" means independent parallel execution of two (sub) expressions, and so it does not set by itself any sequential constraint between its constituent actions.
For the specification language presented in Section 2 we assume that the service decomposition onto the places, may introduce an exchange of synchronization messages for any:
- action prefix ";" and sequential ">>" operators,
- choice operator "[]",
- interrupt operator "[>", and
- process instantiation.

In order to answer the question of what are the places involved in synchronization we have to consider every action and/or expression in its particular context of the service specification and try to find out what are the places related by the above operators. This can be done by assigning **attributes** to every non terminal symbol "x" in the parsing tree of the given specification, according to the context-free grammar defined in Table 1, as follows (see Section 4.1 for more details):

SP(x) the set of places where "x" is initiated, called **S**tarting **P**laces of the non terminal "x";
EP(x) the set of places where the last actions of "x" are executed, called **E**nding **P**laces of "x";
AP(x) the set of places involved in "x", called **A**ll **P**laces of "x".

In addition, a unique attribute **ALL** is defined for a given specification, which describes the set of all places involved in that specification; it is nothing else than the attribute **AP** for the start symbol "SPEC".

The above attributes are used to determine which places need to synchronize their actions. For instance, in the expression "a^1 ; Seq" we assume that after "a^1" is executed, place "1" must send some synchronization messages to the **S**tarting **P**laces of "Seq", and the **S**tarting **P**laces of "Seq" must receive these messages before any action of "Seq" is executed.

The derivation algorithm proceeds in several steps. First, based on the syntax analysis of the service specification, the attributes SP, EP and AP are evaluated for every node of the syntax tree. Then for every place "p" of the service specification a recursive function Tp is applied to the root node of the service syntax tree (see for details Section 4.2). This function traverses the tree

from the top down, selects the nodes to assign to place "p" and generates, when necessary, sending and/or receiving interactions to be inserted in the entity for place "p".

The generation of sending/receiving interactions is conducted following a set of rules, one for each syntactical rule of the specification language (see Table 3 in Section 4.2). As mentioned above, only the operators ";", ">>", "[]", "[>" and process instantiations in the service expression may generate sending/receiving actions in protocol entity expressions. Consequently the remaining of this section is dedicated to the presentation of the principles of protocol derivation for each of these cases. Section 3.1 presents the simplest and the most evident case, i.e. of a service expression containing action prefix ";" and sequential ">>" operators. Section 3.2 is concerned with the derivation of the choice expressions. Section 3.3 presents the principles of derivation for service expressions which contains the disabling operator "[>". The principles of protocol derivation from a service specification including recursion and process invocation are presented in Sections 3.4 and Section 3.5 presents some particular problems which may arise when several instances of a process are invoked.

3.1 Sequence of actions

The obvious need of synchronization is that defined for the distributed execution of some sequential actions. The specification language provides for that the action prefix operator ";" used with atomic actions (events) and the sequential operator ">>" (enabling) used with structured actions. The difference between these operators is syntactical rather than semantic. Let "e_1 op e_2" be a sequential expression, where "op" is either ";" or ">>" and e_1 and e_2 are some expressions. (If "op" is ";" then e_1 must be an event). With these operators we express a sequential constraint, i.e. allow the execution of the right side of the expression only after a complete termination of the actions of the left side. In a distributed environment, we suppose that the actions of the left side will terminate in some places, namely $EP(e_1)$, and the actions of the right side will start in some other places, namely $SP(e_2)$. To implement the sequential constraint between e_1 and e_2, every ending place of e_1 has to send a synchronization message to every starting place of e_2 and conversely every starting place of e_2 should not be allowed to proceed before receiving a synchronization message from every ending place of e_1.

These sending/receiving actions needed for the synchronized execution of a distributed sequence of actions, will be generated in our algorithm by the functions $\mathbf{Synch\_Left_p}$ and $\mathbf{Synch\_Right_p}$ (see tables 3 and 4 in Section 4.2) as follows:

Applying the function $\mathbf{T_p}$ to the given service expression "e_1 op e_2", we derive for every place "p" an expression of the form:

"$\mathbf{T_p}(e_1)$ op ($\mathbf{Synch\_Left_p}(e_1,e_2)$ >> $\mathbf{Synch\_Right_p}(e_1,e_2)$ >> $\mathbf{T_p}(e_2)$)"

where

$\mathbf{Synch\_Left_p}(e_1,e_2)$ = if ($p \in EP(e_1)$) then
send _a _message to every $SP(e_2)$
else "empty" endif

and

$\mathbf{Synch\_Right_p}(e_1,e_2)$ = if ($p \in SP(e_2)$) then
receive_a_message from every $EP(e_1)$
else "empty" endif

"empty" means that no actions are to be generated in the specified place.

As an example consider the following service expression:

SPEC ...a^1 ; exit >> b^2 ; .... ENDSPEC (Example 4)

for which we expect the following protocol specifications:

place 1: SPEC ....a^1 ; ($s_2(x)$; exit) >> (empty)..... ENDSPEC

place 2: SPEC ..... (empty) >> (r_1(x) ; exit) >> b^2 .... ENDSPEC

3.2 Choice between Alternatives

Let us consider expressions of the form "e_1 [] e_2" where e_1 and e_2 are some expressions, which specifies a choice to execute either the actions of e_1 or the actions of e_2. If this expression specifies a service, then we expect that its projection generates a similar choice expression for every place of the service specification, where each alternative will be the projection of e_1 and e_2, respectively. Consequently we are tempted to propose the following derivation rule:

$T_p(e_1 [] e_2) = T_p(e_1) [] T_p(e_2)$

Suppose now that the derivation with this rule, will generate for some particular place an empty alternative of the form "empty [] e_i" which means, "do nothing or, as an alternative, the actions as defined by e_i". This kind of expression may be generally accepted, but when the choice is followed by some other actions, we cannot anymore decide when these subsequent actions have to be started: after the execution of e_i or immediately, without executing e_i. As an example of such a situation let us consider the following service specification:

SPEC A WHERE (Example 5)
PROC A = (a^1 ; b^2 ; A >> c^2 ; d^3 ; exit) [] e^1 ; f^3 ; exit END
ENDSPEC

for which we derive with the above rule the following protocol specifications:

place 1: SPEC A WHERE
PROC A = (a^1.....; A >> .....) [] (e^1 ; ....; exit) END
ENDSPEC

place 2: SPEC A WHERE
PROC A = (..b^2... ; A >>c^2....) [] END
ENDSPEC

place 3: SPEC A WHERE
PROC A = (..... ; A >> ..d^3..) [] (.....; f3 ; exit) END
ENDSPEC

Suppose that place 1, decides to execute a^1. Place 2 will then execute b^2 and following the recursive call to A, place 1 is again required to make a choice. The decision to execute a^1 may be taken several times. Eventually, place 1 may decide to execute e^1 in the right alternative. As the place 2 has not actions defined for this alternative, the derived corresponding expression will be strictly empty, meaning no actions. Consequently the choice made by the place 1 to execute e^1, and implicitly to stop the recursive calls of A, would not be reflected in the place 2. In this case place 2 will never be able to execute c^2, which follows each recursive call of A, because it does not know about the choice made by the place 1. What is required here is the reception of a synchronization message from the place 1, allowing place 2 to continue. We define a general requirement of non-empty alternatives in protocol specification.

This requirement may be easily implemented by a message exchange initiated by the first place of any alternative, which has the purpose to send synchronization messages to all places of the choice expression which do not participate in the alternative. The non-participating places may be determined using the attributes AP (All Places) as follows:

- the non-participating places in the left alternative are AP(e_2) - AP(e_1);
- the non-participating places in the right alternative are AP(e_1) - AP(e_2);

The sending/receiving actions for choice synchronization are generated during the protocol derivation by the function **Alternative** (see Tables 3 and 4 in Section 4.2). Applying the

function $\mathbf{T_p}$ to the service expression "e_1 [] e_2", we derive for every place "p" an expression of the form:

"($\mathbf{T_p}(e_1)$ >> $\mathbf{Alternative_p}(e_1,e_2)$) [] ($\mathbf{T_p}(e_2)$ >> $\mathbf{Alternative_p}(e_2,e_1)$)"

where

$\mathbf{Alternative_p}(u,v)$ **:=** if (p ∈ SP(u)) then
send _a _message to every AP(v) - AP(u)
else
if (p ∈ (AP(v) - AP(u)) then
receive_a_message from every SP(u)
else "empty" endif
endif

For Example 5 we expect protocol specifications of the form:

place 1: SPEC A WHERE
PROC A = (a^1.....; A >> .....) [] (e^1 ; ....; exit) >> (s2(x) ; exit) END
ENDSPEC

place 2: SPEC A WHERE
PROC A = (..b^2... ; A >>c^2....) [] (r1(x) ; exit) END
ENDSPEC

place 3: SPEC A WHERE
PROC A = (..... ; A >> ..d^3..) [] (.....; f3 ; exit) END
ENDSPEC

With our method two restrictions are imposed for the choice expressions of the form "e_1 [] e_2" :

R1 : SP(e_1) = SP(e_2) = {p} , for some p ∈ ALL
R2: EP(e_1) = EP(e_2).

With an expression "e_1 [] e_2" we specify that the service provider offers both the starting events of "e_1" and "e_2", but only one of them must be executed, depending on the choice made by the environment (the users of the service). Once the choice is made, the alternative event should not be offered any more. If SP(e_1) and SP(e_2) are different, the choice becomes more complicated, because we cannot "disable" instantly the not chosen alternative. Suppose the user in SP(e_1) chooses to execute e_1; a synchronization message to prevent the execution of e_2 is not effective, as would be possible for the user in SP(e_2) to make his choice before the message arrives. Restriction R1 simplifies the implementation of the decision of which alternative should be executed: the choice is made locally in the entity where the alternative starts. Possibilities for removing this restriction are discussed in [Kant 92, Kant 93].

