A Protocol Synthesis Algorithm: A Relational Approach

K. ZEROUAL M. El Yassini
Département de mathématiques et d'informatique
Université de Sherbrooke
Sherbrooke, Qc, Canada J1K 2R1

Abstract

The protocol engineering plays an important role in computer networks. Designing a protocol is a challenging activity because of the complexity of the rules defining the interaction between the communicating entities. At this point, formal approaches have been proposed: Analysis and Synthesis. The synthesis approach has the advantage of avoiding errors a priori and ensuring some desirable properties during the protocol design process. In this paper, we propose a relational protocol synthesis method. The method outcome is specifications of all protocol entities serving the different service access points, which are represented in R-net diagrams.

1 Introduction

The widespread use of computer-communication networks means that protocol engineering plays an important role in computer networks and comprises a crucial activity in their building process. Designing a protocol is a challenging activity because of the complexity of the rules defining the interaction between the communicating entities.

Several informal techniques for designing these protocols have been successfully applied. However, these techniques suffer from a number of errors or unexpected and undesirable behavior in the protocols that can be built with them. To cope with these problems, formal approaches for designing protocols have been proposed. Among these is the synthesis approach, a powerful tool for avoiding errors a priori and ensuring some desirable properties during the protocol design process.

Many papers have been published on protocol synthesis. They can be divided into two categories according to the type of specification they take as a starting point. In the first category, we find the approaches [6, 7, 1, 8, 4] that derive a complete protocol specification from a partly specified protocol. These approaches do not take into account the service specification, which is a formal requirement on which the synthesis method should be based. In the second category, we find the approaches [2, 3, 12, 5] that start from a formal service specification and derive a complete protocol. However, the approaches of both categories have some important limitations:

- None of the above approaches take into account the service maintenance process. To update a service, these approaches restart the synthesis process from scratch, whether this is desirable or even possible.
- With the exception of [2, 11], only two-party protocols are considered. Cases involving more than two communicating processes are not well-suited for these approaches.

In this paper, we present a relational method supported by a computerized tool for deriving protocol specifications from a given service specification. Our approach to the derivation of protocol specifications can handle an arbitrary number of protocol entities. The method outcome is specifications of all protocol entities serving the different service access points. These specifications are represented in *R-net* diagrams [9]. Furthermore, our approach allows protocol updating deriving and adding the protocol point that corresponds to the new service primitives needed for the global service specification. The method has been implemented in relational language into an ORACLE package environment.

The remainder of the paper is organized as follows. Section 2 presents a service specification language as well as a protocol specification language. In Section 3, we discuss the method for deriving the protocol specification from a given service specification, illustrated with an example. Finally, we provide some concluding remarks in Section 4.

2 Specification Languages

2.1 Service specification language

To facilitate easier specification of services, we adopted a simple service specification language defined by the following syntax rules:

1° *service-spec* ::= {< *priority-rel* >}
2° *priority-rel* ::=< *event* >< *order* >< *event* >
3° *event* ::=< *action* >^{id}
4° *action* ::=< *letter* > | < *letter* >< *letter* >
5° *letter* ::= $a|b|c|\ldots|z|\#|\$$
6° *id* ::=< *digit* > | < *digit* >< *digit* >
7° *digit* ::= $0|1|2|\ldots|9$
8° *order* ::=; | || |[]

In this language, an event (see Rule 3) stands for the service primitive called "action to be performed at a *SAP* whose identifier is *id*". We define a function

0-8186-7216-1/95 $04.00 © 1995 IEEE

PSap(event) that returns the action label and the *SAP* identifier with which the action label is associated. For example, the event a^1 represents the service primitive "*a*" at the *SAP* 1. A finite or infinite behavior communication service can comprise coordinated concurrent (noted ||) and alternative (noted []) execution of service primitives associated with different SAP_s in addition to the strictly sequential (noted ;) execution of service primitives. This way of formally specifying a service allows us to define only the behaviors of service primitive interactions with *SAP*. A communication service can be defined as finite or infinite behavior depending on whether the service primitives are recursing or not.
Given that a service specification *"service-spec"* is a set of priority relations, and a priority relation represents the ordering of two events, the infinite behavior may be possible by specifying sequential priority relations that are circularly linked. For example, $a^1; b^2$, $b^2; c^3$, and $c^3; a^1$ give rise to an infinite behavior.
In order to simplify the protocol synthesis, we impose, in a similar way as in [3], the following restriction on the form of service specification. For each priority relation where order is "[]", the two events must have the same *SAP* identifier. In other words, the two events must be associated with the same *SAP*. This restriction allows us to avoid synthesizing distributed protocols.

