Executable LOTOS:
Using PARLOG to implement an FDT ‡

David Gilbert

PARLOG Group
Department of Computing
Imperial College of Science and Technology
180, Queen's Gate
LONDON SW7 2BZ
UK

Net mail: drg@doc.ic.ac.uk

A practical investigation of the feasibility of implementing core constructs of the LOTOS specification language in the PARLOG programming language has been made. A program has been written in PARLOG to translate simple LOTOS specifications into executable PARLOG code. A single set of these specifications is used to illustrate the techniques employed to re-express LOTOS constructs as PARLOG procedures and processes. The translator is based on the LOTOS BNF. A large subset of LOTOS constructs which can be mapped on to PARLOG is handled by the translator.

1. Introduction

1.1. Aims

The aims of the research reported here were:

(i) to carry out a practical investigation of the feasibility of using the PARLOG programming language to implement core constructs of the LOTOS specification language.

(ii) to write a translator program which would accept simple LOTOS specifications and generate executable PARLOG code.

1.2. Motivation

Information systems are often characterised by concurrency and non-determinism, and there exist a variety of methods for specifying them. A desirable property of such specifications is that they are succinct, can be analysed mathematically, and permit the rapid prototyping of the product. These prototypes may not be very efficient, but they can be used to evaluate whether the system performs in the way that the specification intended. This work accepts as a basis LOTOS as one such specification method. LOTOS is the Formal Description Technique (FDT) currently being developed by the International Standards Organisation (ISO). It is being applied internationally now, and its use in specifying the ISO Reference Model for Open Systems Interconnection (OSI) is likely to increase in the future. Apart from the problem

‡ Presented at the *IFIP Symposium on Protocol Specification, Testing and Verification VII*, Zurich, Switzerland, May 5-8 1987. To be published in the Conference Procedings by Elsevier Science, North-Holland Press.

of ensuring that descriptions written in LOTOS are "correct", there remains the further difficulty of the practical implementation of LOTOS specifications.

LOTOS could be implemented by designing a compiler which would produce executable machine code from the LOTOS source code; this is a long-term project. An alternative approach is to prototype an implementation of LOTOS using an existing computer language. It might well be that such an implementation would result in inefficient programs which execute very slowly: however this technique would allow the testing of protocols in conditions simulating those of the intended application. This would provide a tool to assist the designers of specifications to discover whether the specification had the intended result. Hopefully this interface between specification and implementation via rapid prototyping would result in a more efficient software production cycle.

The approach taken in the work reported here is that of the second alternative, chosing PARLOG as the practical language in which to implement LOTOS. Systems whose properties include communication and concurrency can be specified in LOTOS, and an implementation of such a specification should exhibit these properties. PARLOG is a member of the family of Parallel Logic Programming Languages. It has sound declarative foundations based on Horn Clause Logic, and incorporates the ability to express concurrency and communication; for all these reasons it makes an excellent candidate as an implementation language for LOTOS. PARLOG programs can be written without regard to the underlying hardware on which they are to be run; hence they may be executed on whatever machine the language has been implemented, either monoprocessor, multiprocessor or a distributed system, unlike most existing PROLOG implementations.

Work has been carried out previously to create executable descriptions of OSI services and protocols using PROLOG as the implementation vehicle. Protocol verification using executable logic specifications in PROLOG has been reported in [SIDHU 1983], and the OSI Transport Service has been specified in PROLOG to create an executable description [LOGRIPPO et al 1985]. More recent work [BRIAND et al 1986] reports a prototype LOTOS interpreter implemented in YACC/LEX, C and PROLOG. The properties of PROLOG allow the interpreter to be used for validation of interaction sequences, and generation of such sequences either randomly or under user control. In practice, the advantage that PROLOG possesses in using backtracking for the generation of sequences is offset by problems of efficiency.

The advantage that PARLOG possesses over PROLOG as an implementation language for LOTOS is that it directly supports non-determinism and concurrency. However, the committed choice feature of PARLOG is not conducive to direct generation of all possible interaction sequences, and an implementation of a LOTOS specification in PARLOG would be more suited to providing a runnable version of the specification. For example, the PARLOG implementation of a gateway protocol could be run on a multiprocessor, allowing the development of the rest of the system to proceed.

