A LOTOS to PARLOG Translator

David Gilbert

PARLOG Group
Imperial College
London SW7 2BZ
U.K.

email: drg@doc.ic.ac.uk

A translator from a subset LOTOS into PARLOG is described, and the process of its development charted. The use of the software is described, and comments made on the difficulties of implementing a direct translator from LOTOS specifications into executable code.

1. Introduction

The specification of communicating systems has become an important task as the ability to construct such systems has evolved. One class of such systems in particular, namely that of computer networks, has become a subject for such specifications since the role of communications and information interchange is vital to the functioning of new generation technology.

However, the ability to specify highly concurrent systems in a formal manner does not guarantee that the path to the construction of such systems from their specifications is either clear or automatic. This paper describes an attempt to construct an automatic translator from specifications in a formal description language into executable code. The formal language selected as a source was LOTOS [ISO87] and the target was the parallel logic programming language PARLOG [Gregory87].

2. LOTOS as a Formal Description Technique.

LOTOS is a process algebra based language developed by the International Standards Organisation (ISO) as a Formal Description Technique (FDT) to specify OSI protocols and services. It has been accepted as an international standard and will be one of the three techniques which can be used in the the description of the OSI model. The language is based on concepts introduced in CCS [Milner80] for the definition of the dynamic behaviour of processes, and also incorporates an abstract data type language, ACT-ONE [Ehrig83]. This paper reports implementation work performed with the dynamic part of LOTOS, and does not deal with the implementation of ACT-ONE. Tutorials on LOTOS may be found in [ISO87, Bolognesi87].

3. Routes to implementing LOTOS.

LOTOS could be implemented by

(i) extending a simulator to produce an interpreter or even a compiler for the language [van Eijk88];

(ii) using traditional compiler methods to produce executable machine code from the LOTOS source code;

(iii) prototyping an implementation of LOTOS using an existing computer language.

The first method might be implementable in a short time, but the resulting interpreter could well be quite inefficient due to the slow speeds of execution of interpreters in general. There are difficulties involved in the second, because the formalism on which LOTOS is based has little relation to the traditional von-Neumann machine architecture. We chose the last route, and selected PARLOG as the target language due to its ability to express concurrency and the possibility of executing PARLOG code on a multi-processor or distributed system. It might well be that such an implementation would result in inefficient programs which execute very slowly. This technique would, however, allow the testing of protocols in conditions simulating those of the intended application, providing 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.

4. PARLOG

PARLOG is a committed choice parallel logic programming language, and can explicitly express both OR and stream-AND parallelism as well as non-determinism. Unlike Prolog, it does not possess the ability to backtrack, due to its committed choice nature, which is implemented by the use of guards. 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. The language is fully described in [Gregory87], and tutorial material can be found in [Gilbert87]; only a minimal description of the language will be given here.

The syntax of PARLOG, given in the table below, is similar to the emerging standard for Prolog, with the addition of explicit sequential operators, a guard operator, and mode declarations.

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.

A PARLOG procedure is a collection of clauses with the same name *n* and arity *k*, and is referred to in this paper as *n/k*. The general form of a PARLOG clause is:

`<head>` **<-** `<guard>` **:** `<body> <or-op>`

Note that <head> is in the form $name(a_1,..,a_k)$, where *name* is the relation name, and $a_1,..,a_k$ 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 naive form of communication in PARLOG is that of asynchronous producer and synchronous consumer, with channels acting as unbounded buffers. However synchronous communication can be expressed in PARLOG using the co-operative construction of binding terms. The producer sends a stream (an incrementally constructed list) of tuples, each of which contains two arguments, one the data item to be sent and the other a variable; the use of a sequential-AND operator then forces the producer to wait until the consumer instantiates the variable to an agreed message. We illustrate this using the tuple **'+'** to encapsulate the message, with the first argument the message item itself, and the second argument the

synchronisation tuple[1]:

```
Item + Reply
```

We then implement synchronous send and receive in PARLOG by the following programs:

Example 1 Synchronous communication in PARLOG.

```
mode synch_send(item?,tuple^).
synch_send(Item,Item+Reply)<- data(Reply).

mode synch_receive(tuple?,item^).
synch_receive(Item+Reply,Item)<- Reply=ok.
```

Note that *data/1* suspends until its argument is ground (ie the top level functor of a data-structure is instantiated). Thus the query *?data(X)* suspends, and *?data([H|T])* succeeds. The call to *=/2* in *synch_receive/2* instantiates the **Reply** variable to **ok**. These programs can be composed as follows:

? synch_send(item , Msg) , synch_receive(Msg , X).

and may be thought of as roughly equivalent to the LOTOS behaviour expression

```
g ! item  || g ? X
```

5. Source-to-source translation.

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.

An initial examination of the syntax and semantics of LOTOS and PARLOG suggests that there might exist some ready similarities between the two languages. However, detailed investigation revealed that a naive approach of mapping operators from the former language directly to those of PARLOG would not be possible [Gilbert87b]. A description of early work undertaken to create constructs in PARLOG which behave like LOTOS constructs is reported in [Gilbert86], and is summarised in [Gilbert87b]. This work was performed, however, using an early definition of LOTOS which does not permit multi-way synchronisation [ISO/BSI86]. We describe in more detail the actual implementation aspects of that work and reports on the experience gained with the new draft standard for LOTOS [ISO87] which does permit multi-way synchronisation.

6. Development of the translator

Only the "dynamic" part of LOTOS was considered for translation into PARLOG, that is the aspects based on CCS [Milner80]. However, PARLOG does possess 'implicit' types (lists, tuples, strings and integers), enabling the interaction between the ADT and the dynamic part of LOTOS and in the areas of "sort-checking" and synchronisation conditions to be incorporated into our translator to an extent[2].

---

[1] Note that **'+'** is infix operator with arity 2

[2] Type checking in PARLOG is performed at run-time, using both pattern matching and the relations *list/1*, *atom/1* and *number/1*.

The translator was developed by first building the 'primitive building blocks', ie atomic actions, and also more complex behaviour expressions including choice and parallel operators.

6.1. Processes in LOTOS and PARLOG.

LOTOS processes are represented as PARLOG processes in the translator. The latter are coded as recursively defined predicates (mutual or self recursion) and possess many of the properties of LOTOS process definitions, including the ability to import and export parameters, and to express recursion. Parameter lists in LOTOS process abstractions may take the form of gate and value parameters as two import lists, and an export list categorised by sorts.

In order to express the facility for communication with the environment, a fourth (output) argument needs to be included in the PARLOG predicate which allows a stream of synchronisation messages (offers) to be exported. These messages are in effect *event offers*. The mode template of the PARLOG process representing the LOTOS process *name* is:

```
mode  name(gates?,val-params?,exports^,offers^).
```

6.2. Event offers

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 [ISO87]. The most important properties of atomic events are that they are indivisible, are capable of synchronous communication and are parameterised by gate-names.

The three kinds of synchronisation in which LOTOS atomic events can participate are *value matching*, *value passing* and *value generation*. An additional constraint which may be applied to synchronisation is that of *selection-predicates*; if used, synchronisation can only occur if the predicate in each partner evaluates to **true**.

6.2.1. Using one-to-one synchronisation.

The technique of cooperative construction of binding terms is used (see above) to enforce synchronous communication, but has been adapted to permit encoding of the synchronisation classes of LOTOS. The signal tuple employed may be either *send/5* or *receive/5*:

```
send(Gate,Value,Sort,Condition,Reply).
receive(Gate,Value,Sort,Condition,Reply).
```

`Gate` and `Sort` are constants, `Value` corresponds to a LOTOS *value expression* in *send/5* and is a variable in *receive/5* at time of transmission. `Condition` is a boolean expression in each case, and `Reply` is the synchronisation variable.

