UNIVERSITY OF BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

Course Code and Title: CEF 424 Software Verification and Validation

Research on LOTOS Formal method. And Case Study of LOTOS

Group members	Matricule
NGECHOP SIMPLICE	FE18A094
TATUH ALEMANJOH FORMIN	FE18A068
CHI CLAUDETTE MAH	FE18A018
OMBUCHUM MICAH ANJI	FE18A059
RAMSES TIBAH NJASAP	FE18A061

Course Instructor: **Mme Aline Tsague**

Contents

LOTOS (Language of Temporal Ordering Specifications)

ABTRACT

LOTOS is a specification language that has been specifically developed for the formal description of the OSI (Open systems Interconnection) architecture, although it is applicable to distributed, concurrent systems in general. In LOTOS a system is seen as a set of processes which interact and exchange data with each other and with their environment. LOTOS is an ISO international standard

INTRODUCTION

LOTOS is a formal specification language to describe communication protocols and distributed systems. It has been standardized by ISO/IEC in 1989 [ISO89]. The design of LOTOS was motivated by the need for a language with a high abstraction level and strong mathematical bases that would allow complex systems to be described precisely and unambiguously, then analyzed using formal methods supported by appropriate software tools.

LOTOS features two clearly separated parts:

- **The data part of LOTOS**, intended to describe data structures, is based on the theory of abstract data types and algebraic specifications (especially the ActOne language defined by Ehrig and Mahr). In this approach, data structures are described by LOTOS *sorts*, which represent value domains, and LOTOS *operations*, which are mathematical functions defined on these domains. The meaning of operations is defined by algebraic *equations*. Value expressions are strongly-typed algebraic terms built from variables and operations. Sorts, operations, and equations are grouped in modules called *types*, which can be combined together using importation (with multiple inheritance), renaming, parametrization, and actualization. The underlying semantics is that of initial algebras**.**

- **The control part of LOTOS** is meant to describe the behaviour of concurrent processes that execute simultaneously, synchronize, and communicate using message-passing rendezvous. LOTOS is based on the process algebra approach for concurrency, and combines the best features of Milner's CCS and Hoare's CSP process calculi. It relies on a small set of basic operators, which express primitive concepts such as sequential composition, non-deterministic choice, guard, parallel composition, rendezvous, etc. These operators are used to build *behavior expressions*, which are algebraic terms that describe the behavior of concurrent systems, complex behaviors being obtained by combining simpler ones. The communication ports for rendezvous are called *gates*

The Nature of LOTOS

- LOTOS is a *specification language*, with a *formal* basis

- LOTOS is widely supported by academic institutions world-wide, and has significant industrial support

- LOTOS was designed for the specification of OSI systems, but is equally suitable for the specification of concurrent or distributed systems generally

- The formal basis of LOTOS ensures precision and analyzability; however, this is bought at the price of needing special training in the language, and more brain- work to understand LOTOS specifications

Application of LOTOS

- LOTOS has been used mainly on OSI, but this is not an intrinsic limitation

- LOTOS is applicable to sequential, concurrent, and distributed systems generally

- The following LOTOS specifications of OSI have been written (CL = *Connection-Less*, CO = *Connection Oriented*)

– **Application Layer** : FTAM (*File Transfer and Manipulation*)and ACSE (*Association Control Service*

Elements)

– **Presentation Layer** : some work (CO)

– **Session Layer** : complete (CO)

– **Transport Layer** : complete (CO)

– **Network Layer** : Service (CL, CO) and Protocol (CL)

– **Data Link Layer** : Service (CL, CO)

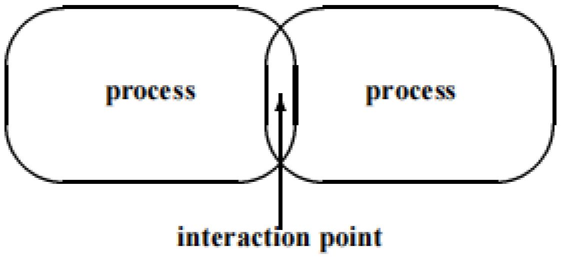
LOTOS is being used on new OSI developments (e.g. ODP (*Open Distributed Processing*) and OSI Management)

Basics of Process Algebra

1. Processes

A *process* is a component of a specification; it is the abstraction of an activity in an implementation, and communicates with other processes

A process is considered to be a black-box at some level of abstraction; only the external behavior of a process is considered.
Processes share a communication mechanism called an *interaction point*, as shown in the figure below; this is the abstraction of an interface in an implementation



`

In LOTOS, the specification concept of interaction point corresponds to the language concept of *event gate* (or just *gate*). Processes are specified by giving a *behavior expression* which defines their externally visible behavior in terms of the permissible sequences of *events* in which they may participate
In these Report, processes are given upper-case names such as:

ACTIVATE_ ALARM DATA_ TRANSFER

2. Events

An *event* represents a *Synchronization* between processes

An *event offer* represents the ability of one process to participate in an event; events external to a process are resolved in conjunction with the *environment* of the process

An event normally requires the participation of two or more processes; three kinds of event are possible: *pure synchronization, value establishment and value negotiation*

BASICS OF PROCESS ALGEBRA

–***Pure synchronization*** - no values are exchanged between the processes
– ***Value establishment*** - one or more processes supply a specific value which lies in the set acceptable to the other process(es)
– ***Value negotiation*** - two or more processes agree on a set of values.

Events are considered to be *atomic*, i.e. at the level of abstraction of the specification, the synchronization of the processes and the associated information are established at the same time; in an implementation, an event may of course correspond to a sequence of elementary steps, the set of events in which a process can potentially participate is called the ***alphabet* of the process**.
The set of events in which a process can immediately participate at any point in the unfolding of its behavior is called the ***initials* of the process**

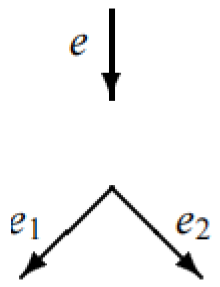
A behavior expression is evaluated in an environment which offers it events to synchronize on; the
Environment may be another behavior expression which it is combined with, or may be something external.

