ソフトウェア工学 90－5
（１９９3. 2. 4）

高信頼ソフトウェア設計のための言語的支援

ドゥシャン　ヨカノビィチ　太田　正孝

(株)　高度通信システム研究所

〒989-32　仙台市青葉区南吉成6丁目6番地の3

あらまし

本報告では，形式的仕様記述言語LOTOSに基づく設計環境について考察する．設計者にとって高信頼ソフトウェアを効率よく開発できる設計環境が必要である．この設計環境を実現ためにLOTOSを拡張し，割りこみ不可能なアクション系列の直接的な記述を可能にした．これによりソフトウェアのエラーリカバリ機構の記述が可能になる．本文では，拡張されたLOTOSの性質を述べ，いくつかの例を示す．すなわち，拡張されたLOTOSにより，いかにして高信頼にプロトコルを記述するかを示す．

LINGUISTIC SUPPORT FOR DESIGNING RELIABLE SOFTWARE

Dusan Jokanovic　　Masataka Ohta

Advanced Intelligent Communication System Laboratories, Ltd.

6-6-3, Minami Yoshinari, Aoba-ku, Sendai, 989-32, Japan

ABSTRACT: *This paper considers design environment based on LOTOS formal specification language. Designers need an environment that effectively supports the development of reliable software. As a first step towards this goal, we introduce an extension of LOTOS that makes it possible to define directly noninterruptable sequences of actions in specifications, that is, atomic actions at any level of abstraction. This allows specifying different software mechanisms for error recovery. The properties of the new language construct are discussed. In addition, some examples of its use are given. Namely, we show how to use the enhanced LOTOS in order to specify a simple reliable protocol.*

Key words :　Distributed computing, Service specifications, Fault tolerance, LOTOS

1. Introduction

Communications, robotics, process control, and other critical computer applications demand reliable software. Software for such systems generally comprises a set of concurrent, cooperating processes. Designers need an environment that effectively supports the development of fault-tolerant software[1]. Several researches have developed methods and tools that use redundancy to help critical systems tolerate errors caused by software faults. The most suitable mechanisms for concurrent systems are programmer transparent coordination and conversation[2],[3]. All four of these approaches are general, apply to any type of computation and they are based on atomic actions. Unfortunately, the few languages that provide adequate syntax and run-time support to implement fault-tolerant mechanisms are still experimental.

In this paper, we consider design environment based on LOTOS, a Formal Description Technique being developed within ISO for the formal specifications of OSI protocols and services[4]. In standard LOTOS, the parallel operator specifies interleaving at the elementary action level. In other words, only elementary actions are viewed as atomic or noninterruptible. Larger atomic actions, called here atomic processes can be specified by using constructs such as semaphores. However, this presents some disadvantages, among others from the point of view of modular design. For example, the parallel composition of two processes may introduce some unwanted interleaving, thus limiting the usefulness of this way of composing modules. Furthermore, one of the main concepts of step-wise development of system is to be able to progressively expand what appears to be a single action *A* at a high level of abstraction into a functionally equivalent, possibly complex compound action *B* at a lower level of abstraction. This presents a problem when concurrency is present: because, if *A* is considered to be atomic, so must be *B* in order to be equivalent to it.

As a first step towards specifying reliable and modular software, we introduce an extension of LOTOS that makes it possible to define atomic processes at any level of abstraction. We extend LOTOS language by some operators apt to define noninterruptible composition of actions. In the resulting enhanced language one can specify behaviors of systems with some sub-behaviors being atomic. This allows specifying mechanisms for error recovery[5],[6].

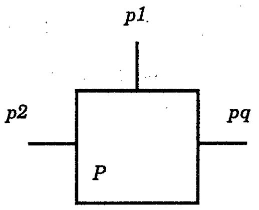
In Section 2, we explain the notion of LOTOS processes briefly and the importance of their atomicity from the aspect of fault tolerance. In Section 3, firstly, we show how to specify atomic processes based on standard LOTOS constructs. Then, a new language construct is proposed which enables specifying atomic processes in LOTOS with less inconvenience. Finally, some properties of the construct are discussed. In Section 4, we show how the enhanced LOTOS can be used in order to specify a reliable protocol. Conclusion is given in Section 5.

