Protocol Synthesis Using Basic Lotos and Global Variables

A. Khoumsi* G.v Bochmann

Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, Québec H3C 3J7, Canada

Abstract

In [6], a method of protocol synthesis, using basic LOTOS (BL) as a specification language, is proposed. In the present paper, we generalize this method. For that, we propose an extended basic LOTOS (EBL) to specify the service and the protocol. With EBL, events are associated to enabling conditions and to transformation functions that depend on global variables. Next, we propose a method to synthesize protocols using EBL as a specification language. This method is inspired by the concept of Transactions *and by the method of [6]. Our method has an advantage : There are cases where it generates a solution while method of [6] is not applicable. This advantage is illustrated in two examples.*

Key Words: *Protocol synthesis, Shared variable, Enabling condition, Transformation function, EBL, Two-phase locking protocol, Partially and fully completed transaction, Disconnection phase.*

1 Introduction

An approach to design of a distributed system is to derive the protocol specification from a specification of the service desired by the user, in such a manner that the protocol is syntactically and semantically correct "by construction". The semantic correctness means that the protocol provides the desired service. The syntactic correctness means that the protocol is deadlock-free (resp. livelock-free) provided that the desired service is deadlock-free (resp. livelock-free), and no unspecified reception errors are possible. Such an approach to design is called *Synthesis*. In [2, 9, 7, 8, 6], different versions of a protocol synthesis method are proposed. Since [6] is the most general and its correctness has been proved, we consider only this reference which uses specifications written in basic LOTOS (BL) [3]. A limitation of the method in [6] is that three restrictions must be respected by a specification of a desired service written in BL, in order to ensure the generation of a correct protocol.

Our first contribution in this paper is the definition of a language which extends basic LOTOS by the use of a set of global variables. Such language is called Extended basic LOTOS and noted EBL. In a specification written in EBL, every event is associated to an enabling condition and a transformation function which possibly depend on a few global variables. An advantage of EBL is that it allows more ways than BL to specify a given system. A specification in BL is just a particular case where no variable is used. EBL being defined, we propose a synthesis method using specifications written in EBL. This method is mainly inspired by the concept of *Transactions using shared distributed variables* [1, 10] and by the protocol synthesis method of [6]. A first advantage of our method is due to the fact that there are more ways to specify a desired service by using EBL than by using BL. Therefore, using EBL increases the possibility to find a specification of a desired service which respects the three restrictions of [6] and then ensures the generation of a correct protocol. A second advantage is that one of these restrictions is not required if a restriction on variables is respected. All these points are discussed in this paper.

The remainder of the paper is organized as follows. In Section 2, we introduce the synthesis method of [6]. In Section 3, we study two simple examples which illustrate the limitations of this method. In Section 4, we present the model EBL. In Section 5, we present our method of protocol synthesis which uses EBL. In Section 6, the examples of Sect. 3 are studied to demonstrate the efficiency of our method. Finally in Section 7, we conclude and propose some future works.

2 Protocol synthesis with BL

We assume that the reader is familiar with basic LOTOS (BL) [3]. The operations of BL which are considered in this paper (and in [6]) are the following, where $B, B1, B2$ are behaviour expressions, a is an event and G is a set of gates (observable events) :

stop	Inaction	*exit*	Termination
$B1 [] B2$	Choice	$a ; B$	Action prefix
$B1 [> B2$	Disabling	$B1 \gg B2$	Sequence

$P[G]$	Process instantiation
$B1 \vert G \vert B2$	Parallel composition with rendezvous on gates of G

Notations 1 .
If G is empty, then operator $|G|$ is noted $|||$;
If G contains all the gates, then $|G|$ is noted $||$.
Every site of a distributed system is identified by a number i and is therefore noted $Site_i$.
An event may correspond to the occurrence of a :

*Supported by an FCAR-NSERC-BNR grant

0-8186-7216-1/95 $04.00 © 1995 IEEE

- Service primitive a in $Site_i$; it is noted a_i.
- Internal event $\hat{\imath}$ in $Site_i$; it is noted $\hat{\imath}_i$.
- Sending of m by $Site_i$ for $Site_j$; it is noted $s_i^j(m)$.
- Reception in $Site_j$ of m from $Site_i$; it is noted $r_j^i(m)$.

In a_i, $s_i^j(m)$ and $r_j^i(m)$, the lower index specifies the site where the event occurs, and m is a message.
Let A be a specification in BL :
- $AP(A)$ is the set of sites involved in A,
- $SP(A)$ (resp. $EP(A)$) is the set of sites where the first (resp. last) event of A may occur.

For example, for the following specification
$A = (a_1 ; b_2 ; exit) ||| (c_3 ; d_4 ; e_1 ; exit)$, we have :
$SP(A) = \{Site_1, Site_3\}$, $EP(A) = \{Site_1, Site_2\}$,
and $AP(A) = \{Site_1, Site_2, Site_3, Site_4\}$.
Boolean operators AND, OR, NEGATION are respectively represented by $\wedge$, $\vee$, $\neg$.

The basic idea of the protocol synthesis in [6] consists in projecting the service specification into each site. The projections are augmented by adding all the messages that must be exchanged between the sites such that the temporal order of events in the different sites ensures the order of service primitives implied by the service specification. The rules of the protocol synthesis are presented in [6]. In the present Section, we only give three examples which illustrate their use. We note that operator " $|G|$" does not generate messages. In the following examples, two sites $Site_1$ and $Site_2$ are involved, $Serv$ is a specification of a desired service, $Prot_i$ is the the specification of the protocol in $Site_i$ generated by the method of [6], for $i = 1, 2$.