Temporal Ordering

Specifications in LOTOS give the *temporal ordering* of events, i.e. the *relative* ordering of events in time
LOTOS abstracts away from absolute timing considerations, e.g. that an event must occur at a specific time or after a specific period

LOTOS specifications use operators which combine behavior expressions to yield more complex behaviour
expressions; these operators obey well-defined laws which enable any specification to be interpreted unambiguously

The data typing part of LOTOS is given semantics using the theory of algebras; LOTOS operators for combining behaviour expressions can be shown to satisfy algebraic laws which characterize them a convenient diagrammatic notation for thinking about behaviour in languages like
LOTOS is the *synchronization tree*, as shown in Figure 1.1



Occurrence of an event *e*
Choice between events *e*1 or *e*2

Synchronization Tree

Background Needed

1. Differences from Programming

Surprisingly little technical knowledge is needed to achieve some familiarity with LOTOS a numerate back
ground, with a small amount of basic mathematics and computing, is probably suffificient
a knowledge of programming languages will also help, but beware that LOTOS being a *specification* language,
treats some concepts in a fundamentally different way

-*variables* in LOTOS are mathematical variables: they are simply names which happen to be bound to a particular value in particular, one does not assign a value to a LOTOS variable as one might assign a value to
a store location; in LOTOS one cannot write, for example:
$x := x + 1$
one must in effect define a new variable (*x*
, say) which is bound to the value of the old *x* plus 1
-*recursion* in programming languages is usually thought of in terms of a stack which records return addresses;

a well-behaved program unwinds this stack before continuing in LOTOS, however, one may exit from within a recursive piece of behaviour to do something completely different; this is because recursion in LOTOS is equivalent to replacing a recursive call with a copy of the text defining the behaviour
-*Efficiency* is an important consideration in programming, but is not appropriate in LOTOS; one should not think of 'implementing' LOTOS literally, and must therefore not be concerned about matters of
effciency
-A good specification clearly defines *what* and not *how*; this contrasts sharply with programming, where one frequently makes design decisions to optimise the use of store, processor, etc.
-*over-specification* must be carefully avoided in LOTOS; one must always concentrate on external requirements
and black-box behaviour, not structures or algorithms to implement these for example, one should specify a queue in terms of the operations on it (enqueue, dequeue, head, etc.) rather than in terms of linked-lists, algorithms for skipping down the queue, etc.

2. Syntax and Semantics

Syntax is usually expressed using a *grammar*; this consists of a set of rules in which *non-terminal symbols* (variables of the grammar) are defined in terms of **non-terminal symbols and *terminal symbols*** (constants of the grammar)
-The definition of LOTOS syntax includes the following constructs:
– Recursion is expressed by, for example:

echo = "hello" echo

which defines **echo** to be **hello** (a terminal symbol) followed by **echo**; in other words, **echo** is an unbounded repetition of **hello**

– a choice is expressed by, for example:
mood = happy | sad
– an optional part of a rule is expressed by, for example:
sentence = subject verb [object]
which says that zero or one occurrences of **object** are allowed
-a set of *derivation rules* is a system of logic for dealing with inferences; such a system is used to express
the semantics of LOTOS operators

-a derivation system has a set of *axioms* (logical formulae from which others are derived), and a set of *assertions* (logical formulae which may or may not be derivable)
-a logical formula is derived by applying *inference rules* of the form:

$$\frac{P_1;\ :::\ ;\ P_n}{Q}$$

meaning that, given *P*1 up to *Pn*, *Q* may be derived
the shorthand notation:

$\vdash P$ means that *P* can be derived from the axioms and the inference rules

Basic Process Expressions

1. Sequence

-the *sequential composition operator* ';' is used to prefix a behavior expression with an event called an *action prefix*; for example:
Button_pressed; RING_ BELL

-action prefixes associate to the right; for example:
connect_ request; connect_ confirm; data; DISCONNECT
means:
connect_ request; (connect_ confirm; (data; DISCONNECT))

the alphabet of this behavior expression is:

{connect_ request; connect_ confirm; data}
plus whatever events there are in DISCONNECT

2. Choice

-The *choice operator* '[]' is used when alternative behaviours are allowed; for example:
(lift_arrived; ENTER) [] (lift_broken; USE_ STAIRS)

-choice is associative (as one would expect); for example:
EAT [] DRINK [] BE_ MERRY

means the same as:
(EAT [] DRINK) [] BE_ MERRY
and:
EAT [] (DRINK [] BE_ MERRY)
but notice that one cannot eat, drink, *and* be merry!
-choice is also commutative (as one would again expect); for example:
PAY_ A_ FINE [] TAKE_ A_ CHANCE
means the same as:
TAKE_ A_ CHANCE [] PAY_ A_ FINE
the initials of the behaviour expression:
(lunch_ bell; EAT) [] (boss_ here; WORK) [] (fire; PANIC)
are:

{lunch_ bell; boss_here; fire}

-In the straightforward case, the choice between alternatives is resolved by the environment of the process;
in the example above, if the environment offers only **fire** then the process will PANIC

3. Parallelism

- **Interleaving**

the *interleaving parallel composition operator* '|||' is used to allow behaviors to unfold completely independently in parallel; the events from each behavior expression are interleaved
for example:
(data_ in; data_ out; BUFFER) ||| (read; mark; digest; BOOK)
includes the following behaviors:

data_in; read; mark; data_out; digest:::

read; mark; digest; data_in:::

-'|||' is associative and commutative (as one would expect, in order to capture the intuitive concept of running in parallel)

- **Synchronization**

-the *synchronizing parallel composition operator* '||' is used where there are events that need to be synchronized; in this case the events (strictly, gates) which occur in *either* of the behavior expressions are obliged
to synchronize, for example:

```
    (bang; start; finish; ATHLETE)
    ||
    (bang; start; finish; STARTER)
```

may engage in the following sequence of events:

bang; start; finish; :::

-if only certain events are to be synchronized, the '|[*:::*]|' form of the operator is used, with the events (strictly, gates) named between '[' and ']'; for example:

```
        (off_ hook; dial; answer; speak; on_ hook; TELEPHONE)
    |[dial]|
        (find_ number; dial; engage_ brain; speak; CALL)
```

will synchronize only on the **dial** event, and will allow **speak** in the second behavior expression before
answer and after **on_ hook** in the first behavior expression (a possibly realistic situation)
'

-|[*:::*]|' with an explicit empty list, or with no gates which occur in the two behaviour expressions, is
equivalent to '|||'
-'|[*:::*]|' with an explicit list which is in fact the union of the two sets of gates is equivalent to '||'
-like '|||', '||' or '|[*:::*]|' is associative and commutative
parallel composition can be used to express independent constraints; for example, the constraints 'breakfast must precede lunch' and 'lunch must precede dinner' can be expressed by:

```
    (breakfast; lunch; AM) |[lunch]|(lunch; dinner; PM)
```

This allows a separation of concerns, not to mention eating habits!