2.2 Protocol specifications language

The synthesized protocol specifications are expressed in *R-net* language. We adopted the *R-net* language because it is appropriate for the specification of distributed system behaviors. *R-nets* was developed as a part of the Software Requirements Engineering Methodology (SREM) [9, 10], in an attempt to formalize and automate techniques for specifying software requirements for a given system. It was designed to remedy to the difficulties of notations such as FSM in adequately showing the sequence of operations performed on a given input message.
R-net consists of three kinds of nodes: *ALPHA*, *subnet*, and *structured*. *ALPHA* nodes, represented by rectangles, specify processing operations, while *subnet nodes*, represented by ovals, are a refinement of *ALPHA* nodes. Structured nodes are *AND*, *OR*, *SELECT*, and *FOR-EACH* nodes. Except for the *FOR-EACH* node, which is a rectangle, all other structure nodes are represented by circles with the appropriate symbol inside. *AND* nodes, noted (&), specify operations that can be executed in parallel; *OR* nodes, noted (+), allow the execution of one of several operations; *SELECT* nodes specify the condition for selecting one node; and *FOR-EACH* nodes allow iteration of each element on a list to be executed. Arcs are used to connect nodes. Figure 17 shows a *R-net* diagram.
R-net can be coupled with a Requirements Statement Language (RSL) and a Requirements Engineering and Validation System (REVS), which is an automated tool system.
The protocol specifications are characterized by the set of send and receive messages that are integrated with the service primitives. These messages serve for synchronization in order to ensure the correct ordering of service primitive interactions.

2.3 Terminology and basic concepts

At this level, we introduce the basic relational operators that enable us to manipulate information in all parts of the service primitive specification in a very flexible and powerful way. Understanding these operators does not require familiarity with programming details. More details on relational algebra may be found in [13]. The relational operators transform either a simple relation or pair of relations into a result that is a relation.
In presenting the basic relational operators, we shall make use of the relations S and T of degree m and n, respectively, denoted: $S(X_1, X_2, ..., X_m)$ and $T(Y_1, Y_2, ..., Y_n)$

1. *Cartesian product(×)*: This operator has two operands, each a relation. Its purpose is to concatenate each and every tuple of its first operand with each and every tuple of its second operand.
   $S \times T(X_1, X_2, ..., X_m, Y_1, Y_2, ..., Y_n)$

2. *Project operator(Π)*: Employs a single relation as its operand. It generates an intermediate result in which the domain listed by name in the command is saved, and all the other domains are ignored.
   $\Pi_{X_i,...,X_j}(S) = \{(X_i, ..., X_j)/X_i, ..., X_j \in S\}$

3. *Selector operator(σ)*: Employs a single relation as its operand. It generates a relation that contains only tuples that satisfy the condition expressed in the command. A condition involves the comparison of two domain values.
   $\sigma_{X_i=X_j}(S) = \{(X_1, X_2, ..., X_m)/(X_1, X_2, ..., X_m) \in S \wedge (X_i = X_j)\}$

4. *Natural-join operator(⋈)*: Employs two relations as its operands. It generates a relation that contains tuples of one operand concatenated with tuples of the second operand, but only when common domains of the operands have the same value.
   $S \bowtie T = \{(X_1, X_2, ..., X_m, Y_k, ..., Y_l)/(X_i \in S) \wedge (X_i \in T) \Rightarrow (X_i(S) = X_i(T))\}$

5. *Equijoin operator([ ])*: Similar to the natural-join operator, except that there is a specified condition on domain to be compared.
   $S[X_3 = Y_4]T = \{(X_1, X_2, ..., X_m, Y_1, ..., Y_n)/ (X_3 \in S) \wedge (Y_4 \in T) \wedge (X_3 = Y_4)\}$

6. To rename (attribute) domains, we proceed as follows:
   $\delta_{X_i,...,X_j \longleftarrow X_{i'},...,X_{j'}}(S)$

7. *Difference operator(−)*: $S - T$ is the relation containing those tuples that are in S but not in T.

3 From Service Specification to Protocol Specification

We present, in this section, a relational synthesis method and an automated tool to derive the protocol specification in *R-net* from the given service specification. The basic idea of the protocol derivation algorithm is that synchronizing service primitives associated with different SAP_s is done by the exchange of synchronizing messages required for the sequencing operator ";". The subexpressions "e_i" and "e_j" of an order relation "$e_i; e_j$" involve service interaction expressed in "e_j" that should not start before service interaction expressed in "e_i" has been finished. This kind of synchronization is formalized by sending a synchronization message from the SAP where the interaction of "e_i" is performed to the SAP where the interaction of "e_j" is to be performed.
To determine which synchronization messages must be exchanged between SAP_s, we should determine, for the given SAP_s, the service primitives that must be synchronized as well as the *send* and *receive* messages according to which those service primitives should synchronized.
For each order relation of the sequence type "$e_i; e_j$" in the service specification, we add the appropriate *send* and *receive* messages of synchronization. This is done, in our approach, by deriving tables 9relations) from parallel service primitives ($Service.P$) , alternative service primitives ($Service.A$), and sequential service primitives ($Service.S$) . The required *send* or *receive* messages, noted as "$message^i_j$," where "$message$" is either "r" or "s", stands for either the synchronizing message sent by the service access point "i" to the service access point "j," or the message received by the service access point "i" from the service access point "j". The derivation process of the synchronizing protocol from a given global service specification is composed of six major steps, each of which is discussed in the following subsections.

