A PARTIAL *RJE* PAD SPECIFICATION TO ILLUSTRATE *LOTOS*

D G Kourie
Department of Computer Science
University of Pretoria, 0001 Pretoria

LOTOS (Language Of Temporal Ordering Specification) is employed to give a partial specification of a system to connect RJE devices across an X.25 network. The system's implementation has been described fully elsewhere [5]. The present purpose is to introduce LOTOS as a specification language, showing how fairly complex time-dependencies may be described in an unambiguous fashion, and also pointing to the way in which LOTOS specifications may be verified.

Key concepts of LOTOS are surveyed, and the underlying model which abstracts the RJE system to be specified is presented. The LOTOS specification of this model is given in outline, with particular emphasis on aspects of the connection phase. The type of verification to which a LOTOS specification may be subjected is briefly discussed and an indication is given of how such verification may be approached.

1.OVERVIEW OF *LOTOS*

1.1 Background

After several years of development (from 1981 to 1984), a draft proposal for the syntax and semantics of LOTOS [9] was produced by ISO FDT Subgroup C in March 1985. Currently, this proposal is undergoing minor revisions. A provisional LOTOS Tutorial has also been provided [12].

LOTOS derives from the seminal work of Milner who describes a Calculus for Communicating Systems (CCS) [15]. It is based on a modification of this formal mathematical calculus, referred to as CCS*. Data structures and value expressions (ie language expressions that describe data values) in LOTOS are represented in the same way as in ACT ONE - an abstract data type language described by Ehrig et. al. [6]. It should be noted that the currently available ISO draft proposal document [9] does not describe ACT ONE, but gives a self-contained though somewhat turgid description of CCS*.

An informal introduction to highlight some of the important CCS* concepts now follows in 1.2 to 1.7. This introduction also serves to describe LOTOS, in that the concepts defined in CCS* apply directly to LOTOS, and the way in which these concepts are represented in LOTOS differs only slightly from the CCS* representations. Some of the main differences are discussed in 1.8.

It should be mentioned that LOTOS bears a strong resembelance to CSP developed by Hoare [8], since both are influenced by Milner's work.

1.2 Processes and interactions

In CCS* processes are viewed as abstract entities which may interact with each other at abstract shared resources called interaction points. At such points primitive atomic synchronised interactions occur which are called events.

1.3 Behaviour Expressions

The behaviour of a process in CCS* is described by a so-called behaviour expression. This expression is primarily (but not exclusively) aimed at describing the way in which a process appears to its external environment. Essentially this means that a behaviour expression describes events and their temporal ordering at interaction points shared by a process and its external environment.

In general, a behaviour expression is built up by conjoining smaller behaviour expressions using operators described below. Hence, if B1 and B2 are behaviour expressions, and one of the

legitimate CCS* operators, denoted by <op> is applied to B1 and B2 to obtain B1 <op> B2, then this represents a new behaviour expression, say B3 which may in turn be conjoined by some operator to another behaviour expression to obtain yet another behaviour expression, etc. The final behaviour expression thus obtained represents the description of some process of interest to the specifier, and is shown in CCS* by the notation :

p(x1,...,xn) := B

Here, p is called a behaviour identifier which gives a name to the process being described. x1,...,xn are variables which occur in the behaviour expressions used to describe the process, and B is the actual behaviour expression which describes the process. The representation p(x1,...,xn) := B is called a process abstraction.

By substituting value expressions E1,...,En for x1,...,xn in B, a specific instance of the behaviour expression B is obtained which is denoted by p(E1,...,En), and which is called a process instantiation. Such a process instantiation is itself a behaviour expression, which may be used as a building block in the construction of another, larger behaviour expression as described above.

1.4 'Atomic' Behaviour Expressions

At the lowest level of the aforementioned essentially recursive procedure to construct behaviour expressions, two 'atomic' building blocks for constructing behaviour expression are found, namely the behaviour expression stop, and a part of a behaviour expression called an action (or event offer) denotation.

The appearance of stop in a behaviour expression, B, signifies that the process described by B terminates at the point where stop is encountered. Of course, B may simply be a behaviour expression which has been conjoined into a larger behaviour expression, say B', so that the appearance of stop in B need not necessarily denote a point where the process described by B' terminates. Whether this is so or not will depend on the operators used to incorporate B into B', as well as on the other behaviour expressions which are incorporated into B'.

An action denotation is used to indicate the potential occurrence of an event. Such a denotation consists of a label, representing an interaction point, followed by zero or more value expressions, each preceded by an exclamation mark symbol. For example :

ip !expA1 !expA2

is an action denotation with label ip representing some interaction point and with expA1 and expA2 being value expressions. The way in which the potential event represented by this action denotation actually occurs is now discussed.