4. Termination

- **Inaction and Success**

-the simplest behavior expression is '**stop**', which offers no events and therefore does nothing; it is used to represent *inaction* or *deadlock*

Successful termination of a behavior expression is represented by '**exit**', which offers a special event ¡ and
then behaves as **stop**

- is part of the underlying mathematical model of LOTOS and is *not* an event which can be explicitly offered;

-may also be an implied initial of a behavior expression

successful termination of a sequence (using the ';' operator) depends on whether the right-most behavior expression is **stop** or **exit**
for example:

clock_ in; clock_ out; **exit**

may terminate successfully, but:

born; died; **stop**
can not.
-successful termination of a choice (using the '[]' operator) depends on the successful termination of *one* of the behavior expressions; for example:

(fail; **stop**) [] (catastrophe; **stop**)

can not terminate successfully, but:

(lift_off; **exit**) [] (armageddon; **stop**)
may succeed

-successful termination of a parallel composition (using the '|||', '||', or '|[*:::*]|' operators) depends on the successful termination of *all* of the behaviour expressions; for all these operators, the special termination
event ¡ is always synchronised — even in the case of '|||'
for example:

(finished; **exit**) ||| (done; **exit**)

may terminate successfully

- **Enabling**

-As a generalisation of the ';' operator, which is used for the sequential -composition of an *event* and a *behaviour expression*, the '>>' operator (pronounced *enables*) combines two *behaviour expressions* in sequence

for example, if process SHOP is:

visit_shop; buy_ food; come_ home; **exit**

and process EAT is:

cook_ food; eat_ food; **stop**

then the process DINE, defined by:

SHOP >> EAT
denotes the possible sequence of events:

visit_ shop; buy_ food; come_ home; cook_ food; eat_food

-exit (i.e. the ¡ event) is absorbed by a following '>>'
if the left-hand behaviour expression does not terminate successfully, the right-hand behaviour expression
will not apply; for example:

(prime_ 5; prime_ 7; (prime_ 9; **stop** [] **exit**)) >> (prime_ 11; **exit**)

can successfully terminate after the sequence of events:

prime_* 5; prime_* 7*; prime_* 11

but will deadlock after the sequence of events:

prime_* 5; prime_* 7*; prime_* 9

-'>>' is associative, but is not of course commutative

- **Disabling**

-A frequent occurrence in specifications is the need to specify behaviour which may be interrupted by something else (e.g. disconnection may terminate data transfer)

-the ‘[>’ operator (pronounced *disabled by*) allows the right-hand behaviour expression to interrupt the left-hand behaviour expression; if this happens, the future behaviour is that of the right-hand behaviour
expression only

-if the left-hand behaviour expression terminates successfully, then the combination also terminates successfully (i.e. disabling is no longer possible); this is to allow for contingencies
for example:

(send_ data; reset_ timer; receive_ acknowledgement; **exit**)
[>
(timer_ expired; sound_ alarm; **stop**)
may terminate successfully if an acknowledgement is received to a message, but may sound an alarm if no acknowledgement is received within some time period
-beware that behaviour may be disabled between its ‘last’ event and **stop** or **exit**
thus, in the example above, **timer_ expired** may occur *before* **reset_ timer** and *after* **receive_ acknowledgement**; this could be realistic in an actual implementation .

5. Non-Determinism

- **Non-Determinism due to Choice**

The events in a process specification are *event offers*; the actual events which happen may be influenced by the environment of the process
for example:

(wake_ up; MAKE_ COFFEE) [] (open_ wine; GET_ DRUNK)

will result in MAKE_ COFFEE or GET_ DRUNK depending on the sobriety of the environment however, identical events may be offered as alternatives: in this case, the environment *cannot* influence which branch is taken; the choice is made non-deterministically
for example:

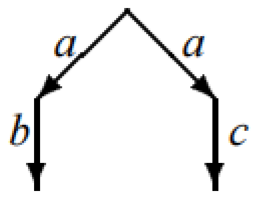
(eat_ out; CHINESE_ MEAL) [] (eat_ out; INDIAN_ MEAL)

will result in CHINESE_ MEAL or INDIAN_ MEAL after the environment has offered **eat_ out**, and cannot be
Influenced

-non-determinism is best understood by considering the synchronization tree for the behavior; for example:

(a; b; **stop**) [] (a; c; **stop**)

has the synchronization tree shown in Figure 2.1



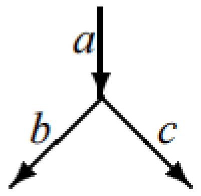
Synchronization Tree with Initial Choice

but:

6. NON-DETERMINISM

a; ((b; **stop**) [] (c; **stop**))

has the synchronization tree shown in Figure 2.2



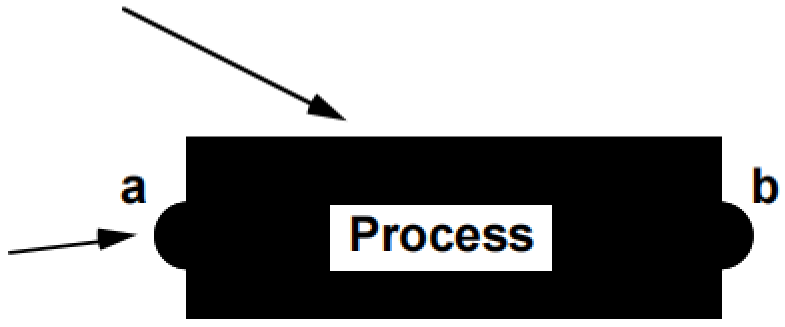
Synchronization Tree with Deferred Choice

-in the first case, the environment offers **a** but has no choice as to what follows: it may be offered **b** or **c**; in the second case, the environment can still decide on **b** or **c** after offering **a.**

- **Basic LOTOS**

  In LOTOS, a distributed, concurrent system is represented by a **process**, possibly consisting of sub-processes.