2. LOTOS, a Formal Description Technique

The name LOTOS is an acronym for Language Of Temporal Ordering Specification, and is intended to convey the fact that LOTOS is a formal specification language, belonging to the class of Formal Description Techniques (FDT). LOTOS is currently defined in [ISO 1987] which contains a tutorial introduction to the language. It has been used to specify portions of the ISO-OSI model, mostly at the Transport and Service layer levels [SCOLLO et al 1985], [TOCHER 1985], [ISO 1986a], [ISO 1986b], [ISO 1986c].

We concentrate on that part of the overall LOTOS language which is used for describing the dynamic aspects of processes, henceforth referred to as "Dynamic LOTOS" in this work. Although the semantics of Dynamic LOTOS is based on Milner's Calculus of Communicating Systems (CCS) [MILNER 1979], many ideas from Hoare's Communicating Sequential Processes (CSP) [HOARE 1985] are incorporated.

The ADT part of the LOTOS language does interact with Dynamic LOTOS in the areas of "sort-checking" and synchronisation conditions. Although we excluded an implementation of the full ADT portion for lack of time, an outline of methods which could be used to accomplish this is to be found in [GILBERT 1986]. The part played by sort-matching in synchronisation was incorporated into our implementation, and is

discussed below.

We base our illustrative LOTOS examples on a subset of the specifications to be found in the LOTOS Tutorial [ISO 1987]:

Example 1

```
process buffer [ in-data, out-data ] :=
        in-data ? x : data ;
        out-data ! x ;
        buffer [ in-data , out-data ]
endproc
```

Example 2

```
process two-slot-buffer [ in , out ] :=
        hide middle in
                buffer [ in , middle ]
                | [ middle ] |
                buffer [ middle , out ]
endproc
```

Example 3

```
process duplex-buffer [ in-a , in-b , out-a , out-b ] :=
        buffer [ in-a , out-a ]
        |||
        buffer [ in-b , out-b ]
endproc
```

The definition of LOTOS has been subject to regular revisions in the past, especially with regard to its semantics, and recently [ISO 1987] a semantics has been given to the language with respect to the parallel and associated operators which is accordance with Milner's latest work [MILNER 1986]. The work reported here was undertaken with the earlier definition [ISO/BSI 1986], but we include a discussion of the effects of the recent changes on our implementation.

3. PARLOG

PARLOG is a logic programming language, and its development was influenced by inputs from both Functional and Logic Programming. This section assumes that the reader has familiarity with the general concepts of logic programming, and specifically a knowledge of PROLOG. Introductory reading to logic programming may be found in [HOGGER 1984] and [KOWALSKI 1983]; standard references for PARLOG are [CLARK and GREGORY 1986] and [GREGORY 1987].

3.1. PARLOG as a Parallel Logic programming language

PARLOG is a language which belongs to the family of committed choice parallel logic programming languages. It can explicitly express both OR and stream-AND parallelism. Committed choice non-determinism is implemented by the use of guards which ensure that committal is made to only one clause, unlike the backtracking ability of PROLOG. These features enable the language to use non-determinism as an evaluation strategy if required by the programmer, and permit the writing of programs which require the separate use of both sequential and parallel evaluations.

The PARLOG syntax used in this article is given in the Table below. Note that variable names start with a capital letter in the examples presented in this paper.

Table 1 PARLOG syntax

box , center,tab (%) ; cB | cB cB10 | l. Symbol%Meaning =
<-%logical implication _ &%sequential-AND _ %parallel-AND _ ;%sequential-OR _ _ :%guard operator _ _ ^%output mode annotation

Mode declarations are used to specify communication constraints on shared variables, which are declared to be either **?** ("input") or ^ ("output"), thus acting as communication channels. These declarations are made once for each relation, and each argument of the relation is annotated:

mode name(a1?,..,ak^,..)

Messages sent along these channels are incrementally constructed from partially determined data structures, usually consisting of lists of terms acting as message streams.

The general form of a PARLOG clause is:

<head> <- <guard> : <body>

Note that <head> is in the form *name(a1,..,ak)* , where *name* is the relation name, and *a1,..,ak* its arguments. The logical implication symbol is ' <-' and ':' the guard operator. Both the guard and the body can be a conjunction of calls, or empty, and the calls are separated by the sequential-AND or parallel-AND operators. A clause is a candidate for evaluation if both input matching in the head and the evaluation of the guard succeeds, whereas in a non-candidate clause either of these fail. A clause can be suspended if either the input matching or guard evaluation suspend waiting for an input variable to become instantiated. A suspended call may eventually become either candidate or non-candidate. Note that no output bindings are made until committal has been made to the clause (ie the guard conditions are satisfied), and committal may be made to only one clause of a procedure.