1.5 Process Synchronisation

Suppose the action denotation given above appears in the behaviour expression of a certain process (say A), and that another action denotation :

ip !expB1 !expB2

occurs in the behaviour expression describing another process (say B). If the value of expA1 = the value of expB1 and the value of expA2 = the value of expB2, then a matching (event) offer is said to have occurred at the interaction point labelled by ip. The action denotation appearing in A's behaviour expression signifies that A will wait at this point until a matching offer is made by some other process (ie B in the example at hand). When this occurs, then both processes involved continue to function, each possibly waiting at the next point at which an action denotation appears in its respective behaviour specification.

1.6 Input/Output Representation

At first sight it may appear that an action denotation as described above is simply a mechanism for describing process synchronisation. This is indeed one of its uses, but seen in conjunction with CCS* operators, it may also be used to describe more complex interactions. To describe such interactions the action-prefix operator (denoted by ;) and the choice operator (denoted by []) are now considered.

If a is an action denotation and B is a behaviour expression, then a;B is a behaviour expression which describes a process behaving exactly like the process which B describes, but only after the action denotation a has received a matching offer.

If B is the set of behaviour expressions {B1, B2,...,Bn}, then

B1 [] B2 [] ... [] Bn

is a behaviour expression which is identical to a non- deterministically chosen element of the set B. Note that an alternative way of writing this expression is by using the so-called summation operator (Σ), applied to B to get ΣB.

Suppose now that x is a variable which can assume values over a given domain s = {s1, s2,...,sn}. (in CCS* terminology x is said to have the sort s.)

The behaviour expression ip!s1;[s1/x]B describes a process which waits for a matching offer of value s1 at interaction point ip, and then behaves as described by the behaviour expression B in which all occurrences of x are substituted by s1. (This substitution is expressed by the [s1/x] notation.) Suppose that a process is meant to input a value for x at the interaction point ip, and then proceeds to function as described by the behaviour expression [si/x]B if the value of x turned out to be si (i = 1,...,n). This may be expressed in CCS* as :

ip!s1;[s1/x]B [] ip!s2;[s2/x]B [] ... [] ip!sn;[sn/x]B

This is clearly a rather cumbersome way of saying that a process must input a value for x, substitute this value for all occurrences of x in the behaviour expression B, and then behave as described by B. CCS* allows for a neater, but equivalent notation, namely ip?x:s;B. This is an example of a so-called extended action-prefix expression.

1.7 CCS* Operators

Apart from the action-prefix operator and the choice operator discussed above, CCS* allows for the following other operators to be applied to behaviour expressions : parallel composition (denoted by |), disabling (denoted by [>), restriction (denoted by \), relabelling, and guarding. These are now informally described.

Consider two behaviour expressions B1 and B2. B1 | B2 is a behaviour expression which describes a process where the processes described by B1 and B2 run independently and in parallel to one another. This means that the order in which subexpressions of B1 and B2 are interleaved cannot be stated a priori. However, the alleged independence of the processes described by B1 and B2 mentioned above must be qualified by the fact that they also will synchronise under the following circumstances : if the behaviour of the processes interleave in such a manner that at some point they are both ready to interact with the same event, then it is assumed that the interaction takes place (since this state of affairs implies that a matching event offer has occured). From the point of view of the environment external to these two processes, the interaction is not directly observed, so that the event that occurs is regarded by the environment as a so-called internal event (denoted by i).

B1 [> B2 is a behaviour expression which describes a process which functions exactly like the process described by B1, unless a matching offer occurs which starts B2 off. If this happens, then the process associated with B1 immediately halts (ie is disabled) and the process described by B2 continues to function to its termination.

Suppose B1 is a subexpression in a larger behaviour expression B2. Suppose too that A is a set of labels denoting interaction points where interactions to B1's external environment occur,

but which are not part of B2's external environment. These interaction points and their associated labels are said to be hidden from B2. Such hiding is shown by means of the restriction operator applied to B2 and A thus : B2\A.

Frequently it is necessary to relabel the interaction points labelled within a behaviour expression. Suppose S is the relabelling function. Then B1[S] is the behaviour expression obtained from B1 by relabelling each label as per S.

Finally, suppose E is some boolean expression. The behaviour expression [E] -> B1 means that the associated process stops if E is false, and is described by B1 if E is true.

Parentheses are used to associate behaviour expressions with their appropriate operators, but may be omitted if allowed by a priority ordering defined on operators as follows : (The ordering is from highest to lowest priority) restriction and relabelling, action-prefix, guarding, choice-composition, parallel-composition, and finally disabling.

Hence, for behaviour expressions B1,...,Bn, the following holds : (B1 | B2) [> (B3 [] B4) is equivalent to B1 | B2 [> B3 [] B4.

1.8 LOTOS

The LOTOS syntax is described in [9] in terms of BNF notation. Syntactically and semantically the language constructs closely parallel representations used in CCS*. Some important differences are now described.