Restriction R2 was defined in [Khen 89] in order to simplify the derivation algorithm. To illustrate, let "(e_1 [] e_2) >> e_3" be a service expression and EP(e_1) ≠ EP(e_2). In this case EP(e_1 [] e_2) cannot be any more expressed as a simple set of places and the synchronization of "(e_1 [] e_2)" with "e_3" becomes more complicated.

3.3 Disabling

Let us consider the disabling operator "[>" in an expression of the form "e_1 [> a^i ; e_2" where "a^i" is an event and "e_1" and "e_2" are some expressions. The LOTOS semantics for this expression defines two properties:
(a) An occurrence of the event "a^i" is possible any time and shall interrupt the execution of "e_1", that is, after the execution of "a^i", no further event of "e_1" may be executed.
(b) If the sequence "e_1" terminates without any interruption by the disabling event "a^i", then "a^i ; e_2" will never be executed.

This semantics is based on the principle of interleaving of individual interactions, which is not a natural model in a distributed environment. Actually, it turns out that this semantics is very difficult to be precisely implemented in a distributed environment, as discussed below.

In the following, we define a distributed implementation of the disabling operator with a slightly modified semantics, which seems to be more intuitive for a distributed environment. The idea of this implementation is to broadcast a signal (message) from the place "i" to all places involved in "e_1", when the execution of "a^i" occurs.

A problem may arise when these actions are executed in a distributed system, where every "p" of the service specification has to execute a sequence of the form "u [> v". It is clear that we should not allow places to freely terminate their "u" sequence, because if this happens the corresponding sequence "v" will be erased. If free termination would be allowed, then for some places the sequence "v" may be erased while for others, it may still be possible. In other words, the execution of the interruption sequence might not be possible for all places, would the disabling event occur. For instance, in Example 6 below, place 1 should not be allowed to terminate before the place 3 executes "c^3". It is clear that the protocol entities must synchronize the termination of "normal" actions, defined by "u". These sending/receiving actions for termination synchronization are generated during the protocol derivation by the function **Rel** (see Tables 3 and 4 in Section 4.2) as follows:

Applying the function $\mathbf{T_p}$ to the service expression "e_1 [> a^i ; e_2" , we derive for every place "p" an expression of the form:

"($\mathbf{T_p}(e_1)$ >> $\mathbf{Rel_p}(e_1)$) [> $\mathbf{T_p}(a^i ; e_2)$ "

where

$\mathbf{Rel_p}(e_1)$:= if (p ∈ EP(e_1)) then
send_a_message to (ALL- p)
"|||"
receive_a_message from (EP(e_1) - p)
else
receive_a_message from (EP(e_1) - p)
endif

$\mathbf{T_p}(a^i ; e_2)$:= $\mathbf{T_p}(a^i)$; ($\mathbf{Interr_p}(a^i)$ >> $\mathbf{Synch\_Left_p}(a^i, e_2)$ >>
$\mathbf{Synch\_Right_p}(a^i, e_2)$ >> $T_p(e_2)$)

$\mathbf{Interr_p}$(e) := if (p ∈ SP(e)) then
send_a_message to (ALL - SP(e))
else
if (p ∈ (ALL - SP(e)) then
receive_a_message from SP(e)
else "empty" endif
endif

$\mathbf{Synch\_Left_p}$ and $\mathbf{Synch\_Right_p}$ will generate synchronization messages for the ";" operator, as discussed in Section 3.1. As an example, we consider the following service specification:

PROC A = (a^1 ; b^2 ; c^3 ; exit) [> (d^3 ;.... exit) END (Example 6)

for which we expect the following protocol specifications :

place
1 PROC A=a^1;..... >> (r_3(x);exit) [> (r_3(y);exit) .....END
2 PROC A=..;b^2;.. >> (r_3(x);exit) [> (r_3(y);exit) .....END
3 PROC A=...;c^3;exit >> (s_1(x);exit ||| s_2(x);exit) [> d^3; (s_1(y);exit ||| s_2(y);exit) .....END

The above protocol specifications show that the place 3 may initiate an interruption, as stated by the service. When the interrupt event d^3 occurs, the place 3 sends a specific message to all other places. On reception, these places will interrupt their "normal" execution and execute their interrupt sequence, if any.

As for the choice expressions, there are some restrictions imposed for the disabling expressions of the form "e_1 [> e_2". First, restriction R2 is extended to disabling expressions and a new restriction R3 is added, as follows:

R2: $EP(e_1) = EP(e_2)$
R3: $EP(e_1) \supset SP(e_2)$

The disrupting event of "e_2" may occur just before the last action in "e_1", and if $SP(e_2) \not\subset EP(e_1)$, we may have a problem similar to the choice between actions executable on different places. Restriction R3 is introduced in order to simplify the implementation of this control.

As noted earlier, this distributed implementation of the disabling operator does not exactly satisfy the LOTOS semantics. It has the following shortcomings:
(i) Property (b) is not satisfied. For the example given at the beginning of this subsection, the execution of "e_1" may terminate before the message signaling the event "a^i" arrives.
(ii) Property (a) is satisfied only approximately due to message delays. For instance, an event of "e_1" may occur after the execution of the event "a^i", because of the message signaling the occurrence of "a^i" has not yet arrived.

Nevertheless, we consider that the above implementation of the disabling operator is reasonable for a distributed environment. First of all, in most cases where the disabling operator is used (for instance, for the disconnecting the data transfer phase of a communication protocol), the expression "e_1" does not terminate. Therefore shortcoming (i) is not relevant. Secondly, in a distributed environment, a more intuitive semantics of the operator should be based on partial-order semantics which captures the cause-effect relationships between the different events in the system (see for instance [Viss 90, Coel 92]). Such a semantics may well be defined in such a manner as to correspond to the implementation described above.

An alternative implementation of interruption that avoids the problem of the message delay could be based on the principle that before "a^i" can be executed, a request for interruption must be issued first. This request is followed by messages sent to all involved sites to interrupt the progress of the events belonging to "e_1" and to return an acknowledgment. When all these acknowledgments are received the interrupt event "a^i" may occur. Such an implementation would satisfy properties (a) and (b), as defined above, and would satisfy trace equivalence with the original LOTOS semantics. However, it would still not be testing equivalent nor observationally congruent to the original LOTOS semantics, as indicated by the following example. Assume that "e_1" consists of concurrent repeated executions of interactions b and c on two different sites. Due to different message delays, the proposed implementation could lead to a system state when b has already been interrupted, while c is still possible. The original LOTOS semantics excludes such a state, because the occurrence of "a^i" interrupts the events at all sites simultaneously.

3.4 Process Invocation and Recursion

Let us consider Example 2, given in Section 2.

```
SPEC A    WHERE
PROC A =  a^i ; A >> b^k ; exit  []  a^i ; b^k ; exit  END
ENDSPEC
```

This specification defines the sequence $(a^i)^n$; $(b^k)^n$, (for same n > 0). Let us follow the execution of this expression and suppose, for instance, that at the beginning, the first alternative (at the left) of the operator "[]" is chosen: after the event "a^i", a new instance of A (say A') is activated. Again, if the first alternative is chosen, another instance of A (say A") is activated. Suppose now that for this new instance, the second alternative (at the right) of the operator "[]" is chosen; this means that process A" will terminate with the execution of sequence "$a^i;b^k$;exit".

Since this process is terminated, process A' will resume in exactly the point where it was interrupted by the activation of A'', i.e. with the event "b^k" specified after the ">>" operator. Process A' eventually ends, and the process A is now reactivated, and "b^k" is again executed.

As stated before, the derivation is defined as a projection of the service expression onto the service places. It is natural to assume that every protocol expression will have the same structure as the service expression, i.e. will define a recursive execution of a local process (see, for instance, Fig. 3).

We have to provide the synchronization between processes A, one in place i and the other in place k. Suppose we have already defined some synchronization between distributed actions, implying that if two successive actions are executed in two different places then the first place will send a message to the second as soon as it finishes its action, and the second will wait for the message from the first one before it starts its own action (see Section 3.1). In this way the synchronization between the pair processes A (in place i and k) is implicitly done. But if these two places are not both involved in actions required before the invocation of A, as in Example 2, the implicit synchronization of processes (at the action level) is not possible anymore, and some new synchronization must be introduced. We call it "synchronization at the process level". For the purpose of this paper we define the process synchronization as taking place at every process invocation.

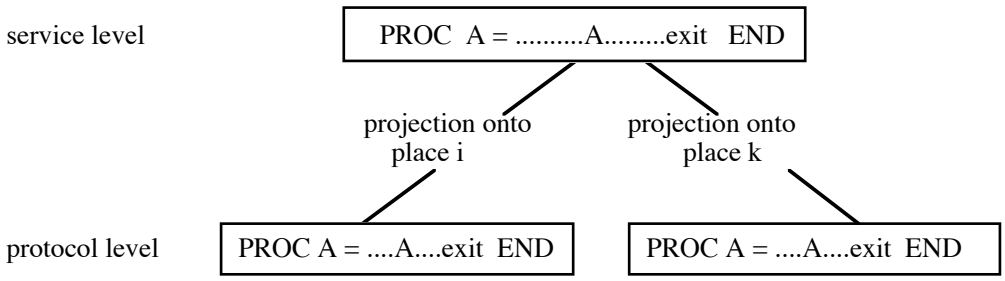

Figure 3 Projection of the service on a set of places

More precisely, a process invocation "A" in a service expression "e" will generate in every place "p" a local process invocation "A", preceded either by:

- sending messages to every place in (ALL - SP(A)), if ($p \in SP(A)$), or by
- receiving a message from SP(A), otherwise.

