Deriving Protocol Specifications from Service Specifications Including Parameters

REINHARD GOTZHEIN and GREGOR VON BOCHMANN
University of Montreal

The service specification concept has acquired an increasing level of recognition by protocol designers. This architectural concept influences the methodology applied to service and protocol definition. Since the protocol is seen as the logical implementation of the service, one can ask whether it is possible to formally derive the specification of a protocol providing a given service. This paper addresses this question and presents an algorithm for deriving a protocol specification from a given service specification. It is assumed that services are described by expressions, where names identifying both service primitives and previously defined services are composed using operators for sequence, parallelism and alternative. Services and service primitives may have input and output parameters. Composition of services from predefined services and service primitives is also permitted. The expression defining the service is the basis for the protocol derivation process. The algorithm presented fully automates the derivation process. Future work will focus on the optimization of traffic between protocol entities and on applications.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture, protocol verification*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*

General Terms: Algorithms, Design, Verification

Additional Key Words and Phrases: Automated protocol design, communication service specification, protocol derivation

1. INTRODUCTION

The service specification concept has acquired an increasing level of recognition by protocol designers (see e.g. [20]). This architectural concept influences the methodology applied to service and protocol definition [6]. Since the protocol is seen as the logical implementation of the service, one can ask whether it is possible to formally derive the specification of a protocol providing a given service. Similar questions have been raised concerning the derivation of synchronization code from given specifications [13, 14].

An architectural model for both service level and protocol level is depicted in Figure 1. A service is realized by a service provider which, according to the

Authors' addresses: G. V. Bochmann, Dept. IRO, University of Montreal, C.P. 6128, Succursale A, Montreal, Quebec H3C 3J7, Canada. R. Gotzhein, University of Hamburg, FB Informatik, Bodensted str. 16, 2000 Hamburg 50, Germany.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1990 ACM 0734-2071/90/1100-0255 $01.50

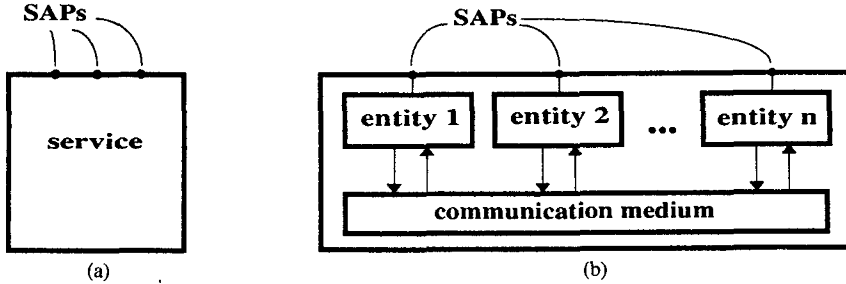

Fig. 1. (a) Service architecture. (b) Protocol architecture.

principle of abstraction is seen as a black box, and made available through Service Access Points (SAPs) (Figure 1a). On the protocol level, some internal structure is given to the black box: Several entities, linked by an underlying transmission medium, may cooperate to provide the service (Figure 1b). We assume the communication medium to be reliable, to maintain the sending sequence of messages and to be connected to each entity by FIFO-queues for transmissions and receptions (also see Section 3.3).

Based on this architectural model, we can phrase the above question in more precise terms. Given a service specification e_s (Figure 2a), is it possible to formally derive the specifications $T_i(e_s)$ for all protocol entities (Figure 2b)?

Services in our approach are described by expressions. Names identifying both service primitives and previously defined services are composed using operators for sequence, parallelism and alternative. Each service primitive is explicitly linked to an interaction point where it is made available to the service user. The expression defining the service then is the basis for the protocol derivation process. An algorithm has been developed which allows us to fully automate this process.

Techniques for systems development and validation can be classified into analytic and synthetic approaches. *Analysis* means decomposition of an existing structure and examination of its components and their relationships. The analysis techniques depend on the nature of the structure and the relevant relationships of its components. Relationships assuring the absence of certain errors are of interest in software engineering. The purpose of the analysis in this area is therefore the detection of these errors. *Synthesis* denotes the process of building a structure that possesses desired properties. The synthesis procedure determines the way components may be put together to form the structure. The purpose of the synthesis method is the prevention of design errors in software engineering.

Both analysis and synthesis techniques have been developed for, and more or less successfully applied to many different areas, among them sequential programs, database relations and communication protocols. Analysis of a sequential program is understood as a formal proof that the program meets its requirements, that is, it is correct with respect to its input/output assertions (program verification). Synthesis, on the other hand, starts from

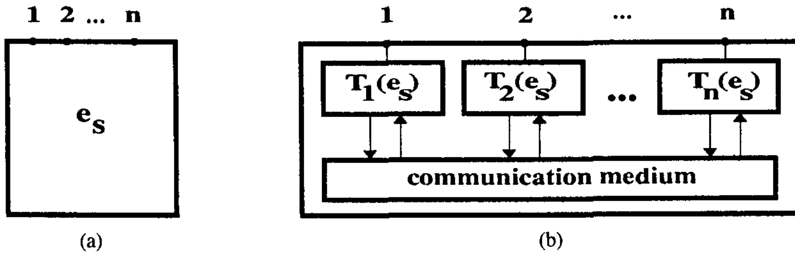

Fig. 2. (a) Service specification. (b) Derived protocol specification.

these assertions and constructs the correct program by stepwise insertion of intermediate assertions and program elements. Analysis of data base relations is used to discover the presence of some undesired properties like insertion, deletion and update anomalies. Synthesis here starts from basic relations and avoids these anomalies by construction [7, 21].

In the area of communication protocols, analysis techniques have been developed and applied to detect design errors such as deadlocks, unspecified receptions, nonexecutable interactions, state ambiguities [24] and nonconformance with the service specification. The best known approach appears to be reachability analysis. This is usually based on the specification of protocol entities as finite state automata, which model certain aspects of message exchange between them. Various types of reachability analysis have been proposed including exhaustive state exploration [2, 22], random state exploration [23] and reduced reachability analysis [19, 25]. Since the analysis of a sufficiently complex protocol specification reveals some of the above design errors, the specification has to be revised and the analysis must be repeated until no more errors are found.

With protocol synthesis one wants to avoid the above errors a priori. So far, not much work has been dedicated to this problem. Existing approaches ([24, 15, 8, 17, 18]) take partly specified protocol entities or complete specifications of some protocol entities as a starting point for the synthesis procedure. The synthesis procedure 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. However, several important limitations apply to each of these approaches:

—With the exception of [15], the service specification is not taken into account. There is no formal requirement on which the synthesis is based. Instead it requires part of the solution to be provided in advance. It is clear that without a formal service definition, conformance with the service is not guaranteed by the synthesis algorithm and must be shown in a separate step.

—Again with the exception of [15], only two party protocols are considered. It seems to be difficult to extend the approaches to an arbitrary number of protocols entities. Thus they are not well suited for high level protocols involving more than two parties.

—[24], [4], and [17] all assume the existence of a reliable communication medium. The latter, however, is extended to cover noisy channels.
—None of the above approaches takes parameters into account. Only a distinction between different message types is possible.
—[24] and [15] do not avoid deadlocks by construction.
—All approaches assume the existence of (incomplete) protocol specifications. None is based solely on the service definition.
—[24] and [15] are quite expensive with respect to computation.

Our approach, introduced in [4] and extended here, is more general in that only the existence of the service specification (Figure 2a) is required. It can handle an arbitrary number of protocol entities. Furthermore, input and output parameters are taken into account, and the possibility of composing new services from previously defined services and/or service primitives supports abstraction and modularity, both on the service and the protocol level. Subsystem failures and unreliable channels, however, are not taken into account. For the derivation of the protocol specification, an assignment of the different services/service primitives to a finite number of service access points must be given. An algorithm then provides specifications of all protocol entities serving these interaction points.

The paper is structured as follows, Section 2 introduces concepts and notations on which our algorithm is based. Section 3 presents the algorithm in two steps, the first focusing on synchronization between the distributed protocol entities, the second describing extensions in order to handle parameters. Section 4 introduces a concept of abstraction by composition of previously defined services and describes how the algorithm of Section 3 can be used for this purpose. In Section 5, a number of examples are presented. Finally, Section 6 discusses the results and gives hints for the extension and application of the algorithm.

2. CONCEPTS AND NOTATIONS

A service ([3, 20]) in our approach is defined by an expression, consisting of the names of service primitives and operators. Let $\mathbf{SP} =_{Df} \{\mathbf{a}^r, \mathbf{b}^s, \ldots, \mathbf{z}^t\}$ denote a set of service primitives. The syntax of expressions is defined by the following production rules of a context-free grammar, where **e** is a nonterminal and starting symbol, and $\mathbf{SP} \cup \{;, |||, [\,]\}$ is a finite set of terminal symbols:

Rule 1: for each terminal symbol $\mathbf{x} \in \mathbf{SP}$: $\mathbf{e} \rightarrow \mathbf{x}$

Rule 2: $\mathbf{e} \rightarrow \mathbf{e} \; ; \; \mathbf{e}$

Rule 3: $\mathbf{e} \rightarrow \mathbf{e} \; ||| \; \mathbf{e}$

Rule 4: $\mathbf{e} \rightarrow \mathbf{e} \; [\,] \; \mathbf{e}$

Each $\mathbf{x} \in \mathbf{S}P$ denotes a service primitive. Each service primitive is linked to an interaction point, in the following called *place*, where it is made available to the service user (and also executed): The notation $\mathbf{a}^4$ means that the service primitive **a** is accessible at place 4.

The operator ";" means that the service defined by the left subexpression must be terminated completely before execution of the service defined by the right subexpression may be started. The operator "|||" means that the services defined by the two subexpressions may be executed in parallel (or in either order). The meaning of "[]" is that either the service defined by the left subexpression or by the right subexpression is to be executed.

In the case of alternative subexpressions (production Rule 4), a decision has to be made regarding which subexpression should be executed. We assume that this decision is made at one place without the consultation of entities at other places. All actions at one place are associated with one entity. We therefore require that the places of the starting operations of the two subexpressions be the same. This is defined more precisely in Section 3.1.2.

We extend this service language by allowing service primitives to have formal input and output parameters. A service primitive then has the following form:

$$\mathbf{x}^{\mathbf{p}}(x_1, \ldots, x_m \mid x_{m+1}, \ldots, x_n)$$

where **x** is the name of the service primitive, **p** is the place where it is accessible, $x_1, \ldots, x_m$ are formal input parameters, and $x_{m+1}, \ldots, x_n$ are formal output parameters. The syntax of this extension could be defined by additional production rules of the context free grammar above.