The enable operator (denoted by >>) is introduced at a lower priority than disabling. If applied to two behaviour expressions B1 and B2, then B1 >> B2 means that once the process that B1 describes has terminated successfully (which is denoted by the special process 'exit'), then the process that B2 describes will begin.

The equivalent of the CCS* | operator is a || symbol in LOTOS. However, LOTOS also allows for operators which lock out the possibility of synchronisation at selective gates. If these gates are a1,a2,...,an, then the operator |[a1,a2,...,an]| may be used. If synchronisation is excluded at all gates then the operator ||| is allowed. Note that these latter two operators imply non-determinism in the following sense : if two processes are in a potentially synchronising position (ie both prepared to interact through a mutually shared gate with the same event) and the environment offers that event at that gate, then one of the processes (chosen non-deterministically) will interact, and the other will not.

Interaction points in LOTOS are referred to as gates. One of the syntactically admissible forms of a process abstraction in LOTOS appears as follows :

$$p[a1,...,an](x1:t1,...,xm:tm) := B$$

where a1,...,an are gates at which interactions occur, and x1,...,xm are variables occurring in the associated behaviour expression indicated above by B. The variables x1,...,xm have sorts t1,...,tm respectively. A process instantiation for this process is p[a1,...,an](E1,...,Em), where E1,...,Em represent value expressions replacing corresponding occurrences of x1,...,xm in the associated behaviour expression. Hence LOTOS differs slightly from CCS* in its representation of process abstraction and process instantiation, in that the gates are explicitly shown in LOTOS, while the corresponding labels do not appear explicitly in the equivalent CCS* expression.

Note that the parenthesised parts of the above representations are optional so that a process abstraction may also appear as p[a1,...,an] := B, with the corresponding process instantiation being p[a1,...,an]. Process abstractions and process instantiations may also be more complex than those given above, but these forms are not discussed here.

LOTOS requires that all sorts, operations, equations be rigorously 'declared'. Loosely speaking LOTOS may be said to be 'strongly typed', in that the sort of all variables must be declared, together with operations which may be performed on these sorts. These matters are not dealt with in detail here, and while they are fully specified syntactically in the relevant draft proposal [9], the semantic explanation is given in [4]. However, earlier LOTOS proposals [10,11] will provide an informal insight as to how typing proceeds.

Other syntax requirements in LOTOS should be deducible from the example specification in 3. below. Note, for example, the occurrences of so-called local definition expressions, which are in actual fact process abstractions preceded by the keyword 'where'. These process

abstractions define the process instantiations which occur 'locally' in a higher level process abstraction.

Note that comments in LOTOS are contained within the symbols (* and *).

2. A MODEL OF THE *RJE* SYSTEM

The RJE system to be specified is described by a process abstraction with behaviour identifier RTX. RTX consists of a number of communicating processes physically located in a hardware device known as an RPAD. The RPAD operates in under control of a (possibly remote) device called the network administrator (NA), which configures each RPAD in an X.25 network, gathers statistical data, enables and disables ports, etc. For the present purposes, a process within RTX will be posited which deals with most of the messages from the Network Administrator. It will be described by the process abstraction with behaviour identifier NA.

Two other processes within RTX are relevant to the present specification. The first is device manager which interfaces to a so-called X780 device. (Such a device may be any IBM 2770, 2780, 3770, 3780, 3740 or equivalent device emulating the BSC transmission protocol.) This process will be described by a process abstraction with behaviour identifier RDM. The second process is a 3305 transport handler, which is described by a behaviour abstraction with behaviour identifier RTH. (cf. [7] for a description of the 3305 protocol.)

The specification identified as RTX_SPEC describes how the processes NA, RDM and RTH interact with each other, as well as with their external environments. The single gate through which RDM and RTH interact is designated by b in the specification. RDM interacts with RTX's external environment through two gates designated a and na. RTH interacts with RTX's external environment through the single gate designated by c. NA also communicates with RTX's external environment through gate na. These gates and their environments are schematically shown in figure 1. Note that the fact that the NA interacts with other RTX processes in the actual implementation does not materially affect the present specification.