These sending/receiving actions for process synchronization are generated during the protocol derivation by the function **Proc_Synch**, as defined below (see Tables 3 and 4 in Section 4.2).

Applying the function $\mathbf{T_p}$ to the service expression "A" , we derive for every place "p" an expression of the form:

"($\mathbf{Proc\_Synch_p}$(A) >> A)"

where

$\mathbf{Proc\_Synch_p}$(e) **:=** if ($p \in SP(e)$) then
send_a_message to every (ALL - SP(e))
else
receive_a_message from every SP(e)
endif

For Example 2, for instance, we expect to obtain protocol entities of the following form:

place i : SPEC A WHERE
PROC A = a^i ; $s_k(x)$; A >> .....exit [] .....exit END
ENDSPEC

place k : SPEC A WHERE

PROC A = $r_i(x)$; A >> .....exit [] .... exit END
ENDSPEC

3.5 Multiple instances of a process

Let us consider the following example:

SPEC B ||| B WHERE
PROC B = (a^1 ; (b^2 ; exit ||| c^3 ; exit)) >> g^4 ; exit END (Example 7)
ENDSPEC

In every place, there are two instances of B; let us call them, respectively, the left instance of B and the right instance of B. In order to execute "g^4" (in any B process), place 4 must get synchronization messages from the preceding places, i.e. from places 2 and 3.

Now, suppose that for the left instance of B, after "a^1" is executed, "b^2" is executed and then a synchronization message is sent from place 2 to place 4 ("c^3" will be subsequently executed). For the right B, after "a^1" is executed, "c^3" is executed and then a synchronization message is sent from place 3 to place 4 ("b^2" will be subsequently executed).

Consequently, place 4 receives two synchronization messages, one from place 2, and the other from place 3. Should this place execute now "g^4", as specified in the definition of B, and if so, in what process B? (Remember, there are two in every place.) The problem here is that the synchronization messages do not identify the originator process. The situation may get worse if process B is recursive, as in the following specification:

SPEC A WHERE
PROC A = (a^1 ; ... A [> b^2 ; d^1 ; exit) [] c^1 ; ... exit END (Example 8)
ENDSPEC

Here, several instances of A may be created, as specified by the recursive expression of the process A; disrupting interactions "b^2" are defined for every one of these instances. The recursive creation of A will end when, in the last instance of A, the place 1 will chose to execute "c^1". In the case a "b^2" event occurs, as shown in section 3.2, a message must be sent from the place 2 to all places in the disrupting expression "(a^1 ; ...A [> b^2 ; d^1 ; exit)". But there are possibly several instances of A, and as the system cannot discriminate between instances of the same process, it cannot properly decide which of these instances must be interrupted.

We conclude that the identification of the originator process in every synchronization message is required. This may be done by a parameterization of every process instance with an occurrence number "s". At every process invocation, which corresponds to the creation of a new instance of the process, the variable "s" is modified (using some numbering scheme that generates unique process numbers) and the new process will receive, as a dynamic attribute, the current value of "s". Every send_a_message or receive_a_message action will be also parameterized with the process occurrence number attribute . If the specification does not contain any explicitly defined process, as in the following case:

SPEC a1 ; b2 ; exit ENDSPEC

then the messages are parameterized with the default occurrence number "0".

4. Definition of the protocol derivation algorithm

The algorithm producing the protocol specifications proceeds through the following steps:

Step 1: Construct the derivation tree for the given service specification. (Transform, if necessary, the disabling expressions in action prefix form - see Section 2).
Step 2: Synthesize attributes SP, EP and AP at each node of the tree (see Section 4.1).
Step 3: For each place p (p = 1..n), apply the function T_p, as defined in Table 3, to the root node of the service specification derivation tree.

The result, for each place, is a character string in the form of a specification containing a set of processes with the same identifiers as in the original service specification; however, the definition of their bodies is changed. Only the service interactions occurring at the place for which the protocol entity is derived will be included in the protocol specification and additional statements for the sending and receiving synchronization messages will be added.

4.1. Attribute evaluation

The second step of the protocol derivation algorithm is the evaluation of attributes for every node of the syntax tree of a given specification. The attributes associated with each node "x" in the syntax tree are SP(x), EP(x) and AP(x) as introduced in Section 3.

The grammar of the specification language presented in Table 1 may be augmented so that attribute evaluation rules are given for each rule of the syntax. These new rules are presented in Table 2. Here the attribute evaluation rules are listed using the convention that the right-hand side of each equation is the definition of the left-hand side. For some rules, subscripts have been used to distinguish between occurrences of the same non terminals. The attributes SP, EP and AP for a non terminal symbol are defined in terms of the attributes of its immediate descendants; therefore they are called synthesized attributes (see [Aho 85] for details on attribute grammars).

Two kinds of leaf nodes may exist. For the leaf nodes generated for the Event_Id symbol the attributes SP, EP and AP are set to the value {place(Event-Id)}. Here "place" is a function from the set of service primitives to the set of places: place(a^i) = i.

The attributes SP, EP and AP of a leaf node corresponding to a process reference (generated by rule (18)) can be considered variables. We equate the variables of such a leaf node, for instance A, with the values obtained by synthesis for the node Process_def corresponding to the same process identifier A. Therefore, the evaluation of SP for a given process, gives rise to an equation defining SP in terms of places corresponding to the explicit events defined in that process, and in terms of variables representing the SP values of those processes which are invoked; and similarly for EP and AP. These equations, which are in general recursive, can be solved by applying the rule that the equation SP(A) := SP(A) $\cup$ X implies the solution SP(A) := X, where SP(A) is the value of the SP attribute for process identified by A.

An iterative method may also be applied to solve these recursive equations. The bottom-up attribute evaluation is performed several times, each pass representing a step of the iteration. For the first step, the values of the SP, EP and AP attributes of the process leaf nodes are set to the empty set. In each subsequent step, the values of these attributes are set equal to the synthesized values obtained for the corresponding process definition nodes. The iteration terminates when the attribute values of all process root nodes have not changed during the last step.

For Example 3 in Section 2, the syntax tree is illustrated in Fig. 4. Some of the attributes are expressed in terms of the variables SP(S), EP(S) and AP(S). We find immediately SP(S) = {1}, EP(S) = {3} and AP(S) = {1,2,3}.
One more attribute must be added to the above list. The synchronization messages received by a given place must be identified with their "purpose", i.e. to which local action are they related. The algorithm will generate these messages in an incremental mode, applying the function $\mathbf{T_p}$ to the root node of the service specification derivation tree, and then to every node of that tree. By generalization we may say that the synchronization is defined between nodes of the derivation tree. A unique identification of the nodes is necessary and this may be done by an attribute **N** associated with each node "x" of the syntax tree. **N(x)** may be, for instance, an integer obtained by numbering the nodes of the tree in a preorder traversal scheme.

Table 2: Attribute Evaluation Rules

Nr.	Syntactic rule			Attribute SP (Starting Places)
(1)	Spec	-->	SPEC Def_block ENDSPEC	SP(Spec) = SP(Def_block)
(2)	Def_block	-->	e WHERE Process_block	SP(Def_block) = SP(e)
(3)	Def_block	-->	e	SP(Def_block) = SP(e)
(4)	$Process\_block_1$	-->	Process_def $Process\_block_2$	$SP(Process\_block_1)$ = SP(Process_def)
(5)	Process_block	-->	Process_def	SP(Process_block) = SP(Process_def)
(6)	Process_def	-->	PROC Proc_Id = Def_block END	SP(Process_def) = SP(Def_block)
(7)	e_1	-->	Dis >> e_2	$SP(e_1)$ = SP(Dis)
(8)	e	-->	Dis	SP(e) = SP(Dis)
(9^1)	Dis	-->	Par [> Mc	SP(Dis) = SP(Par) ∪ SP(Mc)
(9^2)	Mc_1	-->	Pref [] Mc_2	$SP(Mc_1)$ = SP(Pref) = $SP(Mc_2)$
(9^3)	Mc	-->	Pref	SP(Mc) = SP(Pref)
(9^4)	Pref	-->	Event_Id ; Seq	SP(Seq) = {place(Event_Id)}
(10)	Dis	-->	Par	SP(Dis) = SP(Par)
(11)	Par_1	-->	Choice |[event_subset]| Par_2	$SP(Par_1)$ = SP(Choice) ∪ $SP(Par_2)$
(12)	Par_1	-->	Choice ||| Par_2	$SP(Par_1)$ = SP(Choice) ∪ $SP(Par_2)$
(13)	Par	-->	Choice	SP(Par) = SP(Choice)
(14)	$Choice_1$	-->	Seq [] $Choice_2$	$SP(Choice_1)$ = SP(Seq) = $SP(Choice_2)$
(15)	Choice	-->	Seq	SP(Choice) = SP(Seq)
(16)	Seq_1	-->	Event_Id ; Seq_2	$SP(Seq_1)$ = {place(Event_Id)}
(17)	Seq	-->	Event_Id ; exit	SP(Seq) = {place(Event_Id)}
(18)	Seq	-->	Proc_Id	SP(Seq) = see **Note** at the end
(19)	Seq	-->	(e)	SP(Seq) = SP(e)