Process = **black box** able to **interact** with other processes (its environment) and / or to perform internal actions.



Atomic interactions are called **events**
Atomic = instantaneous, one at a time
An event occurs at an **interaction point**, or **gate**
An event may involve the exchange of data
Events imply **process synchronization**
When an event occurs, at least two processes participate in, or experience, that event

Basic LOTOS is a simplified version of the language employing a *finite* alphabet of observable actions. This is so because observable actions in basic LOTOS are identified only by the name of the gate where they are offered, and LOTOS processes can only have a finite number of gates. Three examples of observable actions that we have already met in the previous section are: g ,in2 ,out

The structure of actions will be enriched in *full* LOTOS by allowing the association of data values to gate names, and thus the expression of a possibly infinite alphabet of observable actions.

Basic LOTOS only describes process synchronization, while full LOTOS also describes interprocess value communication.

-Within this proper subset of the language we can appreciate the

expressiveness of all the LOTOS process constructors (operators) without being distracted by interprocess communication

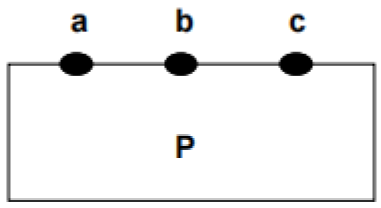
-for basic LOTOS we can give an elegant and, most importantly,

formal presentation of the semantics, without boring the reader with cumbersome notation

-behavioural equivalences are more conveniently introduced at this level.

Process <process_id> <parameter_part> **:=**

<behaviour_expression>
endproc
Ex. : **process** P [a,b,c] := … **endproc**



process_id : an identifier which designates a process, that is a
process name

parameter_part : in Basic LOTOS, a list of interaction points or gates
behaviour_expression : an expression defining the behaviour of the process, i.e.
the allowed orderings of events
Such expressions are built up from more elementary expressions by using LOTOS operators.

- **Behavioral equivalences**

One can describe systems at various levels of abstraction; for example it is possible to describe how they are structured internally in terms of predefined subcomponents, or how they behave from the point of view of a user or of an external observer. In moving within this range of descriptive levels, it is common to distinguish between:
Specifications, which are rather high level descriptions of the desired behavior of the system, e.g. as seen by the user (extensional description);
Implementations, which are more detailed descriptions of how the system works or of how it is constructed starting from simpler components (intensional description).
LOTOS is a specification language which allows the specification of systems at different descriptive levels.

Data types

The representations of values, value expressions and data structures in LOTOS are derived from the specification language for abstract data types (ADT) ACT ONE. The choice of *abstract* data types for LOTOS, as opposed to *concrete* data types, is consistent with the requirement of abstraction from implementation details which has been a guiding principle also in the design of the other component of the language (process definitions). A *concrete* data type implies a description of *how* data values are represented in memory, and *how* some associated procedures operate on them. In other words the data type is defined by explicitly giving its implementation. For example a Pascal queue can be defined as a list of records and a pair of procedures which manipulate it to realize the 'Add' and 'Remove' operations. An *abstract* data type can be seen as the *formal specification* of a class of concrete data types. It does not indicate *how* data values are actually represented and manipulated in memory, but only defines the essential properties of data and operations that any correct implementation (concrete data type) is required to satisfy.

Advantages of LOTOS

1. Although LOTOS is a committee-designed language based on two very different concepts (algebraic data types and process calculi), it achieves a suitable compromise and a fair integration between its various elements. All its language constructs (perhaps with the exception of the choice and par operators on gate lists) derive from concrete needs and are useful in practice.
2. LOTOS is clearly more abstract and higher level than the two other standardized Languages it was competing with (namely, Estelle and SDL), and proved that a specification language could be formal and executable at the same time.

3. The design of LOTOS made it clear that process calculus were not only mathematical notations for studying concurrency theory, but that they could be turned into computer languages used to model real-life systems. LOTOS was indeed the first process calculus in which large specifications of complex systems (e.g., protocols and services of OSI and ISDN networks) were produced.

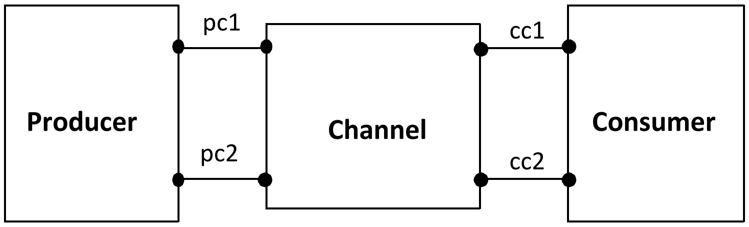
Disadvantages of LOTOS

1. LOTOS failed to gain wide industrial acceptance, mostly due to its so called “steep learning curve".
2. LOTOS is best used by high-level experts rather than average software programmers: this is unfortunately a fatal flaw as far as dissemination (wide spread) is concerned.
3. Despite its status of international standard, LOTOS did not manage to unite the academic community working on process calculi.

CASE STUDY OF LOTOS

The example that we have chosen is the classic problem of a producer and a consumer communicating by means of a channel. In order to explore all LOTOS operators, we will use the same problem with different formulations to suit our illustrative needs.

Formulation 1: A LOTOS specification for the producer-consumer problem with a two-place channel is to be provided. The channel, which handles two types of messages, is FIFO (First In, First Out); it may not lose or reorder messages. The producer must produce exactly two elements and then terminate. Similarly, the consumer must consume both elements and then terminate. The channel synchronizes with the consumer only after it has finished interacting with the producer.



Structure of the Specification

One way to structure the producer-consumer specification is to decompose the problem into three processes, specify each process separately and then compose them to obtain the final solution. At this point, we should not be concerned with the question of how the three processes fit together. Rather, it is more important to understand how each process behaves independently with respect to its own environment.

The Action Prefix Operator

The actin prefix operator, written as a semi-colon; expresses sequential composition of actions. This operator is used to sequentially order actions. For example, *a; B* denotes a behavior where action 'a' must be executed before the behavior expression B. The producer, as a separate entity, can be seen as a black box, which synchronizes with its environment through two synchronization points, Pc1 and Pc2. It performs three actions: it produces two elements and then ***exit***s.