1° Building the tables representing the service specification.

2° Representing the parallel and alternative event in each column of the corresponding tables.

3° Divide the sequential order relations into those associated with the same SAP and those associated with different SAP_s.

4° Adding *send* and *receive* messages to the appropriate sequential service primitives.

5° Adding *send* and *receive* messages to the appropriate parallel and alternative service primitives.

6° Building a *R-net* for each SAP protocol.

3.1 Building the tables of the service specification

In this step, we build three different tables: one for sequential order relations, one for parallel order relations, and one for alternative order relations.

Ev.s.s	Site.o.s	Ev.d.s	Site.d.s
a	1	e	3
a	1	c	3
a	1	d	3
b	2	e	3
b	2	c	3
b	2	d	3
c	3	e	3
c	3	c	3
c	3	d	3
e	3	a	1
e	3	b	2
e	3	c	3
e	3	g	4
c	3	a	1
c	3	b	2
d	3	a	1
d	3	b	2
d	3	c	3
g	4	f	2
g	4	l	4

Figure 1: Table of sequential events

Each of these tables (relations) can be formally expressed as:
$Service.S(Ev.s.s, Site.o.s, Ev.d.s, Site.d.s)$
which stands for the order relations of type sequence $Ev.s.s^{Site.o.s}; Ev.d.s^{Site.d.s}$
$Service.P(Ev.s.p, Site.o.p, Ev.d.p, Site.d.p)$
which stands for the order relations of type parallel $Ev.s.p^{Site.o.p} \parallel Ev.d.p^{Site.d.p}$
$Service.A(Ev.s.a, Site.o.a, Ev.d.a, Site.d.a)$
which stands for the order relations of type alternative $Ev.s.a^{Site.o.a} [] Ev.d.a^{Site.d.a}$
A domain whose label begins with Ev represents an event; a domain whose label begins with $Site$ represents the service access point where an event is executed.
The semantics of the relation $Service.S$ is that the service primitive $Ev.s.s$ at SAP $Site.o.s$ must be executed before service primitive $Ev.d.s$ at SAP $Site.d.s$. The semantics of the relation $Service.P$ is that the service primitive $Ev.s.p$ at SAP $Site.o.p$ is executed concurrently with the service primitive $Ev.d.p$ at SAP $Site.d.p$. The semantics of the relation $Service.A$ is that one of the service primitives $Ev.s.a$ at SAP $Site.o.a$ and $Ev.d.a$ at SAP $Site.d.a$ may be executed. Let consider an example adapted from [3] in which the service specification is:
PROC A = $(a^1 \parallel b^2 \parallel c^3)$; B END
PROC B = $(e^3 \parallel (c^3 [] d^3))$; (A || D) END
PROC D = g^4 ; $(l^4 \parallel f^2)$ END
We represent this service specification in the tables shown in Figures 1, 2, and 3.

3.2 Representation of parallel and alternative events in one column

To facilitate processing the relations $Service.P$, and $Service.A$, we represented all the service primitives that are parallel or alternative in each column of the relations $Service.P$ and $Service.A$, respectively. To complete these tables, we add tuples (a^i, b^j) such

Ev.s.p	Site.o.p	Ev.d.p	Site.d.p
a	1	b	2
a	1	c	3
b	2	c	3
e	3	c	3
e	3	d	3
l	4	f	2

Figure 2: Table of parallel events

Ev.s.a	Site.o.a	Ev.d.a	Site.d.a
c	3	d	3

Figure 3: Table of alternative events

that $(b^j, a^i) \in Service.P$ or $Service.A$. These new tuples are useful in the computation of synchronizing messages of events in the same SAP. The same principle is applied to the service primitives that are alternative; these tables are formally updated as follows:

$$Service.A = Service.A \cup \delta_{Ev.d.a,Site.d.a,Ev.s.a,Site.o.a \leftarrow Ev.s.a,Site.o.a,Ev.d.a,Site.d.a}((\Pi_{Ev.d.a,Site.d.a}(Service.A)) \bowtie Service.A)$$

$$Service.P = Service.P \cup \delta_{Ev.d.p,Site.d.p,Ev.s.p,Site.o.p \leftarrow Ev.s.p,Site.o.p,Ev.d.p,Site.d.p}((\Pi_{Ev.d.p,Site.d.p}(Service.P)) \bowtie Service.P)$$

The contents of theses tables are shown in Figure 4 and Figure 5.