	Attribute EP (Ending Places)	Attribute AP (All Places)
(1)	EP(Spec) = EP(Def_block)	AP(Spec) = AP(Def_block)
(2)	EP(Def_block) = EP(e)	AP(Def_block) = AP(e)
(3)	EP(Def_block) = EP(e)	AP(Def_block) = AP(e)
(4)	$EP(Process\_block_1)$ = $EP(Process\_block_2)$	$AP(Process\_block_1)$ = AP(Process_def) ∪ $AP(Process\_block_2)$
(5)	EP(Process_block) = EP(Process_def)	AP(Process_block) = AP(Process_def)
(6)	EP(Process_def) = EP(Def_block)	AP(Process_def) = AP(Def_block)
(7)	$EP(e_1)$ = $EP(e_2)$	$AP(e_1)$ = AP(Dis) ∪ $AP(e_2)$
(8)	EP(e) = EP(Dis)	AP(e) = AP(Dis)
(9^1)	EP(Dis) = EP(Par) = EP(Mc)	AP(Dis) = AP(Par) ∪ AP(Mc)
(9^2)	$EP(Mc_1)$ = EP(Pref) = $EP(Mc_2)$	$AP(Mc_1)$ = AP(Pref) ∪ $AP(Mc_2)$
(9^3)	EP(Mc) = EP(Pref)	AP(Mc) = AP(Pref)
(9^4)	EP(Pref) = EP(Seq)	AP(Pref) = {place(Event_Id)} ∪ AP(Seq)
(10)	EP(Dis) = EP(Par)	AP(Dis) = AP(Par)
(11)	$EP(Par_1)$ = EP(Choice) ∪ $EP(Par_2)$	$AP(Par_1)$ = AP(Choice) ∪ $AP(Par_2)$
(12)	$EP(Par_1)$ = EP(Choice) ∪ $EP(Par_2)$	$AP(Par_1)$ = AP(Choice) ∪ $AP(Par_2)$
(13)	EP(Par) = EP(Choice)	AP(Par) = AP(Choice)
(14)	$EP(Choice_1)$ = EP(Seq) = $EP(Choice_2)$	$AP(Choice_1)$ = AP(Seq) ∪ $AP(Choice_2)$
(15)	EP(Choice) = EP(Seq)	AP(Choice) = AP(Seq)
(16)	$EP(Seq_1)$ = $EP(Seq_2)$	$AP(Seq_1)$ = {place(Event_Id)} ∪ $AP(Seq_2)$
(17)	EP(Seq) = {place(Event_Id)}	AP(Seq) = {place(Event_Id)}
(18)	EP(Seq) = see **Note** at the end	AP(Seq) = see **Note** at the end
(19)	EP(Seq) = EP(e)	AP(Seq) = AP(e)

Note: The definition of the synthesized attributes for the non terminal symbol Seq in rule (18) is different from the standard attribute evaluation scheme, as explained in Section 4.1.

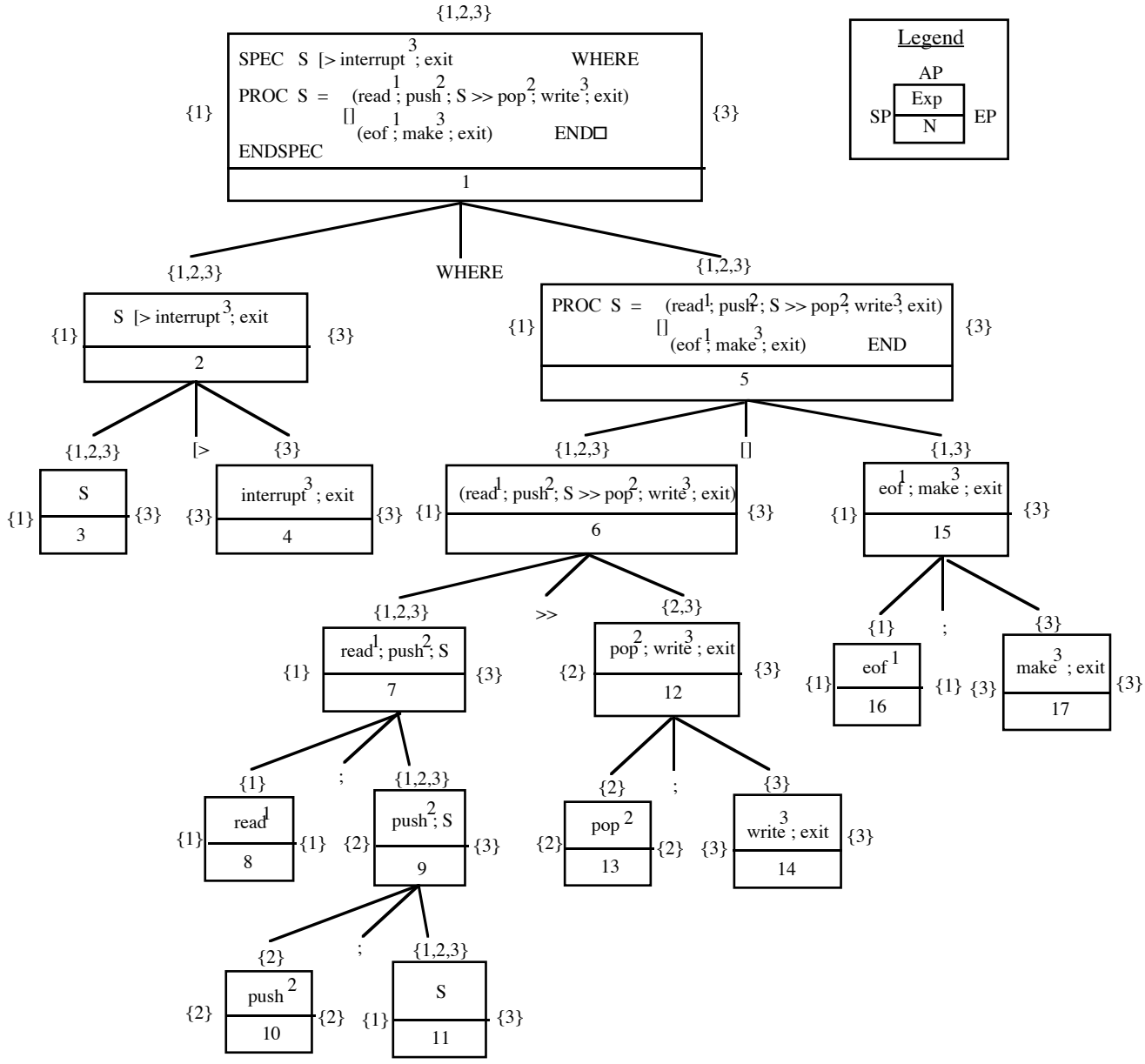

Figure 4 Derivation tree for the specification of Example 3

4.2. Protocol derivation

Tables 3 and 4 give the derivation rules $\mathbf{T_p}$ corresponding to the syntactic rules of the specification language, used in Step 3 of the algorithm. For the specification of each protocol entity we obtain a set of processes with the same identifiers as in the original service specification. For a given place only local events are selected from the service specification; in addition synchronization messages are provided, depending on the context defined by the syntax. Sending/receiving synchronization messages are defined for the sequential operators ";" and ">>", for the disabling operator "[>", for alternative operator "[]" and for process invocation. No synchronization is required for the parallel ("|||" and "|[event_subset]|") operators. The derivation rules, as defined in Table 3, may lead sometimes to a local exchange of messages, for synchronization purposes. [Khen 89] presents some methods to eliminate non-essential messages. The string "empty" represents no action and can be eliminated using the following rules:

"empty ; e" = "e"
"empty >> e = "e"
"e >> empty = "e"
"e ||| empty" = "e"

A Protocol Generator (PG), written in Prolog, has been realized on the basis of the derivation algorithm presented in this section. The PG automatically eliminates un-necessary or irrelevant sequences of characters (as, for instance, the above sequences containing the string "empty") in the derived protocol specifications. Experiments made on several case studies, including a Transport Service Specification [Kant 93], have demonstrated the PG effectiveness. In order to ensure a proper generation of protocol entities, the prototype checks the syntax of the given service specification and its conformance to the restrictions R1, R2 and R3 defined in Section 3. However no automatic decision is taken, nor any suggestion is given on how the user has to proceed to transform its service specification into an acceptable input specifications for the PG. Some possibilities to remove the restrictions are discussed in [Kant 92, Kant 93].

For Example 3 of Section 2 we obtain the following protocol specification, parameterized by the place p:

SPEC (((**Proc_Synch$_p$**(A) >> A) >> **Rel$_p$**(A)) [> (**Proj$_p$**(interrupt^3) ; (**Interr$_p$**(interrupt^3)) WHERE
PROC A =((**T$_p$**(read^1;push^2;A) >> **Synch_Left$_p$**((read^1;push^2;A),(pop^2;write^3;exit)) >>
Synch_Right$_p$((read^1;push^2;A),(pop^2;write^3;exit)) >>
(**T$_p$**(pop^2;write^3;exit)) >> **Alternative$_p$**(read^1;push^2;A >> pop^2;write^3;exit))
[] (**T$_p$**(read^1;write^3;exit) >> **Alternative$_p$**(read^1;push^2;A >> pop^2;write^3;exit)) END
ENDSPEC

Using the definitions of Table 4 and applying the rules of Table 3 again, we obtain finally for the different places the following protocol entity specifications:

Place_1

SPEC (((s2(1);exit ||| s3(1);exit) >> A) >> (r3(1);exit)) [> (r3(2);exit) WHERE
PROC A= (read^1;((s2(6);exit) >> (r2(7);exit) >>
(s2(8);exit ||| s3(8);exit) >> A))
[]
(read^1; (s3(16);exit) >> (s2(19);exit)) END
ENDSPEC

Place2

SPEC (((r1(1);exit) >> A) >> (r3(1);exit)) [> (r3(2);exit) WHERE
PROC A = (((r1(6);exit) >> push^2;((s1(7);exit) >>
(r1(8);exit) >> A)) >> (r3(10);exit) >> pop^2; (s3(11);exit))
[]
(r1(19);exit) END
ENDSPEC