Example 2.1 Protocol synthesis for a service specification containing operator $\gg$:

$$Serv = (a_1 ; exit) \gg (b_2 ; exit)$$
$$Prot_1 = a_1 ; s_1^2(m) ; exit$$
$$Prot_2 = r_2^1(m) ; exit \gg b_2 ; exit$$

Intuitively, operator $\gg$ in $Serv$ requires that b_2 must be executed after a_1. Therefore after the occurrence of a_1, $Site_1$ must send a message to $Site_2$.

Example 2.2 Protocol synthesis for a service specification containing operator $[]$:

$$Serv = (a_1 ; b_2 ; exit) [] (c_1 ; exit)$$
$$Prot_1 = (a_1 ; s_1^2(m1) ; exit) [] (c_1 ; exit \gg s_1^2(m2) ; exit)$$
$$Prot_2 = (r_2^1(m1) ; exit \gg b_2 ; exit) [] (r_2^1(m2) ; exit)$$

Intuitively, when an alternative is selected then a message is sent to all the sites which do not participate in the alternative. Therefore, if the second alternative is selected then $Site_1$ sends $m2$ to $Site_2$. Message $m1$ is used to execute b_2 after a_1.
In order to ensure a proper generation of protocol entities, two restrictions must be respected for the choice expressions of the form $A [] B$ [6] :
R1 : $SP(A) = SP(B)$ and are singletons.
R2 : $EP(A) = EP(B)$ (Not respected here).

Example 2.3 Protocol synthesis for a service specification containing operator $[>$:

$$Serv = (a_1 ; b_2 ; exit) [> (d_2 ; exit)$$
$$Prot_1 = (a_1 ; s_1^2(m1) ; exit \gg r_1^2(m2) ; exit) [> (r_1^2(m3) ; exit)$$
$$Prot_2 = (r_2^1(m1) ; exit \gg b_2 ; exit \gg s_2^1(m2) ; exit) [> (d_2 ; s_2^1(m3) ; exit)$$

Intuitively in $A [> B$, if a first event of B occurs then a message ($m3$ in the example) is sent to all places involved in process A. Besides, all places involved in process A must be informed if A terminates without being interrupted (message $m2$). Message $m1$ is used for ensuring that a_1 is executed before b_2.
As for the choice expressions (Example 2.2), two restrictions must be respected for the disabling expressions of the form $A [> B$ [6] :
R2 : $EP(A) = EP(B)$ and R3 : $EP(A) \supseteq SP(B)$.

3 Two practical examples

Here are two examples where the method of [6] is unable to generate a correct solution.

3.1 First Example

A connection oriented service provided by the transport layer is studied in [6]. It is a simplified version of the OSI Transport Service [5], it involves two users in $Site_1$ and $Site_2$, and contains the three following consecutive phases : the connection establishment, the data transfer, and the disconnection. We consider here the disconnection phase (DP), which may be initiated while data are exchanged between the two sites. During DP, two service primitives are used :
- $T_DISCONNECT_request$, noted $dsreq_i$;
- $T_DISCONNECT_indication$, noted $dsind_i$;

Index i identifies the site where a primitive occurs.
Informally, DP may be specified as follows :

- The user of $Site_i$ may initiate a disconnection by executing a $dsreq_i$ (for $i = 1, 2$).
- If user of $Site_i$ executes a $dsreq_i$, then the user of $Site_j$ receives a $dsind_j$ (for $i, j = 1, 2$ and $i \neq j$).
- $Site_i$ disconnects itself if its user executes a $dsreq_i$ or receives a $dsind_i$ (for $i = 1, 2$).

In [6], the above requirements are formalized in BL by the following specification, which is equivalent to the finite state automaton of Figure 1 :

$$\text{SPEC} \quad DISC = A(1,2) [] A(2,1) \quad \text{WHERE}$$
$$A(i,j) = dsreq_i ; ((dsind_j ; exit) [] (dsreq_j ; C))$$
$$C = ((dsind_1 ; exit) ||| (dsind_2 ; exit))$$

Restriction R1 (Example 2.2) is not respected in $DISC$, and authors of [6] assert that this example is an exception where R1 is not mandatory for applying their protocol synthesis method. But the following protocol specifications $DISC_i$ (in $Site_i$), for $i = 1, 2$, generated by the method of [6] from the service $DISC$, contain a deadlock. In fact, for $i, j = 1, 2$ and $i \neq j$:

$$\text{SPEC} \quad DISC_i = A_i [] B_i \quad \text{WHERE}$$
$$A_i = (dsreq_i ; s_i^j(mi) ; exit) \gg ((r_i^j(pi) ; exit) [] ((r_i^j(qi) ; exit) \gg C_i))$$
$$B_i = r_i^j(mj) ; exit \gg ((dsind_i ; exit \gg s_i^j(pj) ; exit) [] (dsreq_i ; (s_i^j(qj) ; exit) \gg C_i))$$
$$C_i = dsind_i ; exit$$

The deadlock occurs when users of both sites initiate simultaneously a disconnection :
- In $Site_1$: Process A_1 executes $dsreq_1$, sends message $m1$, and then waits for the reception of $p1$ or $q1$.
- In $Site_2$: Process A_2 executes $dsreq_2$, sends message $m2$, and then waits for the reception of $p2$ or $q2$.

In this case, every site is waiting for a message which will never be sent. This deadlock is due to the fact that restriction R1 is not respected, i.e., a choice is not centralized in one site. To avoid this deadlock, processes A_1 and A_2 must be executed in a mutual exclusion. In Sect. 6.1, we show how a correct solution is generated.

3.2 Second Example

Here is a simple example of a non terminating service, noted S, where the method in [6] is unable to give a solution, because restrictions R1 and R2 are not respected (Example 2.2). Informally, S is specified as follows, for $i, j = 1, 2$, and $i \neq j$:

- From the initial state, a_i may be executed.
- $Site_i$ may decide to execute b_i instead of a_i. But b_i cannot be executed before a_j.
- $Site_3$ may execute c_3 if both a_1 and a_2 have been executed.
- After the execution of either b_1 or b_2 or c_3, the initial state is reached.