The search strategy employed by PARLOG is different to that of PROLOG. Firstly the language does not employ automatic left-to-right evaluation, calls in an AND-conjunction being evaluated concurrently. Secondly, a depth-first strategy is not used to explore the search tree, but rather all possible paths are explored concurrently and the first successful answer terminates the search. Finally, if the parallel-OR operator is specified, clauses within a procedure are not explored in the strict textual order in which they are encountered: PARLOG will choose one candidate clause at random to evaluate, and only pursue the paths which arise from that computation. If all the clauses are suspended, then the call itself becomes suspended. The success of a conjunction is dependent on the success of its constituent parts (calls); if none of its calls fails but one is suspended, the conjunction suspends. It should be noted that if one call in a conjunction fails, the evaluation of all the calls in that conjunction must be terminated. The programmer has control over which clause may be chosen as the candidate by the use of input matching and also guards. Moreover there are explicit sequential AND and OR operators which enable the programmer to specify the order of evaluation if desired.

PARLOG provides a facility for metalevel programming, the *call* primitive, which interprets an argument term as a relation call. This facility exists as both in a single argument and three argument form [GREGORY 1987]. We use the former primitive in the work reported here.

3.2. Stream communication

The committed choice aspect of PARLOG give it the ability to express stream communication. This enables PARLOG to be used constructively in applications where concurrency and process state are of importance. Although in a logic program a relation call cannot actively change state, but only reduce to other calls, a relation which calls itself recursively with different arguments can be viewed as a long lived process which changes its state [GREGORY et al 1985]. Since PARLOG allows more than one call to be evaluated concurrently, the language permits computations which comprise the execution of several

concurrent processes.

The basic paradigm of stream communication is that of producers and consumers. PARLOG's ability to perform stream-AND-parallelism gives the language the power to express stream communication in a very simple manner. Shared variables act as communication channels, and messages can be sent by the incremental construction of partly instantiated data structures such as tuples or lists of terms. This incremental communication has an analogy with the lazy and eager parallel evaluation of functional programs. Output variables are produced in an asynchronous manner, but input variables are processed synchronously if an argument contains a partially instantiated term. Mode declarations mean that in a situation involving communication only one PARLOG process can be the producer, binding shared variables, whilst there may be one or more consumers of the communication stream.

An outline program which illustrates this form of stream communication is:

Example 4

```
mode eager-producer( Stream^ ).
eager-producer( [ Item | Stream ]) <-
  produce( Item ) ,
  eager-producer( Stream ).

mode naive-consumer( Stream? ).
naive-consumer( [ Item | Stream ] ) <-
  consume( Item ) ,
  naive-consumer( Stream ).

mode produce(^).

mode consume(?).
```

Note that we do not give the code for the procedures *produce* and *consume*, as these are not relevant to this particular algorithm; we assume that *produce* is 'eager' (asynchronous). The *eager-producer* process can run ahead of the consumer by an arbitrary amount, and we assume the existence of system buffers to permit this.

The PARLOG programmer is, however, able to utilise the completely synchronous communication facilities offered by the language. This is achieved by *back-communication* which can take two forms.

Firstly, the mode of the producer may be reversed, causing it to become "lazy". In this case, the consumer has the output mode on the shared variable, and the producer is declared input on that argument. The consumer than sends the producer a list of variables to be instantiated as messages; unable to run ahead of the consumer, the producer's eagerness is constrained by that process. Lazy evaluation can produce quite succinct algorithms, but often a considerable amount of transformation has to be made to an "eager" algorithm in order to create a "lazy" version which produces the overall desired result. Moreover, the transformed algorithms may themselves not be so easily understood as the original "eager" versions or those using back-communication. Mode reversal was not a technique employed in the research reported here for these reasons.

The second, analogous to "rendez-vous", is that of the co-operative construction of binding terms. The producer sends a stream (eg a list) of tuples, each of which contains two arguments, one the data item to be sent and the other a variable; the use of the sequential-AND operator then forces the producer to wait until the consumer instantiates the variable to an agreed message using the <= primitive. Note that <= instantiates its left argument, which must be a variable, to the right argument (one way unification). The example code below illustrates this; we use *message(Item,Reply)* for the message tuple, and assume the procedures *produce* and *consume* as above.