The process ***Producer*** can then be specified as follows:

process ***Producer*** [pc1, pc2] : **exit** := pc1;

pc2; **exit**

endproc

Producer synchronizes with its *environment* through two gates *pc1* and *pc2*, which are formal gates. Actual gates must be specified at instantiation time. The keyword ***exit***, in the process definition, indicates that the process is able to execute an ***exit*** at the end, i.e. to successfully terminate. In LOTOS terminology, we say that this process has *functionality* ***exit***. The functionality of a process can be either ***exit*** if the process is able to successfully terminate or ***no exit*** otherwise. Successful termination means that execution can continue to other processes.

Similarly, the behavior of ***Consumer*** can be written as:

process ***Consumer*** [cc1, cc2] : **exit** := cc1;

cc2; **exit**

endproc

Finally, the behavior of the ***Channel***, which synchronizes on its left with ***Producer*** and then synchronizes on its right with ***Consumer***, is given by the following process:

process ***Channel*** [pc1, pc2, cc1, cc2] : **exit** :=

pc1; pc2; cc1; cc2; **exit** **endproc** At this point we are ready to compose all three processes to obtain

the complete system.

However, this requires the knowledge of some other operators, called *parallel composition operators,* which are suited for exactly that purpose.

Choice Operator

To illustrate the use of the *choice* operator, let us reformulate the problem.

Formulation 2: Modify the specification so that the channel may deliver the first element, produced by the producer, to the consumer before the second element is put in the channel.

To satisfy this new requirement, we need the choice operator [], which denotes the choice between two or more alternative behaviors. In this case, the ***Channel*** process must first synchronize with the ***Producer*** on gate *pc1* and then either synchronize with the ***Producer*** a second time, on gate *pc2*, or synchronize with the ***Consumer***, on gate *cc1*. Once the choice between *pc2* and *cc1* is made, the other alternative is ignored. In other words, if *pc2* is chosen, then the behavior of ***Channel*** becomes *cc1; cc2;* ***exit*** otherwise, the behavior of the channel becomes *pc2; cc2;* ***exit***. The choice operator is commutative and associative. So, *A [] B* is equivalent to *B [] A* and *A [] B [] C* is equivalent to both (*A [] B) [] C* and *A [] (B [] C).*

In this report, the term behavior *A* is *equivalent to* behavior *B* means that the sequences of actions which behavior *A* is capable of offering, according to an outside observer, are the same as those which behavior *B* is capable of offering. For example, *a; **i**; b; **stop*** is equivalent to *a; b; **stop***.

```
process Channel [ pc1, pc2, cc1, cc2] : exit  := pc1;

(

pc2;     cc1;     cc2;     exit

     []

          cc1;     pc2;     cc2;     exit

     )

endproc
```

Note that the choice can be nondeterministic. For example, compare the behaviors of three vending machines:

```
Insert quarter;  get coffee; stop []  insert dime;  get milk; stop (* 1 *)

insert quarter; get_coffee; stop [] insert_quarter; get_milk; stop      (* 2 *)

insert_quarter; (get_coffee; stop [] get_milk; stop)                    (* 3 *)
```

Machine 1 offers the client (the environment) a choice between inserting a quarter and inserting a dime. If the environment offers a quarter, the behavior of the machine evolves to *get_coffee; stop*. If it offers a dime it evolves to *get_milk; stop*.

Machine 2 accepts quarters only, and after synchronization with the environment (i.e., once the client inserts a quarter), the behavior of the machine can evolve to either *get_coffee; stop* or *get_milk; stop* depending on a *nondeterministic choice* (in other words, once the client inserts a quarter the choice to synchronize on either *get_coffee* or *get_milk* would no longer exist).

Machine 3 accepts quarters only as well, but it is more democratic. Once the client inserts a quarter, the behavior of the machine evolves to *(get_coffee; **stop** [] get_milk; **stop**)*, meaning that the client can still choose between coffee and milk. Another way to express nondeterminism in LOTOS is by using the internal action.

Enable Operator

The LOTOS *enable* operator ***>>*** has a similar function as the action prefix operator, which expresses the sequential composition of an action with a behavior expression. The **>>** is used to express the sequential composition of two behavior expressions. For example, if *P1* and *P2* are two processes, *P1 >> P2* is read *P1 enables P*2. Process *P1* must terminate successfully in order for *P2* to be enabled. This is the only condition under which process *P2* is enabled. Execution of an ***exit*** in *P1* results in an action on a special gate δ. The enable causes δ to become an ***i***, and execution to continue with *P2.* For example, a ; b ; **exit** >> c ; **stop**

is equivalent to *a; b;* ***i****; c;* ***stop***, i.e., *a; b; c;* ***stop***, whereas, a ; b ;
stop >> c; **stop**

is equivalent to *a; b;* ***stop***. The expression on the right hand side of the enable operator cannot be executed, because the expression on the left-hand side cannot terminate successfully.

The process ***Channel*** was written using the action prefix and choice operators only. Using the **>>** operator, we can replace (for the sake of illustration) the process ***Channel*** with a process which exhibits an equivalent behavior. The following informal solution results:

```
process Channel [ . . . ] : exit :=
  ( Get first element;
      (     Get second element;  Put first element; exit
          []
           Put first element ; Get second element; exit
   )
  )
  >>  Put second element   endproc
```

As explained previously, the first synchronization must occur at gate *pc1*. After synchronizing on *pc1*, the ***Channel*** offers to synchronize on *pc2* and *cc1* in any order. Finally, the ***channel*** must synchronize on *cc2*. Translated into LOTOS, we have:

```
process Channel [ pc1, pc2, cc1, cc2] : exit := pc1;
    (
       pc2;    cc1;     exit
       []
       cc1;    pc2;     exit
```

```
        )
        >>      cc2;     exit

    endproc
```

Note that both branches of the choice have an ***exit*** as the last action. Therefore, if either alternative reaches the ***exit***, the behavior *cc2;* ***exit*** becomes enabled.

Internal Action

Let us add new constraints to the requirements of the producer-consumer problem.