3.3 Partition of sequential (events) order relations

Since two sequential service primitives can be associated either with the same SAP or with two distinct SAP_s, we consider two types of service primitive synchronization: synchronization of sequential service primitives in the same SAP and synchronization of sequential primitives in two different SAP_s. Sequential order relations involving two different SAP_s are represented in the table $Service.S.1$; those involving the same SAP are represented in the table $Service.S.2$. The table $Services.S.2$ (see Figure 7), obtained from the selection of tuples in the table $Service.S$ where $Site.o.s = Site.d.s$, represents service primitives that are sequential in the same SAP. The table $Service.S.1$ (see Figure 6), obtained from the selection of tuples in the table $Service.S$ where $Site.o.s \neq Site.d.s$, represents service primitives that are sequential but in different SAP_s. Formally, these tables are obtained as follows:

1. Table of sequential service primitives involving two different SAP_s:
   $Service.S.1 = \sigma_{Site.o.s \neq Site.d.s}(Service.S)$
2. Table of sequential service primitives involving the same SAP_s:
   $Service.S.2 = \sigma_{Site.o.s = Site.d.s}(Service.S)$

Ev.s.p	Site.o.p	Ev.d.p	Site.d.p
a	1	b	2
a	1	c	3
b	2	c	3
e	3	c	3
e	3	d	3
l	4	f	2
b	2	a	1
c	3	a	1
c	3	b	2
c	3	e	3
d	3	e	3
f	2	l	4

Figure 4: $Service.P$ completed

Ev.s.a	Site.o.a	Ev.d.a	Site.d.a
c	3	d	3
d	3	c	3

Figure 5: $Service.A$ completed

Ev.s.s	Site.o.s	Ev.d.s	Site.d.s
a	1	e	3
a	1	c	3
a	1	d	3
b	2	e	3
b	2	c	3
b	2	d	3
e	3	a	1
e	3	b	2
e	3	g	4
c	3	a	1
c	3	b	2
d	3	a	1
d	3	b	2
g	4	f	2

Figure 6: $Service.S.1$

Ev.s.s	Site.o.s	Ev.d.s	Site.d.s
c	3	e	3
c	3	c	3
c	3	d	3
e	3	c	3
d	3	c	3
g	4	l	4

Figure 7: $Service.S.2$

3.4 Adding synchronizing messages to sequential events

To synchronize sequential events in two different SAP_s, we add an event of sending a message to the SAP where an event execution is waiting for the end of execution of the preceding event after each preceding event in a sequential order relation (i.e., event specified in the column $Ev.s.s$ of the table $Service.S.1$). We also add an event of receiving a message from the SAP where the preceding event execution ends before each successor event in a sequential order relation (i.e., event specified in the column $Ev.d.s$ of the table $Service.S.1$).

The table *Send* (see Figure 8), representing sequential events with their corresponding sending messages, is computed from the Cartesian product of the table $Service.S.1$ and an event of sending a message "s," whose origin is the SAP specified in the column $Site.o.s$ and whose destination is the SAP specified

Ev.s.s	Site.o.s	message	Site.o.s/	Site.d.s
a	1	s	1	3
b	2	s	2	3
e	3	s	3	1
e	3	s	3	2
e	3	s	3	4
c	3	s	3	1
c	3	s	3	2
d	3	s	3	1
d	3	s	3	2
g	4	s	4	2

Figure 8: Table *Send*

message	Site.d.s/	Site.o.s	Ev.d.s	Site.d.s
r	4	3	g	4
r	3	1	e	3
r	3	1	c	3
r	3	1	d	3
r	3	2	e	3
r	3	2	c	3
r	3	2	d	3
r	1	3	a	1
r	2	3	b	2
r	2	4	f	2

Figure 9: Table *Receive*

in the column *Site.d.s*.
The table *Receive* (see Figure 9), representing sequential events with their corresponding receiving messages, is computed from the Cartesian product of the table *Service.S*.1 and an event of receiving a message "*r*" whose origin is the *SAP* specified in the column *Site.o.s* and whose destination is the *SAP* specified in the column *Site.d.s*.