2. LOTOS processes

2.1 LOTOS characteristics

In LOTOS, a system as a whole is specified as a single process which may consist of several interacting subprocesses which may in turn be refined into sub-subprocesses, etc. Thus, a LOTOS specification of a system is essentially a hierarchy of process definitions. A process specification in LOTOS describes its behavior, that is, the sequences of observable actions that may occur at its gates -interaction points (Fig. 1). A process performs action either alone on one of its gates or in cooperation with other process(es) on shared gates. In the latter case it is said that participating processes synchronize on these gates by performing rendezvous.

The hierarchical structure of LOTOS specifications suggests appropriate execution model. Namely, we



process *P* [*p1*, *p2* , *pq*] : **exit** :=
p1; (*pq* ; *p2*; **exit**) |[*pq*]| *i* ; *P*
endproc

Fig.1 A process specification in LOTOS

assume that the executable processes are organized in a binary tree structure, called activity tree[7], where each of the leaf processes behaves, at any time, according to corresponding behavior expression. This expression identifies the process current state and indicates the set of alternative actions that it can execute as well as the corresponding terminating state that it reaches on executing each of them. Each of the other nodes in the tree represents a process that is composed of its two child processes, according to the binary behavior construct. Therefore the root process behaves as described by LOTOS specification of the whole system. Behavior expressions of all its successors are formed by recursive binary splits of system specification. This splits can be performed in the various ways dynamically during the execution. The essential is that whenever the behavior associated with an executable process P is defined by an expression $BE=BE_1*BE_2$, "*" being any binary behavior construct, it is assumed that P may give birth to two new processes, say Q and R, such that Q behaves according to BE_1 and R behaves according to BE_2. At the same point of execution, newborn processes Q and R are allocated to some processor(s) which need not be the same which P is assigned to. From that moment on P represents the behavior composition of Q and R, defined by construct "*" and it is in charge of coordinating the activities of Q and R. The operator "*" can be: >>, [>, [], |||, or |[{*synchrogates*}]|, which represent enabling, disabling, alternative choice, independent parallelism and dependent parallelism, respectively. The following situations are possible: 1) Q and R are bound by the enabling construct ($P=Q>>R$). P must first activate only Q, and, when Q ends its execution successfully, P must kill Q and activate R; 2) Q and R are bound by the disabling operator ($P=Q$ [>R). Both processes must be activated by P, but, as soon as R executes its first action, P must kill Q. 3) Q and R are bound by a choice operator ($P=Q$[]R). As soon as one of them executes an action, the other one is killed by P; 4) Q and R are independent concurrent processes ($P=Q$ ||| R); 5) Q and R are bound by a parallel construct ($P=Q$ |[{*synchrogates*}]| R). These processes cannot execute actions on the synchronization gates by their own, but they must have a rendezvous on those gates. In addition to above constructs, LOTOS expression $a;b;P$ notifies temporal ordering in the sense that a process P can be activated only after actions a and b are exercised (first a then b).

2.2 Fault-tolerance and Atomicity

In the system of concurrent processes such as LOTOS execution model, it is essential to prevent domino effect caused by error propagation throughout system. The error confinement can be achieved using atomic software constructs like conversations. Each process that joins a conversation has a recovery point, an acceptance test and alternate algorithm. While a process is in a conversation it may only communicate with other processes in the same conversation. If any process fails an acceptance test or otherwise detects an exception, every process in the conversation performs a rollback to its recovery point, established on entry to the conversation and uses an alternate algorithm. The procedure described above implements backward error recovery scheme. Similarly, forward error recovery in concurrent software is based on atomic actions, as well. Exceptions, signal and raise operations and exception handlers are common mechanisms for providing forward recovery. If any process raises an exception every process in the atomic action invokes an exception handler for the exception. If all processes can recover the process return from the exception handlers and complete the atomic action normally. However, if any of the processes cannot recover, all of the processes complete the conversation abnormally and signal an exception.