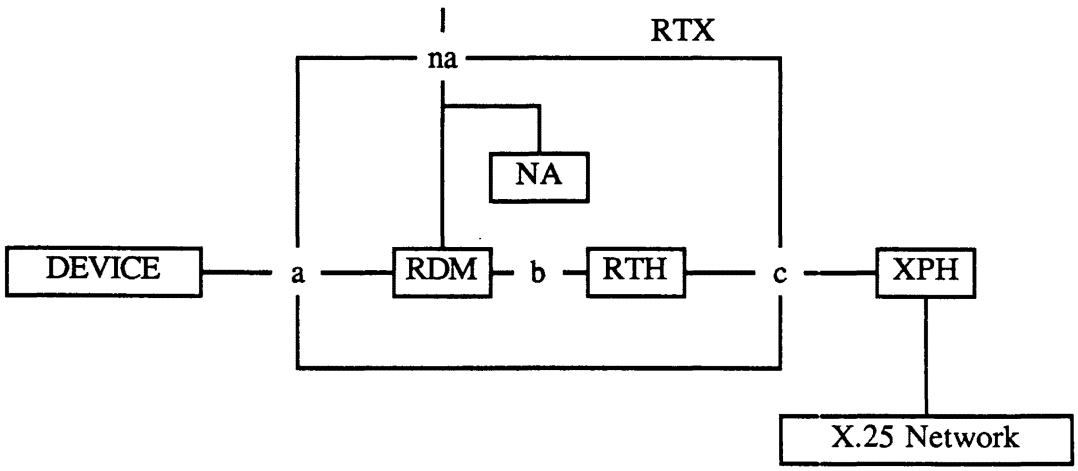

figure 1

RDM interaction through gate a is to the X780 device which communicates with a (remote) peer across an X.25 network using the BSC interface provided by the RPAD. Hence, the specification of the interactions at gate a amounts to a specification of the BSC protocol. Note that from an implementation point of view, a process is needed to assemble the serial incoming message-stream from the device into data structures which RDM accepts and, conversely, to serialise RDM's output to the device. This process is not relevant in the present specification.

RDM interaction through gate na only concerns two Network Administrator commands, namely those which enable and disable communication through gate a. Other interactions through this gate (relating for example to requests for billing or statistical data) do not impinge on the overall communication process, and may essentially be viewed as being handled independently

and in parallel to interactions at other gates. These interactions are dealt with by the process NA.

RTH interaction through gate c is to the so-called XPH process, the X.25 packet handler process which ensures that messages received from RTH are appropriately translated into X.25 packets and sent to the X.25 network, and that X.25 messages received from the network are passed appropriately to RTH. XPH therefore provides the usual network services to the 3305 transport handler, RTH, and is not further specified below.

It should be noted that when the specification mentioned below is implemented, several implementation issues will need to be addressed. In particular, the set of peer X780 devices to which a given device can connect across the X.25 network has to be decided upon, and parameters which characterise the way in which communication is to take place may also have to be provided.

Hence, a device may be configured to connect in a so-called autocall mode, meaning that it always uses the same connection parameters relating to such matters as which application is to be accessed, whether the call will be reverse-charged or not, etc. Alternatively, the device may be configured to send a so-called Call Request Block (CRB) which prescribes what these parameters should be for a specific connection. The temporal ordering of messages between processes in general, and through gate a during connection establishment in particular, will differ according to which configuration option is used. In the specification below, an autocall configuration for the device is presumed.

Furthermore, the way in which a device connects to an application may vary. In some cases, the connection will always be to the same remote port address which offers the application. In this case, the application is designated as a mapped application. On the other hand, a device may be connected to any one of a set of ports offering the desired application. In the latter case the application is referred to as a session application. Which applications are to be characterised as mapped and which are to be considered as session applications is a configuration issue. Part of the function of RTH is to implement a reasonable algorithm to select an initial port offering a desired session application during connection set-up phase, and to re-select a port if a previous connection attempt was unsuccessful. Since RTH is not specified at a detailed level below, the matter of whether the application requested by the autocall device is of type session or mapped is not relevant in the present partial specification of RTX. A full specification of RTH would, however, have to accommodate these alternative application categories.