3.4.1 Synchronizing events in different SAP_s

1. Computation of the table *Send* corresponding to *Service.S*.1 :
   $Send(Ev.s.s, Site.o.s, message, Site.o.s\prime, Site.d.s)$
   where $Ev.s.s^{Site.o.s}; message^{Site.o.s\prime}_{Site.d.s}$ and *message* stands for an event *send*.
   $Send = \Pi_{Ev.s.s,Site.o.s,message,Site.o.s\prime,Site.d.s}$
   $(((\Pi_{Ev.s.s,Site.o.s}(Service.S.1))$
   $\times \{s : message\})$
   $[Ev.s.s = Ev.s.s\prime, Site.o.s = Site.o.s\prime]$
   $(\delta_{Ev.s.s,Site.o.s \longleftarrow Ev.s.s\prime,Site.o.s\prime}(Service.S.1)))$
2. Computation of the table *Receive* corresponding to *Service.S*.1 :
   $Receive(message, Site.d.s\prime, Site.o.s, Ev.d.s, Site.d.s)$
   where $message^{Site.d.s\prime}_{Site.o.s}; Ev.d.s^{Site.d.s}$ and *message* stands for an event *receive*.
   $Receive = \{r : message\} \times$
   $((\delta_{Site.d.s \longleftarrow Site.d.s\prime}(\Pi_{Site.d.s}(Service.S.1)))$
   $[Site.d.s\prime = Site.d.s]$
   $(\Pi_{Site.o.s,Ev.d.s,Site.d.s}(Service.S.1)))$

3.4.2 Synchronizing events in the same *SAP*

We distinguish two cases in synchronizing events in the same *SAP*. In the first case, we determine among sequential events performed in the same *SAP*, those that are successors (*Ev.d.s*) in the table *Service.S*.1 and whose predecessors are executed concurrently with another event. The events thus determined, even if they are sequential to other events in the same *SAP*, must be synchronized to those events that are parallel to their corresponding predecessor's events. The synchronizing messages added in this case are of the type *receive* because the *SAP* associated with the event to be synchronized does not need to send a synchronizing message to itself but needs to receive synchronizing messages from SAP_s associated with an event that is parallel to the predecessor's events (*Ev.s.s*) of the event that must be synchronized.
This is done by building a table *Syn*(*Ev.s.s, Site.o.s, message, Site.o.p, Site.d.p, Ev.d.s, Site.d.s*) whose tuples stand for the following sequence: $Ev.s.s^{Site.o.s}; message^{Site.o.p}_{Site.d.p}; Ev.d.s^{Site.d.s}$.
The table *Syn* is computed from the tables *Service.S*.2 and *Service.P*. For each tuple $a^i; b^i$ in the table *Service.S*.2 and a^i, c^j in the table *Service.P* $(i \neq j)$, we create a new tuple $c^j; b^i$ in table *Inter*. Then, for each tuple $c^j; b^i$ in the table *Inter*, we add a *receive* message $message^{Site.o.p}_{Site.d.p}$ between c^j and b^i. The message $message^{Site.o.p}_{Site.d.p}$ has the same meaning as above.
Formally, the computation of the table *Syn*. is expressed as follows.

$Inter = (\delta_{Ev.s.s,Site.o.s \longleftarrow Ev.d.p,Site.d.p}(Service.S.2))$
$\bowtie$
$(Service.P$
$[Ev.s.p=Ev.s.s, Site.o.p=Site.o.s]$
$Service.S.2)$

$Inter = \sigma_{Site.d.p \neq Site.o.p}(Inter)$

We compute the contents of the table *Syn* from the contents of the table *Inter*:

$Syn = (((\Pi_{Ev.s.s,Site.o.s}(Inter))$
$\times$
$\{r : message\})$
$\bowtie$
$(\Pi_{Site.o.p,Ev.s.s,Site.o.s}(Inter)))$
$\bowtie$
$(\Pi_{Site.d.p,Ev.d.s,Site.d.s,Ev.s.s,Site.o.s}(Inter))$

The second case concerns sequential events in the same *SAP* whose predecessor events do not have parallel events. These sequential events are supposed to be directly sequenced in the same *SAP* and are processed in the substeps of the joining protocol parts.

3.5 Joining protocol parts

In building a protocol specification for a given *SAP*, it may happen that a protocol specification for a given *SAP* is comprised of two or more elementary event sequences that are not explicitly linked. To ensure that these elementary event sequences do not constitute in themselves independent protocols, we infer some global event sequences that can include these elementary event sequences.