Place3

SPEC (((r1(1);exit) >> A) >> (s1(1);exit ||| s2(1);exit)) [> (interrupt^3; (s1(2);exit ||| s2(2);exit))
WHERE
PROC A= (((r1(8);exit) >> A) >> (s2(10);exit) >> (r2(11);exit) >> write^3;exit)
[]
((r1(16);exit) >> write^3;exit) END
ENDSPEC

Table 3. Function Tp

Nr.			Syntactic rule
(1)	Spec	-->	SPEC Def_block ENDSPEC
(2)	Def_block	-->	e WHERE Process_block
(3)	Def_block	-->	e
(4)	$Process\_block_1$	-->	Process_def $Process\_block_2$
(5)	Process_block	-->	Process_def
(6)	Process_def	-->	PROC Proc_Id = Def_block END
(7)	e_1	-->	Dis >> e_2
(8)	e	-->	Dis
(9^1)	Dis	-->	Par [> Mc
(9^2)	Mc_1	-->	Pref [] Mc_2
(9^3)	Mc	-->	Pref
(9^4)	Pref	-->	Event_Id ; Seq
(10)	Dis	-->	Par
(11)	Par_1	-->	Choice |[event_subset]| Par_2
(12)	Par_1	-->	Choice ||| Par_2
(13)	Par	-->	Choice
(14)	$Choice_1$	-->	Seq [] $Choice_2$
(15)	Choice	-->	Seq
(16)	Seq_1	-->	Event_Id ; Seq_2
(17)	Seq	-->	Event_Id ; exit
(18)	Seq	-->	Proc_Id
(19)	Seq	-->	(e)
			Derivation rule
(1)	T_p(Spec)	:=	"SPEC" T_p(Def_block) "ENDSPEC"
(2)	T_p(Def_block)	:=	T_p(e) "WHERE" T_p(Process_block)
(3)	T_p(Def_block)	:=	T_p(e)
(4)	$T_p(Process\_block_1)$	:=	T_p(Process_def) $T_p(Process\_block_2)$
(5)	T_p(Process_block)	:=	T_p(Process_def)
(6)	T_p(Process_def)	:=	"PROC" Proc_Id "=" T_p(Def_block) "END"
(7)	$T_p(e_1)$	:=	T_p(Dis) ">>" **Synch_Left$_p$**(Dis,e_2) ">>" **Synch_Right$_p$**(Dis,e_2) ">>" $T_p(e_2)$
(8)	T_p(e)	:=	T_p(Dis)
(9^1)	Tp(Dis)	:=	"((" T_p(Par) ")>>(" **Rel$_p$**(Par) ^)) [> (" T_p(Mc) ")"
(9^2)	$T_p(Mc_1)$	:=	"(" T_p(Pref) ">>" **Alternative$_p$**(Pref,Mc_2) ") [] (" $T_p(Mc_2)$">>" **Alternative$_p$**(Mc_2,Pref) ")"
(9^3)	T_p(Mc)	:=	T_p(Pref)
(9^4)	T_p(Pref)	:=	**Proj$_p$**("Event_Id") "; (" **Interr$_p$**(Event_Id,Seq) ">>" **Synch_Left$_p$**(Event_Id,Seq)">>" **Synch_Right$_p$**(Event_Id,Seq)">>" T_p(Seq) ")"
(10)	T_p(Dis)	:=	T_p(Par)
(11)	$T_p(Par_1)$	:=	T_p(Choice) "|[" **Select$_p$**(event_subset) "]|" $T_p(Par_2)$
(12)	$T_p(Par_1)$	:=	T_p(Choice) "|||" $T_p(Par_2)$
(13)	T_p(Par)	:=	T_p(Choice)
(14)	$T_p(Choice_1)$	:=	"(" T_p(Seq) ">>" **Alternative$_p$**(Seq,$Choice_2$) ") [] (" $T_p(Choice_2)$ ">>" **Alternative$_p$**($Choice_2$,Seq) ")"
(15)	T_p(choice)	:=	T_p(Seq)
(16)	$T_p(Seq_1)$	:=	**Proj$_p$**("Event_Id") ";" "(" **Synch_Left$_p$**(Event_Id,Seq_2) ">>" **Synch_Right$_p$**(Event_Id,Seq_2) ">>" $T_p(Seq_2)$ ")"
(17)	T_p(Seq)	:=	**Proj$_p$**("Event_Id") "; exit"
(18)	T_p(Seq)	:=	"(" **Proc_Synch$_p$**(Proc_Id) ">>" Proc_Id ")"
(19)	T_p(Seq)	:=	"(" T_p(e) ")"

Table 4: Functions used in T_p

Synch_Left$_p$ (e_1,e_2)	:=	if (p ∈ EP(e_1)) then **send** ((SP(e_2) - {p}),N(e_1)) else "empty" endif
Synch_Right$_p$ (e_1,e_2)	:=	if (p ∈ SP(e_2)) then **receive** ((EP(e_1) - {p}), N(e_1)) else "empty" endif

Rel$_p$ (e)	:=	if (p ∈ EP(e)) then
		send ((ALL - p), N(e))
		"|||"
		receive ((EP(e) - p),N(e))
		else
		receive (EP(e),N(e))
		endif

Interr$_p$ (e_1,e_2)	:=	if (p ∈ SP(e_1)) then
		send ((ALL - SP(e_1) - SP(e_2)), N(e_1))
		else
		if (p ∈ (ALL - SP(e_1) - SP(e_2)) then
		receive (SP(e_1), N(e_1))
		else "empty" endif
		endif

select$_p$ (set)	:=	if set = {} then {}
		else if (there exists a member "e" ∈ set and place(e) = p) then
		{e} ∪ **select$_p$**(set-{e})
		else
		select$_p$(set-{e})
		endif
		endif

Proj$_p$(e)	:=	if (p = place(e)) then e else "empty" endif

Alternative$_p$ (e_1,e_2)	:=	if (p ∈ SP(e_1)) then
		send ((AP(e_2)-AP(e_1)),N(e_1))
		else
		if (p ∈ (AP(e_2) - AP(e_1))) then
		receive (SP(e_1),N(e_1))
		else "empty" endif
		endif

Proc_Synch(e)	:=	if (p ∈ SP(e)) then
		send (ALL - SP(e),N(e))
		else
		receive(SP(e), N(e))
		endif

send (P,N)	:=	if P = {} then "empty" if P = {i,j,...k} then "(s_i (s,N);exit ||| ..||| s_k(s,N);exit)"
receive(P,N)	:=	if P = {} then "empty" if P = {i,j,...,k} then "(r_i(s,N);exit ||| ...||| r_k(s,N);exit)"

4.3 Message complexity

An interesting point of discussion is the amount of synchronization messages generated by our algorithm. The factor which directly determines the number of synchronization messages is the number of places in the service specification. Let "n" be the number of elements of the set ALL. The algorithm will generate:

- for each operator ";" or ">>" at most one message;
- for each operator "[]" at most "n" messages (the worst case is when the sets of places in alternatives are disjoint);

- for each operator "[>":
  with the function **Rel**, at most "n - 1" messages, and
  with the function **Interr**, at most "n - 2" messages, giving a total of at most "2n - 3" messages;
- for every process instantiation at most "n - 1".

The algorithm does not generate any message for the parallel operator, but each parallel expression may be a multiplication factor of 2 for the messages sent to it or received from it. For instance, given the expression:

$$e_1 >> (e_2 \,|||\, e_3) >> e_4$$

the number of messages needed to synchronize "e_1" with the parallel expression "$(e_2 \,|||\, e_3)$" is 2, instead of 1 as defined by the sequential operator ">>" and the number of messages needed to synchronize the parallel expression "$(e_2 \,|||\, e_3)$" with "e_4" is 2, instead of 1 as defined by the operator ">>".

It was shown in Section 3.2 that the algorithm avoids the generation of empty alternatives in choice expressions, if necessary, by some message exchange, as defined by the function **Alternative** (see Tables 3 and 4). The purpose of the empty alternative avoidance is to guarantee a correct synchronization between the actions of the choice expression and its following actions. Let us consider for instance the expression:

$$e_1 >> (e_2 \,[]\, e_3) >> e_4$$

Avoiding empty alternatives by derivation of $(e_2 \,[]\, e_3)$, might request some message exchange involving places in e_2 and e_3.

5. Correctness

As we mentioned in Section 3.3, the disabling operator "[>" is very difficult to be precisely implemented in a distributed environment. Therefore, we have defined a distributed implementation of the disabling operator with a slightly modified semantics, which seems to be more intuitive for a distributed environment. Therefore, in this section, we prove the correctness of our transformation under the assumption that the disabling operator "[>" is not contained in a given service specification.

5.1 Statement of the Problem

It was stated in Section 1 that a communication service described by an expression S, may be seen as being provided by several communicating entities PE_1, PE_2, ...PE_n. Any message sent by an entity PE_i is delivered by a reliable communication medium to the destination entity PE_j. Based on this architectural model we assume that between the service, the entities and the medium there exists the following equivalence relation:

$$S \approx \text{hide } G \text{ in } ((PE_1 \,|||\, PE_2 \,|||\, ...|||\, PE_n) \,|[G]|\, \text{Medium}) \qquad (1)$$

where "≈" means observation congruence [Lotos 89]. If two LOTOS expressions B_1 and B_2 are observation congruent, written $B_1 \approx B_2$, B_1 may be replaced by B_2, and vice versa, in any LOTOS context without changing the meaning. Hiding, with the LOTOS operator "hide", allows to transform some observable actions of a process into unobservable ones. Any occurring action of the set G is transformed into an i-action [Bolo 87]. In our case, G is a list of sending and receiving actions for synchronization of PE_1, PE_2...PE_n (see Section 5.2 below).