The above requirements are formalized by the finite state automaton of Figure 2, and also in basic LOTOS by the following recursive specification :

$$S = ((((a_1 \,;\, exit) \,|||\, (a_2 \,;\, exit)) \gg (c_3 \,;\, exit)) \\ [\,]\, (a_1 \,;\, b_2 \,;\, exit) \,[\,]\, (a_2 \,;\, b_1 \,;\, exit)) \gg S$$

In Sect. 6.2, we show how a solution is obtained.

4 Extended basic LOTOS (EBL)

In the present Section, we propose a model which extends BL, and is therefore noted EBL (extended BL). A specification written in EBL uses a set of variables which may be shared by different sites, and to every event are associated an enabling condition (EC) and a transformation function (TF) which may depend on a few variables. Informally, an event may occur only if its enabling condition has the value $True$, and when it occurs then a few variables are possibly modified by applying the transformation function.

Let then $v^1, v^2, \ldots, v^n$ be n variables respectively defined in the sets $\mathcal{V}^1, \mathcal{V}^2, \ldots, \mathcal{V}^n$.

The $n-$tuplet $(v^1, v^2, \ldots, v^n)$ is noted v and its value is called *variable state*.

The set of variables is noted V, and the set of possible variable states is noted $\mathcal{V}$.

An *enabling condition* θ, w.r.t. V, is any boolean function : $\mathcal{V} \mapsto \{True, False\}$. It is syntactically represented by a boolean expression formed from :

- Canonical expressions $v^i \sim k$, where $k \in \mathcal{V}^i$ and $\sim \in \{=, \leq, <, \geq, >\}$, and
- Boolean operators AND($\wedge$), OR($\vee$) and NOT($\neg$) on canonical expressions.

An enabling condition which is *always* equal to $True$ (resp. $False$) is noted $True$ (resp. $False$).

The set of enabling conditions, w.r.t. V, is noted EC_V.

A *transformation function*, w.r.t. V, is any function $\mathcal{V} \mapsto \mathcal{V}$ syntactically represented by a series of canonical expressions $v^i \leftarrow \phi^i(v)$ separated by the sign $\wr$, where ϕ^i is a function $\mathcal{V} \mapsto \mathcal{V}^i$. Semantically, every expression $v^i \leftarrow \phi^i(v)$ means that the transformation function sets variable v^i to the new value $\phi^i(v)$ which depends on the current variable state v. For the sake of simplicity, $\phi^i(v)$ is noted ϕ^i.

A transformation function which *never* changes any variable is noted Id (for Identity).

The set of expressions defining the transformation functions, w.r.t. V, is noted TF_V.

Henceforth, all definitions, notations and computations are w.r.t. V, EC_V and TF_V. Terms "enabling condition" and "transformation function" are respectively abbreviated by EC and TF.

Example 4.1 An enabling condition and a transformation function.
We consider three integer variables v^i, for $i = 1, 2, 3$.
The EC $(v^1 \leq 3) \vee (v^3 > 0)$ is equal to $True$ if and only if $v^1 \leq 3$ or $v^3 > 0$.
The TF defined by $(v^1 \leftarrow v^1 + 2) \wr (v^2 \leftarrow 0)$ adds 2 to the value of v^1 and sets the value of v^2 to 0. The value of v^3 is not changed.

A specification written in EBL is obtained from a specification in BL if we replace every event σ by a transaction, defined as follows. A *Transaction* is obtained if we associate every event σ with an enabling condition θ and a transformation function ϕ. Such a transaction is noted $\langle \sigma, \theta, \phi \rangle$ and is called :

- *eligible* if event σ is currently possible according to the BL behaviour expression being executed;
- *enabled* if it is eligible and $\theta = True$; and *disabled* if it is not enabled.

The semantics of a transaction $\langle \sigma, \theta, \phi \rangle$ is :
- The transaction may be executed only if it is enabled;
- The execution of the transaction consists of the occurrence of σ followed by the execution of the TF ϕ.
The execution of the transaction is considered completed only after the execution of ϕ.

Example 4.2 Let the specification written in EBL
$\langle a_1, True, v^1 \leftarrow 0 \rangle \,;\, (\langle b_2, True, v^1 \leftarrow 1 \rangle \,|||\, \langle c_2, v^1 = 1, Id \rangle)$.
Transaction $\langle a_1, True, v^1 \leftarrow 0 \rangle$ is initially enabled. Its execution consists of the occurrence of a_1 and the setting of v^1 to zero. After its execution, transactions $\langle b_2, True, v^1 \leftarrow 1 \rangle$ and $\langle c_2, v^1 = 1, Id \rangle)$ become eligible, but only the first one is enabled. $\langle c_2, v^1 = 1, Id \rangle)$ becomes enabled after the execution of $\langle b_2, True, v^1 \leftarrow 1 \rangle$.
Informally, the variables contain a history of the system which is necessary for taking some decisions about the future occurrences of some events.

Notations 2 *(AT(P), ST(P), ET(P))*
Let P be a specification in EBL :
$AT(P)$ is the set of transactions involved in P.
$ST(P)$ is the set of *starting* transactions of P, i.e., the first transactions to be eligible.
$ET(P)$ is the set of *ending* transactions of P. In other words, $Tr \in ET(P)$ means that Tr may be the last transaction of P to be executed, according to the operators of BL. We note that $ST(P)$, $ET(P)$ and $AT(P)$ do not depend on any variable.

Example 4.3 *(AT(P), ST(P), ET(P))*
We consider the following specification $P = (Tr1 \,;\, Tr2 \,;\, exit) \,|||\, (Tr3 \,;\, Tr4 \,;\, Tr5 \,;\, exit)$, where every Tri represents a transaction. Clearly :
$AT(P) = \{Tr1, Tr2, Tr3, Tr4, Tr5\}$,
$ST(P) = \{Tr1, Tr3\}$, and $ET(P) = \{Tr2, Tr5\}$.