The term *service primitive* denotes a unit that is not decomposed further. It is executed at a given service access point as a single, atomic action. The service primitive uses the values of its input parameters for determining the values of its output parameters. Within our language, we do not define the meaning of service primitives. That is, we do not define how the output parameter values depend on the input parameters of a service primitive. This issue is outside the scope of this paper, and could for instance be done by adding input/output assertions.

The values of the input parameters of service primitives are provided by the user of the service, or are obtained as outputs from service primitives executed previously. The output parameters of service primitives are either used as input for subsequently executed service primitives, or are delivered to the service user. For example, in Figure 3a, the following communication service is specified. An **a** primitive at place **p** is followed by a **b** primitive at place **p**′. x_1 is a formal input parameter name, the value of which has to be provided by the service user at place **p**. x_2 is the name of an output parameter of **a** which is not exchanged with a service user, but instead used as input by the subsequent service primitive **b** at place **p**′. We will call such a parameter an *intermediate parameter*. Finally, x_3 will be available to the service user at place **p**′ after execution of **b**.

The above notion of input/output parameters of service primitives is different from what is commonly used for describing communication services ([3] or [20]). Usually, a parameter of a service primitive is either provided by the service user and is input to the service provider or is output from the service provider and is passed to the user. For a reliable communication service, the output parameter provided at the destination place is equal to

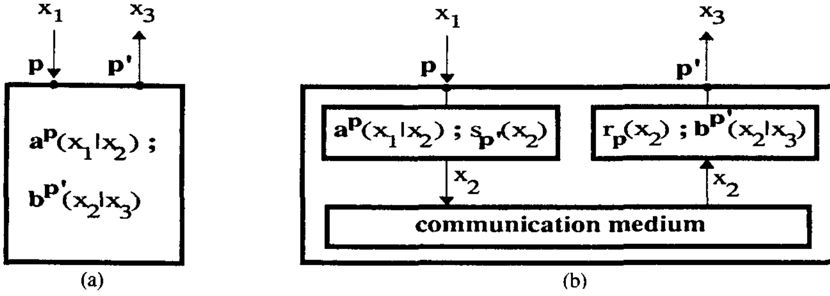

Fig. 3. (a) Service specification. (b) Corresponding protocol specification.

the input parameter provided by the user at the sending place. This service model can be easily expressed in the more general model adopted in this paper. The data transfer example, could be expressed in our model as two service primitives 'send($x_1 \mid x_2$)' and 'receive($x_2 \mid x_3$)' executed at the sending and destination places, respectively, and where for both service primitives the output parameter is equal to its input parameter.

For each syntactically correct service expression, a derivation tree (Figure 4) can be obtained. This tree shows which production rules have been applied to derive the expression. It is used in the definition of the synthesis algorithm (Section 3). As an example, a derivation tree for the service expression $(\mathbf{a}^r ; \mathbf{b}^s) ||| \mathbf{z}^t$ is shown in Figure 4.

To specify protocol entities, we will use the same language as for services, augmented with send and receive operations. Each entity is connected to the underlying medium by FIFO queues for transmissions and receptions (Section 3.3.). When the entity serving place **i** issues a send operation to the entity at place **j**, written s_j, then a new element is added to its FIFO queue for transmissions which has the following contents:

place of sender	place of receiver	synchronization value/ parameter value

The place of the sender is determined implicitly. It is the place where s_j is issued. The place of the receiver is given explicitly as the index **j**. We distinguish between synchronization messages $s_j(z)$, carrying a synchronization parameter value z, and data messages $s_j(x_k)$. The element placed into the transmission queue is eventually conveyed by the medium to the reception queue of its destination. To remove it from that queue, the receiver issues a receive operation, written r_i. This means that a message from place **i** is expected.

3. THE DERIVATION ALGORITHM

The derivation algorithm presented in this section begins with a given service specification. It is assumed to be written in the syntax defined by the context free grammar of Section 2. The principle of the derivation algorithm is to consider for each protocol entity the projection [15] of the service

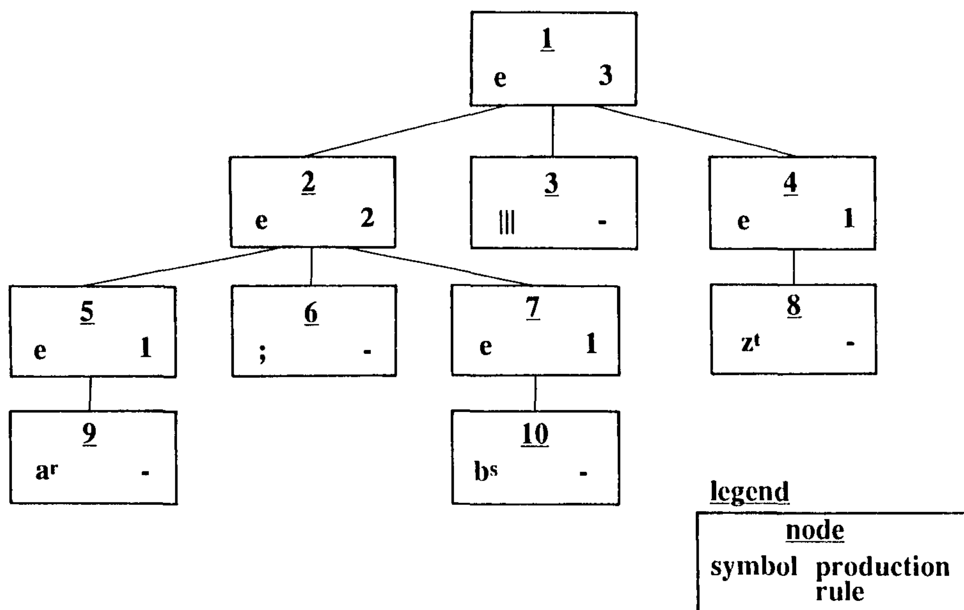

Fig. 4. Derivation tree for $(a^r ; b^s) ||| z^t$.

specification onto the place serviced by that entity. This projection is augmented by appropriate synchronization among the protocol entities, such that, the possible temporal order of operations being executed at different places satisfies the order implied by the service specification. Note that each protocol entity can directly determine the order of actions only at the place it services. Therefore, communication among the protocol entities through an underlying communication medium is required and has to be introduced by the derivation algorithm.

It is desirable to give a proof that the algorithm described here always gives rise to protocol specifications that include properties such as absence of deadlocks and unspecified receptions, as well as provide a communication service equivalent to the given service specification. Without having formulated a proof, if the precautions mentioned in Section 3.3. are taken, we are convinced that the derived protocols exhibit neither deadlocks nor unspecified receptions. The proof that the derived protocols provide a service equivalent to the given service specification is more difficult to establish. It requires the use of formal semantics of the language for the service and protocol specifications, and the determination of the "equivalence" to be considered. The translation of the specification language defined here into LOTOS [10] could be the basis for the semantics [12]. Various forms of equivalences [5, 10] may be used for a comparison of the specifications. The authors think that the service of the derived protocol specification is bisimulation equivalent to the given service specification, however, a proof of this fact goes beyond the work described here.

We will present the derivation algorithm in two steps, first dealing with synchronization only and then adding extensions to handle parameters. In order to define the algorithm, the formalism of attribute grammars [1] is used. An attribute grammar is a context free grammar augmented by attributes which are associated with the nodes of derivation trees. In order to define the values of the attributes at the nodes of a given tree, attribute evaluation rules are associated with the productions of the grammar. These rules are then applied to the instances of the corresponding productions in the derivation tree. Two kinds of attributes are distinguished. An attribute is *inherited*, if its value is determined by the evaluation rule associated with the production rule "above" the node. In many cases, it is immediately obtained from the parent node. Here information is passed down from the root toward the leaves. An attribute is *synthesized*, if its value is determined from the attributes of the immediate descendants and the applied production rule. Here information is passed up from the leaves towards the root.

To define the attribute evaluation rules, a distinction has to be made between the left side of a production rule and the subexpressions on its right side. We introduce indices for this purpose and rewrite the grammar of Section 2 as follows:

Rule 1: for each terminal symbol $\mathbf{x} \in \mathbf{SP}$: $\mathbf{e} \rightarrow \mathbf{x}$

Rule 2: $\mathbf{e} \rightarrow \mathbf{e}_1 \,;\, \mathbf{e}_2$

Rule 3: $\mathbf{e} \rightarrow \mathbf{e}_1 \,|||\, \mathbf{e}_2$

Rule 4: $\mathbf{e} \rightarrow \mathbf{e}_1 \,[]\, \mathbf{e}_2$
where the starting operations of $\mathbf{e}_1$ and $\mathbf{e}_2$ are located at one single place.

This notation does not affect the applicability of production rules. If a rule is applicable to the nonterminal symbol $\mathbf{e}$, then it can also be applied to $\mathbf{e}_1$ or $\mathbf{e}_2$.

3.1 Synchronization Between Places

3.1.1 *Informal Introduction.* As examples, we consider the service expressions $\mathbf{a}^1 ; (\mathbf{b}^2 \,|||\, \mathbf{c}^3)$ and $(\mathbf{a}^1 ; \mathbf{b}^2) \,[]\, (\mathbf{c}^1 ; \mathbf{d}^2)$. Since we want to discuss synchronization only among places to enforce the correct temporal ordering of service primitive executions, parameters are omitted throughout this section. Their addition is discussed in Section 3.2. Figures 5 and 6 also show the specifications of the protocol entities serving the places **1**, **2**, and **3** which the derivation algorithm should produce as results for the examples above.

We use the same specification language to specify protocol entities as for services. We write $\mathbf{T}_i(\mathbf{e}_s)$ for the specification of the protocol entity serving place $\mathbf{i}$, and derived from the service specification $\mathbf{e}_s$. In the case of service $\mathbf{e}_s$ (Figure 5), the specification $\mathbf{T}_1(\mathbf{e}_s)$ expresses that after the execution of $\mathbf{a}^1$, synchronization messages have to be sent to places **2** and **3** in any order to enable the service primitives $\mathbf{b}^2$ and $\mathbf{c}^3$. The notation $\mathbf{s}_2(1)$ denotes a send operation, where a message is sent to place **2**. The integer "1" is a synchronization parameter which is needed in certain cases to identify the received message unambiguously. The corresponding message reception is written

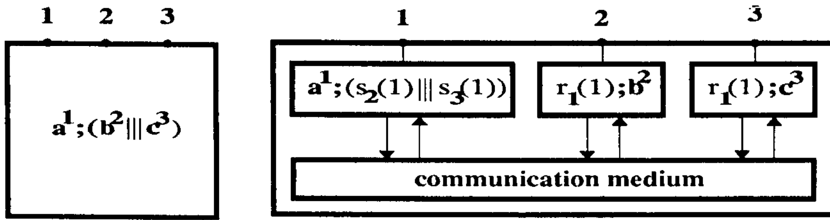

Fig. 5. Service $\mathbf{e}_s = \mathbf{a}^1;(\mathbf{b}^2 ||| \mathbf{c}^3)$ and the specification of the corresponding protocol entities.