Let S be a service expression and $PE_1(S)$, $PE_2(S)$, ... $PE_n(S)$ be the protocol entity expressions derived by a given method from the service expression S. We say that the derivation method is correct if the service expression S and the protocol expressions $PE_1(S)$, $PE_2(S)$, ... $PE_n(S)$ satisfy the equivalence relation above.

Then the following theorem will express the correctness of our derivation algorithm:

Theorem: Let S be a service expression and $T_1(S)$, $T_2(S)$,....,$T_n(S)$ the protocol entity expressions obtained from the algorithm presented in Section 4 . Then:

$$S \approx \text{hide } G \text{ in } ((T_1(S) \;|||\; T_2(S) \;|||\; ... \;|||\; T_n(S)) \;|[G]|\; \text{Medium})$$

An outline of the proof is given in the Section 5.3. For more details see [Kant 92].

5.2 Specification of the Medium

The sending actions were denoted in Section 2 by "$s_k(m)$", meaning: send the message "m" to the entity "k" (the sending entity is not explicitly defined), and the receiving actions by "$r_i(m)$", meaning: receive the message "m" from the entity "i" (again, the receiving entity is not explicitly defined). These forms of the sending and receiving actions, which we call **short forms,** should be used only in the local context of a given entity (as we did so far), or in the case of a system with only two entities. In a global context of a system with more than two entities, it is important to distinguish between two send actions $s_k(m)$, both destined to the entity k, but executed in two different places, let us say "i" and respectively "j". In this case we have to use **long forms** for writing these sending actions as:

$s^i_k(m)$ and $s^j_k(m)$, respectively

The same observation applies to the receiving actions: in order to distinguish between two receiving actions $r_j(m)$, both from the place "j", but executed in two different places, say "i" and "k", we write these actions as:

$r^i_j(m)$ and $r^k_j(m)$, respectively

With this notations we can define G as:

$$G = \{ s^i_j(m),\ r^j_i(m) \mid i=1..n,\ j=1..n,\ i \neq j, m \in M \}$$

where M is a finite set of messages.

The communication medium is composed out of a number of channels, one channel for each pair of places (see Fig. 5). We assume for the proof that at most one message may be in transit over a given channel. We assume also that any message sent by an entity T_i is delivered by the communication medium to the destination entity T_j without loss or duplication or inserted messages; each message is delivered after an arbitrary finite delay.

Consequently we have:

$$\text{Channel}_{jk} = \left[\right]_{m \in M} (s^j_k(m) ; r^k_j(m) ; \text{Channel}_{jk})$$

and

$$\text{Medium} = \left|||\right|_{\substack{j=1..n,\\ k=1..n}} (\text{Channel})_{jk}$$

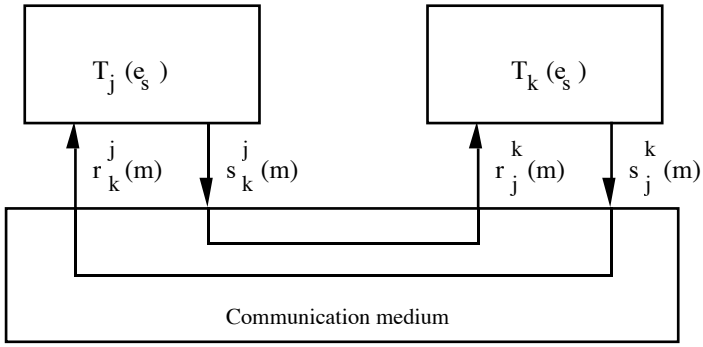

Figure 5. Communication medium

5.3 Proving the correctness of the derivation algorithm by induction

5.3.1 Overview of the method

The proof of the theorem stated in Section 5.1 may be done using the observation congruence laws [Lotos 89] and induction as follows. Let Σ be the set of all service expressions which can be constructed with the syntactic rules of Table 1 from some elementary expressions e_i and operators of the set Ω = {;, [], |||, |[e]|, [>, >>}. Let T_p, with p = 1..n, be a transformation of an expression $S \in \Sigma$ into a set of protocol entity expressions $\{T_1, T_2, ...T_n\}$. We would like to show that the given function T_p derives correct protocol expressions for any service expression $S \in \Sigma$, constructed with an arbitrary but finite number of operators of the set Ω.

In order to prove this we will apply the principle of induction on the syntax tree of the expression S as follows:

1. First we prove the initial conditions (the base of the induction). We consider all service expressions S consisting of a single elementary expression e_i of the form $S = e_i$, and prove that

$$e_i \approx \text{hide G in } (T_1(e_i) \,|||\, T_2(e_i) \,|||\, ... \,|||\, T_n(e_i) \,|[G]|\, \text{Medium}) \qquad (2)$$

2. We prove the inductive step and consider service expressions of the form "$S = S_1$ op S_2", where $S_1, S_2 \in \Sigma$ and "op" is any operator of the set Ω. We prove that

$$S \approx \text{hide G in } (T_1(S) \,|||\, T_2(S) \,|||\, ... \,|||\, T_n(S) \,|[G]|\, \text{Medium}) \qquad (3)$$

holds under the assumption that the derivations $T_p(S_1)$ and $T_p(S_2)$ are correct, that is

$$S_1 \approx \text{hide G in } (T_1(S_1) \,|||\, T_2(S_1) \,|||\, ... \,|||\, T_n(S_1) \,|[G]|\, \text{Medium})$$

and

$$S_2 \approx \text{hide G in } (T_1(S_2) \,|||\, T_2(S_2) \,|||\, ... \,|||\, T_n(S_2) \,|[G]|\, \text{Medium})$$

This induction step must be proven separately for every operator "op" $\in \Omega$. This means that we have to consider, separately, each syntactic rule in Table 1, with its corresponding derivation rule in Table 3.

By induction we can conclude that the function T_p will provide correct protocol expressions for any expression $S \in \Sigma$, constructed by an arbitrary but finite number of operators "op" and elementary expression "e_i".

We give in the following sections the proof of the initial conditions and the proof of the induction step for the sequential operator ">>". For the other operators, the induction step can be proven in a similar manner [Kant 92].

5.3.2 Proving the Initial Conditions

As shown above, the basic case corresponds to elementary service expressions of the form S = e_i, where e_i is defined by the rule (17) of the syntax of the specification language (see Table 1 in Section 2). Therefore we have to consider expressions of the form:

$$S = a^i \text{ ; exit} \qquad (4)$$

where "a^i" is the primitive "a" offered at place "i". In the following we show that the function T_p, defined in Table 3 of Section 4, derives correct protocol expressions for these service expressions.

The function T_p applied to the service expression (4) yields

$$T_p(S) := \mathbf{Proj_p}(a^i) \text{ ; exit } = a^i \text{ ; exit} \qquad \text{for } p = i$$

and

$$T_p(S) := \mathbf{Proj_p}(a^i) \text{ ; exit } = \text{exit} \qquad \text{for } p \neq i$$

where $\mathbf{Proj_p}(a^i)$ is as defined in Table 4 of Section 4. No synchronization messages are generated by the derivation function Tp, therefore G = ∅. We have to prove that:

$$S \approx \text{hide G in (} (T_1(S) \,|||\, ... \,|||\, T_i(S) \,|||\, ... \,|||\, T_n(S)) \,|[G]|\, \text{Medium)}$$

The right side becomes

$$\text{hide G in } (a^i \text{ ; exit}) = a^i \text{ ; exit}$$

which is equal to S. Therefore we conclude that the derivation function Tp applied to elementary expression of the form (4) always yields correct protocol expressions.

5.3.3 Proving the Induction Step for the Sequential Composition Operator ">>"

As an example of the induction step let us consider service expressions constructed by the syntactic rule (7) (see Table 1). In this case the general form "S = S_1 op S_2" becomes "S = S_1 >> S_2" where ">>" stands for the operator "op". The corresponding derivation rule is:

$$T_p(S) := T_p(S_1) \;>>\; \mathbf{Synch\_left_p}(S_1,S_2) \;>>\; \mathbf{Synch\_right_p}(S_1,S_2) >> T_p(S_2)$$

where "$T_p(S_1)$" and "$T_p(S_2)$" are the derivations for the expressions "S_1" and "S_2" respectively, and the functions "$\mathbf{Synch\_left_p}(S_1,S_2)$" and "$\mathbf{Synch\_right_p}(S_1,S_2)$" are as defined in Table 4.