Example 5

```
mode synchronous-producer( Streamˆ ).
synchronous-producer( [ Message | Stream ] ) <-
  produce( Item ) ,
  synch-send( Item , Message ) &
  synchronous-producer( Stream ) .

mode synchronous-consumer( Stream? ).
synchronous-consumer( [ Message | Stream ] ) <-
  synch-receive(  Message , Item )  ,
  consume( Item ) ,
  synchronous-consumer( Stream ).

mode synch-send( item? , messageˆ ) .
synch-send( Item , message( Item , Reply ) )  <-
        wait( Reply ) .

mode synch-receive( message? , itemˆ  ) .
synch-receive(  message( Item , Reply ) , Val )  <-
        Val <= Item ,
        Reply <= ok .

mode wait(?).
wait( ok ).
```

As an example, a non-terminating invocation would be:

synchronous-producer(Stream) , synchronous-consumer(Stream).

In the above example, *synchronous-producer* may only produce one message tuple at a time, and is forced to wait until the *Reply* argument has been instantiated by the call to *wait* in *synch-send*. The overall sequencing of the communication is achieved by the sequential-AND operator in *synchronous-consumer* between the call to synch-send and the self recursive call. Note that the input mode on *wait* ensures that the call to this procedure will suspend until its argument is ground, and that *synch-receive* will only ground *Reply* to *ok* after it has output *Item*.

Reformulating in PARLOG the LOTOS algorithms presented earlier, we employ back communication and the cooperative construction of binding terms, and make use of the procedures for *synch-send*, *synch-receive* and *wait* (above). One problem is to fully capture the operational semantics of the original LOTOS specification.

Example 6

```
mode buffer( in-channel? , out-channelˆ ).
buffer( [] , [] ) .
buffer( [ In-Message | In-Channel ] , [ Out-Message | Out-Channel ] )  <-
        synch-receive( In-Message , Item  ) &
        synch-send( Item , Out-Message ) ,
        buffer( In-Channel , Out-Channel ) .
```

The reader should note that the inclusion of the first clause of the *buffer* procedure implements a buffer which terminates after it receives the last message in the *In-Channel*. In this respect it behaves more like the LOTOS definition of a buffer given below, where *eot* indicates an end-of-transmission message:

Example 7

```
process terminating-buffer [ in , out ] :=
      in ? x : data ;
      out ! x ;
      buffer [ in , out ]
   []
      in ? y : eot ;
      exit
endproc
```

Example 8

```
mode two-slot-buffer(in-channel? , out-channelˆ ).
two-slot-buffer( In-Channel , Out-Channel ) <-
        buffer( In-Channel , Middle ) ,
        buffer( Middle , Out-Channel ) .
```

Example 9

```
mode duplex-buffer( in-a? , out-aˆ , in-b? , out-bˆ ).
duplex-buffer( In-a , Out-a , In-b , Out-b ) <-
        buffer( In-a , Out-a ) ,
        buffer( In-b , Out-b ) .
```

One of the major difficulties with the above PARLOG programs is that channel names cannot be parameterised due to the lack of a Renaming facility in the language. Moreover, the derivation of such PARLOG programs from the LOTOS specifications is not straightforward and certainly not easy to mechanise. For these reasons we chose to attempt a more formal translation from LOTOS into PARLOG.

3.3. PARLOG in the light of Milner's interpretations

We note some correspondences between the concepts Milner expresses in his recent paper [MILNER 1986] and those implemented in PARLOG. The interested reader is referred to [BECKMAN 1986] and [BECKMAN, GUSTAVSON and WAERN 1986] for an analysis of concurrent logic programming languages in a CCS formalism, and to [ELLIS 1986] for a discussion of the translation of PARLOG into CCS.

The "naive" form of communication in PARLOG (asynchronous producer, synchronous receiver) relies on buffering within the system, and does not conform to Milner's Principle 1; co-operative construction of binding terms is excluded for the same reason that Milner discounts the 'rendez-vous' of Ada. However, reverse-mode communication can be considered to conform to the first principle of Milner if we recognise that in a practical programming language there must be some physical means by which processes can communicate.

Due to the ability of a PARLOG computation to permit the existence of more than one concurrent process, the language may be used in a manner which broadly conforms to Principle 2, although PARLOG computational style is not limited to the process view alone. Because of the semantics of logic programs, a PARLOG query which is a conjunction of calls can only succeed when all the calls succeed, and if

programming by side-effects is excluded then Principle 3 is satisfied.