3. A PARTIAL *LOTOS* SPECIFICATION OF *RTX*

```
(*************************************************************)
(* Two type definitions are given below. Each conform strictly *)
(* to LOTOS syntax.                                          *)
(* The first (process_messages) defines five groups of       *)
(* messages which may occur during interactions.             *)
(* The second (process operations) defines the various       *)
(* messages which may occur, categorised by sort-group to which*)
(* these messages belong.                                    *)
(* Note : LOTOS type syntax also provides for the            *)
(*      semantic definition of operators, using the          *)
(*      type definition for equations. Hence one could       *)
(*      define operations such as PUSH and POP on various    *)
(*      sort categories. This is part of ACT ONE. However,   *)
(*      no such definitions have been attempted here,        *)
(*      since such operations were not required for the      *)
(*      present specification.                               *)
(*************************************************************)
type process_messages
  is sorts dev_msg,
       rth_msg,
       xph_msg,
       na_msg,
       dev_int_msg
 endtype
```

```
type process_operations
is opns
   enq,     eot,
   ack0     : -> dev_msg

   connect_request,
   connect_confirm,
   disc_ind,
   disc_req   : -> rth_msg

   clear_ind,
   clear_req  : -> xph_msg

   bill_req,
   bill_respond,
   stats_req,
   stats_respond,
   disable,
   enable     : -> na_msg

   finish_enq,
   finish_eot,
   finish_error : -> dev_int_msg
endtype

(*************************************************************)
(* Abstract : RTX                                            *)
(* This process allows for the interleaving of two major     *)
(* subprocesses. The first (NA) deals with all routine messages*)
(* from the network administrator. The second reponds        *)
(* appropriately to 'stray' messages at gates a and c,       *)
(* recursively returning to process RTX. However, it is also *)
(* prepared to engage in an enable or disable event at gate na,*)
(* thereby transforming to an corresponding enabled or disabled*)
(* state.                                                    *)
(*************************************************************)
process RTX[a,c,na]
    :=
    NA[na]
    |||
    ( ((XPH_EVENT[c] [] DEV_EVENT[a]) >> RTX[a,c,na])
     []
     (na!disable; RTX_DISABLED[a,c,na])
     []
     (na!enable; RTX_ENABLED[a,c,na])
    )
where
(*************************************************************)
(* Abstract : NA                                             *)
(*  This process is partially specified.                     *)
(*  It suggests how responses would be made for billing and  *)
(*  statistics information, requested at the na gate.        *)
(*************************************************************)
process NA[na]
    :=
    ( (na!bill_req; na!bill_repond; exit)
     []
     (na!stats_req; na!stats_respond; exit)
     []
     :
     :
    ) >> NA[na]
 endproc (* NA *)
```

```
(*****************************************************************)
(* Abstract : XPH_EVENT                                          *)
(* This process deals with xph messages which occur before       *)
(* RTX is ready to connect.                                      *)
(*****************************************************************)
 process XPH_EVENT[c]
    :=
    c?xph:xph_msg;
       ([xph = clear_ind] -> exit
        []
        [xph <> clear_ind] -> c!clear_req; exit
       )
 endproc (* XPH_EVENT *)

(*****************************************************************)
(* Abstract : DEV_EVENT                                          *)
(* This process deals with device messages which occur before    *)
(* RTX is ready to connect.                                      *)
(*****************************************************************)
process DEV_EVENT[a])
    :=
    a?dev:dev_msg;
       ([dev = eot] -> exit
        []
        [dev <> eot] -> a!eot; exit
       )
 endproc (* DEV_EVENT *)

(*****************************************************************)
(* Abstract : RTX_DISABLED                                       *)
(* When RTX has been disabled, it must receive an enable         *)
(* message from the gate na before any connection may be         *)
(* established. Messages from gates a and c which arrive         *)
(* before this point are dealt with appropriately.               *)
(*****************************************************************)
process RTX_DISABLED[a,c,na]
    :=
     (na!enable; RTX_ENABLED[a,c,na])
     []
     (XPH_EVENT[c] [] DEV_EVENT[a]) >> RTX_DISABLED[a,c,na]
endproc (* RTX_DISABLED *)

(*****************************************************************)
(* Abstract : RTX_ENABLED                                        *)
(* Once RTX has been enabled its future description is given     *)
(* by the processes RDM and RTH functioning in parallel and      *)
(* sychronising with each other at gate b (which is hidden       *)
(* from the envirnoment of RTX). However, a disable event        *)
(* at gate na interrupts these processes, and places RTX back    *)
(* into a disabled state.                                        *)
(*****************************************************************)
process RTX_ENABLED[a,c,na]
    :=
    (RDM[a,b,na] |[b]| RTH[b,c])\[b]
    [>    (na!disable; RTX_DISABLED[a,c,na])
where
(*****************************************************************)
(* Abstract : RDM                                                *)
(*  RDM applies the choice operator to the two processes         *)
(*  RDM_CALLING (which deals with calls issued at gate a)        *)
(*  and RDM_CALLED (which deals with a call to gate a            *)
(*  from gate b)                                                 *)
(*****************************************************************)
```

```
process RDM[a,b,na]
   :=
   RDM_CALLING[a,b,na] [] RDM_CALLED[a,b,na]

where
(*************************************************************)
(* Abstract : RDM_CALLING                                    *)
(*  Here the first message at gate a is rejected (eot        *)
(*  at gate a if appropriate), and control is passed back to *)
(*  RDM, unless an enq is issued at gate a.                  *)
(*  In the latter case, a connect_request command is issued  *)
(*  at gate b, and CONNECT is invoked.                       *)