Although LOTOS does not always refer explicitly to the sort in an expression such as

```
g!x
```

the following alternative description can be used:

```
g!x ofsort message-sort
```

with the implication that the value part of *all* communications possesses a sort, which if apparently absent must nevertheless be presumed to be of a *default sort*. The translator can insert a default sort **true** if required. If the ADT part of LOTOS were to be treated by the translator, a table of sort declarations would enable the sorts of the value in `g!x` to be determined during the translation process.

Similarly, since the selection predicate is optional, it must always be included in the tuple. If not present in the LOTOS specification, the translator can insert **true** as a condition.

6.2.2. Value matching

An example, *value matching*, is presented below:

Example 2 Value matching.

```
mode match( signal1?, signal2?, result1^, result2^).

match(    send(Gate,Val1,Sort,Condition1,Out1),
         send(Gate,Val2,Sort,Condition2,Out2))<-
     Val1 =:= Val2&
     call(Condition1) & call(Condition2):
     Out1 = ok, Out2 = ok.
```

Note that in this case the Gate and Sort names must be identical. We assume in this example that the Sort is Nat, so that the value expressions Val1 and Val2 are evaluated for equality by the predicate =:=/2. A more complex test of equality could be made if sorts other than Nat were allowed. Both Conditions must be evaluated to **true** using PARLOG's *call/1* before commitment is made to the clause.

6.2.3. Value passing

For the implementation of *value passing* we base our code on the following outline:

Example 3 Value passing.

```
match(    send(Gate,Val1,Sort,Condition1,Out1),
         receive(Gate,Val2,Sort,Condition2,Out2))<-
     Val2 is  Val1&
     call(Condition1) & call(Condition2):
     Out1 = ok, Out2 = ok.
```

Note that in, as in value matching, the Gate names and Sorts must be the same, but Val2 is ground to the value of Val1, using the predicate is/2. The Conditions are then evaluated to **true** before commitment to the body of the clause[3].

6.2.4. Value Generation

A detailed discussion of this is to be found in [Gilbert86], with reference to the older semantics of the parallel operator which permitted 1:1 synchronisation only. The outline code is:

Example 4 Value generation.

```
match( receive(Gate,Val1,Sort,Condition1,Out1),
         receive(Gate,Val2,Sort,Condition2,Out1))<-
         randomise(Val,Condition1,Condition2,Num):
         Val1 = Num, Val2 = Num,
     Out1 = ok, Out2 = ok.
```

Note that this relation generates a random number which satisfies both Conditions, and thus again assumes Sort to be Nat.

6.2.5. The parallel operator.

The semantics of LOTOS's parallel operators differ widely from that of PARLOG. Hence the parallel operators are mapped into a *par* program, written in PARLOG, which emulates their behaviour. 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,

---

[3] In order to preserve clause safety and prevent the binding of a variable in an input position in the guard, the code used in the translator makes a *copy* of Val2 and Condition2 before performing the other guard tests.

or by other *par* processes), and produces a stream of synchronisation signals.

The mode declaration of the *par* program is:

```
mode par(in-1?,in-2?,selected-gates?,hidden-gates?,out-stream^).
```

6.2.6. The parallel operator and multi-way synchronisation.

The code for the parallel operator and synchronisation has to be considerably more complex when value generation is considered along with multi-way communication.

The latest draft standard for LOTOS [ISO87], differs from that used in the original development of the translator [ISO/BSI86], and permits multi-way synchronisation in the use of the parallel operator. This semantics, coupled with 'value generation' and the use of predicates permits a constraint-oriented style of specification. This style, however, introduces aspects of undecidability and implies that the route from specification to implementation is very difficult if not impossible. An example in LOTOS, the solution to which is effectively undecidable, would be [van Eijk88]

```
choice x,y :int [] [3x↑2 + y↑5 = 0 ] -> g;B
```

Constraint-oriented specification is used in the description of multi-partner negotiation for value generation according with restriction by predicates:

```
g?x:nat [x<100] || g?y:nat [y>0] || g?z:nat [z*z > 200]
```

If, on the other hand, the style of specification is oriented towards implementation, and the use of multi-way communication avoided, the implementation of the specifications is much easier. A more detailed discussion of specification styles and their effect on implementability can be found in [van Eijk88].