Milner's Principle 4 (Conjunction) is partly satisfied by PARLOG's stream-AND operator when considering the synchronous communication expressible in the language. However the language only supports a 1 to N form of communication (one producer and many consumers). Moreover, multi-way synchronisation is not the natural paradigm in the language, and it must be enforced by programming techniques, hence making it difficult to incorporate Milner's Principle 9 (Simultaneity) into the analysis of PARLOG. However, Principle 9 is supported to the extent that a PARLOG process may synchronise with more than one other process by the use of several communication channels. Encapsulation (Principle 5) is achieved to the extent that the scope rules of Logic Programming which ensure that calls made within the body of one clause are not "visible" to a procedure which calls that clause. Disjunction (Principle 6) is reflected by the parallel-OR operator in PARLOG, which coupled with the use of guards and input matching can be used to ensure either exclusive clause choice or to allow the system to arbitrarily select one clause for committal if the choice is non- exclusive. Principle 7 (Renaming) is not realisable in PARLOG. Naturally, as a language orientated towards system programming, PARLOG incorporates explicit sequencing operators to allow for such a control where needed, for example in enforcing synchronousity or in i/o operations (Principle 8).

4. Implementing LOTOS in PARLOG

4.1. Source-to-source translation: preliminary considerations

The method used to translate one high level computer language into another may be broadly described by the label "source-to-source translation". LOTOS is very similar to a conventional computer language in its overall design, and is thus a candidate for such a translation process.

One technique that can be used to effect source-to-source translation is *compiler-like translation*. This consists of the derivation from algorithm A in language L, of an algorithm A´ in language L´, each distinct component of A´ being semantically equivalent to the corresponding component in A. In its most low-level form, constructs from the original language are mapped directly into constructs in the target language. One of the advantages of this method is that a "building-block" approach to translation can be used. Little or no "intelligence" is required of the translator process once the low level components have been successfully mapped from A to A´. Implementation of the translator itself as a computer program is thus facilitated. For the reasons given above, we employed compiler-like translation as the method of implementing LOTOS specifications in PARLOG.

4.2. Source-to-source translation: the LOTOS - PARLOG problem.

An initial examination of the syntax and semantics of LOTOS and PARLOG suggests that there might exist some ready similarities between the two languages, making the translation from the former to the latter relatively simple. This is indicated by a comparison of the operators from both languages:

Table 2 Initial correspondences between the operators of LOTOS and PARLOG

center , box ,tab (%) ; cB s| cB s lB l| lB l. LOTOS%PARLOG = ;%action-prefix operator%&%sequential-AND _ >>%enable%&%sequential-AND _ ||%% |[..]|%parallel operator%,%parallel-AND |||%% _ []%choice% %OR-search _ [..]->%guarded expression%:%guard-conditions _ [..]%selection predicate%:%guard-conditions _ l s| l s. process abstraction%procedure declaration _ process instantiation%procedure call

However, such a naive approach possesses many difficulties which originate from the different semantics of the languages. The major problem in translating from LOTOS to PARLOG is that the former language draws heavily on constructs from both CCS and CSP, whilst PARLOG originated as a logic language. LOTOS has inherited constructs from its "parent" languages which are not immediately translatable into PARLOG. Examples are the LOTOS choice and parallel operators, which do not correspond directly to the PARLOG operators bearing the same names, and atomic events, which do not exist in PARLOG. The major work of creating a source-to-source translator from LOTOS to PARLOG lies in the definition of PARLOG algorithms which mirror those constructs.

4.3. Compiler-like translation.

In this section we briefly discuss the implementation of PARLOG process which emulate basic LOTOS constructs.

The basic building block of a LOTOS specification is the *atomic event* which is used to express the synchronised interaction between communicating processes. Thus the atomic event is the first item in the repertoire of LOTOS that requires the attention of the designer of a translation system. Closely associated with atomic events are event gates, or *interaction points*, referred to in the literature on LOTOS as abstract resources shared by processes [ISO/BSI 1986]. In implementing LOTOS specifications in PARLOG, these abstract resources must be represented as PARLOG processes whose run-time behaviour simulates the properties of those resources. This in turn implies that the processes derived from the concept of interaction points will be quite complex. A run-time system incorporating these is necessary to enable LOTOS specifications implemented in PARLOG to be executed.

A complete implementation of LOTOS in PARLOG must also implement operators from the former language as processes in the target language if there is no direct correspondence between operators in both languages. While LOTOS's sequential operators, action-prefix and enable, are equivalent to PARLOG's sequential-AND operator, difficulties exist in the implementation the choice, parallel and disable operators.

Since ACT-ONE was not a target for implementation in PARLOG, the existence of sort definitions has been taken for granted in this work.