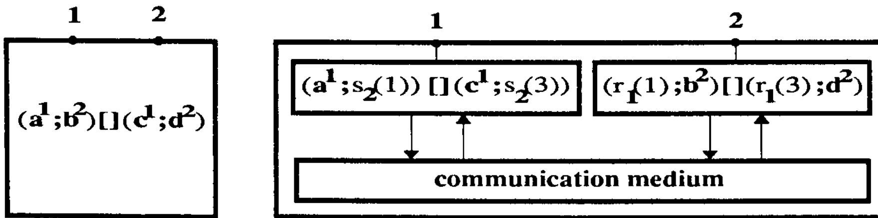

Fig. 6. Service $\mathbf{e}'_s = (\mathbf{a}^1;\mathbf{b}^2)[](\mathbf{c}^1;\mathbf{d}^2)$ and the specification of the corresponding protocol entities.

$r_1(1)$, a message is received from place 1. The send and receive operations constitute a protocol and are part of the specification of the protocol entities. Similarly, the protocol specification for service $\mathbf{e}'_s$ (Figure 6) can be interpreted. Note that it is one of the cases where we need the synchronization parameter to distinguish on the receiving side whether $\mathbf{b}^2$ or $\mathbf{d}^2$ is to be executed next.

In general, synchronization is required in all cases where the operator ";" is used in the service definition. Here, all terminating operations of the left subexpression of ";" have to send synchronization messages to all starting operations of its right subexpression. Similarly, all starting operations of the right subexpression have to receive synchronization messages from all terminating operations of the left subexpression.

In the case of parallel execution of service primitives, "|||", no synchronization is needed. Also, with the constraint concerning the places of starting operations in production Rule 4 (Section 2), no additional synchronization is required in case of alternatives.

The messages carry a synchronization parameter. It is chosen such that all messages signaling the completion of a particular service primitive to other places use the same parameter value. Thus messages related to different service completions are distinguishable by the receiver.

3.1.2 *Definition of the Derivation Algorithm.* The algorithm producing the specification of the protocol entities follows these steps:

Step 1. Construct the derivation tree of the given service expression $\mathbf{e}_s$.

Step 2. Synthesize attributes S and E at each node of the tree.

Step 3. Compute attributes P and F at each node of the tree.

Step 4. For each place p, compute $T_p(e_s)$ which is the specification of the protocol entity serving that place.

The values of the attributes **S**, **E**, **P**, and **F** are character strings that represent expressions of send and receive operations, respectively. Informally, they can be interpreted as follows:

S(e): send operations associated with the *starting places* of subexpression **e** (synthesized)

E(e): receive operations associated with the *ending places* of subexpression **e** (synthesized)

P(e): receive operations from the *preceding places* (inherited)

F(e): send operations to the *following places* (inherited)

Starting and ending places of an expression or subexpression are those places at which execution of the expression or subexpression is begun or finished. Preceding places are those places at which service primitives are executed directly before the execution of the considered expression or subexpressions takes place. This is analogous for following places.

The precise definition of the evaluation rules for the attributes **S**, **E**, **P** and **F** is given in Table I. For each leaf node x, the synthesized attributes **S** and **E** are initialized as follows:

$$\mathrm{S(x)} := \text{"s"}_{\mathbf{place(x)}}\text{"(z)"} \qquad \text{for all } \mathrm{x} \in \mathrm{SP}$$
$$\mathrm{E(x)} := \text{"r"}_{\mathbf{place(x)}}\text{"("N(x)")"} \quad \text{for all } \mathrm{x} \in \mathrm{SP}$$

Here, **place** is a function from the set **SP** of service primitives to the set of places, $\mathbf{place}(\mathbf{x}^{\mathbf{p}}) := \mathbf{p}$. The values of **place** are interpreted as strings. The different strings are implicitly concatenated. Thus, we get string values for the attributes **S** and **E** which later are incorporated into protocol expressions, "$s_p(z)$" or "$r_p(i)$" means that a synchronization message has to be, respectively, sent to, or received from, place **p**.

$N(x)$ is an additional attribute of the leaf nodes. It defines a unique numbering of all leaves of the derivation tree and can be obtained by parsing the tree from left to right. Its value is used for the initialization of the **E** attribute as stated above. The value **N(x)** is carried as a parameter by receive operations associated with the operation **x** in attribute **E**. Furthermore, a synchronization parameter z is associated with send operations (see attribute **S**, initialization) which is replaced by specific values later during the derivation process (definition of $\mathbf{T_p}$, Table II).

The attributes **S** and **E** are synthesized, that is, they are evaluated from the bottom, the leaves, to the top, the root, of the derivation tree. After initialization at all leaf nodes, the rules in Table I are applied to synthesize the values of **S** and **E** for the immediate parent nodes, and all others. For instance, for a parent node to which production Rule 2 has been applied, yielding two immediate descendant nodes, the value of **S** is equal to the value of **S** as evaluated for its left descendant node (see Table I). This process is repeated until the values of **S** and **E** for the root node have been synthesized.

Table I. Evaluation Rules for the Attributes **S**, **E**, **P** and **F**

	S	E	P	F
1	S(e):=S(x) for all x∈SP	E(e):=E(x) for all x∈SP	P(x):=P(e) for all x∈SP	F(x):=F(e) for all x∈SP
2	S(e):=S(e_1)	E(e):=E(e_2)	P(e_1):=P(e) P(e_2):=E(e_1)	F(e_1):=S(e_2) F(e_2):=F(e)
3	S(e):=S(e_1)"|||"S(e_2)	E(e):=E(e_1)"|||"E(e_2)	P(e_1):=P(e_2):=P(e)	F(e_1):=F(e_2):=F(e)
4	S(e):=S(e_1)	E(e):=E(e_1)"[]"E(e_2)	P(e_1):=P(e_2):="empty"	F(e_1):=F(e_2):=F(e)

Table II. Definition of the Function $\mathbf{T_p}$

production rule	T_p
1	T_p(e) := if place(x) = "p" then P(x) "; x ;" F(x)[z/N(x)] else "empty" for all x∈SP
2	T_p(e) := T_p(e_1) ";" T_p(e_2)
3	T_p(e) := T_p(e_1) "|||" T_p(e_2)
4	T_p(e) := if (S(e_1) = S(e_2) = "s_p(z)") then P(e) "; (" T_p(e_1) "[]" T_p(e_2) ")" else T_p(e_1) "[]" T_p(e_2)

The reasoning behind the attribute evaluation rules for **S** and **E** is the observation that synchronization messages have to be transmitted to **place(x)** from all *preceding places* and to be received by all *following places*. In case of production Rule 2, for example, the send operations associated with the *starting places* for the father node in the derivation tree are the same as for the left subexpression of the operator ";." The receive operations associated with *ending places* are the same as for the right subexpressions.

Based on the attribute **S**, we can now precisely define the constraint for production Rule 4:

Rule 4: e → e_1 [] e_2 where S(e_1) = S(e_2) = "s_p(z)" for some place **p**

This also explains why we can simplify the definition of the attribute evaluation rule for **S** in this case.

After having synthesized attributes **S** and **E**, we can now evaluate the inherited attributes **P** and **F** from the root node of the tree towards the

leaves, using the rules defined in Table I. **P** and **F** are initialized to *empty* at the root. The purpose of these rules is to obtain for each leaf node of the tree a pair of strings. These define the receptions that must be performed before the execution of the operation (attribute **P**), and the transmissions to be performed afterwards (attribute **F**). The attributes **P** and **F** depend on the synthesized attributes **S** and **E** as described in the definition covering production Rule 2.

The attributes **P** and **F** are used to derive, from the given service specification, the specification of the protocol entities. Let **p** be an arbitrary place, then the rules shown in Table II, applied recursively to the derivation tree of the service expression, provide a specification for the entity serving the place **p**. The specification is given by $\mathbf{T_p}$ applied to the root node of the service specification. Here, **F(x)[z/N(x)]** denotes **F(x)**, where all occurrences of *z* are substituted by the value of **N(x)**.

For alternatives, the rules of Table II specify that the receptions are inserted before a decision as to which branch to execute. This is the reason why we distinguish two cases in the definition of $\mathbf{T_p}$ covering production Rule 4 (Table II) and why we define $\mathbf{P(e_1)}$ and $\mathbf{P(e_2)}$ to be *empty* for this production rule (Table I).

In order to obtain the specifications for all protocol entities, $\mathbf{T_p}$ has to be applied for each place **p**. Let us consider a first example, the operations $\{\mathbf{a^1}, \mathbf{b^2}\}$ and the service expression $\mathbf{a^1} ; \mathbf{b^2}$. The derivation tree for this service and its attributes can be depicted as follows where "—" represents "**empty**".

The derivation of the protocol specifications for the places **1** and **2** leads to the following result:

$$
\begin{aligned}
T_1(e_s) &= T_1(a^1;b^2)\\
&= T_1(a^1)\ \text{";"}\ T_1(b^2)\\
&= P(a^1)\ \text{"; } a^1 \text{ ;"}\ F(a^1)[z/1]\ \text{"; empty"}\\
&= \text{"empty ; } a^1 \text{ ; } s_2(1) \text{ ; empty"}\\
&= \text{"} a^1 \text{ ; } s_2(1)\text{"}
\end{aligned}
$$

$$
\begin{aligned}
T_2(e_s) &= \ldots = \text{"empty ;"}\ P(b^2)\ \text{"; } b^2 \text{ ;"}\ F(b^2)[z/2]\\
&= \text{"} r_1(1) \text{ ; } b^2\text{"}
\end{aligned}
$$

This is obviously the result we were expecting. The protocol entity at place **1** first executes operation $\mathbf{a^1}$ and then sends a synchronization message $s_2(1)$ to place **2**, while the protocol entity at place **2** first receives this message from place **1** ($r_1(1)$) and then executes operation $\mathbf{b^2}$.