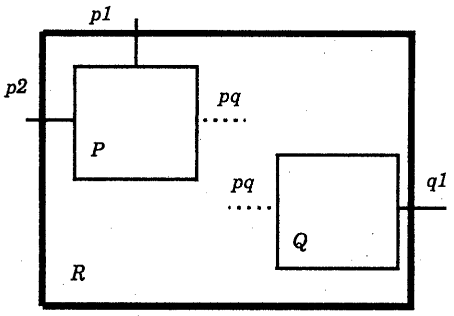
The concept of atomic actions (processes) can be used to structure the temporal activity of the system. An atomic action is an activity, possibly consisting of many steps performed by many different processors, that appears primitive and indivisible to any activity outside the atomic action. To other activities, an atomic action is like a primitive operation which transforms the state of the system without having any intermediate states.

3. Atomic LOTOS processes

3.1 Standard language

As the atomic actions seems necessary for efficient error recovery we propose an atomic action called *LOTOS Atomic Process* or *LAP*. *LAP* represents a distributed control structure that a group of LOTOS processes may join or leave together in synchrony. Inside a *LAP* the processes may communicate with one another, but not with processes outside of the control structure. *LAP* is conceived as a planned atomic action since it has to be decided at the time of system specification

Let us suppose that processes P, Q and R which belong to the system under specification have to perform some critical task in cooperation and concurrently. Firstly, to improve reliability of the task execution a *LAP* for participating processes has to be specified and then, within it, some error recovery mechanism provided. Specification of such error recovery mechanisms are given elsewhere[8]. Here, we



```
process R[p1,p2,q1]: exit:=
behavior
hide pq in
P[p1,p2,pq]|[pq]|Q[q1,pq]
```

Fig. 2 Hiding in LOTOS

merely specify LAP taking advantage of LOTOS specification construct called *hiding*. It allows specifying gates in a process definition as hidden making actions on those gates internal to the process and unobservable from the process environment. Then, only its subprocesses may have rendezvous (multiway synchronization with or without data exchange) on those gates, independently of other processes (Fig. 2).

For each *LAP* the following has to be done at the specification level: 1) for synchronization - two gates for entrance and exit, *lapin* and *lapout*, respectively have to be specified; 2) for error recovery - all input values of parameters of constituent processes have to be saved; 3) for management - a process *CLAP* which would be *LAP* controller has to be determined; 4) for atomicity - all gates of constituent processes have to be hidden. Let us proceed in indicated order and suppose that processes:

P[{*gates*},{*synchrogates*},*parametergate*](*p-parameters*),
Q[{*gates*},{*synchrogates*},*parametergate*](*q-parameters*),
R[{*gates*},{*synchrogates*},*parametergate*](*r-parameters*)

are planned to execute some critical task concurrently by exchanging information through set of synchrogates, without exchanging information with any other processes, and with high reliability. First, define new processes *P'*, *Q'*, and *R'* with common gate *lapin* as follows: *P'*=*lapin*;*P*, *Q'*=*lapin*;*Q* and *R'*=*lapin*;*R*. Thus, in addition to gates of processes *P*, *Q*, and *R*, these processes have a gate *lapin*, as well. This gate serves at the same time as *LAP* identifier. Next, inside the bodies of *P*, *Q*, and *R*, for each of their terminating subprocess we define gate *lapout* before exit, in the pattern....*lapout* ; exit. In addition, for each processor, we define a parallel synchronized process, *SAVE*[*parametergate*](*parameters*), responsible for saving initial values of process input parameters. Now, lets define *CLAP* as a *LAP* controller as follows:

```
process CLAP[lapin,lapout] : exit: =
behavior
hide all synchrogates in
P'|[synchrogates, lapin, lapout]|
Q'|[synchrogates, lapin, lapout ]| R'
endproc
```