Event offers and *synchronisation* in LOTOS were implemented in PARLOG by the technique of back-communication using the co-operative construction of binding terms, resulting in code which is easy to reason about due to the declarative nature of PARLOG. A *signal* program produces a message tuple of the form:

message(send(Gate,Val,Sort,Condition), Reply , Ok)

or

message(receive(Gate,Val,Sort,Condition) , Reply , Ok)

The *parallel operators* are mapped into a *par* program, written in PARLOG, which emulates the right-bracketing properties of the operators defined in [ISO/BSI 1986] and incorporates hiding and selection of gates. Changes can be made to the *par* process to incorporate modifications to the operational syntax of the parallel operators, for example the latest definition of the parallel operator which is associative and does not automatically include hiding [ISO 1987]. Each PARLOG process representing a LOTOS behaviour expression produces a stream of synchronisation signals which are available to an environment, usually represented as a *par* process. Every *par* process consumes two streams of synchronisation signals (produced either by processes representing behaviour expressions, or by other *par* processes), and produces a stream of synchronisation signals.

The mode declaration of the *par* program is:

mode par(instream-1? , instream-2? , selected-gates? , hidden-gates? , out-streamˆ)

Messages with gate names not in the selected or hidden lists are immediately exported via the *out-stream*. Those with gate names on the hidden list may never be exported to the environment via the *out-stream*, and those not on the selected list can never be candidates for matching in that particular *par* process.

All three synchronisation classes of LOTOS are supported in the implementation, as are selection predicates. The *par* process consumes two message streams and performs matching of signals according to the synchronisation classes. The selection predicates of LOTOS are translated as condition relations, (*Condition* in the message tuples) which are evaluated in the *par* process using PARLOG's *call* primitive. *Val* is instantiated in the case of the *SEND* tuple, but uninstantiated in the *RECEIVE* tuple when these tuples are produced by *signal*. In the case of a SEND-RECEIVE match, the value of the RECEIVE *Val* is instantiated to that of the SEND *Val*. A SEND-SEND match requires the two *Val*s to evaluate to the same value, whilst in a RECEIVE-RECEIVE match, a common value is negotiated according to the restrictions imposed by the *Conditions* of each tuple. Moreover only if the gate names and sort names of two signals match and their condition relations evaluate to **true** will any match be successful and synchronisation take place. *Ok* is the synchronisation variable on which the *signal* process suspends until it is successfully ground to the constant *ok* by the matching process.

Process abstraction was mapped on to PARLOG relation declaration, and instantiation of processes on to goal calls. Relations are parameterised by gate-names, value parameters and export lists as in LOTOS. Thus the buffer process would be translated as:

Example 10

```
mode buffer( gates? , value-parameters? , exports^ , stream^ )
buffer( [ In , Out ] , [] , [] , [ Receive , Send | Stream] ) <-
  signal( receive( In , Val , data , true ), Receive) &
  signal( send( Out , Val , data , true), Send ) &
  buffer( [ In , Out ] , [] , Exports , Stream).
```

Note that since the original LOTOS specification did not use selection predicates, these appear as *true* in the translation. The only procedure which produces signals not included in the message stream is that representing the *stop* process in LOTOS.

The *two-slot-buffer* is translated thus:

Example 11

```
mode two-slot-buffer( gates? , value-parameters? , exports^, stream).
two-slot-buffer( [ In, Out] , [] , [] , Stream-out) <-
  buffer( [ In , middle ] , [] , Exports-1, Stream-1) ,
  buffer( [ middle , Out] , [] , Exports-2, Stream-2) ,
  par( Stream-1, Stream-2, [middle], [middle], Stream-out).
```

Note that the *middle* gate is both selected and hidden as in the original specification. In contrast, the LOTOS *duplex-buffer* specification is translated as:

Example 12

```
mode duplex-buffer( gates? , value-params? , exports^ , stream^)
duplex-buffer( [ In-a , In-b , Out-a , Out-b ] , [] , [] , Stream-out) <-
      buffer( [ In-a , Out-a ] , [] , Exports-1 , Stream-1 ),
      buffer( [ In-b , Out-b ] , [] , Exports-2, Stream-1 ) ,
      par( Stream-1 , Stream-1 , [] , [] , Stream-out) .
```

In this program, the empty lists in the third and fourth arguments of *par* indicate that no selection or hiding of gate names has been made.

The *action-prefix* and *enable* operators of LOTOS are supported. They map directly onto PARLOG's sequential-AND operator. Successful termination (*exit*) maps onto PARLOG's successful return from a query. The export of values on exit from a process is implemented by listing the relevant items in the export-list argument of the PARLOG predicate head.