(*************************************************************)
process RDM_CALLING[a,b,na]
    :=
    a?dev:dev_msg;
    (
     [dev = eot] -> RDM[a,b,na]
     []
     [dev = enq] -> b!connect_request;CONNECT[a,b]
     []
     [dev <> enq and dev <> eot] -> a!eot; RDM[a,b,na]
    )
where
(*************************************************************)
(* Abstract : CONNECT                                        *)
(*  This process consists of two subprocesses which          *)
(*  synchronise through an internal gate (hidden from the    *)
(*  environment) called int. The first subprocess is called  *)
(*  DEV_MSG and is described below. The second subprocess    *)
(*  may either interact with an event at gate b and some     *)
(*  time later synchronise with an internal message from     *)
(*  DEV_MSG sent through gate int, its subsequent behaviour  *)
(*  then being described by CONNECT_RESPOND;                 *)
(*  However, the second subprocess might (alternatively)     *)
(*  be informed via gate int (before interaction through     *)
(*  gate b) that an error condition has occured. It then     *)
(*  rejects the next message from RTH (passed through gate   *)
(*  b) and returns to RDM.                                   *)
(*************************************************************)
process CONNECT[a,b]
    :=
    ( DEV_MSG[a,int]      |[int]|
     ( (b?rth:rth_msg;
        int?fin:dev_int_msg;
        CONNECT_RESPOND[a,b](rth,fin)
       )
       []
       (int!finish_error;
        b?rth:rth_msg;
        ([rth <> disc_ind] -> b!disc_req; RDM[a,b,na]\[na]
         []
         [rth = disc_ind] -> RDM[a,b,na]\[na]
        )
       )
     )
    )\[int]
```

```
where
(*************************************************************)
(* Abstract : DEV_MSG                                        *)
(*  This process describes the generation of an arbitrary    *)
(*  string of enq and eot messages of arbitrary length.      *)
(*  The string describes the sequence of interactions at     *)
(*  gate a which occur before synchronisation at gate int    *)
(*  becomes possible.                                        *)
(*  If the string length is 0, or if the string ends in an   *)
(*  enq, then a finish_enq message is used to synchronise at *)
(*  gate int.                                                *)
(*  If the string ends in an eot, then a finish_eot message  *)
(*  is used to synchronise at gate int.                      *)
(*  If, at any point, a message other than eot or enq is     *)
(*  issued at gate a, then the string generation process     *)
(*  stops and a message finish_error is used to synchronise  *)
(*  at gate int.                                             *)
(*  Note that a subprocess DEV_MSG1 accounts for most of the *)
(*  description of DEV_MSG.                                  *)
(*  Note also that after interaction at gate int, this       *)
(*  process deadlocks.                                       *)
(*************************************************************)
process DEV_MSG[a,int]
    :=
    (int!finish_enq;stop)
    []
    DEV_MSG1[a,int]

where
 process DEV_MSG1[a,int]
    :=
    a?dev:dev_msg;
    ([dev = eot] -> (DEV_MSG1[a,int]
                     []
                     (int!finish_eot;stop) )
    []
    [dev = enq] -> (DEV_MSG1[a,int]
                    []
                    (int!finish_enq;stop) )
    []
    [dev <> eot and dev <> enq] -> (a!eot;int!finish_error;stop)
  endproc (* DEV_MSG1 *)
endproc (* DEV_MSG *)

(*************************************************************)
(* Abstract : CONNECT_RESPOND                                *)
(* This parameterised process must be invoked with specific  *)
(* values for rth and fin of the specified sort. If fin has  *)
(* value finish_enq, and rth has value connect_confirm then  *)
(* an ack0 is issued at gate a and the process is further    *)
(* described by DATA_TX. In all other cases, appropriate     *)
(* disconnection action is taken before returning to RDM.    *)
(*************************************************************)
process CONNECT_RESPOND[a,b](rth:rth_msg,fin:dev_int_msg)
    :=
    ([fin = finish_enq] ->
     ([rth = connect_confirm] -> a!ack0;DATA_TX[a,b]
      []
      [rth = disc_ind] -> a!eot;RDM[a,b,na]\[na]
      []
      [rth <> connect_confirm and rth <> disc_ind] ->
                ((a!eot;exit) ||| (b!disc_req;exit))>> RDM[a,b,na]\[na]
     )
```

```
     )
     []
     ([fin = finish_eot or fin = finish_error]
      ([rth <> disc_ind] -> b!disc_req; RDM[a,b,na]\[na]
      []
      [rth = disc_ind] -> RDM[a,b,na]\[na]
      )
     )
   )
where
(*************************************************************)
(* Abstract : DATA_TX                                        *)
(*  This process is not specified.                           *)
(*  It should describe the interactions which occur during   *)
(*  the data transfer phase once a connection has been       *)
(*  established.                                             *)
(*************************************************************)
process DATA_TX[a,b]
        :=
        ..........
endproc (* DATA_TX *)
endproc (* CONNECT_RESPOND *)
endproc (* CONNECT *)
endproc (* RDM_CALLING *)

(*************************************************************)
(* Abstract : RDM_CALLED                                     *)
(*  This process is not specified.                           *)
(*  It should describe the interactions which occur in order *)
(*  to deal with an incoming call to the device.             *)
(*************************************************************)
process RDM_CALLED[a,b,na]
   :=
    ..........
endproc (* RDM_CALLED *)
endproc (* RDM *)

(*************************************************************)
(* Abstract : RTH                                            *)
(*  This process is not specified.                           *)
(*  It should describe the interactions which occur at gates *)
(*  b and c - ie the inputs and outputs to the transport     *)
(*  handler.                                                 *)
(*  Note that any action denotation of the form b!rth        *)
(*  which occurs in this process should have a counterpart   *)
(*  action denotation of the form b?rth:rth_msg, or b!rth    *)
(*  in the above RDM processes.                              *)
(*************************************************************)
process RTH[b,c]
   :=   ..........
endproc (* RTH *)
endproc (* RTH *)
endproc (* RTX_ENABLED *)
endproc (* RTX *)
endspec
```