5 Protocol synthesis with EBL

5.1 Basic idea

Let V be a set of variables, and $Serv_V$ be a specification in EBL (w.r.t. V) of a desired service. Our method of protocol synthesis consists of :

Step 1. Every transaction of $Serv_V$ is designed in such a way that its execution is conceptually equivalent to an instantaneous execution, and that the consistency of the variables is ensured. This justifies the term "transaction". This problem has been addressed by several researchers, and [10] contains a survey of the proposed solutions. Our approach is inspired from one of these solutions [4, 12, 11].

Step 2. We do as if the execution of every transaction consists only of the occurrence of its corresponding event. This approach is correct, because in Step 1 every transaction of $Serv_V$ is designed in such a way that its execution is equivalent to an instantaneous execution. The method of [6] is therefore used to synthesize a specification PS_i in each $Site_i$. This is achieved by projecting $Serv_V$ in each $Site_i$ and by adding all the necessary messages that must be exchanged between the sites to ensure the order of the transactions implied by the operators of basic LOTOS in the specification $Serv_V$.

An advantage of our approach is that two different problems are separated in an elegant way. The first problem is about ensuring the atomicity of the transactions and the consistency of the variables [10], while the second one is about ensuring the order of the transactions implied by the operators of BL in $Serv_V$ [6].

5.2 Step 1

Let us study how the first Step is realized. The aim is to design how the transactions contained in the specification $Serv_V$ of the desired service are executed. Let us imagine that the system specified by $Serv_V$ is evolving from its initial state, by executing its eligible transactions. To execute a currently eligible transaction $Tr = \langle\sigma, \theta, \phi\rangle$, the system must execute the following procedure, noted $\mathcal{P}(Tr)$, in an atomic way :

begin
- Read the variables on which the EC θ depends, and then compute θ,
- If $\theta = False$ then the transaction is not executed,
- Else the execution of the transaction is allowed,
- Its execution consists of the occurrence of σ followed by the execution of ϕ which sets certain variables.

end

Tr is considered completed only after the execution of ϕ. We note that $\mathcal{P}(Tr)$ may be executed several times without executing transaction Tr. This may happen if Tr remains eligible but disabled for a "long while" : a periodic execution of $\mathcal{P}(Tr)$ is then necessary to check if Tr has become enabled. We also note that $(\theta = True)$ is *necessary* but may be *not sufficient* for the execution of Tr. This is the case if two transactions with the same event σ are synchronized (by operator $|[\sigma]|$) and only one transaction is enabled.

To ensure the *atomicity* of $\mathcal{P}(Tr)$ and the *consistency* of the variables, the following conditions must be respected during the execution of $\mathcal{P}(Tr)$:

C1. Variables on which depends θ are not written by another transaction;

C2. Variables written by ϕ are neither read nor written by another transaction.

In order to respect C1 and C2, we propose the *two-phase locking* protocol the correctness of which has been proved [4, 12]. It requires that every transaction :

(1) lock the variables it reads or writes before it actually accesses them, and

(2) not obtain a new lock after it has released a lock.

Two operations are therefore defined :

$lock(X, m)$: where $m = write$ or $m = read$ and X is the variable to be locked. A transaction executes $lock(X, read)$ (resp. $lock(X, write)$) to require a lock on X, before it reads (resp. writes) X. If an operation $lock(X, m)$ succeeds, X is said locked or more precisely m_locked.

$release(X)$: A transaction executes $release(X)$ to remove the lock it has previously executed on X.

The following rules are respected :

Rule 1. A variable which is not locked can be locked,

Rule 2. A variable can be read_locked several times if it is not write_locked,

Rule 3. A write_locked variable cannot be locked.

With this approach, for any transaction $Tr = \langle\sigma, \theta, \phi\rangle$, the corresponding procedure $\mathcal{P}(Tr)$ begins to execute a $lock(X, read)$ for every X to be *only* read, and a $lock(X, write)$ for every X to be written (possibly after being read). Therefore, variables which are to be written by the TF ϕ are write_locked, and variables on which θ depends and which are not to be written by ϕ are read_locked. If all the locks succeed, then $\mathcal{P}(Tr)$ may execute its main body which has been already presented (read variables, computes θ, ..., write variables). At the end, all variables which have been successfully locked are released.

There are several approaches to implement a variable X used by different sites. We propose the one that consists in using a copy of X in each site which uses this variable. Executing a $lock(X, write)$ consists in sending a write_lock request to all copies of X, and waiting a reply from all these copies. A copy replies to a write_lock request if it is not already locked. Executing a $lock(X, read)$ consists in sending a read_lock only to the local copy, and waiting a reply from it. A copy replies to a read_lock request if it is not already write_locked. A site writes X by sending the new value of X to all sites containing a copy of X. And each copy is automatically released as soon as it is updated. Since all the copies are identical, then reading a variable necessitates to read only its local copy.

The two-phase locking ensures that if a transaction is completed then its execution is correct, but it does not ensure that a transaction will be completed. In fact, a deadlock may occur if two transactions attempt simultaneously to lock the same variable. A solution to this problem consists in using a timestamp system [11] : all lock requests are timestamped and the total order on timestamps defines the priorities. The transaction having the less priority will abort.