LOTOS's *choice* operator is not implemented using PARLOG's clause search because the use of PARLOG guards does not produce the output binding necessary for the implementation of synchronisation chosen. Instead, a list of synchronisation choice messages is sent in the message tuple to the *par* process which matches them with those from the corresponding process in the communication. The originator is informed which tuple successfully matched via the *Result* argument of the message tuple. Thus the LOTOS expression:

a ! 3 ; **exit** [] b ? x : nat [x < 5] ; **exit**

would produce a choice list in the message tuple of the form:

message([send(a , 3 , nat , true) , receive(b , X , nat , less(X , 5))] , Result , Ok)

Guarded expressions are implemented by PARLOG guards; when used in combination with the choice operator they are mapped onto guard conditions and OR-parallel search in PARLOG.

Internal events are implemented as synchronisation signals which are always satisfied in the match procedure, but leave the corresponding matching signal to be retried for matching against subsequent signals. In this way, the use of internal events and the choice operator in LOTOS to describe combinations of deterministic and non-deterministic choice can be mapped into PARLOG.

The *disable operator* is not directly mapped into PARLOG constructs; algorithms using this operator must be rewritten using the choice and sequential operators using the expansion method described in the Provisional LOTOS Tutorial [ISO 1986a]. An example expansion is from :

a ; b ; **exit** [> c ; **exit**

to:

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

Disruption of infinite behaviours specified recursively may be expanded by the same method.

5. The Translator

The translator was implemented for a substantial subset of the mappings using PARLOG as the language to write all the necessary modules. It incorporates a tokeniser and parser which together conform to the BNF of LOTOS that was available at the time. Future changes to the BNF can easily be incorporated into the tokeniser and parser since they are written in PARLOG itself. The parser itself was hand-written, but it could be generated using a PARLOG program. The code generator does not implement ACT-ONE and thus does not perform translation-time type-checking. Moreover some of the mappings described in the previous section are not incorporated in the code generator, notably the choice operator in any context other than guarded expressions. This is because of the difficulty of determining at which level of process nesting the choice is offered [GILBERT 1986]. A run-time environment to support block-structured declarations of variables is not implemented: the LOTOS programmer is encouraged to declare explicitly as input all variables needed in any process.

6. Conclusion

LOTOS and PARLOG are languages which have different theoretical bases. The former is a Formal Description Technique with the ability to utilise Abstract Data Types (ACT-ONE), and inherits a formalism derived from CCS and CSP (Dynamic LOTOS). PARLOG is a Parallel Logic Programming language with foundations in Horn Clause logic and influenced by ideas from functional programming.

LOTOS possesses the ability to specify communication systems using general methods which can be adapted to specific situations. It relies heavily on the concept of event-gates. The use of these as input parameters to process instantiations facilitates the construction of systems built out of communicating processes.

The stream communication model employed by PARLOG requires the programmer to explicitly link processes which intercommunicate, linkage being achieved by the use of the logical variable. In order to achieve the flexibility inherent in LOTOS, a PARLOG implementation must create complex software entities which perform the redirection and synchronisation of message streams. These entities are implemented as PARLOG relations. They can cause the execution of the PARLOG code to be inefficient as the messages in the synchronisation streams are routed to the intended destination. The inefficiency is exacerbated by the complex operational semantics of some of the LOTOS operators when used in combination, notably the parallel and hiding operators. However, the question of inefficiency could be overcome by implementing these relations as PARLOG primitives.

Despite the apparent differences between the two languages, mappings into PARLOG from some core constructs in Dynamic LOTOS were made successfully, and a translator implemented. The declarative nature of PARLOG made the accomplishment of these these tasks relatively straightforward, and changes in the semantics of LOTOS are easy to incorporate into the system. We used the software described to translate some simple LOTOS specifications into the PARLOG system. The system is not yet fully developed, and would be improved by debugging tools and other aids. However, given the complex nature of the task, we are satisfied that such a tool described here is useful in the development of system specifications using LOTOS.

A final remark

The execution of this project has inevitably raised the question of whether a logic language like PARLOG might be used to formulate the original specifications. Some work has already been done in this area [BRODA and GREGORY 1984], [GREGORY et al 1985], [RINGWOOD 1984]. The indications are that PARLOG is suitable for the task, but that considerable further work would need to be done to produce PARLOG specifications of large and complex systems.