The new semantics of the parallel operator are implemented in PARLOG as processes which manipulate the streams produced by processes and perform synchronisation across all streams involved. The code is complex, and is under development at present.

6.3. The inactive process: stop

The completely inactive process in LOTOS, **stop**, cannot perform any event. It can be represented in PARLOG by a call suspending on a variable which will never become bound.

6.4. Selection and hiding

Selection is performed by *par/2*, in that only signals naming common gates (or all gates in the case of |||) are considered for synchronisation. Hiding is performed by the multi-way communication predicate, in that signals to gates named as hidden are not made available to the outer environment.

6.5. Action prefix and enable operators

Action prefix is represented as the sequential-AND operator in PARLOG, as is the enable operator. Value passing is not permitted for the latter operator since the ADT part of LOTOS has not yet been incorporated into the translator.

6.6. The choice operator

The choice operator in LOTOS does not map directly into the or-parallel operator of PARLOG, since no output can be made in a guard in PARLOG before commitment to a clause. Instead, choice is represented as a list of possible offers contained within signals produced by processes. 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)
```

The LOTOS choice operator was not fully implemented in the translator with respect of the environment's interaction with the process offering the choice. Although provision has been made in the design of the **signal** predicate for choices to be sent to the *par* process and a reply to be received indicating the successful signal, this facility has not been incorporated into the code generator. In order to implement the full semantics of the choice operator, all behaviour expressions must initially offer one synchronisation message, and if it is selected, the remainder may be offered. Due to the properties of PARLOG guards, no output bindings can be made unless committal has been made to the whole clause. PARLOG representations of behaviour expressions would consist of two predicates, the first offering an initial signal, and the next offering the rest. This would present difficulties if one process invokes others and the choice is offered at lower levels of the instantiation process.

6.6.1. Guarded expressions

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.

6.6.2. Internal events

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.

6.6.3. Disable

The disable operator was originally not implemented, and expansion of disable expressions into those containing choice and sequential operators was employed. However, the present translator makes use of PARLOG's three argument metacall *call/3* [Clark86], to implement LOTOS's disable operator. This metacall is derived from *call/1*, a metalevel programming facility similar to that of Prolog and Lisp.

In its simple form the **call/1** predicate has mode declaration `call(goal?)`. The logical reading of *?call(goal)* is the same as that of *?goal*. A call with a variable as a goal suspends until the variable is instantiated to a term denoting a PARLOG clause body (a relation call or conjunction). A program can thus evaluate calls which are determined at run-time.

The PARLOG system predicate *call/3* has mode `call(Call?, Status^, Control?)`. *Control* is a stream of messages which can be used by a supervisor or monitor program to control the evaluation of the Goal. The acceptable messages on the control stream are: **stop**, **suspend** and **continue**. The Status argument, a variable at the time of the call, is instantiated by the call to a stream of messages reporting key states in the evaluation of the call. The last message will be one of: **failed**, **succeeded** or **stopped**, indicating the form of termination; **stopped** indicating premature termination due to an input **control** message on the **Control** stream.

We use this predicate to implement `[>/2` in our system by adapting *call/3* to output the message **started** on the status stream when the reduction of a call has started:

Example 5 PARLOG Disrupt.