5.3 Step 2

The basic idea of the second Step is introduced in Sect. 2. For a specification $Serv_V$ written in EBL, the method ignores the semantics of all ECs and TFs, but it respects the syntax of EBL. In other words, an expression $\langle \sigma_i, \theta, \phi \rangle$ in $Serv_V$ is processed as if it were simply the name of an event executed in $Site_i$. Therefore, such expression will be contained in the synthesized specification PS_i. Besides, since the syntax of EBL must be respected, every $s_i^j(m)$ (resp. $r_i^j(m)$), generated by rules of [6] (Not. 1) is replaced in our case by $\langle s_i^j(m), True, Id \rangle$ (resp. $\langle r_i^j(m), True, Id \rangle$).
Henceforth, messages generated in the first Step and in the second Step are respectively called transaction messages and synchronization messages.

Theorem 1 The protocol synthesized is semantically and syntactically correct (Sect. 1) if the desired service is syntactically correct.

Proof (informal) : We consider the two steps.
Step 1. Every transaction is designed in such a way that its execution is conceptually equivalent to an instantaneous event. This step is inspired from methods of [12, 11] whose correctness has been proven.
Step 2. Since every transaction is conceptually equivalent to an instantaneous event, the method of [6] may be used. Its correctness is proven in [6].

5.4 Particular cases

Here are two particular cases which help to simplify the design of the transactions. The simplification consists in generating only the useful messages.

5.4.1 First particular case

For a variable X used in a service specification $Serv_V$, let the following condition :

$\mathcal{C}1$: X is updated by only one transaction Tr, and Tr is executed only once in $Serv_V$.

If $\mathcal{C}1$ is respected, let $\mathcal{T}r$ be the set of transactions of $Serv_V$ different than Tr and whose enabling conditions depend on X, and let the following condition :

$\mathcal{C}2$: The transactions of $\mathcal{T}r$ are disabled while X has not been updated.

If $\mathcal{C}1$ and $\mathcal{C}2$ are respected, the two-phase locking protocol may be replaced by a less costly protocol as follows :
- X is initially write_locked,
- Tr may write X without locking it, because X is already locked.
- X is automatically released after its update by Tr.
- After it has been released, X may be read by any transaction of $\mathcal{T}r$ without being locked, because it will no more be written.

This first particular case is considered in Sect. 6.1.

5.4.2 Second particular case

Firstly, we remind that a transaction Tr is assumed completed as soon as its TF is executed, i.e., as soon as the new value of every variable to be updated by Tr is *sent* to the sites containing a copy of the variable. Therefore, the completion of Tr only means that the update of all variables to be updated has been initiated, but not necessarily terminated. Henceforth, Tr is said *fully* completed when the update of the variables is terminated. A transaction being executed is said *partially* completed while it is not fully completed.

In the present Section, we consider a desired service specified by a finite sequence of processes separated by operator $\gg$, i.e., $Serv_V = A1 \gg A2 \gg \cdots \gg An$. To simplify, each of the *successive* processes in this sequence is called "process of $Serv_V$". Our aim is therefore to propose an approach of protocol synthesis which ensures the following condition :

Cond1. Let two any consecutive processes P and Q of $Serv_V$, i.e., $Serv_V = \cdots P \gg Q \cdots$. The synthesized protocol must ensure that, as soon as any transaction of $ST(Q)$ is eligible then no transaction of $AT(P)$ is partially completed.

The respect of Cond1 allows to design the transactions of each process of $Serv_V$ as if the process were executed alone. Respecting Cond1 is therefore interesting because the two-phase locking protocol may be replaced by a simpler protocol. If, for instance, a variable X of P does not need to be locked when P is executed alone, then X will not need to be locked during the execution of the whole system, provided that Cond1 is repected.

For an expression $P \gg Q$ in $Serv_V$, the method of [6] (Step 2) ensures that the termination of P and the beginning of Q are separated by an exchange of synchronization messages from all sites of $EP(P)$ to all sites of $SP(Q)$. A sufficient condition which ensures Cond1 is therefore that no transaction of P is partially completed after the *sending* of the synchronization messages. The solution consists simply in forcing all sites of $EP(P)$ to wait a delay "sufficiently long" before sending their synchronization messages to the sites of $SP(Q)$. This delay is estimated as follows.

We assume that the transit delay of a message between two sites is bounded by a finite value $Trans_delay$. This delay comprises the transmission delay in the network, and also the time passed in the two sites (especially waiting in queues). We also assume that the delay during which a memory storing a copy of a variable is accessed, in order to update this copy, is bounded by a finite value Upd_delay.

During the execution of a transaction, the update of a distant copy of a variable necessitates to send the new value to the corresponding site, and then to access the storage of this copy, in order to update it. Therefore, the delay to update a copy of a variable is bounded by $Trans_delay + Upd_delay$. To set a variable X to a new value necessitates to update its different copies. If we assume that all the copies are updated in parallel, then X is updated in a delay bounded by $Total_delay = Trans_delay + Upd_delay$. Therefore, Cond1 is respected if, after the termination of P, all sites of $EP(P)$ wait a delay greater than or equal to $Total_delay$ before sending their synchronization messages to the sites of $SP(Q)$.

This case is considered in Sect. 6.2. It is interesting because it helps to simplify the design of the transactions by removing, as much as possible, the useless lockings.

5.5 How to specify the desired service

The variables used in a given specification in EBL of a desired service are just a means for expressing the desired constraints. The user does not require that these variables must be used in the designed protocol. For instance, the specification $\langle a_1, True, v^1 \leftarrow 0\rangle$; $(\langle b_2, True, v^1 \leftarrow 1\rangle \,|||\, \langle c_2, v^1 = 1, Id\rangle)$ can be replaced by a_1 ; b_2 ; c_2. In fact, variable v^1 is just a means to specify that c_2 may occur only after b_2. Since there are many ways to specify a same service in EBL, two important questions arise.

First question : What are the restrictions on a specification in EBL of a desired service which ensure a proper generation of protocol entities ?

Second question : When is it possible to specify a desired service in EBL in such a way that all messages are generated in the first Step ? The aim of this question is to know under which conditions our method of synthesis becomes independent on the method of [6].