It should be noted that certain simplifications of expressions obtained during the process of derivation are permitted. Semantically, for arbitrary expressions **e**, $\mathbf{e_1}$, and $\mathbf{e_2}$, the following expressions are equivalent:

$$
\begin{aligned}
e \text{ ; empty} &\approx e\\
\text{empty ; } e &\approx e\\
e_1 \ ||| \ e_2 &\approx e_2 \ ||| \ e_1\\
e \ ||| \text{ empty} &\approx e\\
\text{empty [] empty} &\approx \text{empty}
\end{aligned}
$$

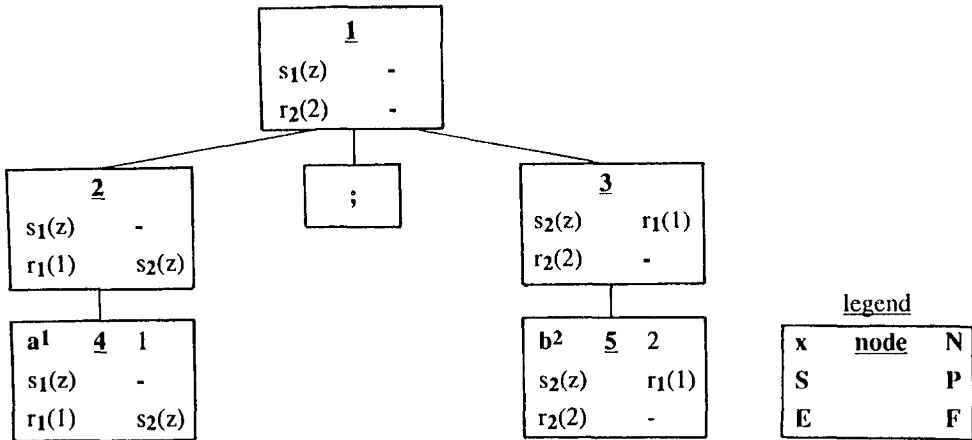

Fig. 7. Derivation tree and attributes for the service $\mathbf{e}_s = \mathbf{a}^1 ; \mathbf{b}^2$.

3.2 Exchange of Data Between Places

3.2.1 *Informal Introduction.* In this section, we discuss the inclusion of parameters in the service primitives. As mentioned in Section 2, a service primitive may have formal input and output parameters. On the protocol level, this requires, in addition to synchronization messages, that parameter values are exchanged among places. Additional send and receive operations are included for this purpose. We assume that a service primitive can only start its execution when the required synchronization messages have been received, and all specified input parameter values are available at its place.

Figure 8 shows the kind of expressions we are now dealing with and the desired result of the protocol derivation process. Note that we do not consider optimization issues at this point. The specification $\mathbf{T}_1(\mathbf{e}_s)$ expresses that after execution of $\mathbf{a}^1$, both synchronization and data messages have to be sent in any order to places **2** and **3** to enable $\mathbf{b}^2$ and $\mathbf{c}^3$. The corresponding receptions are part of $\mathbf{T}_2(\mathbf{e}_s)$ and $\mathbf{T}_3(\mathbf{e}_s)$.

The relationship between output and input parameters is established by the formal parameter name. In Figure 8, x_2 and x_3 are outputs of the service primitive $\mathbf{a}^1$ and inputs to the primitives $\mathbf{b}^2$ and $\mathbf{c}^3$, respectively. This convention allows us to specify where outputs are needed as inputs and thus gives us the necessary information for the protocol derivation. Each output has to be communicated to all places where it is needed as input.

As already mentioned in the explanation of Figure 3, parameter values which are not directly exchanged with the service users are called *intermediate parameters*. In Figure 8, x_2 and x_3 are intermediate parameters. An additional message must be exchanged between the protocol entities of the places where the value of an intermediate parameter is produced as output and the place or places where it is used as input to a service primitive.

3.2.2 *Principles of the Derivation Algorithm.* Since for alternative expressions $\mathbf{e}_1[]\mathbf{e}_2$ only one side is chosen for execution, we call each side *branch*

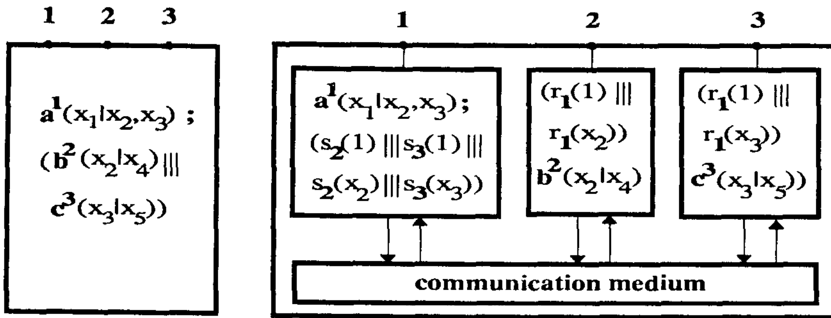

Fig. 8. Service $\mathbf{e}_s = \mathbf{a}^1(x_1 \mid x_2, x_3); ((\mathbf{b}^2(x_2 \mid x_4) ||| \mathbf{c}^3(x_3 \mid x_5))$ and the specification of the corresponding protocol entities.

(or *alternative*) throughout this section. A starting service (primitive) of a branch is denoted *first element* of that branch, all other services and service primitives of a branch are *successive elements.*

The data flow between input and output parameters should be checked for consistency (static semantics). The following restrictions apply:

R1. Parameters names may only denote inputs of a service (primitive) **x** if they occur as outputs of a service (primitive) **y** to be executed before **x**, or if they are supplied by the service user. An exception of this rule applies to alternatives: Here, inputs to successive elements of either alternative must occur as outputs earlier in the same branch.

R2. Parameter names may only be used once to denote an output (uniqueness of service results), except in the case of alternatives. Here, the same parameter name may be used in either branch to denote an output.

R3. Parameter names may only be used to denote outputs if they have not been used to denote an input in a preceding element of the service specification.

The motivation for the exception in case of alternatives and restriction R1 is that if inputs to the second or successive elements of a branch are produced before the branching decision (recall that this decision is taken at the place where the first element of the branch is executed), it is generally difficult to decide whether they are needed as inputs.

Example. $\mathbf{a}^1(x_1 \mid x_2, x_3); ((\mathbf{b}^2(x_2 \mid x_4); \mathbf{c}^3(x_3, x_4 \mid x_5)) [\,] \mathbf{d}^2(x_2 \mid x_5))$ is an expression not fulfilling R1, since it is unclear whether x_3 will be needed as input after execution of $\mathbf{a}^1$.

The exception in restriction R2 is motivated by the observation that parameters needed as inputs after completion of the alternative expression must be available independent of the selected branch.

Example. $(\mathbf{a}^1(x_1 \mid x_2, x_3)[\,]\mathbf{b}^1(x_1 \mid x_2, x_4));\mathbf{c}^2(x_2 \mid x_5)$. Here, x_2 is available as input for $\mathbf{c}^2$, since it occurs as output in both preceding branches.

R3 is due to the derivation formalism.

Example. $\mathbf{a}^1(x_1 \mid x_2);\mathbf{b}^2(x_2 \mid x_1)$ is an expression not fulfilling R3, because of parameter x_1.

We will now extend the definition of $\mathbf{T_p}$ (Section 3.1.2.) such that the derived protocols handle the exchange of parameters. Informally, this means that each parameter value has to be sent (received) from each place where it occurs as output to (at) each place where it is needed as input. We will define several new attributes including **P*** and **F*** which are then used to extend the definition of $\mathbf{T_p}$.

To extract the information for additional transmissions, we proceed through the following steps:

Step 1. Construct a bag containing all tuples $\langle i, x_j \rangle$, where x_j is an input parameter name and **i** is the place where x_j is needed.

Step 2. For each leaf of the derivation tree, select those tuples of the bag constructed in Step 1 for which the service primitive attached to this leaf produces the corresponding output.

Step 3. From the bag resulting from Step 2, construct the expression specifying the transmissions for each leaf.

We use the concept of bags, because tuples may occur more than once in some cases, and we do not consider optimization issues. Steps 1 and 3 will lead to correct results because of restriction R2.

To obtain the information for additional receptions, we proceed as follows:

Step 4. Construct a set of input parameter names for each node of the derivation tree such that:

(a) at nodes, to which production Rule 4 is applied, the set contains the names of input parameters to be received before the decision which branch will be executed is taken, that is, input parameter names of the first element of either branch;

(b) at leaves the set contains the names of input parameters of the attached service primitive, except in cases where the service is the first element of an alternative branch.

Step 5. For each node, construct a bag containing all tuples $\langle i, x_j \rangle$ of the corresponding subtree, where x_j is an output parameter name, and i is the place where x_j occurs.

Step 6. For each node of the derivation tree, select those tuples of the bag constructed in Step 5 which may be received before or during execution of the subtree defined by the considered node (potential receptions).

Step 7. For each node of the derivation tree, select those tuples of the bag, constructed in Step 6, which have to be received by the

considered node before execution may start. These are actual receptions. This is done by comparing each element of the bag with the set obtained in Step 4.

Step 8. From the bag resulting from Step 7, construct the expression specifying the receptions for each node.

Again, we use the concept of bags, because in alternative expressions, the same output parameter name may be used in either branch at the same place.

3.2.3 *Definition of the Derivation Algorithm.* We first of all give an algorithm to check for R1, R2 and R3. It is based on the derivation tree of service expressions and uses attributes **R12**, $\mathbf{R1_l}$, $\mathbf{R1_r}$, $\mathbf{R3_a}$ and $\mathbf{R3_b}$, which have the following meaning:

R12 collects the names of output parameters available *after* the services of the corresponding subtree are executed (synthesized).

$\mathbf{R1_l}$ collects the names of output parameters available *before* the services of the corresponding subtree are executed (inherited).

$\mathbf{R1_r}$ ensures that only output parameters from the same branch are collected as available outputs by $\mathbf{R1_l}$ in case of successive elements in alternatives (restriction R1).

$\mathbf{R3_a}$ collects parameter names used to denote inputs in the corresponding subtree (synthesized).

$\mathbf{R3_b}$ collects parameter names which have been used to denote inputs before the services of the corresponding subtree are executed (inherited).

$$\mathrm{R12(x)} := \{\mathrm{opar(x)}\} \quad \text{for all } \mathrm{x} \in \mathrm{SP}$$
$$\mathrm{R3_a(x)} := \{\mathrm{ipar(x)}\} \quad \text{for all } \mathrm{x} \in \mathrm{SP}$$

The attribute evaluation rules for **R12**, $\mathbf{R1_l}$ and $\mathbf{R1_r}$ are shown in Table III. $\mathbf{R1_l}$, $\mathbf{R1_r}$ and $\mathbf{R3_b}$ are initialized with the empty set at the root node. **R12** and $\mathbf{R3_a}$ are initialized as follows:
Here, $\{opar(\mathbf{x})\}$ denotes the set of formal output parameters, $\{ipar(\mathbf{x})\}$ the set of formal input parameters of the service primitive **x**.