To simplify the situation and without loss of generality we assume that there are three places in the specification, i.e. ALL = {i,j,k}, and that $EP(S_1) = \{i\}$, $SP(S_2) = \{j\}$ and $N(S_1) = m$. This simplification will allow us to temporarily drop the upper index in the notation of the synchronization messages, which will make the following explanation more readable. For the three protocol entities we obtain:

$$T_i(S) = T_i(S_1) >> (s_j(m) \text{ ; exit}) >> T_i(S_2) \qquad (5)$$
$$T_j(S) = T_j(S_1) >> (r_i(m) \text{ ; exit}) >> T_j(S_2) \qquad (6)$$
$$T_k(S) = T_k(S_1) >> T_k(S_2) \qquad (7)$$

The set of synchronization messages generated by the algorithm is

$$G = \{s_j(m),r_i(m),...\} = \{s_j(m),r_i(m)\} \cup G_1 \cup G_2$$

where G_1 and G_2 are subsets of synchronization messages relative to S_1 and S_2, respectively. The medium includes the interactions $s_j(m)$ and $r_i(m)$, that is:

$$\text{Medium} = \text{Channel}_{ij} \;|||\; \text{Med}_1 \;|||\; \text{Med}_2 \qquad (8)$$

where

$Channel_{ij} = s_j(m) ; r_i(m) ; Channel_{ij}$

and Med_1, Med_2 are channel expressions relative to S_1 and S_2, respectively. We have to prove that

$$S \approx \text{hide } G \text{ in } ((T_i(S) \,|||\, T_j(S) \,|||\, T_k(S)) \,|[G]|\, Medium) \quad (9)$$

with the induction hypothesis

$$S_1 \approx \text{hide } G \text{ in } ((T_i(S_1) \,|||\, T_j(S_1) \,|||\, T_k(S_1)) \,|[G]|\, Med_1) \quad (10)$$
$$S_2 \approx \text{hide } G \text{ in } ((T_i(S_2) \,|||\, T_j(S_2) \,|||\, T_k(S_2)) \,|[G]|\, Med_2) \quad (11)$$

Knowing that $T_i(S)$, $T_j(S)$ and $T_k(S)$ do not have any common event, but each of them has common events with Medium, we can write the right side of (9) as follows:

$$\text{hide } G \text{ in } ((T_i(S) \,|||\, T_j(S) \,|||\, T_k(S)) \,|[G]|\, Medium) =$$
$$= \text{hide } G \text{ in } (((T_i(S) \,|[G]|\, Medium) \,|[G]|\, T_j(S)) \,|[G]|\, T_k(S)) \quad (12)$$

We will develop successively the expression (12). Taking first $(T_i(S) \,|[G]|\, Medium)$ and using (5) we have

$$T_i(S) \,|[G]|\, Medium = (T_i(S_1) >> (s_j(m) ; exit) >> T_i(S_2)) \,|[G]|\, Medium$$

Given the Medium expression (8) it is easy, although tedious, to show by expansion (see T1 in Annex A) that:

$$T_i(S) \,|[G]|\, Medium =$$
$$= T_i(S_1) \,|[G_1]|\, Med_1 >> (s_j(m);exit) \,|[s_j(m), r_i(m)]|\, (s_j(m);r_i(m);exit) >> T_i(S_2) \,|[G_2]|\, Med_2 =$$
$$= T_i(S_1) \,|[G_1]|\, Med_1 >> (s_j(m) ; r_i(m) ; exit) >> T_i(S_2) \,|[G_2]|\, Med_2$$

Then, using the last result and (6):

$$(T_i(S) \,|[G]|\, Medium) \,|[G]|\, T_j(S) =$$
$$= (T_i(S_1) \,|[G_1]|\, Med_1 >> (s_j(m) ; r_i(m) ; exit) >> T_i(S_2) \,|[G_2]|\, Med_2) \,|[G]|\, T_j(S) =$$
$$= ((T_i(S_1) \,|[G_1]|\, Med_1) \,|[G_1]|\, T_j(S_1)) >> (s_j(m) ; r_i(m) ; exit) >>$$
$$((T_i(S_2) \,|[G_2]|\, Med_2) \,|[G_2]|\, T_j(S_2))$$

Finally, using (7):

$$((T_i(S) \,|[G]|\, Medium) \,|[G]|\, T_j(S)) \,|[G]|\, T_k(S) =$$
$$= (((T_i(S_1) \,|[G_1]|\, Med_1) \,|[G_1]|\, T_j(S_1)) \,|[G_1]|\, T_k(S_1)) >> (s_j(m) ; r_i(m) ; exit) >>$$
$$(((T_i(S_2) \,|[G_2]|\, Med_2) \,|[G_2]|\, T_j(S_2)) \,|[G_2]|\, T_k(S_2))$$

Using H8 and then H5 from Annex A, the right side of the expression (12) becomes:

$$\text{hide } G \text{ in } (((T_i(S) \,|[G]|\, Medium) \,|[G]|\, T_j(S)) \,|[G]|\, T_k(S)) =$$
$$= (\text{hide } G \text{ in } (((T_i(S_1) \,|[G_1]|\, Med_1) \,|[G_1]|\, T_j(S_1)) \,|[G_1]|\, T_k(S_1))) >> i ; exit >>$$
$$i ; exit >> (\text{hide } G \text{ in } (((T_i(S_2) \,|[G_2]|\, Med_2) \,|[G_2]|\, T_j(S_2)) \,|[G_2]|\, T_k(S_2)))$$

Using the above result in (12) and based on I1 (see Annex A) we have:

$$\text{hide } G \text{ in } ((T_i(S) \,|||\, T_j(S) \,|||\, T_k(S)) \,|[G]|\, Medium) =$$
$$= (\text{hide } G_1 \text{ in } ((T_i(S_1) \,|||\, T_j(S_1) \,|||\, T_k(S_1)) \,|[G_1]|\, Medium)) >>$$
$$(\text{hide } G_2 \text{ in } ((T_i(S_2) \,|||\, T_j(S_2) \,|||\, T_k(S_2)) \,|[G_2]|\, Medium))$$

Substituting (10) and (11) in the above expression we have:

$$\text{hide } G \text{ in } ((T_i(S) \,|||\, T_j(S) \,|||\, T_k(S)) \,|[G]|\, Medium) = S_1 >> S_2$$

which is equal to S.

6. Discussion and Conclusions

We have presented an algorithm for automatic derivation of protocol specifications from formal specifications of a communication service. This algorithm is an extension of previous algorithms [Boch 86, Khen 89, Gotz 90] to a more comprehensive specification language which can handle all operators and unrestricted process invocation and recursion as defined by LOTOS [Lotos 89], a Formal Description Technique developed within ISO for the formal specification of open distributed systems.

One of the nice features of our derivation algorithm is the preservation of the structure of the service specification in every derived protocol specification. Aside from positive methodological aspects for systems design, this feature presents a major advantage in the proof of the correctness of the derivation algorithm, which is outlined in Section 5. The proof is based essentially on the bisimulation congruence relation between the given service specification and the parallel composition of protocol specifications and the communication medium. The proof is done by induction over the abstract syntax tree of the given service specification.

We have built a Protocol Generator (PG) prototype based on the derivation algorithm, which was used for all examples presented in this paper. The experience with this tool demonstrated the effectiveness of our method.

The following discussion relates our approach to other methods of protocol derivation. [Prob 91] describes certain criteria than can be used to classify and compare different protocol synthesis methods. They divide these methods into two broad categories, based on the starting point for the design:

(a) methods starting either from a partial protocol description or from a complete description of one entity and synthesizing the complete protocol [Kaku 88], [Rama 86], [Goud 84], [Sidh 82], [Zafi 80]. The synthesis is based on the duality inherent in message exchange: for each message sent by a protocol entity, there must be a protocol entity prepared to receive it. For all these methods, there is no particular way to specify service requirements that should be satisfied.

(b) methods starting from a given service definition and synthesizing all entities involved in the service definition [Sale 90], [Lang 90], [Chu 88], [Gotz 90], [Khen 89].

Our work belongs to the second category. Compared with the other work in this category, the method described in this paper supports a more powerful specification language. The other methods are based on finite state machines, except for [Lang 90], [Gotz 90 and [Khen 89] which are even further restricted. [Chu 88] and [Lang 90] apply to two-party protocols only.

Our method is based on a different formalism, a so-called "process-algebra" [Miln 80]. These algebras define a rigorous set of transformation rules and equivalence relations that allow a designer to reason formally about the behavior of a system. In particular, this feature allowed us to deal with the formal proof of our derivation algorithm. The proof of correctness given in this paper is an important development of our previous work [Khen 89, Gotz 90]. Formal proofs of other methods are given in [Lang 90] and [Sale 90].

We assumed in this paper a reliable underlying communication medium. For the case of a non-reliable underlying communication service it is possible to use our algorithm as a first step (assuming a reliable medium) and then use a procedure which will systematically transform the error-free protocol into an error-recoverable one. This approach may be similar to [Rama 86], however, the case of multi-party protocols must be considered with care; currently we are working on the integration of such possibilities to our algorithm. The methods described by [Sale 90] and [Chu 88] also provide for error-recoverable protocol specifications.

With the exception of [Gotz 90], none of the synthesis methods considers services with interaction parameters. The extension of the algorithm presented in this paper to service and

protocol specifications with interaction parameters may be pursued along the lines described in [Gotz 90]. This implies the addition of supplementary parameters to the synchronization messages and, in some cases, additional message exchanges between different places. A simple extension for the case with interaction parameters are considered in [Yasu 94].

REFERENCES

[Aho 85] A. Aho, R. Sethi and J. D. Ullman, *Compilers Principles, Techniques and Tools*, Addison-Wesley, 1985.

[Boch 86] G. v. Bochmann and R. Gotzhein, *Deriving protocol specifications from service specifications*, Proceedings of the ACM SIGCOMM '86 Symposium, Vermont, USA, pp.148-156, 1986.

[Boch 90] G. v. Bochmann, *Protocol specification for OSI*, Computer Networks and ISDN Systems 18, pp.167-184, April 1990.

[Boch 90b] G. v. Bochmann, *Specifications of a Simplified Transport Protocol Using Different Formal Description Techniques*, Computer Networks and ISDN 18, pp. 335-377, 1989/90.

[Bolo 87] Bolognesi T. and Brinksma E., *Introduction to the ISO Specification Language LOTOS*, in Computer Networks and ISDN Systems, Vol. 14, No. 1, pp. 25-59, 1987.

[Brin 85] E. Brinksma and G. Karjoth, *A Specification of the OSI Transport Service in LOTOS*, in Protocol Specification, Testing and Verification IV, Y.Yemeni, R. Strom and S.Yemeni (Ed). Elsevier Science Publishers B.V. (North Holland), 1985.

[Coel 92] R. J. Coelho da Costa and J-P Courtiat, *A True Concurrency Semantics for LOTOS*, in Proceedings of the Fifth International IFIP WG 6.1 Conference on Formal Description Technique (FORTE'92), pp.347-362, Oct. 1992.