5.5.1 Answering question 1

Since every transaction is conceptually equivalent to an instantaneous execution, then respecting R1 and R2 for operator $[]$, and R2 and R3 for operator $[>$, ensures the generation of a correct protocol. In this case, no condition on the variables is necessary. Let us study in which cases, restrictions R1, R2 or R3 are not required.

Restriction R1 in an expression $A \,[]\, B$ allows to centralize the choice, in order to "disable" instantly the not chosen alternative. This aim is also achieved if the variables ensure the mutual exclusion between A and B, i.e., if the following restriction $\mathcal{R}1$ is respected :

$\mathcal{R}1$. In an expression $A \,[]\, B$, transactions of $ST(A)$ remain disabled during all the execution of B. And vice versa. (See Not. 2 for $ST(A)$).

Therefore R1 may be "avoided", because any expression $A \,[]\, B$ which respects neither R1 nor $\mathcal{R}1$ can be replaced by an expression $A' \,[]\, B'$ which respects $\mathcal{R}1$. $A' \,[]\, B'$ is obtained if we add to $A \,[]\, B$ the specification of a consistent algorithm of choice, by the use of new variables.

Example : We consider the expression $S = \langle a_1, True, Id\rangle \,[]\, \langle b_2, True, Id\rangle$. If during successive executions of S, a_1 is the first event to be executed, and a_1 and b_2 must be selected alternately, then S may be replaced by $\langle a_1, v^1 = True, v^1 \leftarrow False\rangle \,[]\, \langle b_2, v^1 = False, v^1 \leftarrow True\rangle$, where v^1 is initially equal to $True$.

Restriction R2 may be necessary in expressions $A \,[]\, B$ and $A \,[>\, B$, only if these expressions are followed by operator $\gg$ [9, 6]. Therefore R2 may be "avoided", because any expression $P \gg Q$ may be replaced by an expression $P' \,|||\, Q'$ such that Q' cannot be executed while P' is not terminated.

Informally, P' and Q' are respectively equivalent to (i.e., may execute the sequences of events) P and Q, and the order between the two processes is ensured by the use of new variables, instead of operator $\gg$.

Example : $P = \langle a_1, True, Id\rangle$ and $Q = \langle b_2, True, Id\rangle$. $P ; Q$ may be replaced by $P' \,|||\, Q'$, where $P' = \langle a_1, True, v^1 \leftarrow True\rangle$, $Q' = \langle b_2, v^1 = True, Id\rangle$, and v^1 is initially equal to $False$. Therefore, variable v^1 ensures that Q' is disabled while P' is not terminated.

Restriction R3 is necessary in an expression $A \,[>\, B$. In the present study, we do not still know how the operator $[>$ may be avoided, by the use of new variables.

Therefore for this first question, we deduce that any desired service may be specified in EBL in such a way that restrictions R1 and R2 are not necessary.

5.5.2 Answering question 2

For a desired service specified in EBL by $Serv_V$, let us propose sufficient conditions which ensure that $Serv_V$ may be transformed into an equivalent $Serv'_V$, such that no message is generated at the second Step when our method of protocol synthesis is applied to $Serv'_V$.

Messages generated at the second Step are due to the use of operators ";", "$\gg$", "$[]$" and "$[>$". For operators ";" and "$\gg$", we have seen in Sect. 5.5.1 that they may be avoided by the use of new variables. For operator "$[]$" in an expression $A \,[]\, B$, the method of [6] (our second Step) generates no message if $AP(A) = AP(B)$ (See Not. 1 for $AP(*)$). Therefore, the respect of the two following conditions ensures that $Serv_V$ may be transformed in such a manner that all messages are generated at the first Step.

Condition1 : For any expression $A \,[]\, B$ in $Serv_V$, we have $AP(A) = AP(B)$.

Condition2 : $Serv_V$ does not use operator $[>$.

6 Two examples of protocol synthesis

Our method of protocol synthesis is applied to the two examples presented in Section 3.

6.1 Example of the disconnection phase

The desired service, specified in Sect. 3.1 by the automaton of Fig. 1, contains useless sequences. Its specification may be simplified by removing states 7, 8 and 9 of the automaton of Fig. 1. This means that, after he has executed a disconnect request, a user is *automatically disconnected* and then *cannot receive a disconnect indication*. The simplified service is defined in BL as follows, where restrictions R1 and R2 are not respected :

$$\text{SPEC } DISC = A(1,2) \,[]\, A(2,1) \text{ WHERE}$$
$$A(i,j) = dsreq_i \,;\, ((dsind_j \,;\, exit) \,[]\, (dsreq_j \,;\, exit))$$

This service is defined in EBL by the following specification which respects R1 and R2. Two boolean variables v^1 and v^2 are used, and their initial value is $False$.

$$\text{SPEC } DISC = A(1,2) \,|||\, A(2,1) \text{ WHERE}$$
$$A(i,j) = (\langle dsreq_i, True, v^i \leftarrow True\rangle \,;\, exit) \;[]\; (\langle dsind_i, v^j = True, Id\rangle \,;\, exit)$$

Informally : (for $i, j = 1, 2$ and $i \neq j$)

- User of $Site_i$ may execute a $dsreq_i$ or receive a $dsind_i$.
- User of $Site_i$ may receive a $dsind_i$ only if user of $Site_j$ has executed a $dsreq_j$ ($v^j = True$).

Restrictions R1 and R2 are respected by processes $A(1,2)$ and $A(2,1)$. The synthesized protocol can be specified as follows , for $i, j = 1, 2$ and $i \neq j$:

$$Prot_i = (\langle dsreq_i, True, v^i \leftarrow True\rangle \,;\, exit) \;[]\; (\langle dsind_i, v^j = True, Id\rangle \,;\, exit)$$

v^1 and v^2, which are initially $False$, do not need to be locked because we are in the particular case of Section. 5.4.1. In fact, for $i,j = 1,2$ and $i \neq j$:
- Each v^i is written only once, by transaction $\langle dsreq_i, v^i = False, v^i \leftarrow True\rangle$, and
- Transaction $\langle dsind_j, v^i = True, Id\rangle$ is disabled while v^i has not been written.