Formulation 3: Modify the specification so that the channel may lose either or both elements. Losing an element can be modelled by an internal action of the channel. Once an element is put in the channel, it may either be consumed by the consumer or lost as a result of an unexplained internal action of the channel. Note that the internal action ***i*** is not controlled by the environment and therefore, in conjunction with the *[]* operator, it represents a nondeterministic choice. Informally, the process ***Channel*** becomes:

```
        process Channel [ . . . ] : exit  := (
        Get the first element;
           (   Get the second element ;  []      Put the first element;          exit      (* 1 *)

               Get the second element;           Lose the first element;         exit      (* 2 *)
               []
               Put the first element; []         Get the second element;         exit      (* 3 *)

               Lose the first element;           Get the second element;         exit       (* 4  *)
    )
        )
        >>  ( Put the second element; exit  [] Lose the second element; exit )  endproc
```

Note that from an observational point of view, there is no difference between 2 and 4. This is an application of an equivalence law by which *a;* ***i; exit*** *[]* ***i;*** *a;* ***exit*** is equivalent to ***i;*** *a;* ***exit****.*
Therefore, these two alternatives are merged into a single behavior expression. In LOTOS, we have:

```
    process Channel [pc1, pc2, cc1, cc2] : exit  := pc1;  (  pc2 ;  cc1;  exit  [] cc1;
        pc2;  exit  []  i;  pc2;  exit  )

            >>  ( cc2;  exit  []         i;          exit)
    endproc
```

However, after changing the ***Channel*** specification, the ***Consumer*** is no longer correct, because the ***Channel*** is now ready to synchronize first on *cc1*, if the first element is not lost, or on *cc2*, if the first element is lost. Therefore, the consumer must be specified to synchronize on the following sequences of actions: *cc1, cc2* then ***exit***, *cc1* then ***exit***, *cc2* then ***exit***, or simply ***exit*** when both elements are lost. In LOTOS:

```
process Consumer [ cc1, cc2] : exit :=    cc1; (cc2;
    exit [] exit)
    []
    cc2;
        exit     []

    exit
endproc
```

Being able to execute independently of the environment, internal actions provide an additional way of specifying nondeterminism in LOTOS.

coffee; **exit** [] milk; **exit** (*1*)

is a process that is ready to synchronize on both *coffee* and *milk*.

i; coffee; **exit** [] milk; **exit** (*2*)

is a process that is ready to synchronize on *coffee*, but may not be able to synchronize on *milk*. If the environment proposes *milk,* synchronization may be impossible if the process has already decided to execute ***i***; however, if the environment proposes *coffee*, it can be assumed that the internal action will be executed eventually, and synchronization will then occur. Similarly, **i**; coffee; **exit** [] **i**; milk; **exit** (*3*)

is a process that may be unable to synchronize on either *coffee* or *milk*, depending on an internal decision.

An interesting application of this concept is the specification of priorities. (*2*) can be interpreted that *coffee* has priority over *milk*. If, in addition, one wants to specify that *milk* is still possible after *coffee*, this could be written

i; coffee; milk; **exit** [] milk; **exit**

By using more complicated cascades of alternatives with internal actions, one can specify priorities with respect to any predetermined and finite number of events.

There are three parallel composition operators in LOTOS: a basic one, the *selective composition* operator, and two derived ones, the *interleaving* and the *full synchronization* operators.

The *interleaving* operator (|||) is used to express the concept of parallelism between behaviors when no synchronization is required.

(out1; out2; **exit**) **|||** (in1; in2; **exit**)

is equivalent to

```
out1;    (out2; []     in1;       in2;       exit

         in1;          (out2;          in2 ;      exit      []     in2;      out2;    exit)     )
[]
in1;     (in2; []      out1;     out2;      exit

         out1;         (in2;           out2; exit       []     out2;    in2;       exit       )
```

The interleaving operator mimics a divorced couple, where lifestyles are completely independent. Some married couples maintain the engagement to do together certain things, for example breakfasts and movies every Thursday night. When processes must synchronize on common actions, the *selective parallel* operator, denoted by *|[L]|*, is used, where *L* is the list of actions on which synchronization must occur. For example:

```
    a; b; c; exit          (* subprocess 1 *)
|[a]|
    d;  a; c; exit         (* subprocess 2 *)
```

is equivalent to or to
or of course to

d; a; (b; c; **exit** **|||** c; **exit**)

d; a; (b; (c; c; **exit** [] c; c; **exit**) [] c; b; c; **exit**)

d; a; (b; c; c; **exit** [] c; b; c; **exit**)

This is so because both subprocesses execute independently until one of them reaches a common action, at which point it must wait to synchronize with the other subprocess. Once the second subprocess reaches the same point, synchronization is possible (depending on the context in which the process occurs, participation of other processes may be necessary also) and if it occurs both subprocesses proceed to offer their next actions. In this example, the first action of subprocess 1 is *a*. Since action *a* is common to both subprocesses, it must wait for subprocess 2 to reach action *a*, before offering action *b*. On the other hand, action *d* of subprocess 2 is not in the synchronization set, so no synchronization is required. The same is true for action *c* which is offered independently by both subprocesses. Therefore, after synchronization on *a*, both subprocesses continue independently, with all possible interleavings of the remaning actions. Clearly, when *L* is the empty list the selective parallel composition operator becomes the interleaving operator.

A special case is provided by the action δ, which is produced by ***exit****s*. The action δ is always considered to be a common gate, for any parallel composition operator. Therefore, all behaviors composed in parallel must synchronize on their ***exits***. In the case of enable, the action δ is transformed into an internal action, after the synchronization with other ***exit****s* has occurred. For example,

a; **exit** ||| b; c; **exit**

is equivalent to a; b; c; **exit** [] b; (a; c; **exit** [] c; a; **exit**)

Both of these behaviors are ready to synchronize with the environment on any of the following sequences:

a b c δ *b a c* δ *b c a* δ

Let's take another example:

(a; **exit** ||| b; **exit)** >> c; **stop** is
equivalent to a; b; **i**; c; **stop** [] b; a; **i**; c;
stop
after synchronization on δ which has been transformed into ***i*** by the enable operator. The behavior *c; **stop*** is enabled only after the two ***exit**s* synchronize.