The attributes $\mathbf{R3_a}$ and $\mathbf{R3_b}$ are evaluated like **R12** and $\mathbf{R1_l}$, respectively, except

- rule 4 for $\mathbf{R3_a}$ which is $\mathbf{R3_a(e) := R3_a(e_1) \cup R3_a(e_2)}$
- rule 1 for $\mathbf{R3_b}$ which is $\mathbf{R3_b(x) := R3_b(e) \cup \{ipar(x)\}}$ for all $\mathbf{x} \in \mathbf{SP}$
- and rule 2 for $\mathbf{R3_b}$ which is $\mathbf{R3_b(e_1) := R3_b(e)}$

$$\mathbf{R3_b(e_2) := R3_b(e) \cup R3_a(e_1)}$$

Now we can define the restrictions R1, R2 and R3 precisely, as shown in Table IV.

Note that the case where input parameters are supplied by the service user is not taken into account by this definition of R1. This is done automatically when the "dummy" service primitive $\mathbf{i^P}$ is introduced in Section 4.

Table III. Evaluation Rules for the Attributes **R12**, **R1$_l$** and **R1$_r$**

	R12	R1$_l$	R1$_r$
1	$R12(e) := R12(x)$ for all $x \in SP$	$R1_l(x) := R1_l(e)$ for all $x \in SP$	$R1_r(x) := R1_l(e)$ for all $x \in SP$
2	$R12(e) := R12(e_1) \cup R12(e_2)$	$R1_l(e_1) := R1_l(e)$ $R1_l(e_2) := R1_r(e) \cup R12(e_1)$	$R1_r(e_1) := R1_r(e)$ $R1_r(e_2) := R1_r(e) \cup R12(e_1)$
3	$R12(e) := R12(e_1) \cup R12(e_2)$	$R1_l(e_1) := R1_l(e_2) := R1_l(e)$	$R1_r(e_1) := R1_r(e_2) := R1_l(e)$
4	$R12(e) := R12(e_1) \cap R12(e_2)$	$R1_l(e_1) := R1_l(e_2) := R1_l(e)$	$R1_r(e_1) := R1_r(e_2) := \{\ \}$

Remarks:

1. The evaluation rule for **R12** and production rule 4 expresses that only outputs occurring in *both* branches of the alternative are assumed to be available afterwards.
2. The evaluation rule for **R1$_l$** and production rule 4 expresses that previous outputs are permitted as inputs for the *first* service primitive of each branch of the alternative.
3. The evaluation rule for **R1$_r$** and production rule 4 expresses that only outputs of the same branch are available as inputs for subsequent services/service primitives (restriction R1). Therefore, all names of previous output parameters are removed.
4. In the definition of **R1$_l$** and **R1$_r$** for production rule 2, $R1_r(e) \cup R12(e_1)$ is exactly the set of output parameter names available after the preceding part of the current branch according to restriction R1.

Table IV. Definition of the Restrictions **R1**, **R2** and **R3**

	R1	R2	R3
1	$\{ipar(x)\} \subseteq R1_l(x)$ for all $x \in SP$	-	$R3_b(x) \cap \{opar(x)\} = \{\ \}$
2	-	$R12(e_1) \cap R12(e_2) = \{\ \}$	-
3	-	$R12(e_1) \cap R12(e_2) = \{\ \}$	-
4	-	-	-

The following attributes are defined in Table V:

S* constructs a bag containing all tuples $\langle i, x_j \rangle$, see Step 1 in Section 3.2.2 (synthesized).

F* selects tuples according to Step 2 in Section 3.2.2 (inherited).

Table V. Evaluation Rules for the Attributes S* and F*

	S*	F*
1	$S^*(e) := S^*(x)$ for all $x \in SP$	$F^*(x) := F^*(e) \setminus_b \{ <i,x_j> \mid i \text{ is a place} \wedge x_j \notin \{opar(x)\} \}$ for all $x \in SP$
2	$S^*(e) := S^*(e_1) +_b S^*(e_2)$	$F^*(e_1) := F^*(e)$ $F^*(e_2) := F^*(e) -_b S^*(e_1)$
3	$S^*(e) := S^*(e_1) +_b S^*(e_2)$	$F^*(e_1) := F^*(e) -_b S^*(e_2)$ $F^*(e_2) := F^*(e) -_b S^*(e_1)$
4	$S^*(e) := S^*(e_1) +_b S^*(e_2)$	$F^*(e_1) := F^*(e) -_b S^*(e_2)$ $F^*(e_2) := F^*(e) -_b S^*(e_1)$

Remarks:

1. We use operators "$-_b$", "$+_b$"and "$\setminus_b$" to express operations on bags: "$-_b$" subtracts a bag from a bag; "$+_b$" denotes the union of two bags; "$\setminus_b$" subtracts a set from a bag. "$\{\}_b$" denotes the empty bag.
2. S* is initialized with $\{ <i,x_1>,\ldots,<i,x_n> \}_b$ for all $x^i \in SP$, where $ipar(x^i)=x_1,\ldots,x_n$, at all leaf nodes.
3. F* is initialized with the value of S* at the root node.
4. The evaluation rule for F* and production rule 2 could be simplified to $F^*(e_2) := F^*(e)$, because R1 enforces that outputs may occur only *before* they are needed as inputs. However, the given rule yields shorter attribute values.
5. The evaluation rule for F* and production 3 could be simplified to $F^*(e_{1,2}) := F^*(e)$, because R1 enforces that outputs in one branch must not be inputs in the other branch. However, the given rule yields shorter attribute values.
6. The evaluation rule for F* and production rule 4 expresses that one alternative has been chosen, and therefore the inputs for the other branch are not needed.

Step 3 of Section 3.2.2 is included in the new definition of T_p; see Table VIII.
The following attributes are defined in Tables VI and VII:

- I_s* collects input parameter names as needed to define I_i* (synthesized)
- I_i* constructs a set of input parameter names as required by step 4 in Section 3.2.2. (inherited)
- E* constructs bags of tuples $<i,x_j>$ according to step 5 in Section 3.2.2. (synthesized)
- O* selects tuples according to step 6 in Section 3.2.2. (inherited)
- P* selects tuples according to step 7 in Section 3.2.2. (synthesized)

Step 8 of Section 3.2.2 is included in the new definition of T_p (Table VIII). This is an extension of the definition given in Table II. It is extended by

Table VI. Evaluation Rules for the Attributes I_s^* and I_i^*

	I_s^*	I_i^*
1	$I_s^*(e) := I_s^*(x)$ for all $x \in SP$	$I_i^*(x) := I_i^*(e)$ for all $x \in SP$
2	$I_s^*(e) := I_s^*(e_1)$	$I_i^*(e_1) := I_i^*(e)$ $I_i^*(e_2) := I_s^*(e_2)$
3	$I_s^*(e) := I_s^*(e_1) \cup I_s^*(e_2)$	$I_i^*(e_1) := I_s^*(e_1)$ $I_i^*(e_2) := I_s^*(e_2)$
4	$I_s^*(e) := I_s^*(e_1) \cup I_s^*(e_2)$	$I_i^*(e_1) := I_i^*(e_2) := \{\ \}$

Remarks:

1. I_s^* is initialized with $\{ipar(x)\}$ at each leaf, where $x \in SP$ is the service primitive attached to that leaf.
2. I_i^* is initialized to be empty at the root node.

adding transmissions and receptions of parameter values using the attributes **F*** and **P***, respectively.

Both the parameter values to be sent and their destination are determined by the attribute **F***. A parameter value should be sent immediately after it is produced as an output. If more than one parameter value results from a service primitive or one value has to be sent more than once, the required messages may be sent in arbitrary order. Therefore, we transform the value of **F*** into the following string which is then included into the protocol specification (Table VIII, production Rule 1):

$$\mathbf{trans}(F^*(.)) := \begin{cases} \text{"empty"} \quad \text{if } F^*(.) = \{\ \}_b \\ \text{"}s_{i_1}(x_{j_1}) \| s_{i_2}(x_{j_2}) \| \ldots \| s_{i_n}(x_{j_n})\text{"} \quad \text{if } F^*(.) = \{<i_1, x_{j_1}>, \ldots, <i_n, x_{j_n}>\}_b \neq \{\ \}_b \end{cases}$$

Both the parameter values to be received and their source are determined by the attribute **P***. Each of these parameter values may only be produced by one service primitive, except in case of alternatives where it has to be produced in either branch (restriction R2). Since only one branch is executed dynamically, this choice must be reflected in the string to be incorporated in the protocol specification. Each parameter may be received from different

Table VII. Evaluation Rules for the Attributes **E***, **O*** and **P***

	$\mathbf{E}^*$	$\mathbf{O}^*$	$\mathbf{P}^*$
1	$\mathbf{E}^*(e) := \mathbf{E}^*(x)$ for all $x \in \mathbf{SP}$	$\mathbf{O}^*(x) := \mathbf{O}^*(e) -_b \mathbf{E}^*(x)$ for all $x \in \mathbf{SP}$	$\mathbf{P}^*(e) := \{\ \}_b$
2	$\mathbf{E}^*(e) := \mathbf{E}^*(e_1) +_b \mathbf{E}^*(e_2)$	$\mathbf{O}^*(e_1) := \mathbf{O}^*(e) -_b \mathbf{E}^*(e_2)$ $\mathbf{O}^*(e_2) := \mathbf{O}^*(e)$	$\mathbf{P}^*(e) := \{\ \}_b$
3	$\mathbf{E}^*(e) := \mathbf{E}^*(e_1) +_b \mathbf{E}^*(e_2)$	$\mathbf{O}^*(e_1) := \mathbf{O}^*(e) -_b \mathbf{E}^*(e_2)$ $\mathbf{O}^*(e_2) := \mathbf{O}^*(e) -_b \mathbf{E}^*(e_1)$	$\mathbf{P}^*(e) := \{\ \}_b$
4	$\mathbf{E}^*(e) := \mathbf{E}^*(e_1) +_b \mathbf{E}^*(e_2)$	$\mathbf{O}^*(e_1) := \mathbf{O}^*(e) -_b \mathbf{E}^*(e_2)$ $\mathbf{O}^*(e_2) := \mathbf{O}^*(e) -_b \mathbf{E}^*(e_1)$	$\mathbf{P}^*(e) := \mathbf{O}^*(e) \setminus_b \{ \langle i,x_j \rangle \mid \mathbf{i}$ is a place $\wedge\, x_j \notin \mathbf{I}_i^*(e) \}$

Remarks:

1. $\mathbf{E}^*$ is initialized with $\{ \langle i,x_i \rangle,\ldots,\langle i,x_m \rangle \}_b$, where $x^i \in \mathbf{SP}$ is the service primitive attached to that leaf, and $opar(\mathbf{x^i}) = x_i,\ldots,x_m$.
2. $\mathbf{O}^*$ is initialized with the value of $\mathbf{E}^*$ at the root node.
3. $\mathbf{P}^*$ is initialized with $\mathbf{O}^*(x) \setminus_b \{ \langle i,x_j \rangle \mid \mathbf{i}$ is a place $\wedge\, x_j \notin \mathbf{I}_i^*(x) \}$ at each leaf.
4. The evaluation rules for $\mathbf{I}_s^*$ express that only the input parameter names of the first element (see production rule 2) of both alternatives (see production rule 4) are considered.
5. The evaluation rules for $\mathbf{O}^*$ and production rules 2, 3 and 4 reflect the fact that outputs of $\mathbf{e_1}$ ($\mathbf{e_2}$) are not available as inputs for $\mathbf{e_2}$ ($\mathbf{e_1}$) and therefore are no potential receptions. Exception: $\mathbf{e_2}$ in production rule 2.

places according to the chosen branch. Therefore, we regroup and transform the value of **P*** as follows:

$$
\mathbf{rec}(\mathbf{P}^*(.)) := \begin{cases} \text{"empty"} \ \text{ if } \ \mathbf{P}^*(.) = \{\ \}_b \\[1em] \text{"}(r_{i_{1,1}}(x_{j_1})[]\ldots[]r_{i_{n,1}}(x_{j_1})) \mid\mid\mid (r_{i_{1,2}}(x_{j_2})[]\ldots[]r_{i_{n,2}}(x_{j_2})) \mid\mid\mid \ldots \\ \quad \mid\mid\mid (r_{i_{1,m}}(x_{j_m})[]\ldots[]r_{i_{n,m}}(x_{j_m}))\text{"} \\ \qquad \text{if } \ \mathbf{P}^*(.) = \{\langle i_{1,1},x_{j_1}\rangle,\ldots,\langle i_{n,1},x_{j_1}\rangle,\ldots,\langle i_{n,m},x_{j_m}\rangle\}_b \neq \{\}_b \\ \qquad\quad \wedge (\forall k,l)(1 \le k < l \le m \Rightarrow x_{j_k} \neq x_{j_l}) \end{cases}
$$

As shown in Table VIII, receptions have to be incorporated before a service primitive (production Rule 1) and before an alternative (production Rule 4).

Table VIII. Definition of the Function T_p Including Transmission of Parameters

production rule	T_p
1	$T_p(e)$:= if place(x) = "p" then "(" P(x) "|||" rec(P*(x)) ") ; x ; (" F(x){z/N(x)} "|||" trans(F*(x)) ")" else "empty" for all x∈SP
2	$T_p(e) := T_p(e_1)$ ";" $T_p(e_2)$
3	$T_p(e) := T_p(e_1)$ "|||" $T_p(e_2)$
4	$T_p(e)$:= if ($S(e_1) = S(e_2)$ = "$s_p(z)$") then "("P(e) "|||" rec(P*(e)) ") ; (" $T_p(e_1)$ "[]" $T_p(e_2)$ ")" else $T_p(e_1)$ "[]" $T_p(e_2)$

In the latter case, the definition of T_p assures that all necessary receptions occur before the choice is made, which means that input parameter values for both branches are received.

3.3 Assumptions about the Communication Medium

In this paper we assume that the underlying communication medium provides for reliable message transmission between different service access points (called *places*). Concerning the order of message reception, the following three schemes may be assumed:

(1) For every given pair of source and destination places, the messages are received in the same order as they have been sent. However, at each place, there is single queue with FIFO discipline through which all incoming messages are received by the protocol entity.

(2) As above, but at each place there is a separate input queue for each source.

(3) Messages from any source, as they arrive at the destination place, are stored in a reception buffer from which they may be received by the local protocol entity in an order determined by the behavior of the protocol entity. No assumption about the transmission order is made.

The reception scheme (1) may lead to an unspecified reception in the sense of [24] for certain protocols derived by this method; an example is given in [12]. The reception scheme (2) is compatible with the derived protocols as long as no parameters are involved. In the presence of parameters, unspecified receptions may occur in some cases. These problems can be avoided by deriving protocol specifications based on an optimized message exchange protocol [12], or by using the reception scheme (3).

4. COMPOSITION OF SERVICES

So far, we have assumed that services are described by expansions which are composed of service primitives and operators. By giving names to these expressions we extend our service language: The names of already defined (composed) services in addition to service primitives, may be used to compose new service expressions. Note that a composed service, like service primitives, is associated with the place where its execution is initiated.

The inclusion of parameters into this concept is straightforward. A service may have formal input and output parameters just as a service primitive. Input values are assumed to be available at the place where the service is accessible. Output values will be made available by the service at the same or a remote place after execution. Thus, we can abstract from the fact that the execution of the service might actually involve several places.

Since we allow output values of a service to be available at a remote place, we have to extend our notation. Instead of writing $\mathbf{x}^{\mathbf{p}}$ (Section 2), we write $\mathbf{x}^{\mathbf{p},\mathbf{p}'}$, meaning that the service $\mathbf{x}$ is accessible at place $\mathbf{p}$ and provides its results at place $\mathbf{p}'$. For notational convenience, we continue writing $\mathbf{x}^{\mathbf{p}}$ instead of $\mathbf{x}^{\mathbf{p},\mathbf{p}}$.

Example: $\mathbf{y}^{1,2}(x_1 \mid x_3) := \mathbf{b}^1(x_1 \mid x_2); \mathbf{c}^2(x_2 \mid x_3)$
$\mathbf{z}^2(x_4 \mid x_5, x_6) := \mathbf{y}^{1,2}(x_4 \mid x_5) \parallel\!| \ \mathbf{d}^3(x_4 \mid x_6)$

In the example, a service $\mathbf{y}^{1,2}$ is defined as the sequential execution of the service primitives $\mathbf{b}^1$ and $\mathbf{c}^2$ at the places **1** and **2**, respectively. This service is then used to specify another service $\mathbf{z}^2$. Inputs on the left side of the service definition may occur as inputs on the right side. Outputs on the left side must be produced by the right side. Additionally, parameters denoting intermediate results may be used on the right side. In the case of service $\mathbf{y}^{1,2}$, for instance, the x_2 parameter produced as output by the $\mathbf{b}^1$ service primitive must be transfered to place **2** where it is used as input for the $\mathbf{c}^2$ service primitive.

A service primitive is a unit which can not be decomposed further. It is important to note that while service primitives have to be executed at the place where they are made available, this is not the case for (composed) services. As can be seen in the example, the service $\mathbf{z}^2$ is made available at place **2**, but uses services and service primitives accessible via places **1** and **3**.

How can we handle this concept of abstraction with our derivation algorithm? Recall that we assume input parameter values to be available at the place where the (composed) service is initiated. We want output parameter values to be made available at the same or a remote place after termination. To incorporate this assumption and requirement into the derivation process, we rewrite an arbitrary service expression as follows:

$$\mathbf{x}^{\mathbf{p},\mathbf{p}'}(x_1, \ldots, x_m \mid x_{m+1}, \ldots, x_n) := \text{expression}$$

$$\mathbf{x}^{\mathbf{p},\mathbf{p}'}(x_1, \ldots, x_m \mid x_{m+1}, \ldots, x_n) := \mathbf{i}^{\mathbf{p}}(\mid x_1, \ldots, x_m);$$
$$(\text{expression}; \mathbf{o}^{\mathbf{p}'}(x_{m+1}, \ldots, x_n \mid))$$

Here, $\mathbf{i}^{\mathbf{p}}$ is a "dummy" service primitive whose only purpose is to make the input values to $\mathbf{x}^{\mathbf{p},\mathbf{p}'}$ formally available at the place $\mathbf{p}$, such that the

derivation algorithm can handle it. Also, this convention makes restriction R1 as defined in Section 3.2.3 applicable to user-supplied inputs. Similarly, the only purpose of $\mathbf{o}^{\mathbf{p}'}$ is to enforce a data exchange such that output values become available at place $\mathbf{p}'$ independent of where they actually were produced. Both $\mathbf{i}^{\mathbf{p}}$ and $\mathbf{o}^{\mathbf{p}'}$ can later be removed from the resulting protocol specification.

Some minor changes to the definition of the attribute evaluation rules and the definition of $\mathbf{T_p}$ (Section 3) become necessary, because we allow services providing their results at a remote place. Let $\mathbf{SP}' =_{Df} \{\mathbf{a}^{\mathbf{r},\mathbf{r}'}, \mathbf{b}^{\mathbf{s},\mathbf{s}'}, \ldots, \mathbf{z}^{\mathbf{t},\mathbf{t}'}\}$:

a) Instead of a single function **place** (see Section 3.1.2.), we define two functions $\mathbf{place_s}$ and $\mathbf{place_e}$ from the set **SP'** to the set of places: $\mathbf{place_s}(\mathbf{x}^{\mathbf{p},\mathbf{p}'})$ = "**p**", $\mathbf{place_e}(\mathbf{x}^{\mathbf{p},\mathbf{p}'})$ = "**p'**".

b) The attributes S and E (see Section 3.1.2.) are initialized at all leaf nodes as follows:

S(x) := "s"$_{\mathbf{place_s}(x)}$"(z)"   for all $x \in$ **SP'**
E(x) := "r"$_{\mathbf{place_e}(x)}$"("N(x)")"   for all $x \in$ **SP'**

c) The definition of $\mathbf{T_p}$ covering production rule 1 (see Table 2) is changed to:

```
Tp(e) := if   places(x) = placee(x) = "p"
              then P(x) "; x ;" F(x)[z/N(x)]
              else if  places(x) = "p"
                       then P(x) "; x"
                       else if  placee(x) = "p"
                                then F(x)[z/N(x)]
                                else "empty"
```

d) The attributes S* and E* (see Section 3.2.3.) are initialized at all leaf nodes as follows:

$\mathbf{S}^*(\mathbf{x}^{\mathbf{i},\mathbf{j}}) := \{<i,x_1>,\ldots,<i,x_m>\}_b$ for all $\mathbf{x}^{\mathbf{i},\mathbf{j}} \in$ **SP'** where ipar($\mathbf{x}^{\mathbf{i},\mathbf{j}}$) = $x_1,\ldots,x_m$

$\mathbf{E}^*(\mathbf{x}^{\mathbf{i},\mathbf{j}}) := \{<j,x_{m+1}>,\ldots,<i,x_n>\}_b$ for all $\mathbf{x}^{\mathbf{i},\mathbf{j}} \in$ **SP'** where opar($\mathbf{x}^{\mathbf{i},\mathbf{j}}$) = $x_{m+1},\ldots,x_n$

e) The definition of $\mathbf{T_p}$ covering production rule 1 (see Table 8) is changed to:

```
Tp(e) := if   places(x) = placee(x) = "p"
              then "("P(x) "|||" rec(P*(x)) "); x ;(" F(x)[z/N(x)] "|||" trans(F*(x)) ")"
              else if  places(x) = "p"
                       then "("P(x) "|||" rec(P*(x)) "); x"
                       else if  placee(x) = "p"
                                then "("F(x)[z/N(x)] "|||" trans(F*(x)) ")"
                                else "empty"
```

The approach described above includes a nice property. The abstraction established on the service level is maintained on the protocol level. Therefore, we can apply our algorithm to each service expression separately independent of its usage in other service expressions.

