SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 18(4), 365–385 (APRIL 1988)

An Interpreter for LOTOS, A Specification Language for Distributed Systems*

L. LOGRIPPO, A. OBAID, J. P. BRIAND AND M. C. FEHRI
Department of Computer Science, University of Ottawa, Ottawa, Ontario, Canada K1N 9B4

SUMMARY

LOTOS is an executable specification language for distributed systems currently being standardized within ISO as a tool for the formal specification of open systems interconnection protocols and services. It is based on an extended version of Milner's calculus of communicating systems (CCS) and on ACT ONE abstract data type (ADT) formalism. A brief introduction to LOTOS is given, along with a discussion of LOTOS operational semantics, and of the executability of LOTOS specifications. Further, an account of a prototype LOTOS interpreter is given, which includes an interactive system that allows the user to direct the execution of a specification (for example, for testing purposes). The interpreter was implemented in YACC/LEX, C and Prolog. The following topics are discussed: syntax and static semantics analysis; translation from LOTOS external format to internal representation; evaluation of ADT value expressions and extended CCS behaviour expressions. It is shown that the interpreter can be used in a variety of ways: to recognize whether a given sequence of interactions is allowed by the specification; to generate randomly chosen sequences of interactions; in a user-guided generation mode, etc.

KEY WORDS Specification languages Distributed systems Interpretation Communications protocols

INTRODUCTION

The theory and practice of specification languages for data communications protocols and services (often called formal description techniques or FDTs) has been the object of much recent interest. Formal and exact specifications of protocols and services are useful in every phase of the protocol development life-cycle. Even more, they are essential for protocols and services that are international standards meant to be implemented in compatible ways across the world. The specification must capture those features of an implementation that are necessary for it to be able to communicate with other implementations. Therefore, it is important that the specification be implementation-independent.

Several approaches have been proposed for the specification of data communications systems, which range the whole gamut from very implementation-independent logic specifications[1] to specifications written in implementation languages such as C.[2]

* Preliminary versions of parts of this paper have appeared in the Proceedings of the 1986 ACM SIGCOMM Symposium and in B. Sarikaya and G.v. Bochmann (eds) *Protocol Specification, Verification, and Testing VI*, North-Holland, Amsterdam, 1987.

0038–0644/88/040365–21$10.50

Received 8 September 1986
Revised 13 March and 23 November 1987

Much could be said on the advantages and disadvantages of executable and non-executable specification techniques. Executable techniques provide a 'running prototype' of what is specified and thus have the advantage of allowing testing to begin with specification. On the other hand, they force the specifier to make some implementation choices that are premature at the specification stage and that may puzzle the implementor. Non-executable techniques have the opposite advantages and disadvantages. In addition, non-executable specifications, such as temporal logic specifications, can give explicitly the constraints that the system should respect (for example, that a certain state of affairs always follows another), which may be difficult to extract from executable specifications. Our feeling is that both types of specifications are needed. Further discussion on this topic can be found in References 3–5.

The specification language LOTOS (Language Of Temporal Ordering Specifications), an FDT that is in the process of being standardized for the formal specification of open systems interconnection (OSI)[6] protocols and services, has already been the subject of several papers.[7–9] Even though OSI was the immediate motivation for LOTOS, the language is general in scope and could be used for the specification of most types of distributed systems. The definition of LOTOS used here is that in Reference 10.

This paper presents the philosophy of executability underlying LOTOS and reports on our experience in implementing a prototype LOTOS interpreter. Since LOTOS is expected to become an International Standard of the International Organization for Standardization (ISO), presumably there will be several implementations in the future, and therefore we hope that our experienc will benefit future implementors.

Reference 11 contains a survey of other related systems that have been developed recently. Among them, those by Berthomieu[12] and Karjoth[13] are worth mentioning as sources of interesting ideas. Further background information is contained in Reference 14. Our system is the only one we know that accepts the language of Reference 10 almost in its entirety, including complete static semantics checking (the only features not implemented were some ACT ONE features whose semantics could not be determined from References 10 or 15).

LOTOS CONCEPTS

Since LOTOS is a relatively new language, it is appropriate to present a brief overview of its main concepts. For a full understanding of this paper, however, familiarity with a more complete tutorial[7] or with Reference 16 is necessary. A shorter tutorial, with examples and discussion of LOTOS formal semantics, is given in Reference 17. The reader should be aware of the fact that LOTOS was being substantially enhanced at the time of writing of this paper.

LOTOS is based primarily on the principles of the calculus of communicating systems (CCS),[16] which provided the inspiration for the representation of behaviour expressions. However, it is also influenced by the related work on Communicating Sequential Processes (CSP).[18] The augmented CCS on which LOTOS semantics are based is called CCS*. For the definition of data abstractions (i.e. values, data structures and operations on them), LOTOS is based on the ADT formalism of ACT ONE.[15]

CCS and ACT ONE could then be considered the two parts that constitute LOTOS: the first for describing the 'control', and the second for describing the 'data'. These two parts are mutually independent, except that the control part makes reference to

the data part for the semantics of value expressions. One advantage is that a LOTOS interpreter can be written in two independent parts, where the control part has to call on the data part every time a value expression has to be evaluated. Another possible advantage is that users who prefer other data formalisms to ACT ONE could use such formalisms in lieu of ACT ONE (this, however, is likely to require important changes to the static semantics of LOTOS).

LOTOS is a recursive language without side-effects. The structure of a typical LOTOS specification could be graphically represented as a tree made up of many small, nested processes. LOTOS favours stepwise decomposition of specifications. For example, a service specification may, at the top level, consist of four processes, a sending entity, a receiving entity and two bidirectional queues. These processes may themselves be subdivided, and this subdivision continues down to small processes, such as the service primitives themselves. Some of these processes may share 'gates' through which communication occurs. Also, various control relationships may hold between processes. One process may be alternative with relation to another, or may be in parallel with another, or may be capable of 'disabling' another. All this is described in greater detail below.