There are couples where the two partners are so attached to each other that they always do everything together. If at a given moment a possible behavior of one partner is impossible for the other partner, then the first partner will not exercise that behavior. If no possible common behavior can be found, an impasse (a deadlock) occurs. The *full synchronization* operator, denoted *||*, is used when the processes involved in synchronization must synchronize on every observable action. Clearly, when *L* is the list of all gates, the selective parallel composition operator becomes identical to the full synchronization operator. For example, the behavior: *a; b; c; **exit*** will synchronize with the behavior: *a; b; c; **exit***. Therefore, *a; b; c; **exit** || a; b; c; **exit*** is equivalent to *a; b; c; **exit***. As a second example, the behavior *a; b; **exit** || d; a; c; **exit*** is equivalent to ***stop*** (deadlock!), because the left hand side offers *a* while the right hand side offers *d*. However, a; b; **exit** || (a; c; **exit** [] a; b*;* **exit**) is equivalent to a; **stop** [] a; b; **exit**

in other words it can lead to either deadlock or success, depending on a nondeterministic choice:

if synchronization occurs on the first and second *a*, further synchronization is impossible.

In the presence of a choice, all the alternatives that lead to a deadlock are not considered. This can be expressed by the law: *B [] **stop*** is equivalent to *B*. For example, a; b; **exit**
|| (a; c; **exit** [] c; b; **exit**)
is equivalent to *a; **stop***. The first alternative was selected because the second would have led to immediate deadlock, however the deadlock occurred after the first action.

As a further example, note that
(a; b; **stop** [] c; d; **stop**) |[a,b]| (a; b; **stop** [] d; f; **stop**) is

equivalent to a; b; **stop** [] (c; d; **stop** ||| d; f; **stop**)

It is also interesting to consider the role of the internal action in this respect. For example, a;
exit || (a; **exit** [] **i**; b; **exit**)
is equivalent to a; **exit** []
i; **stop**

i. e., it may deadlock if the system chooses to execute the internal action before agreeing on *a*. On the other hand,

(a; **exit** [] b; **exit**) || (a; **exit** [] **i**; b; **exit**) is
equivalent to a; **exit** [] **i**; b; **exit**
i.e. it will not deadlock (if the environment cooperates), while a; **exit**
|| (**i**; a; **exit** [] **i**; b; **exit**)
is equivalent to

i; a; **exit** [] **i**; **stop**

i.e. it may deadlock depending on what internal action is executed.

These examples show that, as already mentioned, there is only interleaving on internal actions.

The parallel composition operators */// and //* are commutative and associative. *|[L]|* is commutative, and may be associative depending on the gates in *L*.

Putting The Modules Together

Now that we have gained some experience with LOTOS operators and specified the behavior of each process separately, we can compose them using the parallel composition operators. But first, let us look at the complete specification and then elaborate on it.

```
1       specification Producer_Consumer  [ pc1, pc2, cc1 cc2 ] : exit
2
3               behavior
4               (
5               Producer [ pc1, pc2 ]
6               |||
7               Consumer [ cc1, cc2 ]
8               )
9               ||
10              Channel [pc1, pc2, cc1, cc2]
11
12            where
13            process Producer [ out1, out2 ] : exit :=  . . .       (*As defined previously*)
14            process Consumer [ in1, in2 ] : exit :=  . . .         (*As defined previously *)

15            15        process Channel [ le1, le2, , re1, re2 ] : exit := . . .  (*As defined previously *) 16
              endspec
```

Before introducing the rest of LOTOS operators, we must explain some of the notions that we have just introduced. Line 1 introduces the **specification *Producer_Consumer***, which synchronizes with the environment through four gates *pc1, pc2, cc1, cc*2. Line 3, **behavior,** indicates the beginning of

Producer_Consumer's behavior expression. Any global data abstractions would have to be declared on line 2. The behavior expression of this specification is the composition of three instances of three processes. Line 12, the **where** clause, defines the processes that are used in the behavior expression of the specification. Lines 13, 14 and 15 introduce the definitions of the three processes ***Producer, Consumer*** and ***Channel.*** Note that each process definition has a list of formal gates, which must be relabelled with actual gates.

It is important to know that relabelling is done dynamically as each action is executed, rather than statically as the process is instantiated. For example, let:

```
 process      P [a, b, c] : noexit :=
            a; b; stop |[a]|  a; c; stop
endproc
```

The static relabelling of instantiating ***P*** with ***P****[c, c, a]* would be equivalent to *c; c;* ***stop*** *|[c]| c; a;* ***stop***, i.e. to *c; a;* ***stop***. In LOTOS' dynamic relabelling instead, the substitution is done before the actions are offered to the environment of the relabelled behavior. So, the execution of *a; b;* ***stop*** *|[a]| a; c;* ***stop*** results in *a; (b;* ***stop*** *||| c;* ***stop)***, which is equivalent to *a; (b; c;* ***stop*** *[] c; b;* ***stop)***. This , of course, becomes *c; (c; a;* ***stop*** *[] a; c;* ***stop)*** after the relabelling.

The behavior given on lines 3 to 12 can be replaced with the following equivalent behavior.

```
behavior
     (

           Producer [ pc1, pc2 ]

       |[pc1, pc2]|
             Channel [pc1, pc2, cc1, cc2]

       |[cc1, cc2]|
           Consumer [ cc1, cc2 ]

     )
   where  . . .
```

Note that the parallel composition operators have provided us with a powerful new tool to better specify the concepts introduced above. For example, a channel that can take and deliver, in any order, can be written as:

```
pc1; cc1; exit   |||   pc2;  cc2;  exit
```

while a similar channel, which can also lose an element, can be written as:

pc1; (cc1; **exit** [] **i**; **exit**) ||| pc2; (cc2; **exit** [] **i**; **exit**)

Disable operator

The LOTOS *disable* operator *[>* models an interruption of a process by another process. So, ***P1*** *[>* ***P2*** means that, at any point during the execution of ***P1***, there is a choice between executing one of the the next actions from ***P1*** or one of the the first actions from ***P2***. Once an action from ***P2*** is chosen, ***P2*** continues executing, and the remaining actions of ***P1*** are no longer possible. If ***P1*** terminates unsuccessfully, the first actions from ***P2*** are offered, while if ***P1*** terminates successfully, ***P2*** does not start execution. For example, a; b; **exit** [> c; d; **stop**

is equivalent to a; (b; (**exit**   []   c; d; **stop**) [] c; d; **stop**) []
c; d; **stop**

As a final requirement, let us assume that the channel may also fail at any time.