Example. The service definitions above lead to the following protocol specifications:

$T_1(y^{1,2}(x_1|x_3))$ = "$(s_1(1) ||| s_1(x_1))$; $(r_1(1) ||| r_1(x_1))$; $b^1(x_1|x_2)$; $(s_2(2) ||| s_2(x_2))$"
$T_2(y^{1,2}(x_1|x_3))$ = "$(r_1(2) ||| r_1(x_2))$; $c^2(x_2|x_3)$; $(s_2(3)|||s_2(x_3))$; $(r_2(3) ||| r_2(x_3))$"

$T_1(z^2(x_4|x_5,x_6))$ = "$(r_2(1) ||| r_2(x_4))$; $y^{1,2}(x_4|x_5)$"
$T_2(z^2(x_4|x_5,x_6))$ = "$(s_1(1) ||| s_3(1) ||| s_1(x_4) ||| s_3(x_4))$; $(s_2(2) ||| s_2(x_5))$;
$(r_2(2) ||| r_3(3) ||| r_2(x_5)) ||| r_3(x_6))$"
$T_3(z^2(x_4|x_5,x_6))$ = "$(r_2(1) ||| r_2(x_4))$; $d^3(x_4|x_6)$; $(s_2(3) ||| s_2(x_6))$"

5. EXAMPLES

In the following, we give some additional examples intended to demonstrate both the application and the capabilities of the algorithm. Example 1 considers synchronization between places only. Examples 2 and 3 also take parameters into account.

Example 1.

s^1 := $(a^1 ||| b^2)$; $(c^3 [] d^3)$
$T_1(s^1)$ = "$(s_1(1) ||| s_2(1))$; $r_1(1)$; a^1 ; $s_3(2)$; $(r_3(4) [] r_3(5))$"
$T_2(s^1)$ = "$r_1(1)$; b^2 ; $s_3(3)$"
$T_3(s^1)$ = "$(r_1(2) ||| r_2(3))$; $(c^3$; $s_1(4) [] d^3$; $s_1(5))$"

Here, a service $\mathbf{s}^1$ is defined which is accessible via place 1. The derived protocol specification clearly defines that the initiative for the execution of the service comes from place 1 ($\mathbf{T}_1(\mathbf{s}^1)$), and after the service is completely executed, control returns to place 1. This is obtained by inserting "dummy" service primitives $\mathbf{i}^1$ and $\mathbf{o}^1$ as described in Section 4 before starting the derivation process.

Example 2. $\mathbf{s}^1(x_1 \mid x_3) := \mathbf{a}^1(x_1 \mid x_2) ; \mathbf{b}^2(x_2 \mid x_3)$. In the derivation tree (Figure 10), the values of the attributes $\mathbf{S}^*$, $\mathbf{F}^*$, $\mathbf{I}^*_s$, $\mathbf{I}^*_i$, $\mathbf{E}^*$, $\mathbf{O}^*$ and $\mathbf{P}^*$ are shown for each node. The derivation leads to the following protocol expressions:

$T_1(s^1(x_1|x_3))$ = "$(s_1(1) ||| s_1(x_1))$; $(r_1(1) ||| r_1(x_1))$; $a^1(x_1|x_2)$;
$(s_2(2) ||| s_2(x_2))$; $(r_2(3) ||| r_2(x_3))$"
$T_2(s^1(x_1|x_3))$ = "$(r_1(2) ||| r_1(x_2))$; $b^2(x_2|x_3)$; $(s_1(3) ||| s_1(x_3))$"

Example 3. $\mathbf{s}^1(x_1 \mid x_4) :=$
$(\mathbf{a}^1(x_1 \mid x_2) ||| \mathbf{b}^2(x_1 \mid x_3)) ; (\mathbf{c}^3(x_2 \mid x_4) [] \mathbf{d}^3(x_3 \mid x_4))$

$T_1(s^1(x_1|x_4))$ = "$(s_1(1) ||| s_2(1) ||| s_1(x_1) ||| s_2(x_1))$; $(r_1(1) ||| r_1(x_1))$; $a^1(x_1|x_2)$;
$(s_3(2) ||| s_3(x_2))$; $((r_3(4) [] r_3(5)) ||| (r_3(x_4) [] r_3(x_4)))$"
$T_2(s^1(x_1|x_4))$ = "$(r_1(1) ||| r_1(x_1))$; $b^2(x_1|x_3)$; $(s_3(3) ||| s_3(x_3))$"
$T_3(s^1(x_1|x_4))$ = "$(r_1(2) ||| r_2(3) ||| r_1(x_2) ||| r_2(x_3))$;
$(c^3(x_2|x_4)$; $(s_1(4) ||| s_1(x_4))) [] (d^3(x_3|x_4)$; $(s_1(5) ||| s_1(x_4)))$"

Here, we have augmented the service of Example 1 by parameters. The derived protocol specification clearly shows how the entity associated with

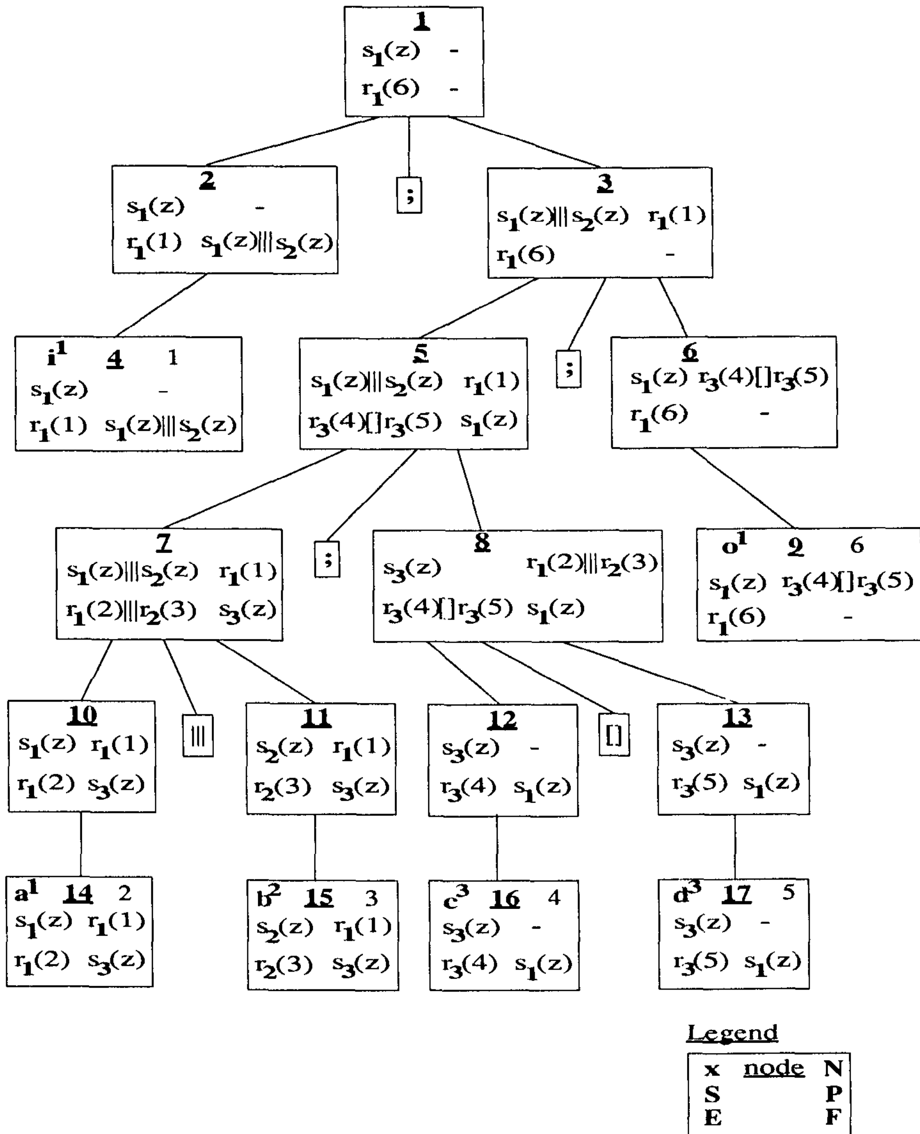

Fig. 9. Derivation tree with attributes for Example 1.

place 1 initiates the service execution and how it collects the results afterwards.