```
mode ? [> ?.
A [> B <-
        call(A,S1,C1), call(B,S2,C2), arbitrate(S1,C1,S2,C2).

mode arbitrate(S1?,C1^,S2?,C2^).
arbitrate([success|S1],C1,S2,[stop|X]).
arbitrate(S1,[stop|X],[started|S2],C2).
arbitrate([Other|S1],C1,S2,C2)<-
        Other \== success, S2 \== [started|X]:
        arbitrate(S1,C1,S2,C2).
```

The first clause of *arbitrate/4* will detect if A has terminated successfully, and binds C2 to **[stop|X]**, thus preventing the evaluation of B. However, *arbitrate/4* may in its second clause detect when the evaluation of B starts (in a non-suspended state, if A has not yet terminated), and bind C1 to **[stop|X]**, aborting the evaluation of A.

7. Present state of the translator

A trial version of a source-to-source translator was designed, capable of converting a subset of LOTOS in to PARLOG. The implementation was constrained both by the incompleteness of the LOTOS to PARLOG mappings described in the section on Development, and by problems encountered in the construction of the translator itself (see below). The ADT part of the LOTOS language was not implemented, and there is no type-checking incorporated in the parser.

The translator was modularised, allowing flexibility in development, promoting ease of testing and facilitating software maintenance. All the software was written in PARLOG. The modules developed were:

(i) Tokeniser
(ii) Parser
(iii) Code Generator

The modules are executed in parallel, exploiting the stream-AND parallelism of PARLOG; a typical usage would be the call:

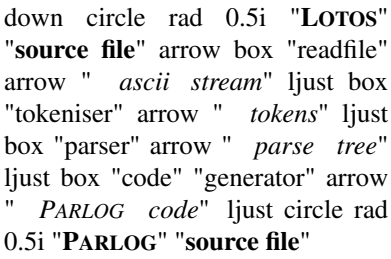
Example 6 .

```
mode translate(LotosFilename?, ParlogFilename?).
translate(LotosFilename, ParlogFilename)<-
     readfile(LotosFilename,Asciistream),
     tokeniser(Asciistream,Tokens),
     parser(Tokens,ParseTree),
     code-generator(ParseTree,ParlogCode),
     write-code(ParlogFilename,ParlogCode).
```

Note that while *tokeniser* passes the tokens in a stream to *parser*, the latter process incrementally produces a complex structure (channel)[4], the parse tree, thus permitting more concurrency than would a stream in its consumption by *code-generator*. A pictorial representation of the translation process is given in the figure below.

[4] A *stream* (an incrementally constructed list) is a special form of a channel.

Figure 1 Translator program.



7.1. Reading a file to produce a stream of ascii characters.

The routine which reads a file and produces a stream of ASCII characters uses the PARLOG primitives which are implemented in "C". Sequential-AND operator are used in the code for **readfile** to ensure the correct sequence of file operations.

7.2. The Tokeniser

The **tokeniser** processes the stream of integers representing ASCII characters produced by **readfile**. In implementing this program the BNF for LOTOS has been adhered to, but all printable characters except for spaces and tabs are included in the output stream. The **tokeniser** predicate consumes a list of integers representing ASCII characters, and recursively replaces those groups of characters which represent LOTOS lexical tokens by the appropriate token. Detection of the empty list on the input stream determines the termination of the execution of **tokeniser**. Comments in LOTOS, which are delimited by "(*" and "*)" are not processed.

7.3. The Parser

7.3.1. Overall design

The **parser** was implemented as a top-down recursive descent parser, using the parallel logic programming technique of difference-streams. A tuple (data-structure) is produced representing the parse-tree, whose general form is:

```
spec(specname(Specid),Params,Externaltypes,Definition,Rest)
```

where Rest is a list of local definitions, the members of which are tuples of the form:

```
process(procname(Name),Params,Definition,Defns-rest)
```

Each of the **process** tuples has its own local definition part, Defns-rest, which is itself a list of **process** tuples. The attributes of each tuple are at first left uninstantiated until ground later during the parsing. This technique has some similarities with "fixup" which is used when compiling: addresses of parameters can be left undefined until fixed during a subsequent pass.

7.3.2. Error checking

No error checking algorithms are incorporated in the parser which behaves like a standard predicate in PARLOG: when called it either succeeds and the parse-tree tuple is produced as output, or the call fails. The parser was not designed to include diagnostic abilities for the time being. As it was decided to exclude the ADT part of LOTOS from the implementation, type-checking is not performed by the parser.

7.3.3. Implementing the BNF

The sections from the LOTOS BNF which were implemented in the parser are:

```
6.2.1  Specification
6.2.2  Definition block
6.2.6  Process Definitions
6.2.7  Behaviour Expressions[5]
6.2.8  Value-expressions
6.2.9  Declarations
6.2.10 special-identifiers
```

Those sections not implemented at all were:

```
2.3  Data-type-definition
2.4  P-expressions.
```

Section 2.5 (sorts, operations and equations) was only implemented for *sort-list* and *simple-equation*.

7.3.4. Amendments to the BNF

The BNF contains one statement which is not LL1, and so presents difficulties for a straightforward implementation. The relevant section is 6.2.8:

```
value-expression =  [value-expression operation-identifier]
                     simple-expression
```

To facilitate writing the PARLOG code this was transformed into the following equivalent form:

```
value-expression = simple-expression <empty>
        | simple-expression operation-identifier value-expression
```

7.4. The Code Generator

The code generator was implemented as a recursive traverse of the parse-tree, creating a list structure suitable for input to the PARLOG library predicate **write_source**.

PARLOG predicates described above were used to map structures encountered in the parse tree on to PARLOG programs. Atomic events, value matching, value passing, value generation and the stop process are all translated successfully by the code generator. LOTOS expressions which can be translated are:

(i) Enable expressions (no value passing)
(ii) Parallel expressions
(iii) Hiding & gate selection expressions
(iv) Choice expressions (in the context of guarded expressions only)
(v) Guarded expressions
(vi) Action-prefix expressions
(vii) Atomic expressions
(viii) Value expressions (a limited subset)

In addition, the block-structure of LOTOS as regards process declarations has been implemented in the translations. At present only self-recursion is translated correctly. No run-time support is included for variable references: outline methods to implement both this and correct referencing of process call are outlined later in this paper.

8. Using the lopar software

The software runs under Unix™ BSD 4.2+ and requires the PARLOG SPM system [Foster86]. Input to the translator is from a file containing a LOTOS specification. The translator will accept full LOTOS syntax, but only the dynamic part is converted from the parse tree into abstract code; the latter is then written out as

[5] Not sum-expressions or par-expressions.

PARLOG source code to file. The PARLOG code can then be compiled by the SPM compiler, producing a file containing PARLOG 'object' code acceptable to the SPM emulator.

In order to execute the PARLOG object code produced by the translator, it must be loaded into the SPM along with library routines needed, for example the code for the parallel operators and synchronisation routines. Also required is the LOTOS supervisor, whose function is equivalent to an observer of the execution of the specification. This supervisor enables the observations to be made available to the user, either as a trace on the console, or as output to file.

9. Improvements

Implementation of the ADT part of LOTOS would also enable improvements to be made in the language translator. Sort checking would then become possible. This would facilitate the inclusion of error checking facilities in the parser, allowing it to be used in a diagnostic mode to verify correct use of variables and expressions in a specification. Included in the diagnostic capabilities should be the ability of the parser to flag errors, to attempt to correct them and to carry on with the parsing without halting at the first error it encounters. A user friendly feature would be to return the user to the LOTOS source code under the control of a standard text editor at the point of the first non-recoverable error.

More work has to be done in the area of code generation. Translation of the choice operator needs to be included. The generator is lacking a full implementation of the scope rules and block structuring of LOTOS; some guidelines have been given in [Gilbert86] regarding the implementation of an environment stack both at translation time and at run time. This would be best implemented after the mapping of ACT-ONE into PARLOG has been achieved.

10. Comments on implementing LOTOS using PARLOG