[Chu 88] P. M. Chu and M.T. Liu, *Protocol Synthesis in a State Transition Model*. Proceedings IEEE COMPSAC ' 88, pp. 505-512, 1988.

[Gotz 90] R. Gotzhein and G.v. Bochmann *Deriving protocol specifications from service specifications including parameters*. ACM Transactions on Computer Systems, Vol. 8, No. 4, pp. 255-283, 1990.

[Goud 84] M. Gouda and Y. Yu *Synthesis of communicating finite-state machines with guaranteed progress*. IEEE Transactions on Communications, COM-32, No.7, pp.779-788, July 1984.

[Hals 88] Halsall F., *Data Communications, Computer Networks and OSI*, Addison-Wesley, 1988.

[ISO 8072] ISO, *Information Processing System - Open Systems Interconnection-Transport Service Definition* IS 8072, 1985.

[ISO 8073] ISO, *Information Processing System - Open Systems Interconnection, Connection Oriented Transport Protocol Specification*, IS 8073, 1985.

[Kaku 88] Kakuda Y.,Wakahara Y., *Component-based synthesis of protocols for unlimited number of processes*, in Proc. COMPSAC'87 pp. 721-730, 1988.

[Kant 92] Kant C. *Deriving Protocol Specifications from Service Specifications Written in LOTOS*, Research Report #805, Dept. I.R.O., Université de Montreal, 1992.

[Kant 93] Kant C. *Deriving Protocol Specifications from Service Specifications*, Ph.D. thesis, Université de Montreal.

[Khen 89] F. Khendek, G. v. Bochmann and C. Kant, *New results on deriving protocol specifications from services specifications*, Proceedings of the ACM SIGCOMM'89, pp.136-145, 1989.

[Kurs 89] Kurshan R.P. and McMillan K., *A Structural Induction Theorem for Processes,* ACM, 1989.

[Lang 90] R. Langerak, *Decomposition of functionality; a correctness-preserving LOTOS transformation,* in Proceedings of the Tenth International IFIP WG 6.1 Symposium on Protocol Specification, Testing and Verification, Ottawa, pp.229-242, June 1990.

[Lotos 89] ISO, *Information Processing System - Open Systems Interconnection - LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behavior*, IS 8807, Jan. 1989.

[Merl 83] P. Merlin and G. v. Bochmann *On the construction of sub module specifications and communication protocols*. ACM Trans. on Programming Languages and Systems, No.1, pp.1-25, Jan.1983.

[Miln 80] Milner R., *A Calculus of Communicating Systems,* Springer-Verlag LNCS 92, Berlin 1980.

[OSI 84] ISO, *Information Processing System - Open Systems Interconnection - Basic Reference Model*, ISO 7498, 1984.

[Prob 91] R. Probert and K. Saleh, *Synthesis of Communication Protocols: Survey and Assessment*, IEEE Transactions on Computers, Vol. 40,No. 4. pp. 468-475, 1991.

[Rama 85] C.V. Ramamoorthy, S. T. Dong and Y. Usuda *An Implementation of an Automated Protocol Synthesizer (APS) and its Application to the X.21 Protocol.* IEEE Transactions on Software Engineering, Vol. SE-11, No.9, pp. 886-908, Sept.1985.

[Rama 86] C.V., Ramamoorthy, Y. Yaw, R. Aggarwal and J. Song, *Synthesis of Two-Party Error-Recoverable Protocols.,* Proceedings of the ACM SIGCOMM '86 Symposium, Vermont, USA, pp.227-235, 1986.

[Rana 83] S.P. Rana *A Distributed Solution of the Distributed Termination Problem,* Information Processing Letters 17, pp. 43-46, 1983.

[Sale 90] K. Saleh and R. Probert *A Service-based Method for the Synthesis of Communication* Int. J. Mini and Microcomput. Special Issue on Distributed Systems, vol. 12, No. 3, pp. 97-103, 1990.

[Scol 86] Scollo G.,Pappalardo G.,Logrippo L.,Brinksma E., *The OSI Transport Service and its Formal Description in LOTOS,* in L. Csaba, K. Tarnay, T. Szentivanyi (eds.) Computer Network Usage: Recent Experiences, pp. 465-488, North-Holland, Amsterdam 1986.

[Scol 87] Scollo G., Formal *Description of the OSI Transport Service in LOTOS,* contrib. to ISO/TC97/SC6/WG4 special group on FDT in LOTOS, Berlin 20-23, Oct., 1987.

[Sidh 82] Sidhu D.P., *Protocol Design Rules*, in Proc. Second IFIP Int. Symp. Protocol Specification, Testing and Verification, pp. 283-300, 1982.

[Tane 88] Tanenbaum A., *Computer Networks,* Prentice Hall, 1988

[Viss 85] C. Vissers and L. Logrippo, *The importance of the concept of service in the design of data communications protocols*, in Proc. of the Fifth IFIP Workshop on Protocol Specification, Verification and Testing, Toulouse, pp.3-17, 1985.

[Viss 88] C. Vissers, G. Scollo and M.v. Sinderen, *Architecture and Specification Style in Formal Descriptions of Distributed Systems*, in: Proc. IFIP Symposium on Protocol Specification, Testing and Verification, Atlantic City, pp.189-204, 1988.

[Viss 90] C. Vissers, *FDT's for open distributed systems, a retrospective and a prospective view*, in Proceedings of the Tenth International IFIP WG 6.1 Symposium on Protocol Specification, Testing and Verification, pp.341-362, Ottawa, June 1990.

[Yasu 94] K. Yasumoto, T. Higashino and K. Taniguchi, *Software Process Description using LOTOS and Its Enaction*, in Proc. of the 16th Int. Conf. on Software Engineering (ICSE-16), pp.169-178, May 1994.

[Zafi 80] P. Zafiropulo, C.H. West, H. Rudin, D.D. Cowan and D. Brand *Towards analyzing and synthesizing protocols*. IEEE Transactions on Communications, Vol. COM-28, No.4, pp.651-661, April 1980.

Annex A:

Observation Congruence laws.[1]

In $B_1 = B_2$, the expressions B_1 and B_2 are observation congruent; B1 may be replaced by B_2 in all LOTOS context without changing the meaning.

Choice.

B_1 **[]** B_2 = B_2 **[]** B_1 (C1)

B_1 **[]** (B_2 **[]** B_3) = (B_1 **[]** B_2) **[]** B_3 (C2)

B_1 **[]** B_1 = B_1 (C3)

Parallel:

B_1 **|** B_2 = B_2 **|** B_1 (P1)

B_1 **|** (B_2 **|** B_3) = (B_1 **|** B_2) **|** B_3 (P2)

B_1 **|[**list**]|** B_2 = B_1 **|[**list'**]|** B_2 (P3)
where list' is any list containing the same elements as list

B_1 **|[**list**]|** B_2 = B_1 **||** B_2 if $L(B_1) \cap L(B_2) \subseteq$ list (P4)
L(B) is the set of free gate identifiers

B_1 **|[]|** B_2 = B_1 **|||** B_2 (P5)

Hiding:

hide list **in** B = **hide** list' **in** B (H1)
where list' is any list containing the same elements as list

hide list **in** B = **hide** list' **in** B where list' = list $\cap$ L(B) (H2)

hide list **in hide** list' **in** B = **hide** list'' **in** B (H3)
where list'' list $\cup$ list'

hide list **in** B = B if list $\cap$ L(B) = $\varnothing$ (H4)

hide list **in** (a ; B) = i ; **hide** a **in** B if a $\in$ list (H5)

hide list **in** (B_1 **[]** B_2) = (**hide** list **in** B1) **[]** (**hide** list **in** B2) (H6)

hide list **in** (B_1 **|[**list'**]|** B_2) = (**hide** list **in** B_1) **|[**list'**]|** (**hide** list **in** B_2) (H7)
if list $\cap$ list'= $\varnothing$

hide list **in** (B_1 **>>** B_2) = (**hide** list **in** B_1) **>>** (**hide** list in B2) (H8)

hide list **in** (B_1 **[>** B_2) = (**hide** list **in** B_1) **[>** (**hide** list in B2) (H9)

Enabling:

exit >> B = i ; B (E1)

(B_1 **>>** B_2) **>>** B_3 = B_1 **>>**(B_2 **>>** B_3) (E2)

Disabling:

B_1 **[>** (B_2 **[>** B_3) = (B_1 **[>** B_2) **[>** B_3 (D1)

(B_1 **[>** B_2) **[]** B_2) = (B_1 **[>** B_2) (D2)

exit **[>** B = exit **[]** B

---

[1] from [LOTOS 89]

Internal actions:

$a ; i ; B = a ; B$ (I1)
$B [] i ; B = i ; B$ (I2)
$a ; (B_1 [] i ; B_2) [] a ; B_2 = a ; (B_1 [] i ; B_2)$ (I3)

Expansion theorems:

Notation:

$B_1 [] B_2 [] ...[] B_n$ is written $[]\{B_1, ...B_n\}$
Let $B = [] \{ b_i ; B_i \mid i \in I \}$
$C = [] \{ c_j ; C_j \mid j \in J \}$

$B|[A]|C = [] \{ b_i ; (B_i | C) \mid b_i \notin A , i \in I \}$ (T1)
$[] [] \{ c_j ; (B | C_j) \mid c_j \notin A , j \in J \}$
$[] [] \{ a ; (B_i | C_j) \mid a = b_i = c_j , a \in A \}$

$B [> C = C$ (T2)
$[] [] \{ b_i ; (B_i [> C) \mid i \in I \}$

hide A in $B = [] \{ b_i ;$ hide A in $B_i \mid b_i \notin A , i \in I \}$ (T3)
$[] [] \{ i ;$ hide A in $B_i \mid b_i \in A , i \in I \}$