Ev.s.s	Site.o.s	message	Site.o.p	Site.d.p	Ev.d.s	Site.d.s
c	3	r	3	1	e	3
c	3	r	3	2	e	3
c	3	r	3	1	c	3
c	3	r	3	2	c	3
c	3	r	3	1	d	3
c	3	r	3	2	d	3

Figure 10: Table *Syn*

Let $Service.S.I(Ev.s.s, Site.o.s, Ev.d.s, Site.d.s)$ denotes derived sequential events.
Add to $Service.S.I$ events sequences specified in $Sevice.S.2$:
$Service.S.I = \sigma_{Ev.s.s \neq Ev.d.s}(Service.S.2)$
$Service.S.1.2 = \delta_{Ev.d.s, Site.d.s \longleftarrow 1.Ev.s, 1.Site.o}(Service.S.1)$
$i = 1$
REPEAT
Determine sequential events that constitute a path:
$Service.S.1.2 = Service.S.1.2 \bowtie (\delta_{Ev.s.s, Site.o.s \longleftarrow i.Ev.s, i.Site.o}(Service.S.1))$
Add to $Service.S.I$ sequential events $(Ev.s.s^{Site.o.s}; Ev.d.s^{Site.d.s})$ such that $(Site.o.s = Site.d.s) \wedge (Ev.s.s \neq Ev.d.s)$ in $Service.S.1.2$:
$Service.S.I = (Service.S.I) \cup \Pi_{Ev.s.s, Site.o.s, Ev.d.s, Site.d.s}(\sigma_{(Site.o.s = Site.d.s) \wedge (Ev.s.s \neq Ev.d.s)}(Service.S.1.2))$
$Service.S.1.2 = \sigma_{(Site.o.s \neq Site.d.s)}(Service.S.1.2)$
$Service.S.1.2 = \delta_{Ev.d.s, Site.d.s \longleftarrow (i+1).Ev.s, (i+1).Site.o}(Service.S.1.2)$
Delete from $Service.S.1.2$ sequential events specified twice in a tuple t_i in $Service.S.1.2$:
$j = 1$
REPEAT
$Service.S.1.2 = \sigma_{(j.Site.o \neq (i+1).Site.o) \vee (j.Ev.s \neq (i+1).Ev.s)}(Service.S.1.2)$
$j = j + 1$
UNTIL $(j = i + 1)$
$i = i + 1$
UNTIL $(Service.S.1.2 = NULL)$
Delete from $Service.S.I$ events that are either parallel or alternative:
$Service.S.I = (Service.S.I$
$-\delta_{Ev.s.a, Site.o.a, Ev.d.a, Site.d.a \longleftarrow Ev.s.s, Site.o.s, Ev.d.s, Site.d.s}(Service.A))$
$-\delta_{Ev.s.p, Site.o.p, Ev.d.p, Site.d.p \longleftarrow Ev.s.s, Site.o.s, Ev.d.s, Site.d.s}(Service.P)$

Figure 13: Algorithm for computing the table *Service.S.I*.

To build a global events sequence, we perform a join operator on the relation *Service.S.*1 and itself such that the attributes *Ev.d.s*, *Site.d.s* and *Ev.s.s*, *Site.o.s* are respectively equal. We remove from the result of this "join-operator" all the tuples whose attributes *Ev.d.s*, *Site.d.s* and *Ev.s.s*, *Site.o.s* are respectively equal. Then, we gather in the table *Service.S.I* tuples whose attributes *Ev.s.s* and *Ev.d.s* are not equal, and *Site.o.s* and *Site.d.s* are equal. In other words, we keep in the table *Service.S.I* tuples representing the sequences of different events executed in the same *SAP*. We proceed in a similar fashion with the table thus obtained until the table obtained is empty. When the table *Service.S.*1.2 is empty, we verify whether the ending event of an elementary event sequence and the starting event of another elementary event sequence are both sequenced in a tuple of the table *Service.S.I*. If the verification is positive, we concatenate the two elementary event sequences. The ending and starting events do not take into account the synchronizing messages(i.e., events of the type *send* and *receive*).
To illustrate how to join two elementary event sequences, let us consider the two event sequences shown in Figure 12. Applying the algorithm shown in Figure 13 to the table *Service.S.*1 gives the table *Service.S.I*, shown in Figure 11.
Given that the ending event b^2 and the starting event f^2 are sequential events in the table *Service.S.I*, we concatenate the two elementary event sequences and we obtain the protocol specification shown in Figure12.

Ev.s.s	Site.o.s	Ev.d.s	Site.d.s
b	2	f	2
g	4	l	4

Figure 11: *Service.S.I*

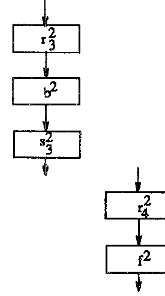

Figure 12: Protocol parts of SAP 2

3.6 Building *R-net* diagrams for protocol specifications

At this level, we build a set of *R-net* diagrams representing protocol specifications of the service access points (SAP_s) from the tables *Send*, *Receive*, and *Syn*.
Each tuple t_i in the tables *Send*, *Receive*, and *Syn* stands for a sequence of transition (arc) in the *R-net* diagram representing the protocol specification of the *SAP i*.