In LOTOS, interprocess communication occurs by means of a two-way 'rendezvous' mechanism, called 'interaction'. In an interaction, each one of the participating processes specifies the name of a gate and whatever information it wishes to provide on the values to be established. This can range from a specific value to a 'sort' (i.e. data type) only. Processes can participate in an interaction only if, at the same time, they are ready for an interaction where they specify the same gate and compatible sorts. For example,

```
g ?x:int !(3+5) !true                    (1)
```

is an 'action denotation' in LOTOS. It describes an 'interaction offer' at gate g, i.e. the process executing this action denotation is ready to interact with other processes in the environment, which are or will become ready to execute complementary ('matching') interaction offers. If it occurs, the interaction establishes three values: the first of these values, x, is not known to this process, hence the question mark or 'query'. Only its sort is known to be 'integer'. The next two values are known, hence the exclamation marks, or 'shrieks'. The first is the integer 8, and the second is the boolean true. A matching offer could be something such as

```
g !3 ?y:int ?z:bool                      (2)
```

If a process B2 becomes ready to execute (2) at the same time that another process B1 is ready to execute (1), an interaction may occur, and one can say that process B2 passes to B1 the value 3 (which becomes the value of x in B1), and receives the values 8 and true, which become the values of y and z in B2. After the interaction, each process proceeds independently. Another matching offer for (1) would be

```
g ?y:int ?z:int !true                    (3)
```

In this case, the sorts of x and y match, but neither offer provides a value. An interaction can still occur as it is assumed that 'some' integer value will be established

by the interaction mechanism and sent to both processes. It is also interesting to note the way the two offers match in their third component, where they both agree on the value true. In this case, there is no exchange or generation of values, but just synchronization on a value known by both.

An example of non-matching offer would be

```
g !3 ?y:bool ?z:bool
```

because the processes cannot agree on whether the second value should be a boolean or an integer.

The LOTOS concept of interaction is a generalization of the traditional concept of output with matching input (a shriek with a matching query). One can have interactions where both parties already have all the information (two shrieks), and others where neither party has it (two queries).

A process that executes an offer will wait for a matching offer to occur. If the latter cannot occur, the process containing the offer is said to be in deadlock.

The symbol i (the 'internal' action) denotes an action that needs no co-operation from the environment in order to occur. It takes time, but it does not involve any variables. It is also used to represent the external view of an interaction that has occurred between two processes (it is visible to the external environment as lapsed time only). On the other hand, the symbol stop denotes a process that does nothing. It models a deadlock.

The important characteristics of the LOTOS interprocess communication mechanism are: it is symmetric (a query can be defined as a disjunction of shrieks); it is anonymous, i.e. neither side knows the name of the other; it does not involve queuing; it is atomic, although not (necessarily) instantaneous; it is non-deterministic, since at a given point several rendezvous may be possible and several different values could be agreed on.

Action denotations are the basic building blocks of LOTOS. These building blocks can be composed in increasingly larger blocks (called 'behaviour expressions') by means of certain operators:

1. The operator ; is used to prefix an action denotation to a behaviour expression (first perform an offer, then proceed to the behaviour expression).
2. The operator [] expresses alternatives. B1 [] B2, where B1 and B2 are behaviour expressions, means do either B1 or B2. The choice may be specified to be either completely non-deterministic (this is useful for example to express 'spontaneous transitions', such as in the case where B1 or B2 start by i); or determined by the environment (in this case, each choice starts by an action denotation, and the choice is determined by what complementary offers are provided by the environment); or determined by a 'guard', i.e. a condition (the notation for guards is: [(condition)]-> B, where B is a behaviour expression).
3. The operator || expresses parallel composition. B1 || B2, where B1 and B2 are behaviour expressions, means: do B1 and B2 in parallel. Furthermore, B1 and B2 may interact, by the mechanism described above, through the gates they share. There is a way to restrict the set of gates through which B1 and B2 can communicate.
4. The operator >> performs the sequential composition of behaviour expressions ('enabling'). B1>>B2, where B1 and B2 are behaviour expressions, means do B1, then B2. There is a mechanism by which B1 can pass values to B2, namely

```
B1 >> def x1, ... ,xn in B2
```

states that B1 'exports' a vector of n values that become the values of variables x1,. . .,xn in B2.

5. The operator [> is called 'disabling' and represents situations, common in data communications systems, where a process is allowed to interrupt another process. The intuitive meaning of the expression B1 [> B2, where B1 and B2 are behaviour expressions, is that process B2 may disrupt or prevent the execution of process B1. B2 may or may not be executed, depending on a non-deterministic choice. If it is not executed, B1 executes normally and the whole construct terminates. However, at any time during the execution of B1, B2 can take over. In this case, B1 will not be resumed, and B2 runs to completion.
6. Gate hiding. A LOTOS system can be seen as a 'black box', say A, communicating with the external environment through gates. A itself may contain other black boxes, say B and C, that also communicate through gates. Some of the gates of B and C may also be gates of A. Others instead may be for only internal communication between B and C. These latter gates must be hidden inside A, which is accomplished by the operator \. Hence, A\[X], where A is a behaviour expression, and X is a list of gate names, indicates that gates in the list X are invisible outside A.

Furthermore, LOTOS has mechanisms to name behaviour expressions (similar to procedure declarations in conventional programming languages), and mechanisms to instantiate behaviour expressions, similar to procedure calls in conventional languages. Named behaviour expressions are called 'process abstractions'. Particularly noteworthy is the 'gate parametrization' mechanism, by which formal gates can play different actual roles in different instantiations. Gate parameters are shown between square brackets after the name of a process abstraction. For example, the following line:

```
process check[retry,deliv](Expect:int, Mes:int|int) :=
```

names a process check that has two formal gate parameters (retry and deliv) and two formal variable parameters, Expect and Mes, that are of sort int. Also, the vertical bar indicates that the process exports an integer value (see item 4 above).

As mentioned above, LOTOS has also a data part, which is used to specify the data structures and the operations on them. This portion of LOTOS is taken almost verbatim from well-known approaches to data abstraction specification.[15, 19] Accordingly, the description of a data abstraction consists of two parts: a syntactic part, where the arguments and the functionality of each operator are described, and a semantic part where the meaning of each operator is described implicitly by means of axioms.

EXECUTABILITY OF LOTOS

LOTOS has been designed as an executable specification language. However, because precise and clear specification, rather than executability, was the primary goal of its design, LOTOS does not particularly cater to ease of translation or efficiency of execution. For example, implementing the LOTOS interprocess communication mechanism requires in principle that, at each step of execution, all pairs of active processes be examined to determine which ones are able to participate in an interaction.

More seriously, there are several features of LOTOS that, if used, may prevent executability altogether. A first such feature is the possibility of defining infinitely branching execution trees, such as the ones generated by

```
process P[g] := (g; stop) * P[g] endproc
```

where * can be the parallel operator, the alternative, or the disable operator (this example is due to Elie Najm). Note that process P is not 'guardedly well-defined'.

Further, in LOTOS one can have infinitely branching trees generated by a non-deterministic choice among the elements of an infinite set of values, such as in example (3) above. In this case, however, the non-determinism can be solved by intervention of the user who specifies the value(s) to be used, as is done in our interpreter.