4. VERIFICATION ISSUES

Verification is generally understood to mean the procedure of proving that some predicate (called a post-condition) will hold at the end of a program given that some other predicate (called a pre-condition) holds at the start. Indeed, if this proof can be carried out successfully, then an enunciation of the pre-condition and post-condition, together with a qualifying statement about conditions under which the program terminates, can be regarded as a correct specification of the program. These matters represent some of the most challenging aspects of Computer Science, and have been intensively studied, resulting in several systems which automate the proof procedure to prove sequential programs correct. (cf. [13,17]) The proof procedure becomes considerably more difficult in the case of concurrent programming, but studies of how verification may take place, and even be automated, in these cases have proceeded apace. (cf. [14,1])

However, a specification of a system in LOTOS is clearly different from a specification in the aforementioned sense, even if at some philosophical level the two types of specification are equivalent. (There seems, for example, to be some underlying commonality between the Modal Logic described by Manna and Pnueli [3] and the semantics of a LOTOS specification.) A LOTOS specification is in fact a specification of the temporal ordering of events at gates within a system, together with a description of what these events are. If a system is implemented in such a way that it exhibits the same behaviour towards its external environment (ie at its gates) as is prescribed by a LOTOS specification of the system, then the implementation may be deemed to be correct. Mechanisms for proving correctness at this level is not what is meant by verification in the present context. Rather, what is sought is a means of verifying that a LOTOS specification itself actually specifies what was intended. Indeed, the terminology used by Hoare [8] is perhaps somewhat clearer. He distinguishes between the definition of a system, (for example a CSP or LOTOS definition) the specification of the defintion (in terms of pre- and post- conditions) and the verification that the definition conforms to the specification. Often the specification is in terms of a statement about the nature of the so-called derivations (Hoare calls them traces) which characterise the system.

A system of inference rules are proposed in CCS* based on so-called action predicates. This provides an approach to identifying and generating derivations of a system defined in LOTOS. An example of an action predicate is the following statement :

> *The process described by B1 (a behaviour expression) interacts at an interaction point labelled g by virtue of a matching offer g!v1,...,vn arising at the interaction point, and then subsequently behaves as prescribed by the behaviour expression B2.*

This action predicate may clearly be true or false, and is abbreviated to the notation :

B1 -gv1...vn-> B2

Note that v1,...,vn are values (eg 8, 10, 'string' etc.) and not value expressions (eg 4*2, x+5, etc.). gv1....vn is called an experiment .

An example of a CCS* inference rule, where a denotes some experiment and B1, B2 and B2' are behaviour expressions, is the following :

$$\frac{\text{B2 -a-> B2'}}{\text{B1 [> B2 -a-> B2'}}$$

This states that if the action predicate B2 -a-> B2' is true, then the action predicate B1 [> B2 -a-> B2' is true. The inference rule in effect defines the semantics of the interrupt operator, [>. As an example of how this inference rule may be used, consider the behaviour abstraction RDM_ENABLED[a,b,na] given in the LOTOS specification in 3. above, where the interrupt operator is utilised. Appropriate translation of this LOTOS specification to CCS* (the translation procedure is described in [9]) will show that RDM_ENABLED describes a process which is always prepared to interact at the na gate with the event 'disable'. Should such an interaction be offered, the specification states that RDM_ENABLE will henceforth be described by the

behaviour abstraction RDM_DISABLED, ie (using the notation [[B]] to designate the translation of the LOTOS expression B to its corresponding CCS* expression) :

[[RDM_ENABLED]] -na!disable-> [[RDM_DISABLED]]

since :

[[na!disable;RDM_DISABLED]] -na!disable-> [[RDM_DISABLED]]

This verification procedure may be forward-propagated by submitting another experiment to [[RDM_DISABLED]], deriving the resulting behaviour expression, and using the inference rules to draw further conclusions about what holds by virtue of the sequence of experiments. The sequence of experiments is referred to as a derivation. Hence, by examining a variety of derivations one may investigate whether they lead to legitimate (ie expected) behaviour expressions or not.