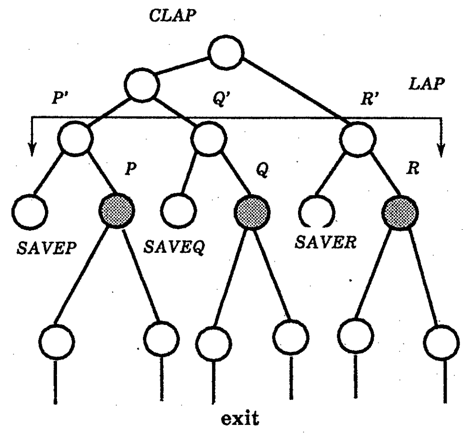


Fig. 3 LOTOS Atomic Process

Since all gates in *CLAP* are hidden the information smuggling across *LAP*'s border is prevented. This has to be supported by run-time mechanism, too. After process substitution *CLAP* behavior is equal to the following one:

lapin;(*SAVEP*[*parametergate*](*p-parameters*)||*P*)
|[*synchrogates, lapin, lapout*]|
lapin;(*SAVEQ*[*parametergate*](*q-parameters*)||*Q*)
|[*synchrogates, lapin, lapout*]|
lapin;(*SAVER*[*parametergate*](*r-parameters*)||*R*)

Thus, after performing rendezvous on gate *lapin* the processes *P'*, *Q'* and *R'* enter *LAP* synchronously giving birth to processes *P*, *Q* and *R* accompanied with corresponding *SAVE* processes, respectively (Fig. 3). Then, they continue performing assigned task exchanging information through exchange gates. Notice that the children of processes are members of their parents' *LAP*. *LAP* terminates successfully as soon as all its members perform multirendezvous on gate *lapout* and exit. Then, the other processes dependent on the results of *LAP* may be enabled with initial values of parameters computed in *LAP* and conveyed through exit. *CLAP* checks if all processes supposed to enter the *LAP* have applied for it and then performs: synchronized *LAP* initiation, *LAP* isolation, and synchronized exit. Moreover, it has to support system recovery after an unsuccessful attempt to assembling all planed processes in *LAP*.

3.2 Enhanced language

Hiding is not satisfactory enough for specifying atomic processes, since it hides actions, which otherwise might have to be visible in the environment. Therefore, we propose a new construct for specifying atomic LOTOS processes. It stands for strong sequencing construct and it is denoted by "●". For example, $a \bullet b$ means that after action a is executed, action b must immediately follow. The behavior of this new construct has to be defined in terms of the two LOTOS operators that involve concurrency, that is, parallel composition and disabling. To this end, we have defined auxiliary constructs expressing the fact that, in the presence of strong sequencing, evaluation of an expression must continue along the subexpression where strong sequencing is present.

Concerning composition, the auxiliary constructs are: 1) The left composition construct, denoted by $|l$, is like the LOTOS composition construct except that it takes its new action from the left process. In the similar way, 2) the right composition construct, noted $|r$ takes its next action from the right process.

Concerning disabling, the semantics of the disable with atomicity is expressed as follows: if a strong sequence is disabled by a process then the disabling should be delayed until the end of the atomic process. For this purpose we introduce a new derived disable construct called left disable, and we denote it by $l>$.

Before giving formal definition of the semantics of these constructs, we provide some examples of their use. We will use the following identities:

$B \|$ stop $= B$, stop $\| B = B$, and stop $[> B =$ stop

In examples, indentation expresses sequencing and two alternatives at the same indentation level express a choice between derivations.

Example 1:

```
a ● b;stop || c ● d;stop
-a-> (b ; stop |l c ● d ; stop)
    -b-> (stop || c ● d ; stop)
        -c-> d ; stop
            -d-> stop
-c-> (a ● b ; stop |r d ; stop)
    -d-> (a ● b ; stop || stop)
        -a-> b ; stop
            -b-> stop
```

The starting expression specifies a composition between two atomic sequences "$a \bullet b$" and "$c \bullet d$". As we can see from the derivations, action "a" is always followed by action "b" and action "c" is always followed by "d" and both atomic actions are executed in any order. Please notice that if, the environment offers "c" after having offered "a", the result is a deadlock.