6. DISCUSSION, EXTENSIONS, AND APPLICATIONS

We have presented an algorithm that fully automates the derivation of protocol specifications from service specifications. Since it is a constructive approach, the algorithm aims at avoiding certain design errors a priori. For

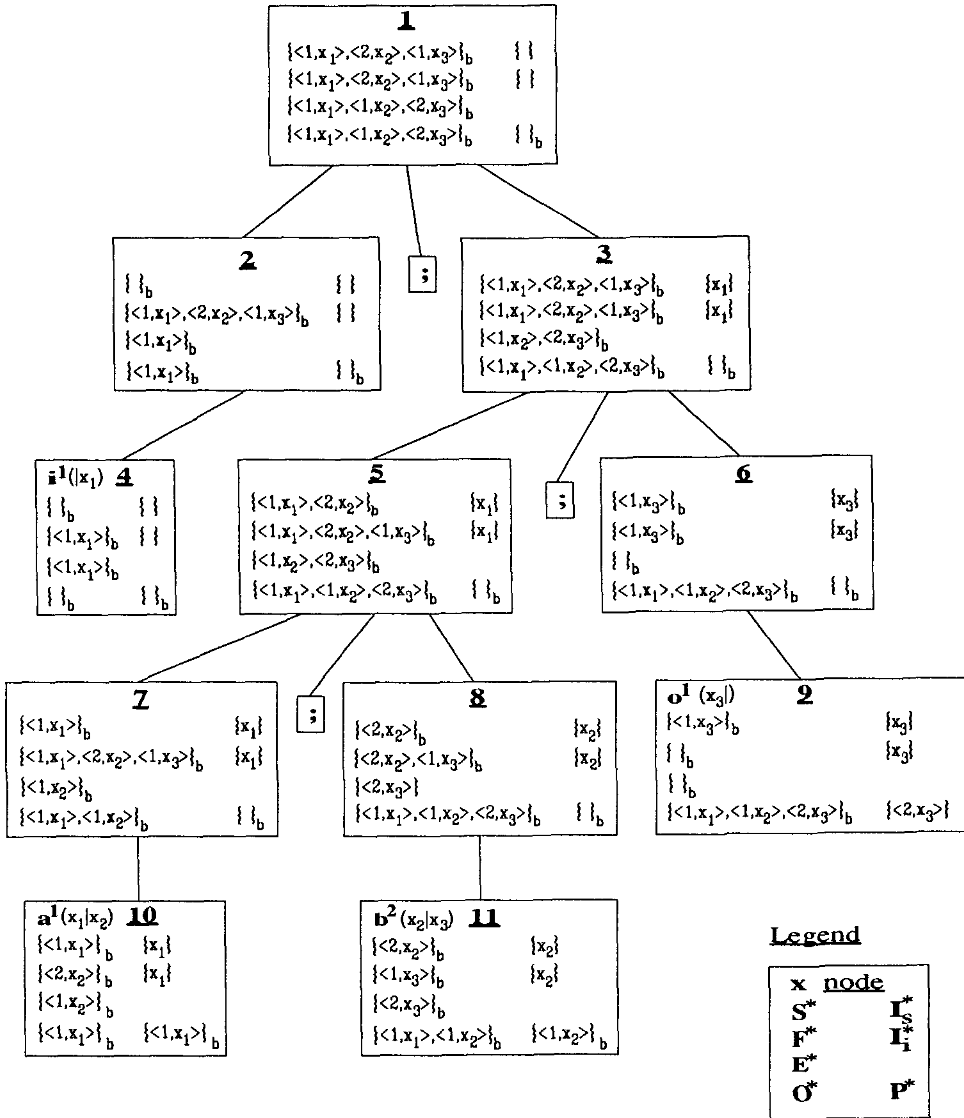

Fig. 10. Derivation tree with attributes for Example 2.

instance, unspecified reception errors are avoided if a particular reception scheme is chosen. For each sent message, the protocol specification contains a corresponding reception such that all messages in transit will eventually be consumed. Also, conformance with the service specification is achieved by construction, and the specifications of the protocol entities are free of nonexecutable interactions, deadlocks and state ambiguities.

We assume the availability of a reliable message transmission service between participating protocol entities. This assumption also appears,

explicitly or implicitly, in earlier, less general approaches to synthesizing protocols [8, 24] and is not unrealistic for high-level protocols. The service of the transport protocol, layer 4 of the OSI reference model, is designed to enable communication preserving the correct order and avoiding unnoticed loss of messages. For lower level two party protocols, it is possible to first assume correct delivery and later augment the derived protocol with means for error recovery incrementally [18].

Subsystem failures are not handled by the algorithm. Such failures may prevent successful completion of services in execution, if they go along with a loss of the entity's memory. Here, the introduction of a stable store, a store that can survive subsystem crashes, is a possibility to enable a restart and successful completion of the service [11]. However, a treatment of these aspects is beyond the scope of the paper.

The protocol derivation algorithm may be applied in different areas. Logical connections would be established between involved protocol entities before the derived protocol is executed. Within the OSI reference model, this situation can be satisfied for the application layer. It is therefore expected that the algorithm could be useful in areas such as distributed databases, process control, etc. The situation is also satisfied for message exchange via interprocess communication facilities. Here, entities would be associated with processes running within the same, possibly distributed, operating system.

Application of the derivation algorithm could also be attractive in systems where services are defined dynamically during the operation of the system. This is the case for distributed data base management systems, where the end users issue queries or update commands. Since the view of the end user would be centralized, his queries typically would not indicate at which places the required operations (selection, projection, join, division) have to be executed. This decision would have to be made on the basis of information available only to the database management system before the derivation algorithm can be applied.

Further research should include the following areas:

(a) It would be desirable to formally define the semantics of the language used to specify services and protocols and to give a formal proof that the presented algorithm always yields correct results.

(b) Extensions should concern the optimization of traffic necessary to synchronize operations and to pass parameter values. It is for instance not necessary to pass messages to synchronize subsequent operations at the same place. Also, synchronization messages and data messages may be combined.

(c) To define a service, we have used operators for sequence, parallelism and alternatives. Such operators can also be found in languages like LOTOS [10], CCS [16] or CSP [9]. The presented algorithm could therefore be applied to certain specifications in those languages.

(d) It could be useful to include recursion or iteration into our language for the specification of services. The impact of this extension on the derivation algorithm must be carefully examined.

(e) The derivation algorithm should be extended to cope with an unreliable medium. It could then be applied to lower level services.

From a systematic point of view, synthesis certainly has its advantages over analysis. However, it is not easy to find an algorithm for the construction of correct protocol specifications that is sufficiently general. We hope that the algorithm presented here will lead to the practical application of protocol synthesis methods.

ACKNOWLEDGMENTS

The idea of deriving protocol specifications from service specifications arose in discussions with H. Ichikawa and M. Itoh during a visit of G. v. Bochmann at the NTT Mosashino ECL in Tokyo. We thank H. Ichikawa and T. Murakami for their detailed study and helpful comments on an earlier version of the algorithm presented here. We also thank the referees for several constructive comments which helped us to improve the paper. Financial support from the Natural Science and Engineering Research Council of Canada is gratefully acknowledged.

REFERENCES

1. Bochmann, G. v. Semantic evaluation from left to right. *Commun. ACM 19*, 2 (Feb. 1976), 55-62.
2. Bochmann, G. v. Finite state description of communication protocols. *Comput. Networks 2* (1978), 361-371.
3. Bochmann, G. v., and Sunshine, C. A. Formal methods in communication protocol design. *IEEE Trans. Commun. COM-28*, 4 (Apr. 1980), 624-631.
4. Bochmann, G. v., and Gotzhein, R. Deriving protocol specifications from service specifications. In *Communications, Architectures & Protocols, Proceedings of the ACM SIGCOMM '86 Symposium*. ACM, New York, 1986, 148-156.
5. Bolognesi, T., and Smolka, S. A. Fundamental results for the verification of observational equivalence: A survey. In *Protocol Specification, Testing, and Verification, VII, Proceedings of the IFIP WG 6.1 Workshop* (Zurich, May 5-8, 1978). H. Rudin, C. H. West, Eds., North-Holland, 165-180.
6. Chung, R. A methodology for protocol design and specification based on an extended state transition model. In *Proceedings ACM SIGCOMM Symposium*, (Montreal, June 1984). *Comput. Commun. Rev. 14*, 2 (1984), 34-41.
7. Date, C. J. *An Introduction to Database Systems*. 3rd. ed., Addison-Wesley, Reading, Mass., 1981.
8. Gouda, M., and Yu, Y. Synthesis of communicating finite-state machines with guaranteed progress. *IEEE Trans. Commun. COM-32*, 7 (July 1984), 779-788.
9. Hoare, C. A. R. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, N.J., 1985.
10. *LOTOS—A Formal Description Technique*. DIS 8807, 1989.
11. *Information Processing Systems—Open Systems Interconnection—Service Definition for the Commitment, Concurrency and Recovery Service Element*. ISO/IEC JTC1/SC21, DIS 9804, 1988.
12. Khendek, F. Dérivation de spécifications de protocole à partir de spécifications de service, dans un sous-ensemble LOTOS. M.Sc. Thesis, Univ. of Montreal, 1989.
13. Laventhal, M. S. A constructive approach to reliable synchronization code. In *Proceedings of the 4th International Conference on Software Engineering* (1979), 194-202.

14. MACKERT, L. Modelling, specification and correct realization of asynchronous systems. Dissertation Arbeitsberichte des IMMD Bd.16, Nr.7. Univ. of Erlangen, 1983 (in German).
15. MERLIN, P., AND BOCHMANN, G. V. On the construction of submodule specifications and communication protocols. *ACM Trans. Program. Lang. Syst. 1* (Jan. 1983), 1-25.
16. MILNER, R. *A Calculus of Communicating Systems. Lecture Notes in Computer Science 92*, Springer-Verlag, Berlin, 1980.
17. RAMAMOORTHY, C. V., DONG, S. T., AND USUDA, Y. An implementation of an automated protocol synthesizer (APS) and its application to the X.21 protocol. *IEEE Trans. Softw. Eng. SE-11*, 9 (Sept. 1985), 886-908.
18. RAMAMOORTHY, C. V., YAW, Y., AGGARWAL, R., AND SONG, J. Synthesis of two-party error-recoverable protocols. In *Communications, Architectures & Protocols, Proceedings of the ACM SIGCOMM '86 Symposium* (1986). ACM, New York, 1986, 227-235.
19. RUBIN, J. AND WEST, C. H. An improved protocol validation technique. *Comput. Netw. 6* (1982), 65-73.
20. VISSERS, C. A., AND LOGRIPPO, L. The importance of the service concept in the design of data communications protocols. In *Protocol Specification, Testing, and Verification, V, Proceedings of the IFIP WG 6.1 Workshop* (Toulouse-Moissac, June 10-13, 1985). M. Diaz, Ed., North-Holland, Amsterdam, 1986, 3-17.
21. WEDEKIND, H. Data base systems I. Reihe Informatik Bd. 16, B.I. Wissenschaftsverlag, 1981 (in German).
22. WEST, C. H. An automated technique of communication protocol validation. *IEEE Trans. Commun. COM-26* (1978), 1271-1275.
23. WEST, C. H. Protocol validation by random state exploration. In *Protocol Specification, Testing, and Verification, VI, Proceedings of the IFIP WG 6.1 Workshop* (Montreal, June 10-13, 1986). B. Sarikaya and G. v. Bochmann, Eds., North-Holland, Amsterdam, 1986, 233-242.
24. ZAFIROPULO, P., WEST, C. H., RUDIN, H., COWAN, D. D., AND BRAND, D. Towards analyzing and synthesizing protocols. *IEEE Trans. Commun. COM-28*, 4 (Apr. 1980), 651-661.
25. ZHAO, J., AND BOCHMANN, G. V. Reduced reachability analysis of communication protocols: A new approach. In *Protocol Specification, Testing, and Verification, VI, Proceedings of the IFIP WG 6.1 Workshop* (Montreal, June 10-13, 1986). B. Sarikaya and G. v. Bochmann, Eds., North-Holland, Amsterdam, 1986, 243-254.

Received August 1986; revised April 1988 and February 1989; accepted September 1990