If $Ev.s.s^{site.s.s}$ is a label of a transition T_i in a *R-net* G_i,
Then If $message^{Site.o.s'}_{Site.d.s}$ is the label of a transition T_j that is an immediate successor of T_i;
Else Create a T_j with label $message^{Site.o.s'}_{Site.d.s}$ as an immediate successor of T_i;
Else Create a sequence of two transitions T_i and T_j with labels $Ev.s.s^{Site.o.s}$ and $message^{Site.o.s'}_{Site.d.s}$ respectively.
If there is a parallel or an alternative event T_l to transition T_i,
Then Link T_i to T_l either as a parallel or as an alternative transitions in G_i.

Figure 14: Algorithm for building transitions from the table *Send*.

If $Ev.s.s^{site.d.s}$ is a label of a transition T_j in a *R-net* G_i,
Then If $message^{Site.d.s'}_{Site.o.s}$ is the label of a transition T_i that is an immediate predecessor of T_j;
Else Create a T_i with label $message^{Site.d.s'}_{Site.o.s}$ as an immediate predecessor of T_j;
Else Create a sequence of two transitions T_i and T_j with labels $message^{Site.d.s'}_{Site.o.s}$ and $Ev.s.s^{Site.o.s}$ respectively.
If there is a parallel or an alternative event T_l to transition T_j,
Then Link T_j to T_l either as a parallel or as an alternative transitions in G_i.

Figure 15: Algorithm for building transitions from the table *Receive*.

If $Ev.s.s^{Site.d.s}$ is a label of a transition T_j in a *R-net* G_i,
Then If $message^{Site.o.p}_{Site.d.p}$ is the label of a transition T_i that is an immediate predecessor of T_j;
Then If $Ev.s.p^{Site.o.p}$ is the label of a transition T_{i-1} that is an immediate predecessor of T_i;
Else Create a T_{i-1} with label $Ev.s.p^{Site.o.p}$ as an immediate predecessor of T_i;
Else Create a sequence of two transitions T_{i-1} and T_i with labels $Ev.s.p^{Site.o.p}$ and $message^{Site.o.p}_{Site.d.p}$ respectively that precedes immediately T_j;
Else Create a sequence of three transitions T_{i-1}, T_i, and T_j with labels $Ev.s.p^{Site.o.p}$,$message^{Site.o.p}_{Site.d.p}$,and $Ev.d.s^{Site.d.s}$ respectively.
If there is a parallel or an alternative event T_l to transition T_j,
Then Link T_j to T_l either as a parallel or as an alternative transitions in G_i.

Figure 16: Algorithm for building transitions from the table *Syn*.

To build a *R-net* diagram (G_i), we determine the *SAP* identification by getting the value of the attribute *Site.o.s* in a tuple t_i. We create, whenever it does not exists in G_i, a transition T_i whose label is the same as that in an attribute in t_i such that there exists a sequence of transitions that corresponds to the sequences of events in tuple t_i. To integrate the transition sequences in G_i, we link to the transition T_i the transitions T_j that are either parallel or alternative.

Each table (i.e., *Send*, *Receive*, *Syn*) is processed by its appropriate algorithm (see Figures 14-16).

Integrating protocol parts

During the derivation of a protocol specification, we must deal with situations in which a protocol specification is obtained from the integration of several chunks of a protocol specification. To integrate parts in a protocol specification, we use the information in the table *Service.S.I* that allows us to determine junction points between protocol specification parts.
For each tuple in the table *Service.S.I* , we identify the *SAP* in which the integration process can be performed. Then we check whether the two sequential events in the same *SAP* are concatenated or not. If there are two or more chunks, each of which contains one event, we concatenate the chunks according to the order of the events in the tuple in the table *Service.S.I*. An illustration of this case is the derivation of protocol *SAP* 2 (Figure 17) from protocol specification chunks shown in Figure 12 and the table *Service.S.I* shown in Figure 11.

4 Conclusion

We propose in this paper an algorithm to derive protocol specification from a service specification based on a relational approach. The research reported herein allows more flexible specification of services and protocols. The representation of a protocol specification is represented in a diagrammatic form that is easy to understand and to validate.
In this approach, updating a service does not require restarting the synthesis process from scratch. We can derive only the protocol specification parts that are involved in the modification and integrate it into whole protocol specification.
The implementation of this synthesis algorithm is simple because it requires only some basic concepts in relational database and any relational software database. We implemented this algorithm as a prototype system on a Sun workstation in a relational language supported with an ORACLE database system.

References

[1] Zhang Y., Takahashi K., Shiratori N., Noguchi S., "An Interactive Protocol Synthesis Algorithm Using a Global State Transition Graph," *IEEE Trans. Soft. Eng.*, Vol. 14, No. 3, pp. 394-404, 1988.