Example 2:

```
a ● b ; stop [> c ● d ; stop
-a-> (b ; stop l> c ● d ; stop)
    -b-> (stop [> c ● d ; stop)
        -c-> d ; stop
            -d-> stop
```

Sequence "$a \bullet b$" cannot be disabled by sequence "$c \bullet d$" The formal semantics of the ● construct is complicated by the presence of the choice construct [], since a process can be defined as a choice between two different alternatives, one of which is atomic and the other not, such as the expression:

$(a \bullet b$; stop [] c ; d ; stop) $\|$ $e \bullet f$; stop

Above expression shows a case where it is undefined which parallel composition to apply after each action. If left composition is applied, then "c ; d" is treated as atomic, while if normal composition is applied, then "$a \bullet b$" will be interrupted. Intuitively, both situations are undesirable. In order to deal with this problem, we introduce an operational semantics with attributes, which will carry the atomicity nature of each expression In order to be able to build an inference system we use parametarized inference rules. The parameter can be considered as a synthesized attribute which will indicate the nature of the derivative of an expression. This attribute has two possible values: "*at*" to indicate an atomic derivation and "*int*" to indicate an interruptible derivation. For simplicity we will use the name "*any*" to denote a variable that ranges over {*at*, *int*}. The values "*int*" and "*at*" represent the atomicity of derivations. In the following, by an action (e.g. $a,b,...$) we mean an action consisting of a gate name followed by its list of input/output events or an internal action "i". Next we give only set of selected inference rules.

For sequencing:

R1: $a \bullet B$ -a-(at)-> B,

R2: a ; B -a-(int)-> B

Rule 1 states that the derivation by strong sequencing is atomic. Rule 2 says that the derivation by sequencing is interruptible. In the similar way inference rules can be derived for other constructs:

For choice:

R3: *A* [] *B* -*a*-(*any*) -> *A*' if *A*-*a*-(*any*) -> *A*'

R4: *A* [] *B* -*a*-(*any*) -> *B*' if *B*-*a*-(*any*) -> *B*'

Rules 3 and 4 state that the derivative of a choice and its atomicity is the same as the derivative and the atomicity of one of its alternatives.

For parallel composition:

R5: *A* || *B* -*a*-(*at*) -> *A*' |*l* *B* if *A* -*a*-(*at*)-> A'

R6: *A* || *B* -*a*-(*at*) -> *A* |*r* *B*' if *B* -*a*-(*at*)-> *B*'

R7: *A* || *B* -*a*-(*int*) -> *A*' || *B* if *A* -*a*-(*int*)-> *A*'

R8: *A* || *B* -*a*-(*int*) -> *A* || *B*' if *B* -*a*-(*int*)-> *B*'

R9: *A* || *B* -*a*-(*any*) -> *A*' || *B*'
  if *A* -*a*-(*any*)-> *A*' and *B* -*a*-(*any*)-> *B*'

Please note that, according to this semantics atomic sequences can compose only with atomic sequences.

For left composition:

R10: *A* |*l* *B* -*a*-(*int*)- > *A*' || *B* if *A*-*a*-(*int*) -> *A*'

R11: *A* |*l* *B* -*a*-(*int*)- > *A*' || *B* if *A*-*a*-(*int*) -> *A*'

Thus, the atomicity of derivation is the same as the atomicity of derivation of the left process.

For disable operation:

R12: *A* [> *B* -*a*-(*at*) -> *A*' l> *B* if
  A -*a*-(*at*)-> *A*' and Label(*a*) $\neq$ exit

R13: *A* [> *B* -*a*-(*int*) -> *A* [> *B* if
  A -*a*-(*int*)-> *A*' and Label(*a*) $\neq$ exit

R14: *A* [> *B* -*a*-(*any*) -> *A*' if *A* -*a*-(*any*)-> *A*'
  and Label(a) = exit, where exit is termination gate;

R15: *A* [> *B* -*a*-(*any*) -> *B*' if *B* -*a*-(*any*)-> *B*'

When the disabled process is atom-prefixed, the next action is to be taken from it (Rule 12) if it has not terminated. If a process is terminated then the disabling has no effect (Rule 13). Please notice that Label(*x*) returns the gate name that occurs in action *x*.