Finally, it is of course possible to define sets of ADT axioms that do not have the termination property.[20, 21] For example, sets including permutative rules such as plus(A,B) = plus(B,A) that may cause the value expression evaluator to enter an infinite loop.

The problem of detecting or 'repairing' such specifications presents serious theoretical difficulties, and was not attacked at the present stage of work. However, the Knuth–Bendix algorithm is an option in our system, as mentioned in the next section. In addition, as discussed above, one could observe that forcing full executability on a specification language is likely to reduce its expressive power. A specifier that is interested in using our interpreter will have to choose a constructive and computationally efficient style of specification, and be aware of such pitfalls.

Therefore, it is not practical to use LOTOS executable specifications in lieu of implementations. An additional reason is that a specification usually contains non-determinism to specify implementation choices. Such non-determinism must be resolved at implementation time. The usefulness of our approach, therefore, is limited to prototyping and to preliminary testing of a specification.

STRUCTURE OF THE INTERPRETER

The main aim underlying our work was to produce a full LOTOS interpreter in a relatively short time and with limited resources. We wanted a tool to experiment with, rather than a production tool. Because the definition of LOTOS is not stable yet, we wanted a system that could be adapted to change easily. The environment had to be fairly widely used in research establishments, so the intrepreter must be portable. Furthermore, the language was to be implemented in a straightforward way, close to its semantic definition, and avoiding all but the most obvious short cuts or optimizations. The language definition[10] simplifies this task by providing constructive semantics. In this way, our interpreter is a running test bench of the language definition, and the several minor changes that can be expected at this stage are likely to result in equally minor changes to the interpreter.

The general structure of our interpreter is shown in Figure 1. The control part is kept separate from the data part. Further, both parts are implemented by first translating LOTOS code into an internal representation (intermediate CCS* code and ADT rewriting rules, respectively). Thus, the first part of the interpreter (the top four boxes in Figure 1) is a compiler. The real interpreter works on the internal representations and is made up of two parts: the CCS* interpreter and the ADT rewriting rule interpreter. The ADT interpreter is a subordinate of the CCS* interpreter in the sense that it evaluates ADT functions when requested. The interpreter is called SINAPS,

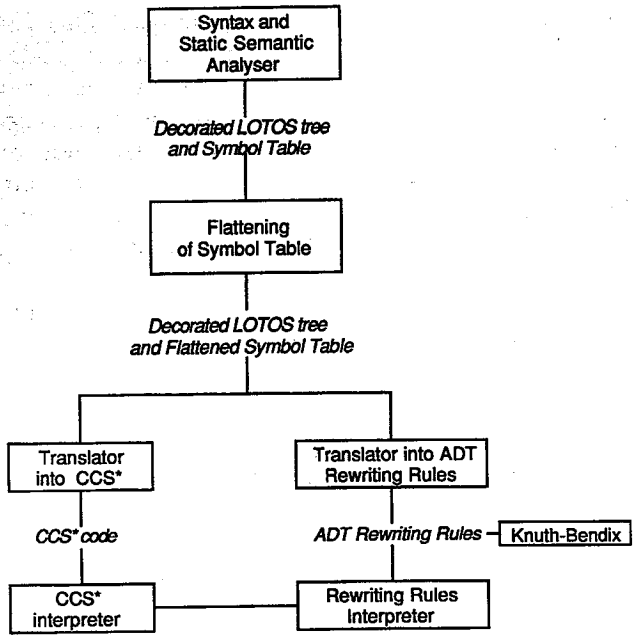

Figure 1. The structure of the interpreter

for Simulation and INference APplication System.

We used UNIX[22] and the language C, with the related compiler-writing aids such as LEX and YACC, for the compiler. The CCS* interpreter and the ACT ONE interpreter are written in Prolog. Various reasons motivated the use of Prolog.[3, 5, 23] The most important reason was that the operational semantics of both CCS* and ACT ONE are defined in terms of rewriting rules involving variable replacements, which can be easily programmed in Prolog. The other was that Prolog programs can be made reversible, i.e. capable both of recognizing whether a set of arguments satisfies a condition, and of generating a set of arguments that satisfies the same condition. Further, the same program can be used to traverse a tree both 'top down' and 'bottom up'. These features are used to advantage.

Syntax analysis, static semantics analysis and the LOTOS tree

The compiler is made of four modules (or passes), each of them corresponding to a chapter in the formal definition of the syntax and static semantics of the language.[10] Since LOTOS allows forward references to processes and sorts prior to their definitions, syntax and static semantics analysis cannot be performed (at least not at low cost) in a single pass. On the other hand, the four passes make the compiler easy to update and modify, which is an important advantage for a language that is still unstable.

As in MENTOR,[24] a specification is internally represented by an attributed tree (the LOTOS decorated tree), the skeleton of which is the abstract syntactic tree of the specification. This tree, which is updated at each step, is the central data structure of the compiler, and is the interface between its four passes. This internal representation is potentially useful not only for the interpreter, but also for other future applications, such as verifiers, graphic interfaces, etc.

The first step is parsing. The kernel of this module is the parsing function produced

by YACC from the BNF representation of the grammar. This function builds an incomplete abstract syntactic tree (i.e. with no semantic attributes).

The second step processes instances of identifiers that are declarations by checking their uniqueness and building a symbol table entry of the right scope.

The third step processes all remaining instances of identifiers by looking in the symbol table for the corresponding declaration. It also builds the attributes of value-identifiers (sorts of the values), operation-identifiers (sorts of the arguments and sort of the results) and process-identifiers (number of gates, sorts of the arguments and functionality).

The last step checks if all value-expressions and behaviour expressions are well-formed with respect to their sort and functionality.

The structure of the LOTOS decorated tree was designed according to the following requirements:

(a) It has to be free of syntactic or semantic errors.
(b) It has to be structured: it should eliminate the need of sequentially scanning the input to obtain specific information.
(c) It has to be equivalent to the input: one must be able to 'decompile' the internal representation into the external one and get the original specification, or at least a similar one.

Each node in the tree corresponds to an entity in the abstract syntax and holds three kinds of data: identification data, housekeeping data (arity of the node, source line number, pointers) and synthesized semantic attributes. The latter are built by semantic routines. Furthermore, certain nodes are associated with local symbol tables. These tables group particular sets of attributes and collect information on every relevant symbol in the specification (identifiers, constants . . .). They are built according to the scope rules given in Reference 10, where a scope is defined as a part of text associated with a non-terminal (a node). Therefore, the table associated with a node gathers all the symbols whose scope is the tree under the node. The synthesized attributes in the symbol tables are used to build and check the sorts and functionality of value and behaviour expressions, respectively.