[2] Philip Merlin & Gregor V. Bochmann, "On the construction of submodule specifications," *in ACM Trans. on Programming Languages and Systems*, January 1983, vol. 5, No.1, pp. 1-25.

[3] G.V. Bochmann, R. Gotzhein "Deriving protocol specifications from services specifications," *Proceedings of the ACM SIGCOMM Symposium*, pp. 148-156, 1986.

[4] Peeinder P. Sidhu & Juan Aristizabal, "Constructing Submodule Specifications and Network Protocols," *IEEE Transactions on Software Engineering*, vol. 14, n 11, November 1988.

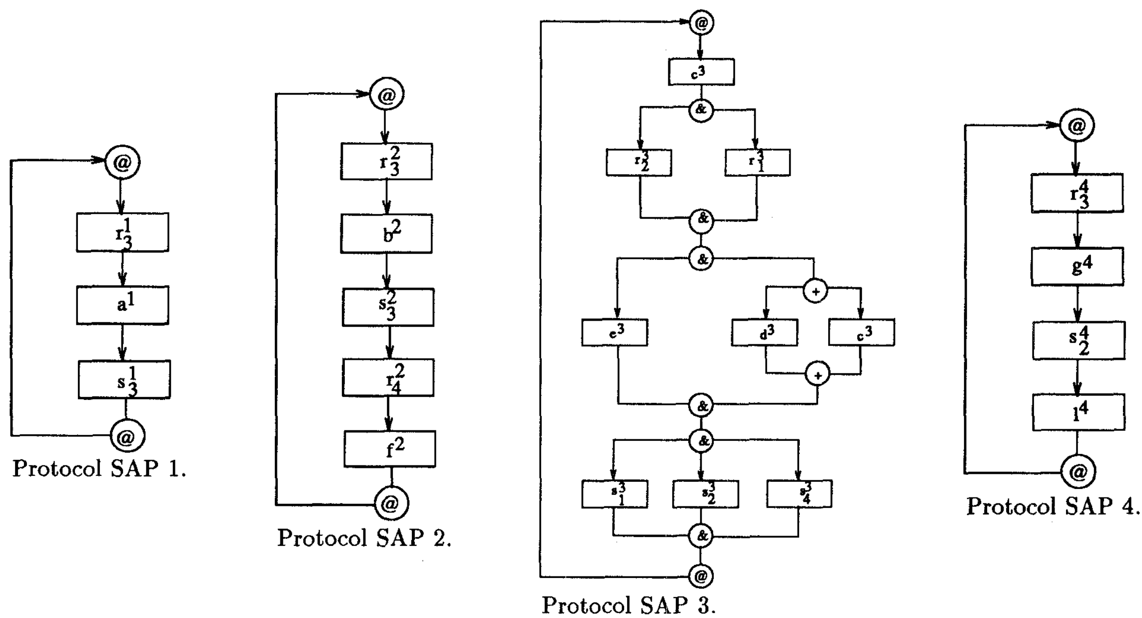

Figure 17: Derived protocol specifications

[5] K. Zéroual, T. Mesbah, A. Baziramwabo, "A Formal Specifications Synthesis," to appear in *IEEE Trans. Soft. Eng. Journal.*

[6] P. Zafiropolo, "Towards analyzing and synthesizing protocols," *IEEE Trans. Comm.*, Vol COM-28, no. 4, pp. 651-660, 1980.

[7] Mohamed G.Gouda & Yao-Tin Yu, "Protocol Validation by Maximal Progress State Exploration," *IEEE Trans. on Comm.*, Vol. COM-32, NO1, Januaray 1984.

[8] C.V. Ramamoorthy, S.T. Dong, Y. Usuda, "An implementation of an automated protocol synthesizer (APS) and its application to the X-21 protocol," *IEEE trans. Software Engineering*, Vol. SE-11, no. 9, pp. 886-908, 1985.

[9] M. Alford, "SREM at the Age Eight: The Distributed Computing Design System," *IEEE Computer*, vol. 18, no. 4, pp. 36-46, 1985.

[10] T. Bell, D. Bixler, M. Dyer, "An Extendable Approch to Computer-Aided Software Requirements Engineering," *IEEE Trans. Software Engineering*, vol. SE-3, no. 1,pp. 49-60, 1977.

[11] F. Khendek, G.V. Bochmann, C. Kant, "New results on deriving protocol specifications from service specifications," Tech. Report No. 692, Dep. IRO, Faculté des arts et des sciences, Université de Montréal, 1989.

[12] Peil-Ying M. Chu, Ming. T. Liu, "Protocol Synthesis in a State-Transition Model," Proceedings of the COMPSAC, pp. 505-512, 1988.

[13] Jeffrey D. Ullman, "Principles of Database Systems," *Computer Science Press*, 1982.