Rules 5 (respectively 6) states that if a process *A* (respectively *B*) derivates in an atomic way to another process *A*' (resp. *B*'), then the composition of process *A* (resp. *B*) with another process *B* (resp. *A*) is atomic and the next action of the composition will be taken from *A*' (resp. *B*'). These two rules express the interleaving.

Now, as an example, we use this new formalism to derive the behavior of process defined as follows:

<u>Example 3:</u>

```
(a ● b ; stop [] c ; d ; stop) || e ● f ; stop
-a-(at)-> (b ; stop |l e ● f ; stop)
    -b-(int) -> (stop || e ● f ; stop)
        -e-(at )-> f ; stop
            -f-(int)-> stop

    -c-(int)-(d ; stop || e ● f ; stop)
        -d-(int) -> (stop || e ● f ; stop)
            -e-(at)-> f ; stop
                -f-(int)-> stop

        -e-(at) -> (d ; stop |r f ; stop)
            -f-(int)-> (d ; stop || stop)
                -d-(int )-> stop

-e-(at)-(a ● b ; stop [] c ; d ; stop) |r f ; stop
    -f-(int)-> (a ● b ; stop [] c ; d ; stop) || stop
        -a-(at)-> b ; stop
            -b-(int ) -> stop
        -c-(int)-> d ; stop
            -d -(int)-> stop
```

The sequences *a* ● *b* and *e* ● *f* are not interruptible while *c* ; *d* can be interrupted by *e* ● *f*.

4. <u>Specification examples</u>

First, we will show how to specify atomic sequences of actions within a process. Those actions should not overlap with other actions of other processes. In the example below, processes *A* and *B* have atomic actions denoted by *as11* ● *as12* and *as21* ● *as22*, respectively.

A: = *a1* ; *as11* ● *as12* ; *A*

B: = *b1* ; *as21* ● *as22* ; *B*

We are interested to find a process equivalent to the composed process *A* || *B*, defined as follows:

A||*B* = *a1*; (*as11* ● *as12*; *A*||*B*) [] *b1*; (*A* || *as21*● *as22*;*B*)

Let *P* = (*as11*● *as12* ; *A*||*B*) and *Q* = (*A* || *as21* ● *as22* ; *B*)

By substituting and then expanding we get:

P =

as11● (*as12*; *A* |*l* *B*) [] *b1*; (*as11*● *as12*; *A*|| *as21*● *as22*;*B*)

Q =

a1;(*as11*● *as12*; *A*|| *as21*● *as22*; *B*)[] *as21*● (*A* |*r* *as22*; *B*)

Let *R* = (*as11*● *as12*;*A* || *as21*● *as22*;*B*).

Then by laws: *as12*; *A* |*l* *B* = *as12*; (*A* || *B*)

and *A* |*r* *as22*; *B* = *as22*; (*A* || *B*), it follows:

P = *as11*● *as12* ; (*A*||*B*) [] *b1*; *R* and

Q = *as21*● *as22* ; (*A*||*B*) [] *a1*; *R*, where

R = *as11* ● *as12* ; (*a1* ; *R* [] *as21* ● *as22* ; (*A* || *B*))
  [] *as21* ● *as22* ; (*b1* ; *R* [] *as11* ● *as12* ; (*A* ||*B*))

At last, we get:

A||*B* = *a1* ; (*as11* ● *as12* ; (*A* || *B*) [] *b1* ; *R*)
  [] *b1* ; (*as21* ● *as22* ; (*A* || *B*) [] *a1* ; *R*)

Thus, it is clear that the resulting behavior is equal to interleaving of the behaviors of the two processes without the overlapping of their atomic actions.

As mentioned, in LOTOS there are some high-level composition constructs that can be used in splitting systems into modules that can be:

1) executed sequentially using the enabling operator>>,
2) executed concurrently using the composition operator ||,
3) executed by having a process disabling another using operator [>.

In doing so, the design process becomes in principle easy. However, as mentioned in Introduction, usually it is not easy to split systems into several concurrent components because of the high level of the concurrency that is involved in LOTOS.