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. Multiway synchronisation is a boon to specifiers, but is a very hard construct to implement; problems in this area are related to those of implementing constraints in logic programming [Lassez87, Choquet87]. It may well be that the construction of a LOTOS translator would be facilitated if only implementation-oriented styles of specification were acceptable as input, and the use of multi-way communication avoided.

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 processes. 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.

11. Conclusions

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. Neither the tokeniser nor parser perform any kind of error checking, syntax or otherwise. The code generator does not implement ACT-ONE and thus does not perform translation-time type-checking. Moreover some of the LOTOS constructs which can be mapped into PARLOG are not incorporated in the code generator, notably the choice operator in any context other than guarded expressions. LOTOS's scope rules are incorporated in the code generator regarding

process declarations, but recursive process instantiations are only supported for self-recursion. No provision is made for scope rules applied to the declaration of variables, as a run-time environment is not implemented: the LOTOS programmer is encouraged to declare explicitly as input all variables needed in any process.

The declarative nature of PARLOG facilitated the task of building the prototype translator, and modifications are relatively easy to perform. The system was used successfully to translate some simple LOTOS specifications, and 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.

Acknowledgements

This work was partly carried out while I was in receipt of an SERC grant, studying for the MSc in Computing, and also while I was funded by Alvey on "Implementation and Applications of PARLOG", Project number 043/098.

References

Bolognesi87.
Tommaso Bolognesi and Ed Brinksma, ‘‘Introduction to the ISO Specification Language LOTOS,’’ *Computer Networks and ISDN Systems*, vol. 14, no. 1, pp. 25-59, Elsevier Science Publishers, 1987.

Choquet87.
Nicole Choquet, *Need for a Prolog handling constraints in a software engineering application,* CR-CGE, Laboratoires de Marcoussis, Marcoussis, France, 1987.

Clark86.
Keith Clark and Steve Gregory, ‘‘PARLOG: Parallel programming in Logic,’’ *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 1, Jan 1986.

Ehrig83.
H. Ehrig, W. Frey, and H. Hansen, ‘‘ACT ONE: An algebraic specification language with two levels of semantics,’’ Bericht Nr 83-03, Techische Universitaet, Berlin, 1983.

Foster86.
Ian Foster, Steve Gregory, Graem Ringwood, and Ken Satoh, ‘‘A Sequential Implementation of PARLOG,’’ *3rd International Conference on Logic Programming, London. July 1986*, Dept of Computing, Imperial College, London. UK, March 1986.

Gilbert86.
David Gilbert, ‘‘Implementing LOTOS in PARLOG,’’ MSc Thesis, Department of Computing, Imperial College, London, UK, September 1986.

Gilbert87.
David Gilbert, ‘‘PARLOG: a tutorial introduction.,’’ *Proceedings of Parallel Processing and Supercomputing*, Begian Institute for Automatic Control, Antwerp, Belgium, November 19-20, 1987.

Gilbert87b.
David Gilbert, ‘‘Executable LOTOS: Using PARLOG to implement an FDT,’’ *Proceedings of IFIP Protocol Specification, Testing and Verification: VII, Zurich, Switzerland, 5-8 May 1987*, Elsevier Science, North-Holland, Amsterdam, Netherlands, 1987.

Gregory87.
Steve Gregory, *Parallel Logic Programming in PARLOG: The Language and its Implementation,* Addison-Wesely, London, UK, 1987.

ISO87.ISO, *Revised Text of 2nd ISO / DP 8807 - LOTOS,* ISO, March 1987.

ISO/BSI86.
ISO/BSI, ‘‘Working document DP 8807 LOTOS/86/1 FDT:278,’’ IST/21/1/3 BSI 86/61840, April 1986.

Lassez87.
Catherine Lassez, ‘‘Constraint Logic Programming,’’ *BYTE*, pp. 171-176, BYTE, August 1987.

Milner80.
Robin Milner, *A Calculus of Communicating Systems,* Lecture Notes in Computer Science, 92, Springer-Verlag, Berlin, 1980.

van Eijk88.
Peter van Eijk, ‘‘Software tools for the specification language LOTOS,’’ PhD thesis, Department of Informatics, University of Twente, Enschede, Netherlands, January 1988.