Acknowledgements

I would like to thank my supervisor, Graem Ringwood, who contributed many useful ideas, and Rob Neely from ICL, who proposed the investigation. I would also like to express my appreciation to all of the PARLOG group at Imperial College and especially to Steve Gregory for his sound and helpful suggestions. In addition, my sincere thanks go to Chris Hankin of Imperial College and Danny Crookes from Queen's University, Belfast, both of whom gave me a lot of their time and were always willing to answer my questions, and also to David Freestone of British Telecom. This work was carried out while I was in receipt of an SERC grant, studying for the MSc in Computing.

References

Beckman, L. [1986], "Towards a formal semantics for concurrent logic programming languages". In *Third International Conference on Logic Programming*, London, Shapiro E. (ed), Springer-Verlag Lecture Notes in Computer Science, July 1986

Beckman, L. and Gustavson, R. and Waern, A. [1986], "An algebraic model of parallel execution of logic programs". In *Procs Symposium of Logics Comp Sci*, Cambridge June 1986.

Briand, J.P., Fehri, M.C., Logrippo, L. and Obaid, S. [1986], "Executing LOTOS Specifications". In *Protocol Specification, Testing and Verification VI*, B. Sarikaya and G.V. Bochman (eds), pp 73-84, Elsevier Science Publishers (North-Holland) 1986

Broda, K. and Gregory, S. [1984]. "PARLOG for discrete event simulation". Imperial College Research report DOC 84/5

Clark, K.L. and Gregory, S. [1986], "PARLOG: Parallel programming in Logic". In *ACM Transactions on Programming Languages and Systems*, Vol 8, No.1, Jan 1986

Ellis, M. [1986], "A Relational language into CCS". MSc Thesis, Department of Computing, Imperial College, London 1986

Gilbert, D.R. [1986], "Implementing LOTOS in PARLOG". MSc thesis, Department of Computing, Imperial College, London 1986

Gregory, S. [1987], "Parallel Logic Programming in PARLOG". Addison-Wesely 1987

Gregory, S. and Neely, R. and Ringwood, G. [1985], "PARLOG for specification, verification and simulation". In *7th International Symposium on Computer Hardware Description Languages and their Applications*, Tokyo, August 1985.

Hoare, C.A.R. [1985], "Communicating Sequential Processes". Prentice Hall, UK, 1985

Hogger, C. J. [1984], "Introduction to Logic Programming". Academic Press, 1984.

ISO. [1986a], "Provisional LOTOS tutorial". ISO/TC 97/SC 21 N 1539, September 1986

ISO. [1986b], "Formal Specification in LOTOS of ISO 8072" ISO/TC 97/SC 6 N 4395

ISO. [1986c], "Formal Specification in LOTOS of ISO 8073" ISO/TC 97/SC 6 N 4396

ISO. [1987], "LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior". ISO/TC 97/SC 21 N 1573, February 2, 1987

ISO/BSI [1986], "Working document DF 8807 LOTOS/86/1 FDT:278". IST/21/1/3 BSI 86/61840, April 1986

Kowalski, R.A. [1983], "Logic Programming". In *Proceedings of the IFIP Congress 1983*, pp 133-145, 1983.

Logrippo, L., Simon, D. and Ural, H. [1985], "Executable Description of the OSI Transport Service in Prolog". In *Protocol Specification, Testing and Verification IV*, Y. Yemeni, R. Strom and S. Yemeni (eds), pp 279-293, Elsevier Science Publishers (North-Holland) 1985

Milner, R. [1979], "Calculus of communicating systems". Springer-Verlag 1979

Milner, R. [1986], "Process Constructors and Interpretations". In *Proceedings of IFIP 10th International World Computer Congress*, Dublin, September 1-5 1986. Vol 10, pp 507-514, North Holland 1986.

Ringwood, G.A. [1984]. "The Dining Logicians". MSc thesis, Imperial College, 1984

Scollo, G. and Pappalardo, G. and Logrippo, L. and Brinksma, E. [1985], "The OSI Transport service and its formal description in LOTOS". SECDOS/CI/31, 17 June 1985

Sidhu, D. [1983], "Protocol Verification via Executable Logic Specifications". In *Protocol Specification, Testing and Verification III*, H. Rudin and C.H. West (eds), pp 237-248, Elsevier Science Publishers (North-Holland) 1983

Tocher, A.J. [1985], "OSI transport service: a constraint-orientated specification in extended LOTOS". SEDOS/CI/WP/11/1k, ICL, Nov 1985