For instance, suppose that one wants to build a send-receive protocol by performing the parallel composition of a sender and a receiver. This will normally not be possible because of the interleaving that is involved in the language, namely sender and receiver will be able to interleave at each elementary operation and there is no way of preventing them from doing so, short of using semaphores. In other words, a protocol specification described by existing LOTOS language operators is either ambiguous and unreliable or cumbersome. In order to specify reliable protocol in elegant way we make use of proposed LOTOS extension.

The protocol entity (Fig. 4) that we want to specify should provide a data sending service as well as data receiving service. The entity will get data from a user via gate "*from__user*", send it via a medium at gate "*to__ medium*", receive data from a medium via gate "*from__medium*" and deliver it to a user via gate "*to__user*".

First, let us specify the sender:

sender : = *from__user* ● *to__medium* ; *sender*

then the receiver:

receiver : = *from__medium* ● *to__user* ; *receiver*

The whole entity will be then:

protocol : = (*sender* || *receiver*)

By expansion we get:

sender || *receiver*

= *from__user* ● (*to__medium* ; *sender* |*l* *receiver*)

[] *from__medium* ● (*sender* |*r* *to__user* ; *receiver*)

= *from__user* ● *to__medium* ; (*sender* || *receiver*)

[] *from__medium* ● *to__user* ; (*sender* || *receiver*)

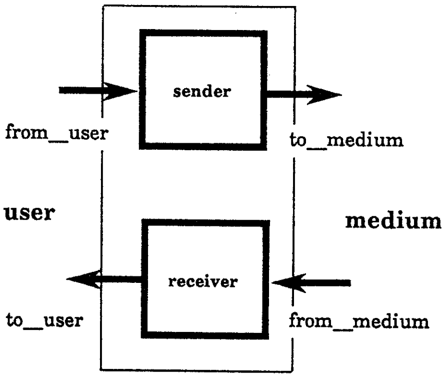


Fig. 4 Protocol entity

At last, *protocol* : = *from__user* ● *to__medium* ; *protocol*

[] *from__medium* ● *to__user* ; *protocol*

Therefore, by combining in parallel the two processes, we have obtained a system that provides the two functions, unambiguously.

5. Conclusion

We research specification mechanisms to allow some critical part of LOTOS processes to recover after a manifestation of an error. Namely, a notation for specifying an atomic action, named LOTOS Atomic Process *LAP* is proposed. Automatic implementation and execution of LOTOS specifications is important step in design of distributed software systems. It can be especially useful in rapid prototyping in distributed environment where system functionality checking is in focus and efficiency is less important. The results of our research can be applied for designing simulators of LOTOS specifications.

References:

[1] B. Randell," *System Structure for Software Fault Tolerance*," IEEE Trans. Software Eng., Vol. 1, pp. 221-232, June 1975.
[2] T. Anderson and P. A. Lee, **Fault Tolerance, Principles and Practice**, Prentice-Hall, 1981.
[3] F. Christian," *Exception Handling and Software Fault Tolerance*," IEEE Trans. Comp., Vol. 31, pp. 531-540, 1982.
[4] T. Bolognezi and E. Brinksma," *Introduction to the ISO Specification Language LOTOS*," Computer Networks and ISDN Systems, vol. 14, North-Holland, 1987.
[5] P. Jalote, R. Campbell, "*Atomic Action for Fault Tolerance Using CSP*," IEEE Trans. Software Eng., vol 12, no. 1, pp. 59-68, Jan 1986.
[6] C. A. R. Hoare," *Communicating Sequential Processes*," Commun. ACM, vol. 21, no. 8, pp. 666-677, Aug. 1978.
[7] G. Bochman, Q. Gao and C. Wu, "*On the Distributed Implementation of LOTOS*," in Proc. 2nd Int. Conf. Formal Description Techniques, Canada, Dec. 1989.
[8] D. Jokanovic and M. Ohta" *Supporting fault tolerance in distributed LOTOS environment*," Tehnical Report of IEICE, FTS 92-12 (1992-09).