The three requirements mentioned above are met. The attributes ensure the correctness of the specification (i.e. from the point of view of static semantics), whereas access to a specific item is achieved by simple tree-traversal algorithms. Furthermore, we were able to write a decompiler as described above.

As for most modern specification languages, LOTOS rules allow different objects having different meanings or defined in different parts of the specification to have the same name. Hence the need for a flattening function, i.e. a renaming function that transforms the structured name space of LOTOS into a flat one where any two different objects have different names. This function is applied on the decorated tree just before the translation phases described below.

Translation and execution of the data part

ADT definitions can appear in different places in a LOTOS specification and hence may have different scopes. Our interpreter, however, removes this scope structure first of all by flattening and then by performing the union of all the sets of equations included in a specification into a single unstructured specification. Further, the equations are

translated into rewriting rules by traversing the internal representation of the equations in the LOTOS decorated tree and orientating the equations from left to right in such a way that the set of variables in the right side of the rewriting rule is included in the set of variables of the left side. (Unfortunately, this may not be possible, and even if it is possible, it does not guarantee termination of the system).

The ADT Rewriting Rule Interpreter is the part of the LOTOS interpreter that evaluates value expressions. It executes the rewriting rules by a rewriting algorithm of the type 'leftmost-innermost'. In other words, it reduces value expressions beginning from the innermost subterm at the leftmost side of a term. During the reduction, our algorithm memorizes all the reductions accomplished since its invocation in order to avoid reducing an expression that was previously encountered and completely reduced.

For this evaluation method to be effective, there are certain properties that must be satisfied by the rewriting system in use; it must be confluent and terminating. Termination is necessary both in order to guarantee termination of the evaluation and for checking confluence. It is undecidable, but can often be proved by various methods.[20] Confluence means that the result of the computation does not depend on the choice of rules to be applied, or the order in which they are applied. This property can be checked by executing the Knuth–Bendix algorithm,[25] which is an option in our system. When a rewriting system is not confluent, the Knuth–Bendix algorithm can also be used to help transform it into a confluent one. This algorithm is based on the use of equations as rewriting rules and on the computation of critical pairs when left members of rules overlap. If a critical pair has distinct irreducible forms, a new rule must be added and the procedure applies recursively until it stops. This algorithm was implemented in our system. It executes interactively; in other words, the user is prompted to provide information such as the preferred orientation of the rules and additional rules that may be needed.

As an example of LOTOS ADT definition we provide the specification of the queue in the simple service provider, which is discussed below.

```
type buffer is int1  with boolean1 with boolean with
    sorts  queue
    opns    new  :                -> queue
            error:                -> int
            add  :  int, queue -> queue
            rem  :       queue -> queue
            first:       queue ->  int
            empty:       queue -> bool
            if_then_else: bool,queue,queue -> queue

    eqns
    forall M:int, Q:queue, Q1:queue, Q2:queue in
      rem (new)       = new
      rem(add(M,Q))   = if_then_else(empty(Q),new,add(M,rem(Q)))
      first(new)      = error
      first(add(M,Q)) = if_then_else_int(empty(Q),M,first(Q))
      empty(new)      = true
      empty(add(M,Q)) = false
      if_then_else(true,Q1,Q2)  = Q1
      if_then_else(false,Q1,Q2) = Q2

endtype
```

Translation and execution of the control part

In Reference 10, LOTOS semantics are defined by a translation function that maps instances of LOTOS behaviour expressions to CCS* behaviour expressions. Operators that exist in LOTOS but have no direct correspondent in CCS* (an example is 'enable') must be expanded. Properties of an occurrence of a LOTOS behaviour expression are defined to be the properties of the occurrence of the corresponding CCS* behaviour expression. In our implementation, the translation process consists of translating the control part of the renamed LOTOS decorated tree in the CCS* internal representation, which is conceived in terms of the need for efficient execution in Prolog.[26, 27]

In Reference 10, CCS* semantics are defined as operational semantics, i.e. as an idealized interpreter capable of executing certain types of rewriting rules. In CCS style, interactions are defined to cause the transformation of a behaviour expression into another behaviour expression (its derivative), which defines what has to be done after the interaction has occurred. The type of transformation depends on the operators contained in the behaviour expression. For example, if a behaviour expression B is of the form a; B1 where a is an action denotation and B1 is a behaviour expression, the effect of the occurrence of a matching offer for a is defined by the following transformations on B: B becomes B1 (a has been taken out since it has been executed); and, if the interaction has determined the value of some variables, this value is replaced for the variables everywhere in the variables' scope in B1.

As a further example, the following are the inference rules for the CCS* choice operator + (which is the translation of the LOTOS operator []): they state that a derivative of a choice is the same as the derivative of the branch on which the derivation has been performed (B, B1 and B2 are behaviour expressions, whereas a is an interaction offer). Either one of the two rules can be selected non-deterministically if a applies to both sides of an alternative.

if B1 -a-> B2 then B1 + B -a-> B2 (rule 1)

if B1 -a-> B2 then B + B1 -a-> B2 (rule 2)

According to CCS* semantics, at any instant of execution, one can determine the set of possible next actions of a CCS* process.

Inference rules were implemented in Prolog.[26, 27] For instance, rule 1 above can be written in Prolog as

```
infer(choice(B1,B), Action, B2):-
      isaction(Action),
      infer(B1,Action,B2).
```

where isaction verifies if Action is a legal action.

As mentioned below, the non-determinism in the set of inference rules is resolved either by the user (in the 'one-stepper' mode) or automatically by random selection.

Our interpreter, therefore, is inherently a uniprocessor, and this is consistent with CCS*, where global system states are captured by behaviour expressions.

An example

As an example, a simple service provider is described in LOTOS and CCS*. The example may seem somewhat contrived; however, it shows several LOTOS features.