Because all concepts and inference rules in CCS* have been formally defined and because there is a formal relationship between CCS* and LOTOS, there appears to be a firm basis for formal verification of systems defined in LOTOS. Such verification would seek to establish that the set of derivations for a given definition conforms to expectations.

Note, however, that these are basically reachability properties which have also been studied from the finite state-machine point of view [18]. LOTOS does not currently provide for the expression and verification of fairness and liveliness properties (expressible in modal logic [3]), but these issues are being researched [16]. However, LOTOS does provide for the expression of non-determinism, a theme not emphasised in this paper. Verification of non-deterministic systems defined in LOTOS would involve examining not only the set of possible derivations, but also the set of those derivations which could also be refused by the system, by virtue of some non-deterministically chosen action.

5. CONCLUSION

It is interesting to note that, to date, LOTOS has been primarily used as a specification mechanism. (cf. [2] for a specification of the OSI Transport Service, and [4] for a specification of a Presentation Service.) These studies demonstrate the language's expressive power, its modularity and conciseness, all of which properties were specifically aimed for by LOTOS' designers.

In [2], mention is made of a project at Twente University to develop a number of LOTOS tools, including a syntax checker, single-step simulator, trace checker, specification integrator, test generator and possibly a specification compiler. At Pretoria University a syntax checker has been written [19] and progress is being made towards a trace generator.

It is to be anticipated that the future availability of such tools will stimulate a shift in the research focus from specification to verification issues, in which the formal well-defined character of LOTOS will play a central role.

REFERENCES

1. Babich A. F., .br; Proving Total Correctness of Parallel Programs. *IEEE Transactions on Software Engineering*, **SE-5**, No. 6, November 1979, pp558-574.
2. Brinksma E. and Karjoth G., A specification of the OSI transport service in LOTOS. *Proceedings of the IFIP WG 6.1 Fourth International Workshop on Protocol Specification, Testing, and Verification*, (Editors : Yemini Y., Strom R., and Yemini S.) North-Holland (1985).
3. Boyer R. S. and Moore J. S. (Editors), *The Correctness Problem In Computer Science*. Academic Press (1981) .
4. Carchiolo, V., Le Moli G., Palzzo S. and Pappalardo G. Modelling and Specifying a Presentation Protocol by Temporal Ordering. *Proceedings of the IFIP WG 6.1 Fourth International Workshop on Protocol Specification, Testing, and Verification*, (Editors : Yemini Y., Strom R., and Yemini S.) North- Holland (1985).
5. Carey C. W., Hattingh C., Kourie D. G., Van den Heever R. J. and Verkroost R. F, .br; The Development of

an RJE / X.25 PAD : A Case Study., Quæstiones Informaticæ, **4**, No 3, 1986.
6. Ehrig H., Fey W. and Hansen H., *ACT ONE : An Algebraic Specification Language with two levels of semantics*, Bericht-Nr. **83-03**, Technical University of Berlin (1983)
7. GTE TELENET, *A Packet Assembly / Disassembly Protocol Specification for Binary Synchronous Communications* (BPAD/3305), April 1980.
8. Hoare C.A.R., *Communicating Sequential Processes*, Prentica-Hall (1985).
9. *ISO* 8807, Information Processing Systems - Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, March 1985
10. *ISO* TC97/SC16 WG 1 N299, Definition of the Temporal Ordering Specification Language LOTOS, May 1984.
11. *ISO* TC97/SC16 WG 1 N157, Draft Tutorial Document. Temporal Ordering Specification Language, January 1984.
12. *ISO* TC97/SC 21 N 938, Provisional LOTOS tutorial, November 1985.
13. Johnson S. D., *A Computer System for Checking Proofs*, UMI Research Press, Ann Arbor, Michigan 1982.
14. Lubechevsky B. D., An Approach to Automating the Verification of Compact Parallel Coordination Programs. (I). *Acta Informatica*, **21**, 1984, pp 125-169.
15. Milner R., *A Calculus of Communicating Systems*, Springer-Verlag. (1980) .
16. Parrow J. and Gustavsson R., Modelling Distributed Systems in an Extension of CCS with Infinite Experiments and Temporal Logic. *Proceedings of the IFIP WG 6.1 Fourth International Workshop on Protocol Specification, Testing, and Verification*, (Editors : Yemini Y., Strom R., and Yemini S.) North-Holland (1985).
17. Polak W., Compiler Specification and Verification, *Lecture Notes in Computer Science*, No 124, Springer-Verlag, 1981.
18. Rudin H., An Introduction to Automated Protocol Validation, *Proceedings of the Joint CSSA / IFIP Conference on Data Communications*, Johannesburg, September 1982.
19. Van der Vegte L., A LOTOS syntax checker, Paper presented at the 1986 South African Computer Science Lecturers' Association.