To model this fact, we put a disable [**>** at the end of the channel's behavior expression. We use the internal action **i**, to express an internal decision by the channel.

```
process Channel [ pc1, pc2, cc1, cc2] : exit :=
(
        (* same behavior expression as in Section 2.6 *)
)
 [> (     i; (* channel goes down *)
          exit
    )
endproc
```

Also, to ensure that the ***Consumer*** terminates successfully when the ***Channel*** goes down, we must disable the existing behavior of the ***Consumer*** with an **exit** which is always ready to synchronize with the **exit** after the [> in the ***Channel***. The ***Consumer*** then becomes: **process** ***Consumer*** [cc1, cc2] : **exit** :=

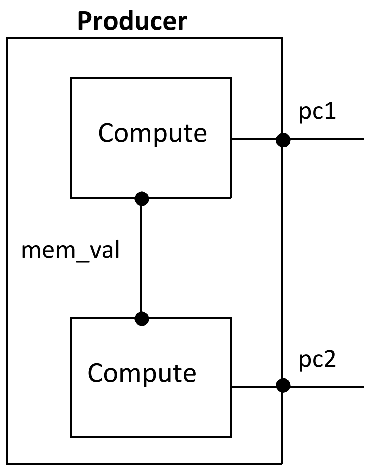
```
  (
        (* same behavior expression as in Section 2.6 *)
     )
    [> exit
endproc
```

Note that the ***Consumer*** behavior is still equivalent to the previous one. The difference would appear if the ***Consumer*** enabled itself recursively.

A corresponding change is necessary in the ***Producer.***

Hiding operator

The ***hide*** operator allows abstracting from the internal functioning of a process, by hiding actions that are internal to it. In particular, when the top-down approach is used, the designer can compose the system using LOTOS operators while hiding the details of interprocess communications that are irrelevant at a higher level of abstraction. For the sake of illustration, let us assume that the process ***Producer*** is composed of two subprocesses (actually two instances of the same process)



```
process Producer [ pc1, pc2 ] : exit  := hide
    mem_val  in
    (
        Compute [pc1,  mem_val]
        |[mem_val]|
        Compute [mem_val,  pc2]
    )
    where
        process Compute [v1, v2] : exit  := v1;  v2;
            exit
        endproc
endproc
```

The expansion of ***Producer*** is: *pc1; **i**; pc2; **exit**,* which is the behavior that we wish to model. It offers the observable action *pc1,* it performs an internal action ***i*** which represents a hidden action (the result of synchronization between *v2* and *v1* relabelled *mem_val*), then it offers action *pc2.* Note that the cooperation of the environment is required for both actions *pc1* and *pc2*, but not for ***i.*** Generally speaking, an action requires cooperation of all processes that must synchronize on that action by virtue of the parallel composition operators, unless a ***hide*** hides it from external processes. If an action is not hidden with respect to the environment, cooperation of the environment is also required. The attentive reader will have noted that the absence of hiding in our ***Producer_Consumer*** specifications makes it necessary for the environment to participate in actions on gates *pc1, pc2, cc1, cc2*. This can be prevented by hiding these four gates at the top level of the specification.

As an additional example of the ***hide***, note the following:

(**hide** b **in** a; b; c; **exit**) || a; c; **exit**

is equivalent to *a;* ***i****; c;* ***exit*** because *b* is turned into an internal action which does not synchronize.

Also,

hide b **in** (a; b; c; **exit** || a; c; **exit**)

is equivalent to *a;* ***stop***.

Conclusions

We have presented the specification language LOTOS. The language has a strong algebraic nature and the first impact with the apparently complex symbology of specifications may be discouraging. However, we hope we have proved, with the examples given, that once the user has achieved some familiarity with the operators of the language, he/she can specify systems in a natural way which reflects quite directly the way the system's structure and behaviour are conceived at the intuitive level. The specifier, in general, does not feel forced to express unnecessary details with respect to his/her *abstract* view of the processes being specified.
LOTOS has the merit (and takes the risks) of being based on relatively new and powerful theories, which so far have mainly been confined to academic environments. The wide exposure that the language is currently undergoing by its application to the specification of OSI protocols and services.

References

[1] I. Ajubi, "Draft Formal Specification of the OSI Connection-Oriented Session Protocol in LOTOS", ISO/TC 97/SC 21 N. 1486, February 1986.

[2] E. Brinksma, "A Tutorial on LOTOS", in: M. Diaz (ed.), Proceedings of IFIP Workshop

'Protocol Specification, Testing, and Verification V', pp. 171-194, North-Holland, Amsterdam,

1986.

[3] Guy Leduc -Université de Liège ”Ingénierie du logiciel dans les réseaux informatiques”,Chapter 2 The LOTOS Language

[4] *Tommaso Bolognesi,"* Introduction to the ISO Specification Language LOTOS" Via S. Maria 36 - 56100 Pisa, Italy

[5] *Kenneth J. Turner. The Formal Specification Language LOTOS: A Course For*

Users. Department of Computing Science and Mathematics, University of Stirling, Scotland, August 1989

[6] Hubert Garavel and Holger Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Proceedings of the 11th International. Symposium of Formal Methods Europe FME'2002 (Copenhagen, Denmark), volume 2391 of Lecture Notes in Computer Science*, pages 410-429. Springer, July 2002. Full version available as INRIA Research Report 4492.

[BB 87] Bolognesi, B., and Brinksma, E. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems* 14 (1987) 25-59. Also reprinted in [VVD89] 23-73.

[BB 89] Bochmann, G.v., and Bellal, O. Test Result Analysis in Respect to Formal Specifications, Proc. 2nd Int. Workshop on Protocol Test Systems, Berlin, Oct. 1989, pp.272-294.

[BC 89] Bolognesi, T., and Caneve, M. Equivalence Verification: Theory, Algorithms, and a Tool. In [VVD] 303-326.

[BDD 90] Bochmann, G.v. , Desbiens, D., Dubuc, M., Ouimet, D., and Saba, F. Test Result Analysis and Validation of Test Verdicts. To appear in the Proceedings of the Workshop on Protocol Test Systems, McLean, Virginia, (Oct. 1990).