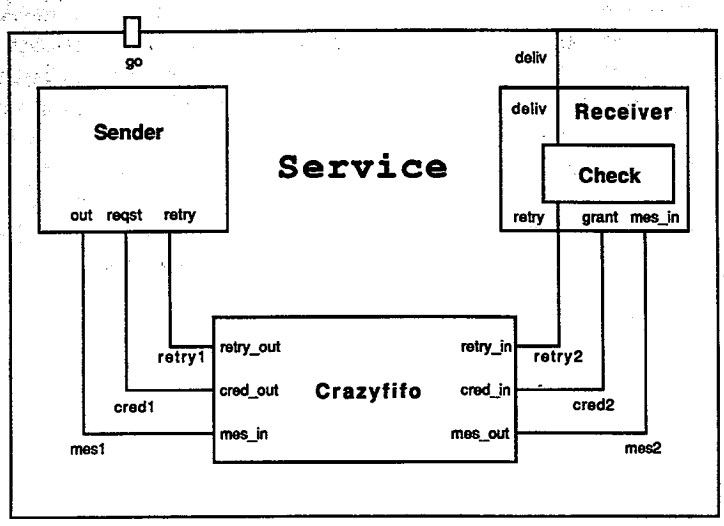

Figure 2. A simple service provider

The general structure of the system is shown in Figure 2. Reasons of space prevent us from showing the full LOTOS specification (the interested reader is referred to Reference 28 for further details). The body of the paper concentrates on its CCS* counterpart, which is the one that is actually interpreted. For clarity, standard CCS* notation is used, in lieu of the more obscure CCS* internal reprsentation used by our system.[29] There are sender and receiver processes, which communicate with each other by an unreliable channel, represented by a crazyfifo process. Upon prompting from one of the users, and upon reception of credits from the receiver, the sender starts sending integer-valued messages, numbered from 1 onwards. These messages must be received by the other user in the order they were sent. Therefore, the reciever will perform a sequence check and can ask the sender to transmit starting with a designated integer. Whenever the sender runs out of credits, it blocks until credits are sent to him by the receiver.

Proceeding bottom up, the LOTOS specification of the sender is as follows:

```
process sender[out,reqst,retry] (Mes:int,Cred:int) :=
                    (* sender sends if it has credits
  [gt(Cred,zero)] -> out!Mes;
                    sender[out,reqst,retry](succ(Mes),pred(Cred))
  []                 (* or it waits for credits
  [eq(Cred,zero)] -> reqst?New_cred:int;
                    sender[out,reqst,retry](Mes,New_cred)
  []                 (* or it retransmits if requested
  retry?Expect:int ; sender[out,reqst,retry](Expect,Cred)
endproc
```

The corresponding text in CCS* is:

```
sender(mes,cred) :=

       [cred > 0] -> out!mes; sender(mes+1, cred-1)
       +
       [cred = 0] -> reqst?new_cred:int; sender(mes,new_cred)
       +
       retry?expect:int; sender(expect, cred)
```

The crazyfifo process uses the queue ADT shown above. It is in this ADT that the operations add, rem, etc. are defined; add(mes,q) is the queue that results of adding message mes to queue q; first(q) yields the message in q; and rem(q) is the queue that results of removing from q its first element. crazyfifo can either input a message at gate mes_in or, if the queue is not empty, output its first element or lose its head. It can also play the role of one-place buffer from the receiver to the sender (the last two alternatives), which is used in order to carry retransmission and grant messages (for which we did not wish to specify queuing or possible loss). Note that the choice between regular output and loss is non-deterministic, and this is expressed by the internal event i.

```
crazyfifo(q) :=
    mes_in?mes:int; crazyfifo(add(mes,q))              (* input
    +
    [not(empty(q))] ->
            ( mes_out!first(q); crazyfifo(rem(q))     (* regular delivery
              +
              i; crazyfifo(rem(q))                    (* loss
            )
    +
    retry_in?expect:int; retry_out!expect; crazyfifo(q)
    +
    cred_in?cred:int; cred_out!cred; crazyfifo(q)
```

The following process is the receiver. Its parameter expect is the expected message number. receiver can either receive a new message or grant five credits to sender. In the first case, it calls the local process check, which checks if the received message is the expected one. In the affirmative, the message is delivered to the end user via gate deliv. Otherwise, check asks for retransmission starting from the expected message number. Gate d (called d1 outside check) is used to represent sequencing from check to receiver. To understand its function, it must be recalled that CCS* does not have the enable construct >>. Therefore, a LOTOS construct where a process B1 enables a process B2 is translated into CCS* by a construct where B1 and B2 are run in parallel, but B2 must wait for transmission of values from B1 over an internal gate. This gate must be hidden from the outside environment.

```
receiver(expect) :=
    mes_in?mes:int;
            ( check(expect,mes) [d1/d]
            || d1?new_expect:int; receiver(new_expect)) \[d1]
    +
    grant!5; receiver(expect)                          (* granting credits

check(expect,mes) :=
    [mes = expect] -> deliv!mes_in; d!expect+1; stop (* a "good" message
    +
    [mes > expect] -> retry!expect; d!expect; stop   (* a "bad" message
```

In order for the processes to communicate, they must be composed in parallel and linked via certain common gates. For this purpose, one must rename some gates using common gate names. Below, one can see that sender and crazyfifo can now communicate via gates mes1, cred1 and retry1; crazyfifo and receiver can communicate via mes2, cred2 and retry2. At the same time, one must worry about other processes present in the environment also being able to communicate over these gates. To prevent this,

they were made invisible by the hiding operator \. Note that all gates are hidden, with the exception of the two externally visible ones (go and deliv). Note also that for the three processes to become active a go_signal must be provided by the user on gate go.

```
service :=
  go!go_signal;
    (  sender(succ(zero),zero) [mes1/out, cred1/reqst, retry1/retry]
    || crazyfifo (new) [mes1/mes_in, mes2/mes_out,cred1/cred_out,
                        cred2/cred_in,retry1/retry_out, retry2/retry_in]
    || receiver (1) [mes2/mes_in, cred2/grant, retry2/retry]
    ) \ [mes1, mes2, cred1, cred2, retry1, retry2]
```