Therefore, the transactions are designed as follows, for $i,j = 1,2$ and $i \neq j$:

- v^1 and v^2 are initially write_locked.
- Transaction $\langle dsreq_i, True, v^i \leftarrow True\rangle$:
  - Executes $dsreq_i$,
  - Sets to $True$ and release the copy of v^i in $Site_i$,
  - Sends a message from $Site_i$ to the other $Site_j$ in order to set to $True$ and to release the local copy of v^i in $Site_j$.
- Transaction $\langle dsind_i, v^j = True, Id\rangle$:
  If the local copy of v^j in $Site_i$ is released, then : Reads it and, if it is $True$ then executes $dsind_i$.

The service is therefore provided by using at most two messages. The aim of each one is to inform a site that a $dsreq$ has been executed in the other site, by setting a variable to $True$ and by releasing it. Therefore, the number of messages is two only if users of both sites execute simultaneously a $dsreq_i$.

6.2 Example 2

Example of Section 3.2 (Fig. 2) may be specified in EBL, by the use of two boolean variables v^1 and v^2, as follows, where $\psi^i = (v^i = True)$, for $i = 1,2$, and $\theta = (\psi^1 \wedge \psi^2)$. The service does not depend on the initial values of v^1 and v^2 :

SPEC $SS = A \gg B \gg C \gg SS$ WHERE
$A = A(1) ||| A(2)$
$B = B(1,2) ||| B(2,1)$
$A(i) = (\langle \hat{\imath}_i, True, v^i \leftarrow False\rangle ; exit)$
$B(i,j) = (\langle a_i, True, v^i \leftarrow True\rangle ; exit)$
$\quad [] (\langle b_i, \psi^j, Id\rangle ; exit)$
$C = (\langle c_3, \theta, Id\rangle ; exit) [] (\langle \hat{\imath}_3, \neg\theta, Id\rangle ; exit)$

Informally :

- Process A set variables v^1 and v^2 to $False$ when the initial state is reached (Fig. 2).
- v^1 (resp. v^2) is set to $True$ if the user of $Site_1$ (resp. $Site_2$) *executes* a_1 (resp. a_2).
- The user of $Site_i$ may (for $i,j = 1,2$ and $i \neq j$) :
  Either execute a_i;
  Or execute b_i, but only if v^j is equal to $True$.
- The user of $Site_3$ may execute c_3 if both v^1 and v^2 are equal to $True$.

We can easily check that restrictions R1 and R2 are respected by processes $B(i,j)$ and C, which contain the operator []. In the obtained protocol specifications, we use the two following notations :

$rcv_i^j(m) = \langle r_i^j(m), True, Id\rangle ; exit$ and
$snd_i^j(m) = \langle s_i^j(m), True, Id\rangle ; exit$.

The protocol specifications $\mathcal{P}rot_i$, for $i = 1,2$, and $\mathcal{P}rot_3$ can then be specified as follows ($j = 1,2$ and $j \neq i$):

$SPEC\ \mathcal{P}rot_i = A_i \gg M_i \gg B_i \gg M_i' \gg \mathcal{P}rot_i$
$\quad WHERE$
$A_i = \langle \hat{\imath}_i, True, v^i \leftarrow False\rangle ; exit$
$B_i = (\langle a_i, True, v^i \leftarrow True\rangle ; exit)$
$\quad [] (\langle b_i, v^j = True, Id\rangle ; exit)$
$M_i = snd_i^j(mi) \gg rcv_i^j(mj)$
$M_i' = snd_i^3(pi) \gg rcv_i^3(m3)$

SPEC $\mathcal{P}rot_3 = M_3 \gg C_3 \gg M_3' \gg \mathcal{P}rot_3$ WHERE
$M_3 = (rcv_3^1(p1) ||| rcv_3^2(p2))$
$C_3 = (\langle c_3, \theta, Id\rangle ; exit) [] (\langle \hat{\imath}_3, \neg\theta, Id\rangle ; exit)$
$M_3' = snd_3^1(m3) ||| snd_2^3(m3)$
ENDSPEC

In this example, we are in the particular case of Sect. 5.4.2. In fact, the desired service is specified by $SS = A \gg B \gg C \gg SS$, where $A = A(1) ||| A(2)$ and $B = B(1,2) ||| B(2,1)$. According to the approach proposed in Sect. 5.4.2 , for i=1,2 :

- After the execution of A_i, $Site_i$ waits a time equal to $Total_delay$ before sending mi.
- After the execution of B_i, $Site_i$ waits a time equal to $Total_delay$ before sending pi.
- After the execution of C_3, $Site_3$ waits a time equal to $Total_delay$ before sending $m3$.

With this approach, the transactions are designed as if each process A, B or C were executed alone.

Process A : Each variable is written only once and is never read. Therefore, its locking is not necessary.

Process B : It is similar to the disconnection phase (Sect. 6.1) which respects the particular case of Sect. 5.4.1. The variables are initially $False$ because process B follows process A. According to the solution proposed in Sect. 5.4.1, the two variables must be initially write_locked. Therefore, A must write_lock them before the beginning of B. The variables will be released after being set to $True$. They may be read only after being released.

Process C : Each variable is only read and is never written. Therefore, its locking is useless.