A possible derivation sequence for process service above is shown, where for brevity the different renamings are denoted by the substitutions [S1], [S2] and [S3], and the set of hidden gates by A. Note that only the first and last interactions are with the external environment. All the others are internal to the system and so, according to CCS* semantics, they result in the internal action i.

```
service

 go!go_signal    (* we start, the only process that can proceed is receiver
 ----->(  sender(1,0) [S1]
        || crazyfifo(new) [S2]
        || receiver (1) [S3]
           ) \ A

   i      (* credits are trasmitted to crazyfifo
 ----->(  (sender(1,0)) [S1]
        || (cred_out!5 ; crazyfifo(new)) [S2]
        || receiver (1) [S3]
        ) \ A

   i      (* 5 credits are received by sender
 ----->(  sender(1,5) [S1]
        || crazyfifo(new) [S2]
        || receiver(1) [S3]
        ) \ A

   i      (* 1 transmitted to the queue
 ----->(  sender(2,4)[S1]
        || crazyfifo(add(1,new)) [S2]
        || receiver(1) [S3]
        ) \ A

   i      (* 1 is lost
 ----->(  sender(2,4) [S1]
        || crazyfifo(new) [S2]
        || receiver(1) [S3]
        ) \ A

   i      (* 2 is transmitted to the queue
 ----->(  sender(3,3) [S1]
        || crazyfifo(add(2,new)) [S2]
        || receiver (1) [S3]
        ) \ A

   i      (* 2 is received, check will fail
 ----->(  sender(3,3) [S1]
        || crazyfifo(new) [S2]
```

```
          || ((check(1,2)[d1/d]|| d1?new_expect:int; receiver(new_expect))
             \[d1]) [S3]
          ) \ A

    i      (* receiver asks crazyfifo for retransmission of 1
------>(  sender(3,3) [S1]
          || (retry_out!1; crazyfifo(new)) [S2]
          || ((d1!1; stop||d1?new_expect:int; receiver(new_expect))
             \[d1])[S3]
          ) \ A

    i      (* crazyfifo asks sender for retransmission of 1
------>(  sender(1,3) [S1]
          || crazyfifo(new) [S2]
          || ((d1!1;stop||d1?new_expect:int; receiver(new_expect))
             \[d1]) [S3]
          ) \ A

    i      (* 1 is sent via gate d1
------>(  sender(1,3) [S1]
          || crazyfifo(new) [S2]
          || receiver(1) [S3]
          ) \ A
```

and the process restarts again from message 1. It is hoped that this time message 1 will not be lost and will be delivered. Assuming that no other messages were sent in the meantime, one would have

```
deliv! 1
-------->( sender(2,2) [S1]
           || crazyfifo(new) [S2]
           || receiver(2) [S3]
           ) \ A
```

and so on.

USE OF THE INTERPRETER

The use of the interpreters of most traditional languages is straightforward: some data are read, and some are printed. This will happen in a predetermined order each time, and the internal mechanism is deterministic. The situation is not so simple for interpreters of specifications of distributed systems, because these describe systems that are highly non-deterministic in nature. For example, suppose that the purpose of the exercise is testing the design of a two-user data link service provider. It would not be sufficient for testing purposes to provide two sequences of messages to be sent from each end to the other, because in addition to regular delivery one would like to be able to test various internal conditions such as errors, etc. In many cases, the user is interested in exercising all the possible (or many of the possible) different orders of execution of the elementary operations, and there is a large number of these. Systematic exploration of the global state space of the system specified is possible with our interpreter using the backtracking feature of Prolog.[4, 5] Unfortunately, in most cases, efficiency problems would prevent us from completing this procedure. The question then is how to direct the interpreter to take various 'interesting' internal choices.

Our system can be used in at least two different ways: to validate interaction sequences, that is to tell whether an interaction sequence that is submitted is a

valid behaviour for the specified entity, and to generate interaction sequences, either exhaustively, or at random or (more commonly) under human direction. For a related approach based on different principles, see Reference 30.

Validation of interaction sequences

Validating sequences of actions is useful whenever one wants to test a specification on some concrete cases, for example, to help increasing the confidence that the desired process behaviour was specified. For instance, the sequence of derivation of service in the example above can be represented as

```
                go!go_signal; i; ...; deliv!1
service  ================================>( sender(2,2) [S1]
                                          || crazyfifo(new) [S2]
                                          || receiver(2) [S3]
                                           ) \ A
```

In order to exercise a sequence the following Prolog clause was defined:

```
exerce(B1,[],B1).
exerce(B1,[Action|RestOfActions],B11):-
        infer(B1,Action,B2),
        exerce(B2,RestOfActions,B11).
```

The first clause stops the program with the empty sequence. The validation process consists of providing an initial behaviour expression that will instantiate B1, a list of actions that will instantiate [Action|RestOfActions], and the resulting behaviour expression that will instantiate B11. The result will be either acceptance or rejection according to whether the sequence is valid or not. By not instantiating one of the three parameters one obtains the (possibly infinite) set of all possible values for that parameter that satisfy the relation exerce: by not instantiating two of them one obtains the set of all possible pairs of values that satisfy the same relation, etc. For example, by not instantiating the list of actions one obtains the set of all possible lists of actions leading from B1 to B11.

Below, a transcript of a terminal session is shown, giving a simple validation sequence for the receiver process by itself. The user submits a sequence to the receiver process, where the latter is made to offer the value '5' on gate grant three times, after which it is made to accept an integer on gate mes_in. The system responds with the resulting behaviour expression and states that the sequence submitted is valid.

Note that the sequence is shown in our internal CCS* representation, in which (among other things) ? and ! are represented by $ and #, respectively.

```
| ?- call_validator(receiver(succ(zero)),
                    [[grant,[#,succ(succ(succ(succ(succ(zero))))),int]]
                    ,[grant,[#,succ(succ(succ(succ(succ(zero))))),int]]
                    ,[grant,[#,succ(succ(succ(succ(succ(zero))))),int]]
                    ,[mes_in,[$,mes,int]]],B11).

Initial behavior expression :
receiver(succ(zero))

The resulting behavior expression is :
```

```
(((check(succ(zero),ms))[[d],[dl]])
 || (dl?new_expect:int;(receiver(new_expect)))
)\[dl]

 Valid sequence
```

Other actions, such as the ones in process check, could not have been tried, because they involve the evaluation of guards on values that are not known. In order to try such actions, it would be necessary to compose check with another process capable of providing these values.

If the users so prefer, they can select an option where they provide the list of the interactions with the environment only. In this case, the system will automatically generate all possible sequences of intermediate internal actions.

Generating sequences of interactions

It was mentioned above that our LOTOS interpreter can be used in order to systematically generate the tree of all possible sequences of actions for a CCS* specification as defined by the inference rules. When one deals with systems of realistic size, however, random exploration may be more practical.[31] To implement a non-deterministic choice between two or more rules, these rules are combined in a compound rule and one of them is chosen at run-time by generating a random number. Furthermore, since a process might loop, a mechanism for limiting the number of calls to a procedure is included.

Below, a random selection of increasingly longer sequences generated for service is shown. Credits are passed from receiver to sender, and then crazyfifo loses the first two messages.

```
| ?- call_generator(service).
 Initial behaviour expression :
   service
 Generation starts :

 Applied action sequence :
            < go!go_signal:signal >

 Applied action sequence :
            < go!go_signal:signal >
            < i >  : cred2 ?![succ(succ(succ(succ(succ(zero)))))]

 Applied action sequence :
            < go!go_signal:signal >
            < i >  : cred2 ?![succ(succ(succ(succ(succ(zero)))))]
            < i >  : cred1 ?![succ(succ(succ(succ(succ(zero)))))]

                . . . .

 Applied action sequence :
            < go!go_signal:signal >
            < i >  : cred2 ?![succ(succ(succ(succ(succ(zero)))))]
            < i >  : cred1 ?![succ(succ(succ(succ(succ(zero)))))]
            < i >  : mes1 ?![succ(zero)]
            < i >
            < i >  : mes1 ?![succ(succ(zero))]

                . . . .
```

```
Applied action sequence :
          < go!go_signal:signal >
          < i >  : cred2 ?![succ(succ(succ(succ(succ(zero)))))]
          < i >  : cred1 ?![succ(succ(succ(succ(succ(zero)))))]
          < i >  : mes1 ?![succ(zero)]
          < i >
          < i >  : mes1 ?![succ(succ(zero))]
          < i >
          < i >  : mes1 ?![succ(succ(succ(zero)))]

             . . . .
```

Another way of selecting consists of directing the execution of a process by giving for each execution step a number indicating a particular choice. This is similar to the method discussed in the next section; however, the selection is fixed by the user at the beginning, rather than during execution. This method of operating will not be illustrated.

Simulation (the 'one-stepper')

Interactive execution is made possible by displaying the set of all possible next actions a process can execute at each step, and requesting a choice. This is done by generating the set of possible actions linked by an inference to a given behaviour expression. This method of step-by-step user-controlled execution was chosen to make it possible for the user to have full control on the execution by selecting the branches of interest. Also, in this way the user is able to prevent the system from running into infinite loops such as those possible in cases such as P := i; P + Q. The one-stepper is the most commonly used option of our system.

Three types of choices are possible: interactions with the environment, internal communications, and the internal action i. In the first case, the user would be prompted with something such as

```
gate !succ(zero):int ?x:bool
```

for which a possible valid user answer would be

```
gate ?y:int !true:bool
```

An example of the second case would be

```
i [gate ?! zero]
```

for an internal communication where the value zero is agreed on.

Once the list of possible actions is displayed, the user chooses one of them. The first few steps of simulation of our service provider example follow. User responses are preceded by |: (of course, other choices could have been taken, leading to a different simulation).

```
Choose an action or a command:
1 : go!go_signal:signal

|: 1>go!go_signal:signal.          (* user gives starting signal
 Ok ...
```

```
Choose an action or a command:
1 : i [cred2?![succ(succ(succ(succ(succ(zero)))))]]
     <-> Between: crazyfifo and receiver

|: 1.                                  (* receiver grants credit of 5
 Ok ...

Choose an action or a command:
1 : i [cred1?![succ(succ(succ(succ(succ(zero)))))]]
     <-> Between: crazyfifo and sender

|: 1.                                  (* sender now has 5 credits
 Ok ...

Choose an action or a command:
1 : i [mes1?![succ(zero)]]
     <-> Between: sender and crazyfifo
2 : i [cred2?![succ(succ(succ(succ(succ(zero)))))]]
     <-> Between: crazyfifo and receiver
|: 1.                                  (* sender sends message 1
 Ok ...

Choose an action or a command:
1 : i
2 : i [mes1?![succ(succ(zero))]]
     <-> Between: sender and crazyfifo
3 : i [mes2?![succ(zero)]]
     <-> Between: crazyfifo and receiver
4 : i [cred2?![succ(succ(succ(succ(succ(zero)))))]]
     <-> Between: crazyfifo and receiver

|: 1.                                  (* user decides loss of message 1
 Ok ...
```

Alternatively, the user can ask the system to make a random choice between several alternatives not involving interactions with the environment. This can be done for a single step, or for a series of steps up to a specified maximum.[12]

The system normally saves all the behaviour expressions derived during the simulation process. At any point, the user can ask the system to print a list of these expressions, and then can direct the system to return to any of them in order to select a different choice. Alternatively, the user can bypass this mechanism, and ask the system to save only the behaviour expressions of interest.

In all cases, the event sequences are recorded to remember the execution history.

Yet another option available allows the user to declare dynamically during execution new behaviour expressions to be connected in parallel with the current one to play the role of the environment.

The system provides some twenty-five commands. A list of them is given in Reference 27.

It was by using the simulator that the possibility of a deadlock in our service provider was discovered. Notice that the receiver can decide to send credits, and crazyfifo may attempt to transmit them, even if the sender is not ready to accept them. In such a situation, crazyfifo and sender cannot co-operate, because the first will be attempting to deliver credits, whereas the second will be attempting to send messages. An easy way to eliminate this deadlock is to remove the guard [cred = 0] in the sender so that the latter is always ready to accept credits.

CONCLUSIONS AND CURRENT WORK

Although the ideas in LOTOS and in our interpreter are not new, their combination is new. The language combines two powerful specification techniques. It was specified semantics-first and its semantics are formal and constructive. Further, the language has already proved its usefulness, since extensive specifications are being written in it within the OSI project (among others, the Transport layer[8] and the Session layer[32]).

By using the interpreter, a LOTOS specification becomes a prototype of the system specified. This prototype can then be exercised in several ways: to validate interaction sequences, to generate sequences of interactions, and in interactive simulation. This helps increasing the specifier's confidence that the specification is correct, and allows early detection of specification errors.

The issue of using our system for testing or verification is currently being investigated. In order to make our prototype really useful for these purposes, several features must be added. Several such features have been considered and most of them would not be difficult to add, due to the high-level approach followed.

It is possible to define LOTOS operational semantics in terms of LOTOS constructs directly, rather than in terms of CCS*.[33] This approach, which has been adopted by the forthcoming LOTOS Draft Proposal, suggests an interpreter that does not need the intermediate step of translation into CCS*. Such an interpreter would be more convenient to use because it removes the need to become acquainted with CCS*, which exists for users wishing to take full advantage of our system. We are already working on an interpreter that follows this philosophy. Its structure is very similar to the structure of the interpreter discussed in this paper, but CCS* is replaced by an internal, Prolog-oriented representation of the LOTOS source. The user will never see the latter, because it can be easily translated back into LOTOS. At the same time, we are extending the system's capability to execute more general sets of ADT axioms.[34]

ACKNOWLEDGEMENTS

LOTOS is the product of an ISO committee, called ISO/TC97/SC21/WG1/FDT Ad Hoc Group/Subgroup C. We are grateful to the colleagues of the group, and especially to its two successive chairmen, Chris Vissers and Ed Brinksma, for the collaboration that has led to LOTOS and for many useful discussions. We are also grateful to Gregor v. Bochmann, Robert L. Probert, Hasan Ural and Peter Van Eijk for their interest and useful suggestions. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, by the Canadian Department of Communications, and by the Canadian Department of National Defense. We are also indebted to Digital Equipment Corporation of Canada for an equipment grant.