Therefore, the transactions may be designed as follows, where $i,j = 1,2$ and $i \neq j$:

- Transaction $\langle \hat{\imath}_i, True, v^i \leftarrow False\rangle$ (in process A) :
  - Executes the internal action $\hat{\imath}_i$
  - Sets to $False$ the local copy of v^i in $Site_i$ (without locking it),
  - Sends a message from $Site_i$ to $Site_j$ in order to set to $True$ and to write_lock the copy of v^i in $Site_j$. Therefore in process B, the local copy will be initially write_locked.
  - Sends a message from $Site_i$ to $Site_3$ in order to set to $True$ the copy of v^i in $Site_3$.
- Transaction $\langle a_i, True, v^i \leftarrow True\rangle$ (in process B) :
  - Executes a_i,
  - Sets to $True$ and release the copy of v^i $Site_i$,
  - Sends a message from $Site_i$ to $Site_j$, in order to set to $True$ and to release the copy of v^i in $Site_j$.
  - Sends a message from $Site_i$ to $Site_3$, in order to set to $True$ the local copy of v^i in $Site_3$.

- Transaction $\langle b_i, v^j = True, Id \rangle$ (in process B) :
  If the local copy of v^j in $Site_i$ is released, then :
  Reads it and, if it is $True$ then executes b_i,
- Transaction $\langle c_3, \theta, Id \rangle$ (in process C) :
  - Reads copies of v^1 and v^2 in $Site_3$, computes θ.
  - If $\theta = True$ then executes c_3.
- Transaction $\langle \hat{\imath}_3, \neg\theta, Id \rangle$ (in process C) :
  - Reads copies of v^1 and v^2 in $Site_3$, computes θ.
  - If $\theta = False$ then executes $\hat{\imath}_3$.

The messages exchanged during one cycle can then be separated in two groups :

Six synchronization messages (Step 2) ($i, j = 1, 2$ and $i \neq j$):
- One message mi from $Site_i$ to $Site_j$;
- One message $m3$ from $Site_3$ to $Site_i$.
- One message pi from $Site_i$ to $Site_3$;

Six or Eight transaction messages (Step 1) :
- Two messages by each transaction $\langle \hat{\imath}_i, True, v^i \leftarrow False \rangle$, $(i=1,2)$;
- Two messages by each transaction $\langle a_i, True, v^i \leftarrow True \rangle$ if it is executed, $(i=1,2)$;

It is interesting to note that more than half messages (eight) are due to the fact that the service is cyclic. These messages are $m1$, $m2$, $m3$, and those generated by transactions $\langle \hat{\imath}_i, True, v^i \leftarrow False \rangle$, $(i=1,2)$.

7 Conclusion

In this paper, we develop a formalism called EBL which extends basic LOTOS by the use of global variables. Then, we propose a method of protocol synthesis which extends the method of [6], by using specifications written in EBL. An advantage of our method is that there are cases where it generates a correct solution while method in [6] is not applicable. Two examples illustrate this advantage. We recognize that many aspects remain to be studied. Some of them are :

1. To define several service structures which allow to simplify the design of transactions. Two cases are proposed in Sect. 5.4 and are illustrated in Examples of Sect. 6.
2. To extend EBL with timing requirements.
3. To work on more complex examples.

Acknowledgements

We thank Anindya Das and Benoit Caillaud for helpful comments on a first version of the paper.

References

[1] P.A. Bernstein, V. Hadzilagos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[2] G.v. Bochmann and R. Gotzhein. Deriving protocols specifications from service specifications. In *Proceedings of the ACM SIGCOMM Symposium*, USA, 1986.

[3] T. Bolognesi and E. Brinskma. Introduction to the iso specification language lotos. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[4] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Comm. of the ACM*, 19(11), November 1976.

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

[6] C. Kant, T. Higashino, and G.v. Bochmann. Deriving protocol specifications from service specifications written in lotos+. Technical Report 805, Université de Montréal. Département IRO, Montréal, Québec, Canada, January 1992.

[7] M. Kapus-Kolar. New results on deriving protocol specifications from service specifications. In *Proceedings of MELECON*, pages 1093–1096, Ljubljana, Yugoslavia, 1991.

[8] M. Kapus-Kolar, J. Rugelj, and M. Bonač. Deriving protocol specifications from service specifications. In *Proceedings of IASTED INT. SYMP. APPLIED INFORMATICS*, pages 375–378, Innsbruck, 1991.

[9] F. Khendek, G.v. Bochmann, and C. Kant. New results on deriving protocol specifications from services specifications. In *Proceedings of the ACM SIGCOMM Symposium*, pages 136–145, 1989.

[10] M. Raynal. *Gestion des données réparties : problèmes et protocoles.* Eyrolles, 1992.

[11] M. Raynal. *Synchronisation et état global dans les systèmes répartis.* Eyrolles, Collection EDF, 1992.

[12] I.L. Traiger, J. Gray, C.A. Galtieri, and B.G. Lindsay. Transactions and consistency in distributed database systems. *ACM TODS*, 7(3):323–342, 1982.

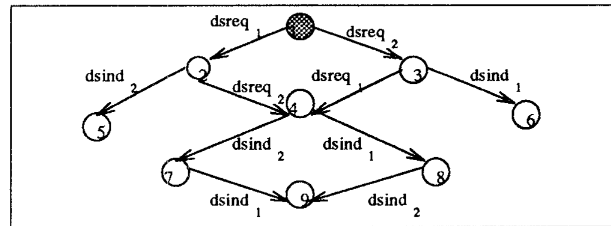

Figure 1: Disconnection phase : Service specification.

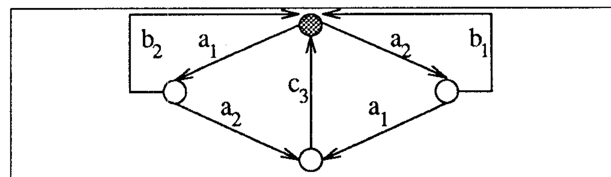

Figure 2: Specification of a non terminating service.