REFERENCES

1. R. L. Schwartz and P. M. Melliar-Smith, 'From state machines to temporal logic: specification methods for protocol standards', *IEEE Trans. Comm.*, **COM–30**, 2486–2496 (1982).
2. R. Tenney, 'Specification technique', in *Formal Description Techniques for Network Protocols*, Report No. ICST/HLNP–80–3, National Bureau of Standards, June 1980.
3. L. Logrippo, 'Constructive and executable specifications of protocol services by using abstract data types and finite-state transducers', in H. Rudin and C. H. West (eds), *Protocol Specification, Testing, and Verification III*, North-Holland, 1983, pp. 111–124.
4. L. Logrippo, D. Simon and H. Ural, 'Executable description of the OSI transport service in Prolog, in Y. Yemini, R. Strom and S. Yemini (eds), *Protocol Specification, Testing and Verification IV*, North-Holland, 1985, pp. 279–293.

5. D. P. Sidhu, 'Protocol verification via executable logic specifications', in H. Rudin and C. H. West (eds), *Protocol Specification, Testing, and Verification, III*, North-Holland, Amsterdam, 1983, pp. 431–436.
6. H. Zimmermann, 'OSI reference model — the ISO model of architecture for open systems interconnection', *IEEE Trans. Comm.*, **COM-28**, (4), 425–432 (1980).
7. E. Brinksma, 'A tutorial on LOTOS', in M. Diaz (ed.), *Protocol Specification, Testing, and Verification, V*, North-Holland, Amsterdam, 1986, pp. 171–194.
8. Scollo, G., G. Pappalardo, L. Logrippo and E. Brinksma, 'The OSI transport service and its formal description in LOTOS', in L. Csaba, K. Tarnay and T. Szentivanyi (eds), *Computer Network Usage: Recent Experiences*, North-Holland, 1986, pp. 465–488.
9. V. Carchiolo, A. Faro and G. Scollo, 'A temporal ordering specification of some session services', in *Communications Architectures & Protocols, Proceedings of the 1984 SIGCOMM Conference*, pp. 107–114.
10. International Organization for Standardization (ISO), *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique based on the Temporal Ordering of Observational Behavior (DP 8807)*, March 1985.
11. P. Van Eijk, 'A comparison of behavioural language simulators', in B. Sarikaya and G.v. Bochmann (eds), *Protocol Specification, Testing, and Verification VI*, North-Holland, 1987, pp. 85–96.
12. B. Berthomieu, 'Le langage CCS et son interprete; une implantation experimentale de CCS', *Rapport Technique CNRS-LAAS*, Toulouse, 1985.
13. G. Karjoth, 'An interactive system for the analysis of communicating processes', in M. Diaz (ed.), *Protocol Specification, Testing and Verification V*, North-Holland, Amsterdam, 1986, pp. 91–102.
14. P. Van Eijk, 'Software tools for the design of protocol systems, part I', *Memorandum INF-85-21*, Technical University of Twente, Department of Informatics.
15. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications* 1, Springer-Verlag, Berlin, 1985.
16. R. Milner, 'A calculus of communicating systems', Lecture Notes in Computer Science, Springer-Verlag, 1980.
17. V. Carchiolo, A. Faro, O. Mirabella, G. Pappalardo and G. Scollo, 'A LOTOS specification of the PROWAY highway service', *IEEE Trans. Computers*, **C–35**, 949–968 (1986).
18. C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
19. J. Guttag, E. Horowitz and D. Musser, 'Abstract data types and software validation', *Comm. ACM*, **21**, 1048–1064 (1978).
20. G. Huet, 'Confluent reductions: abstract properties and applications to term rewriting systems', *18th IEEE Symposium on Foundations of Computer Science*, 1977, pp. 30–45.
21. R. Hansen, D. Schmidt and D. Sidhu, 'A critique of LOTOS', *ISO/TC97/SC21/WG1/N51*, 1985.
22. B. W. Kernighan and R. Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, 1984.
23. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1984.
24. B. Melese, V. Migot and D. Verove, 'The Mentor — V5 Documentation', *Rapport Technique INRIA, No. 43*, January 1985.
25. D. E. Knuth and P. B. Bendix, 'Simple word problems in abstract algebra', in J. Leech (ed.), *Computational Problems in Abstract Algebra*, Pergamon Press, 1969, pp. 263–297.
26. A. Obaid, 'Application of the inference rules of CCS and LOTOS', *Technical Report TR–85–14*, University of Ottawa, Department of Computer Science.
27. A. Obaid, 'SINAPS: a simulator of communicating systems', *Technical Report TR–86–14*, University of Ottawa, Department of Computer Science, 1986.
28. J. P. Briand, M. C. Fehri, L. Logrippo and A. Obaid, 'Executing LOTOS specifications', in B. Sarikaya and G.v. Bochmann (eds), *Protocol Specification, Testing, and Verification VI*, North-Holland, 1987, pp. 73–84.
29. J. P. Briand, M. C. Fehri, L. Logrippo and A. Obaid, 'Structure of a LOTOS interpreter'. *Communications Architectures and Protocols. SIGCOMM '86 Symposium*, pp. 167–175.
30. U. Ural and R. L. Probert, 'Step-wise validation of communications protocols and services', *Computer Networks and ISDN Systems*, **11**, 183–202 (1986).
31. C. H. West, 'Protocol validation by random state exploration', in B. Sarikaya and G.v. Bochmann (eds), *Protocol Specification, Testing, and Verification VI*, North-Holland, 1987, pp. 233–242.
32. G. Scollo and M. Van Sinderen, 'On the architectural design of the formal specification of the session standards in LOTOS', in B. Sarikaya and G.v. Bochmann (eds), *Protocol Specification, Testing, and Verification VI*, North-Holland, 1987, pp. 3–14.

33. T. Bolognesi, R. De Nicola and D. Latella, 'Elements for the revised draft proposal', *Unpublished Document (Document LOTOS/85/26*, November 1985.
34. R. Forgaard, 'A program for generating and analyzing term rewriting systems', *Master's Thesis*, MIT Lab. for Computer Science, Cambridge, Mass., 1984.